J3W for Linux 入門講座

水谷純
http://www.nk.rim.or.jp/~jun/index.html

第4回　継承と階層構造

前回（2001年11月号）では、三次元の物体の形状を作成する方法について説明しました。形状を作成する上で、頂点の座標と面を登録するのは結構大変です。今回は、いろいろな形状を簡単に生成できるように「回転体を生成するクラス」を作成します。その回転体クラスを使って、クラスの継承の機能と物体の階層構造(多関節オブジェクト)を利用した簡単なプログラムを紹介します。物体の形状をプログラミングする必要があるJ3Wですが、複雑な形状と動きの3Dアニメーションを、数十キロバイト程度のデータ（j3dファイル）を配布するだけで済むという利点を持っています。

■クラスの継承

J3C言語は「同じクラスのインスタンスでも自分自身のデータメンバとメソッドしかアクセスできない」という特徴を持っています。別のクラスのメソッドやデータを操作することはできません。C++やDelphiでは、クラスの継承を利用しなくてもCやPascalのプログラムとしてアプリケーションが作成できてしまいます。これはCやPascalのプログラマには利点でもありますが、一方ではオブジェクト指向の考え方を身に付けるには邪魔にもなります。

幸か不幸か、J3Cでは継承はほとんど不可欠です。三次元の物体をプログラムしていく上で、例えば右手と左手というように、よく似た性質の物体が必要になることが多くあります。また、位置や大きさ、色だけ異なる物体が必要な場合もあります。既存のクラスを継承して、異なる部分だけを再定義すれば、簡単にクラスを追加できます。既存のクラス（物体の定義）を再利用しやすくするには、継承する場合のことを考えてクラスを設計しなければなりません。

■回転体クラスと継承

形状の作成は、頂点と面を1つずつ指定するのではなく、ある規則に従って頂点を生成するプログラムを作成すると楽になります。回転体とは、二次元の形状をある回転軸を中心に回転させることによって生成する三次元形状です。

図1は、X軸とY軸を含む平面（XY平面）に4つの頂点を置いたところを示しています。頂点0と頂点1はX軸上（YZ座標がともに0）にあり、頂点2と頂点3はX軸から離れたところ(Y座標が正、Z座標が0)にあるものとします。ここでX軸を中心に紙面奥に向かって回転させると立体ができます。この立体を「回転体」と呼びます。

4つの頂点をX軸の周りに回転させると頂点3の軌跡は円を描きます。これを一度に60度ずつ回転させると六角形になります。図2はX軸上に視点を置いて、Xが増加する方向を見た場合を示しています。頂点1から頂点0の方向を見ていると考えてください。このとき頂点2も頂点3と同様に、頂点4、頂点6、頂点8、頂点10、頂点12といった頂点で六角形を描いています。このようにして頂点を生成すると図3の左側のようになり、面を貼れば右側の回転体が作成できます。Z軸は紙面奥が正の方向になります。

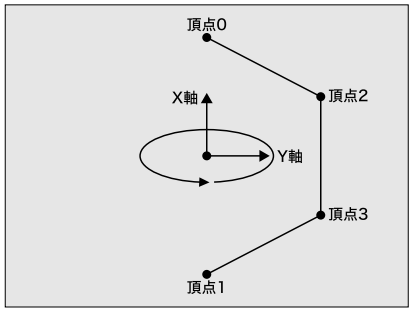
それでは実際にCRevolutionという名の回転体クラスを作成します（リスト1）。まずXY平面上の頂点を図1のように設定します。X軸上に並ぶ2つの頂点として、X座標をvx配列に、Y座標をvy配列に分けて格納します。頂点0「vx[0],vy[0]」と頂点1「vx[1],vy[1]」に続いて、必要なだけ頂点を配列に設定します。続いてvColor配列に対応する頂点の色を指定します。

Setupメソッドに、頂点0と頂点1を除いた頂点数と回転方向の分割数をセットして呼び出すと、頂点と面が回転体として登録されます。CRevolutionクラスは、配列と変数で100のデータメモリを使っています。データメモリの領域はスタックとしても使いますから、インスタンスを生成する場合には「new(CRevolution, 200)」のように150以上のデータメモリを確保してください。

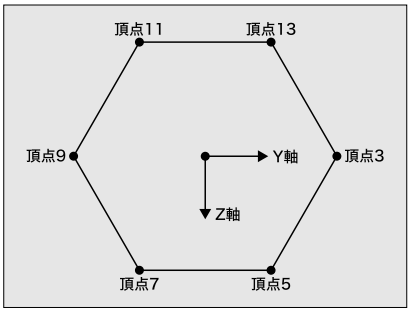
XY平面で与えた頂点を回転させて、X、Y、Zの座標を求めるために次の式を使っています。頂点番号iの頂点がj回の回転（一度にTだけ回転）した場合の座標を求めています。

```
X = vx[i]
Y = vy[i] * cos(T*j)/10000
Z = vy[i] * sin(T*j)/10000
```

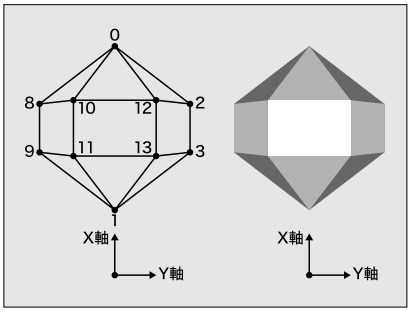
【図1】XY平面上に4つの頂点を置いたところ



【図2】X軸の周囲を回転する頂点3の軌跡



【図3】60度ずつ回転してできた回転体



サイン（sin）、コサイン（cos）、平方根などの算術関数では結果が1万倍されて返るため、それを調整するために10000で割る必要があります。リスト1では、各座標値を求める式をそのままVertexメソッドの引数として渡しています。

CRevolutionクラスは、INIT()、RUN()、EVENT()のどのメソッドも空のため、new組み込み関数でインスタンス化しても何もしないクラスとなります。具体的な形状や動作はCRevolutionを継承するクラスで定義します。

リスト2はCRevolutionを継承して球体を生成するsphereクラスの定義です。INITメソッドから形状定義のためのsphereメソッドを呼び出して球体を生成しています。NewObject(3000,3000)で3000頂点、3000面を確保していますが、頂点や面の数が増えてもいいように余裕を持たせています。

CRevolutionに追加したsphereメソッドは、球の半径radius、経線の分割数div、球の色colorを指定して呼び出すことによって、半円上の頂点をvx、vy配列に設定します。その後、CRevolutionから継承したSetupメソッドを14（＋2）個の頂点で、断面が30角形になるような回転体の頂点と面を登録します。RUNメソッドでは球体を運動をさせています。ここでは簡単に8秒間で360度のピッチ回転と3秒間で18度のヘッド回転を繰り返しています。

リスト3でmainクラスを定義します。INITメソッドでsphereクラスのインスタンスを生成し、mainクラスは視点となりますが、形状を持つ必要がないためNewObject(0, 0)で空の物体を生成しています。視点のように、位置や姿勢の情報が必要なインスタンスは、頂点や面を登録しない場合でもNewObjectの実行が必須になります。RUNメソッドでは、キー入力を調べて、ESCキーが押されるとすべてのインスタンスを終了させるようにしています。結果としてプログラムの実行が終了します。リスト3を次のコマンドでコンパイルして実行し、動作を確認してください。

```
j3cc -r rv_main.j3c
```

さて、球体が回転しても動きが分かりづらいため、実行してもあまり面白くありません。そこで、CRevolutionの「帯状に色が異なる回転体」を生成する機能を使い、sphereクラスを継承したsphere2クラスで新しい球体を作成します。球体の形状は、sphereクラスのsphereメソッドをそのまま使うことができますから、INITメソッド中でsphereメソッドで指定された色を部分的に変更することにします（リスト4）。

リスト5は、リスト3がsphere2クラスを使うように「<<<< 変更」の部分を変更したもの

【リスト1】回転体クラスCRevolution（c_revolv.j3c）

```
class CRevolution {
    final int MAX = 30;
    volatile int vx[MAX];                   // 頂点のX座標
    volatile int vy[MAX];                   // 頂点のY座標
    volatile int vColor[MAX];               // 面の色
    volatile int vWork[6];                  // 作業用配列
    volatile int T, m, i, j;

    int Vertex(int x, int y, int z) {       // 頂点の登録
        RX = x; RY = y; RZ = z;
        Point();
    }

    //--------------------------------------
    // 面と頂点を登録
    // n は頂点数 -2, m は円周の分割数
    //--------------------------------------
    int Setup(int n, int m) {
        for(i = 0; i <= n+1; i=i+1)         // 初期頂点（n+2個）の登録
            Vertex(vx[i], vy[i], 0);

        // 頂点座標を計算して、頂点を登録
        T = 2880 / m;                       // 1ステップの角度
        for(j=1; j<m; j=j+1)                // j=0の座標は初期値で登録済み
            for(i=2; i<=n+1; i=i+1)
                Vertex(vx[i], vy[i] * cos(T*j)/10000, vy[i] * sin(T*j)/10000);

        // 面を登録
        if (n > 1) {                // 上下の錐体を除いた面の定義
            vWork[1] = 4;                   // 四角形
            for(j=0; j<=m-2; j=j+1)         // 経線ごと
               for(i=2; i<=n; i=i+1) {      // 緯線ごと
                   vWork[0] = vColor[i];    // 面の色を指定
                   vWork[2] = j*n+i;        // 頂点番号指定
                   vWork[3] = j*n+i+1;
                   vWork[4] = (j+1)*n+i+1;
                   vWork[5] = (j+1)*n+i;
                   Plane(vWork);            // 面の登録
               }

            for(i=2; i<=n; i=i+1) {         // m-1から0への戻り
                vWork[0] = vColor[i];       // 面の色を指定
                vWork[2] = (m-1)*n+i;       // 頂点番号指定
                vWork[3] = (m-1)*n+i+1;
                vWork[4] = i + 1;
                vWork[5] = i;
                Plane(vWork);               // 面の登録
           }
        }

                                            // 上部（X大）の錐体
        vWork[1] = 3;                       // 三角形
        vWork[0] = vColor[0];               // 面の色を指定
        for(j=0; j<=m-2; j=j+1) {
            vWork[2] = 0;                   // 頂点番号指定
            vWork[3] = j*n+2;
            vWork[4] = (j+1)*n+2;
            Plane(vWork);                   // 面の登録
        }
                                            // m-1 から 0 への戻り
        vWork[2] = 0;                       // 頂点番号指定
        vWork[3] = (m-1)*n+2;
        vWork[4] = 2;
        Plane(vWork);                       // 面の登録

                                            // 下部（X小）の錐体
        vWork[0] = vColor[1];               // 面の色を指定
        for(j=0; j<=m-2; j=j+1) {
             vWork[2] = 1;                  // 頂点番号指定
             vWork[3] = (j+2)*n+1;
             vWork[4] = (j+1)*n+1;
             Plane(vWork);
        }
                                            // m-1 から 0 への戻り
        vWork[2] = 1;                       // 頂点番号指定
        vWork[3] = n +1;
        vWork[4] = m*n+1;
        Plane(vWork);                       // 面の登録
    }

    int INIT() { }
    int RUN() { }
    int EVENT() {  }
}
```

【リスト2】球体を生成するsphereクラス（sphere.j3c）

```
import "c_revolv.j3c";

class sphere extends CRevolution {
    int sphere(int radius, int div, int color) {
        volatile int i;
        vx[ 0] =  radius;       //   頂点0
        vy[ 0] =  0;
        vx[ 1] = -radius;       //   頂点1
        vy[ 1] =  0;
        for (i=0; i<div-1 ; i=i+1) {
            vColor[i] = color;
        }
        for (i=1; i<div-1 ; i=i+1) {
            vx[i+1] = radius * cos(i*1440/(div-1)) / 10000;
            vy[i+1] = radius * sin(i*1440/(div-1)) / 10000;
        }
    }

    int INIT() {
        NewObject(3000,3000);
        sphere(500, 16, 3);
        Setup(14, 30);          // 頂点と面を登録
        ClearRegisters();
        SetPosition();          // 位置と姿勢を指定
    }
    int RUN() {
        RotPitch(8000, 2880);   // 360度/8秒でピッチ回転
        RotHead(3000,144);      // 18度/3秒でヘッド回転
    }
}
```

です（2行変更しています）。このように、sphere2はsphereを継承し、sphereはCRevolutionを継承することによって、継承元のクラスに手を加えずに回転体の形状や動作を変更することができます。

先ほどと同様にして、リスト5（rv_main2.j3c）をコンパイルして実行します。すると今度は、帯状に異なる色のある球体が表示されます（画面1）。リスト3のときより動きが分かりやすくなったと思います。リスト4にRUNメソッドを加えて、sphereクラスと異なる動きをさせることもできます。いろいろ試してみてください。

【リスト3】回転体の実行（rv_main.j3c）

```
import "sphere.j3c";

class main {
    volatile int key;
    int INIT() {
        new(sphere, 3000);      // 球のインスタンスの生成
        NewObject(0, 0);        // 視点用の物体の生成
        ClearRegisters();       // RX - RBに0を代入
        RX = -1000;             // 位置の指定 10m 後方
        SetPosition();          // 位置と角度を視点に設定
        See();                  // このオブジェクトを見る
        BackgroundColor(0x000); // 背景色を設定
        GraphMode();            // グラフィックウインドウを開く
        Zoom(0);                // 視野角53度（2倍ズーム）
        Specular(200);
    }

    int RUN() {
        key = InKey();          // キーコード入力
        if (key == 0x1B) Stop();// ESCキーで終了
    }

    int EVENT() {}
}
```

■継承を利用する上での注意点

一般のオブジェクト指向言語には、実行時に呼び出し先のメソッドを決める「動的結合（dynamic binding）」または「遅延結合（late binding）」と呼ばれる機能があります（C++の仮想関数に相当する機能）。しかしJ3C言語にはこの機能はなく、コンパイル時に呼び出すメソッドが決まります。これを「静的結合（static binding）」または「早期結合（early binding）」と呼びます。従って、クラスを継承して一部のメソッドを再定義した場合には、呼び出し元のメソッドも再定義する必要があります。

【画面1】sphere2クラスで色分けした球体

リスト6のプログラムは、クラスBのインスタンスを生成して、10秒待って終了するものです。このプログラムを読んで実行結果を予想してみてください。J3Cのプログラムを解読するには、class mainのINITから始めます。

mainから生成されたクラスBのインスタンスは、クラスAから継承したINITが再定義されていないので、クラスAのINITを実行します。IntroduceMyselfメソッドは、クラスBで再定義されていることから、どうやらこのプログラムの作者は、継承したINITからクラスB用のIntroduceMyselfメソッドが実行されることを期待しているようです。クラスBのインスタンスが実行するクラスAのINIT中のIntroduceMyselfメソッドは、クラスAとクラスBのどちらのIntroduceMyselfメソッドを実行するのでしょうか？

実行すると、画面には「I am A.」と表示されます。残念ながらプログラムの作者が期待した動作にはなってないないようです。この例から分かるように、クラスBのインスタンスが実行するクラスAのINITは、あくまでクラスAのINITです。従って、実行結果は当然クラスAのIntroduceMyselfメソッドを実行することになります。クラスBでもINITを再定義するようにすれば、期待する動作になります。

汎用のメソッドを作成する場合には、条件を

【リスト4】回転体に色を着ける（sphere2.j3c）

```
import "sphere.j3c";

class sphere2 extends sphere {
    int INIT() {
        NewObject(3000,3000);
        sphere(500, 16, 12);
        vColor[0] = 8;          // 部分的に色を変更
        vColor[5] = 2;
        vColor[9] = 10;
        vColor[11] = 1;
        vColor[1] = 14;
        Setup(14, 30);          // 頂点と面を登録
        ClearRegisters();
        SetPosition();          // 位置と姿勢を指定
    }
}
```

【リスト5】rv_main.j3cの変更（rv_main2.j3c）

```
import "sphere2.j3c";           // <<<< 変更

class main {
    volatile int key;
    int INIT() {
       new(sphere2, 3000);      // 球のインスタンスの生成  <<<< 変更
       NewObject(0, 0);         // 視点用の物体の生成
       ClearRegisters();        // RX - RBに0を代入
       RX = -1000;              // 位置の指定 10m 後方
       SetPosition();           // 位置と角度を視点に設定
       See();                   // このオブジェクトが見る
       BackgroundColor(0x000); // 背景色を設定
       GraphMode();             // グラフィックウインドウを開く
       Zoom(0);                // 視野角53度(2倍ズーム)
       Specular(200);
    }

    int RUN() {
        key = InKey();            // キーコード入力
        if (key == 0x1B) Stop();  // ESCキーで終了
    }

    int EVENT() {}
}
```

【リスト6】クラス継承の実験（class.j3c）

```
class A {
    int IntroduceMyself() {
        String("I am A.");
        Char(13);
    }
    int INIT() {
        IntroduceMyself();
    }
    int RUN(){}
    int EVENT(){}
}

class B extends A {
    int IntroduceMyself() {
        String("I am B.");
        Char(13);
    }
}

class main {
    int INIT() {
        new(B);
        Pause(1000);
        Stop();
    }
    int RUN(){}
    int EVENT(){}
}
```

承元（基底）クラスでは、与える引数の値で動作が変わるようなメソッドを用意し、派生クラスでは、位置や姿勢の指定、色の設定、大きさの指定などをINITやRUNから引数で指定して呼び出すようにします。

なお、EVENTメソッドは内部でRUNメソッドから呼び出されるため、EVENTを再定義した場合は、RUNも再定義するようにしてください。EVENTだけの再定義はできません。

■物体の階層構造

人体のように関節を持つ物体を表現するためには、階層構造を持つオブジェクトが必要になります。腰をひねれば肩も腕も頭も動きます。つまり、肩、腕、首の関節を動かさなくても、地面に対する物体の位置や角度は変化します(画面2)。これは、腰の座標系（フレームと呼ぶ場合もある）上に肩の座標系が固定され、肩の座標系に頭の座標系が固定され、という具合に親子関係が形作られているからです。

リスト7は、rodクラスを継承したParent（親）、Child（子）、GrandChild（孫）クラスが3層の階層構造になっている例です。rodクラスはリスト1のCRevolutionクラスを継承して回転体を生成します。Parentクラスのインスタンスは、階層構造の下位に5個のChildクラスのインスタンスを持ち、Childクラスは GrandChildクラスのインスタンスを1つ持っています。リスト7を実行すると画面3のようになります。誌面がモノクロなので分かりにくいですが、実際にはクラスごとに色が変えてあります。中心部がParentクラスで、Parentクラスの動きに合わせてすぐ外側のChildクラス、その外側のGrandChildクラスが連動して動きます。

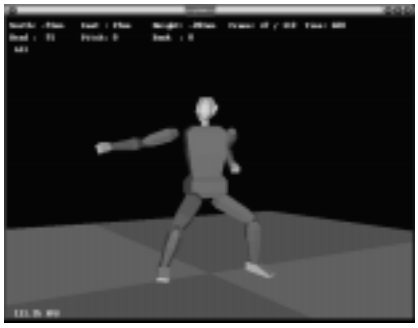
リスト7は、約500の面で構成された物体が、10関節、60自由度で動作します。これはESCキーが押されるまで動き続けます。どの関節もrodクラスのRUNメソッドで定義された7つの回転命令で同じ動きをしていますが、全体としては、かなり複雑な動きをしていると思いませんか？　それぞれのクラスの動きをRUNメソッドで別に定義すれば、いろいろな動きを試すことができます。

階層構造の物体を実現する組み込み関数はchildだけです。childは、インスタンスを生成するnewと、インスタンスが物体として存在するためのNewObjectという2つの組み込み関数の機能を持つ関数です。ただし「生成されるインスタンスがchild組み込み関数を実行したインスタンスのオブジェクト（の座標系）に属している」という点が異なります（図4）。

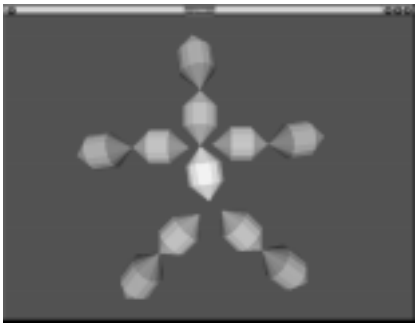
childで生成されたインスタンスは、座標系が親オブジェクトの局所座標系に属している以外は、全く独立した物体として動作します。親となる物体が先に消滅した場合は、その子供たちはそれぞれ空間内で独立した（ワールド座標系に存在する）物体になります。

ここで注意しておきたいことがあります。クラスの継承による階層と、物体の座標系の親子関係による階層構造を混同しないようしてください。未定義クラスエラーが発生する場合には混同している可能性があります。リスト7でrodクラスはインスタンス化されていません。継承されてParent、Child、GrandChildという派生クラスのインスタンスが作られています。従って、Parent、Child、GrandChildの各クラス間に継承関係はありません。これを例えば、「ParentからChildを派生させて、Childから

【画面2】関節を持つ物体は「腰の座標系」を持つ



【画面3】リスト7を実行した様子



【図4】child組み込み関数の役割

GrandChildを派生させて……」とすると、物体の親子関係を定義するchild組み込み関数でクラス間に相互参照が発生し、クラス宣言時またはインスタンス生成時に「クラスが未定義」というエラーが発生します。すでに宣言された識別子（クラス名、メソッド名、変数名などの名前）だけしか参照できないのはJ3Cの仕様です。

■終わりに

クラスの継承と物体の階層構造の実現方法を一気に紹介しました。クラスの継承関係を上手に設計すると、少ないコーディング量で多くのことを実行する効率的なプログラムを作るための強力な武器になると思います。

次回は、物体を外部からコントロールするメッセージの送信とメッセージを処理するEVENTメソッドの使い方、そしてグラフィックウィンドウに文字や線分を重ね書きする方法を解説する予定です。

【リスト7】階層構造の例（hierarch.j3c）

```
import "c_revolv.j3c";

class rod extends CRevolution {
    int rod(int length, int radius, int color) {
        volatile int i;
        vx[ 0] =  length;
        vy[ 0] =  0;
        vx[ 1] =  0;
        vy[ 1] =  0;
        vx[ 2] =  length - radius;
        vy[ 2] =  radius;
        vx[ 3] =  radius;
        vy[ 3] =  radius;
        for (i=0; i < 4; i=i+1) vColor[i] = color;
    }

    int def_shape(int color) {
        rod(250, 80, color);
        Setup(2, 16);
    }

    int INIT() {
        NewObject(3000,3000);
        def_shape(5);
        ClearRegisters();
        SetPosition();
    }
    int RUN() {
        RotPitch(1000,  176);        // 22度/秒でピッチ回転
        RotPitch(1000, -352);        // 44度/秒でピッチ回転
        RotPitch(1000,  176);        // 22度/秒でピッチ回転
        RotHead(2000,  240);         // 15度/秒でヘッド回転
        RotHead(2000, -480);         // 30度/秒でヘッド回転
        RotHead(2000,  240);         // 15度/秒でヘッド回転
        RotBank(1000, 1440);         // 18度/秒でヘッド回転
    }
}

class GrandChild extends rod {
    int INIT () {
        def_shape(2);
        ClearRegisters();            // RX - RBに0を代入
        RX =  250;
        SetPosition();               // 初期位置と角度の設定
    }
}

class Child extends rod {
    int INIT () {
        SetPosition(); // 親からRX-RBで渡される初期位置と角度
        def_shape(3);
        child(GrandChild, 150, 3000, 3000);
    }
}
```

```
class Parent extends rod {
  int INIT () {
      NewObject(3000, 3000);
      def_shape(15);
      ClearRegisters();              // RX - RBに0を代入
      RZ = 150;
      RP = 720;
      SetPosition();                 // 初期位置と角度の設定
      ClearRegisters();              // RX - RBに0を代入
      RX=250;
      child(Child, 150, 3000, 3000); // 子オブジェクト1
      RX=250;
      RY=50;
      RH=720;
      child(Child, 150, 3000, 3000); // 子オブジェクト2
      RY=-50;
      RH=-720;
      child(Child, 150, 3000, 3000); // 子オブジェクト3
      ClearRegisters();              // RX - RBに0を代入
      RX=-50;
      RY= 50;
      RH=1080;
      child(Child, 150, 3000, 3000); // 子オブジェクト4
      RY= -50;
      RH=-1080;
      child(Child, 150, 3000, 3000); // 子オブジェクト5
    }
}

class main {
  int INIT () {
      new(Parent, 150);       // 共通の親を生成
      NewObject(0, 0);        // 視点用の物体の生成
      ClearRegisters();       // RX - RBに0を代入
      RX = -900;              // 位置の指定 9m 後方
      SetPosition();          // 位置と角度を視点に設定
      See();                  // このオブジェクトが見る
      BackgroundColor(0x844); // 背景色を設定
      GraphMode();            // グラフィックウインドウを開く
  }
  int RUN() {
      volatile int key;
      key = InKey();           // キーコード入力
      if (key == 0x1B) Stop(); // ESCキーで終了
  }
  int EVENT() {}
}
```

■j3wの3Dエンジンの使用法

これまでj3wの3Dエンジン部分の概要を説明してきましたが、今回はj3wのソースを使って独自の3Dアニメーションのプログラムを作成してみましょう。j3w-643/source/j3w/以下のファイルのうち、表Aの19ファイルを使用します。これらのファイルがj3wの3Dエンジンとなっています。

リストAのプログラムmy_3d.cppと表Aのファイルを作業用のディレクトリにコピーしてください。また、gccコマンドでコンパイルしようとすると、コマンドラインで指定するファイルが多く面倒ですから、Makefile(リストB)も用意します。リスト中の「⇒」はTABを表します。空白（スペース）ではエラーになります。TABを使う必要があるのはmakeコマンドの仕様です。作業用のディレクトリで次のように実行すると、my_3dという実行ファイルが作成されます。

```
make -f Makefile.my_3d
```

次のコマンドを入力すると実行できます。

```
./my_3d
```

単に3角形が回転するだけのプログラムです(画面A)。回転速度はCPUの速度で決まります。forループ中の「p->Axis.rotate(3, 1, 2)」で回転を行っていますが、この部分を「p->Axis.move(Vector(x,y,z))」とすると、三角形は表示されるたびに「(x,y,z)」で指定した方向に平行移動します。「eye->Axis.rotate(..)」、「eye->Axis.move(..)」とすれば視点が移動します。　（水谷純）

【表A】使用する関数

ヘッダファイル	C++ファイル
axis.h	axis.cpp
hobj3d.h	hobj3d.cpp
misc3d.h	misc3d.cpp
object3d.h	object3d.cpp
pal256.h	pal256.cpp
screen.h	
scrnx.h	scrnx.cpp
spaceh3d.h	spaceh3d.cpp
tpolygon.h	tpolygon.cpp
vertex.h	vertex.cpp

【画面A】リストAの実行例

【リストA】3Dアニメーションプログラム（my_3d.cpp）

```
#include "scrnx.h"
#include "spaceh3d.h"

const int COLOR = 3;                                    // 物体の色
TSpaceH3D sp(2, 50, 4000000);                           // 空間生成
ScrnX sc;                                               // ウィンドウ生成
int tmp[16];                                            // 頂点指定用
THObj3D *eye, *p;                                       // 物体のポインタ

int main() {
    sc.set_mode(1);
    sc.SetPaletteEntry(255, 60<<8, 40<<8, 90<<8);       // 背景色の指定
    sp.set_screen(&sc);                                 // スクリーンの登録
    int n = sp.append_object(0, 0);                     // 視点の生成
    eye = sp.get_object(n);                             // 視点の取得
    eye->Axis.set_origin(-400, 0, 0);                   // 視点の位置
    eye->Axis.attitude(0, 0, 0);                        // 視点の姿勢
    sp.set_eye(eye);                                    // 視点の登録
    n = sp.append_object(20, 10);                       // 物体の生成
    p = sp.get_object(n);                               // 物体の取得
    p->Axis.set_origin(0, 0, 0);                        // 物体の位置
    p->Axis.attitude(0, 0, 0);                          // 物体の姿勢
    p->Polygon.Vertex.Add(Vector(0,0,200));             // 物体の頂点登録
    p->Polygon.Vertex.Add(Vector(0,200,-200));          // 物体の頂点登録
    p->Polygon.Vertex.Add(Vector(0,-200,-200));         // 物体の頂点登録
    tmp[0] = 0; tmp[1] = 1; tmp[2] = 2;                 // ポリゴン頂点指定
    p->Polygon.Add(3, COLOR, tmp);                      // ポリゴン登録
    tmp[0] = 2; tmp[1] = 1; tmp[2] = 0;                 // ポリゴン頂点指定
    p->Polygon.Add(3, COLOR-1, tmp);                    // ポリゴン登録
    for(int i=0; i<10000; i++) {
        p->Axis.rotate(3, 1, 2);                        // 物体の回転
        sp.display(0);                                  // 3D空間の構築
        sc.update();                                    // 表示
    }
}
```

【リストB】Makefile（Makefile.my_3d）

```
TARGETNAME = my_3d

CC   = g++

CFLAGS = -O2 -Wall -DLINUX
CFLAGS += -I/usr/X11R6/include -L/usr/X11R6/lib

SRCS = axis.cpp  hobj3d.cpp  \
    scrnx.cpp misc3d.cpp  object3d.cpp  spaceh3d.cpp \
    tpolygon.cpp  vertex.cpp pal256.cpp my_3d.cpp

LIBS = -lm -lX11

OBJS = $(SRCS:.cpp=.o)

.SUFFIXES: .c .cpp .o

.cpp.o :
⇒       ${CC} ${CFLAGS} -c $<

all : ${TARGETNAME} ${TARGETNAME_ASM}

${TARGETNAME} : $(OBJS)
⇒       $(CC) ${CFLAGS} -o ${TARGETNAME} $(OBJS) $(LIBS)

clean:
⇒       rm -f core
⇒       rm -f *.o
⇒       rm -f ${TARGETNAME}
```

注）⇒はTABを表します。空白（スペース）ではエラーになります。