J3W for Linux 入門講座

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

第5回　メッセージ

前回までの解説で、いろいろな形状の物体をアニメーションできるようになりました。今回は、物体が「勝手に」アニメーションするのを見ているだけではなく、今回は物体を「リアルタイムに操作」する方法を見ていきましょう。

■メッセージの送受信

J3Cでは、クラスは独立して動作し、他のクラスからメソッドを呼び出したり、データメンバ（クラスの変数）を操作することはできません。クラス（のインスタンス）間で情報を受け渡すには「メッセージ」を使います。メッセージは32ビットの整数です。このメッセージを特定のインスタンスに送ると、受信したインスタンスのEVENTメソッドが呼び出されます（図1）。

これまでのサンプルでは、mainクラスのRUNメソッドで「ESCキーが押されるのを待って終了」という動作をしていました。この場合は単に「プログラムの終了」という動作のため、mainクラスが「キー入力を監視して押されるのを待つ」方法で十分でした。これを「ポーリング」と呼びます。しかし、動作中の物体（インスタンス）がキー入力に反応する場合でも、物体の回転や移動という仕事とキー入力の監視を同時に処理しなければならないのでしょうか？　物体の運動に関する部分だけに専念して、キー入力で動作の変更があったときにだけ、特別に決められた動作ができると便利です。

この「特別な場合」に実行する命令を処理するために、J3Cには「EVENTメソッド」が用意されています。EVENTメソッドの役割は「割り込み処理」です。割り込み処理は、他のインスタンス（物体）から送られるメッセージによって起動します。割り込み処理を起動する「割り込み」がメッセージです。これはUNIXのシグナルとシグナルハンドラの関係と同じです。

メッセージを使うと、例えば10秒間前進するという動きをRUNメソッドで繰り返している場合でも、キー入力された瞬間に90度向きを変えることができます。それには、キー入力を監視しているインスタンスが、特定のキーが押されたことを検出した場合に「特定の」インスタンスにメッセージを送るようにします。そのインスタンスは、メッセージに対応するための動作をEVENTメソッドに記述しておきます。メッセージを受け取ると、どのような動作をしていても優先的にEVENTメソッドでプログラムされている動作をします。

メッセージを送るには、Send（インスタンス番号，メッセージ）の形でSend組込み関数を実行します。「インスタンス番号」とは、インスタンスのRQレジスタ変数に設定した数値のことで、インスタンス自身が自由に設定することができます。

new組み込み関数で生成されたインスタンスは、インスタンス番号（J3Wのマニュアルでは「プロセスID」と呼んでいる）が0に初期化されるため、そのままでは生成されたインスタンスを区別できません。同一のクラスから生成された複数のインスタンスに、別々にメッセージを送る方法を見ていきましょう。

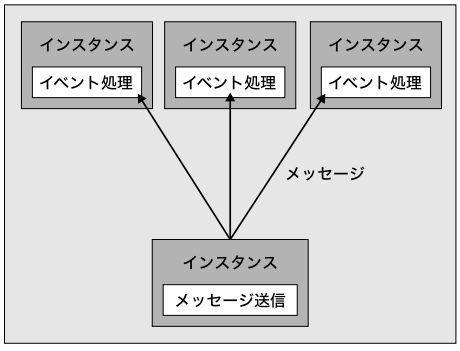
リスト1は、mainクラスが3つのReceiver1クラスのインスタンスを生成し、キー入力された値によって、特定のReceiver1のインスタンス、またはすべてのReceiver1のインスタンスにメッセージを送るものです。Receiver1のインスタンスを区別するには、インスタンス番号をRQレジスタ変数に設定する必要があります。そこで、newを実行したインスタンス（リスト1ではmain）のレジスタ変数は、生成されるインスタンスに引き継がれることを利用します。

リスト1では、RXレジスタ変数の値を毎回変えてnewを実行しています。生成されたReceiver1の各インスタンスは、INITメソッドでRXレジスタをRQレジスタに代入してインスタンス番号を設定しているため、3つのインスタンスが区別できるようになります。次のようにして実行すると、自分のインスタンス番号に続いて受信したメッセージの内容が表示されます。

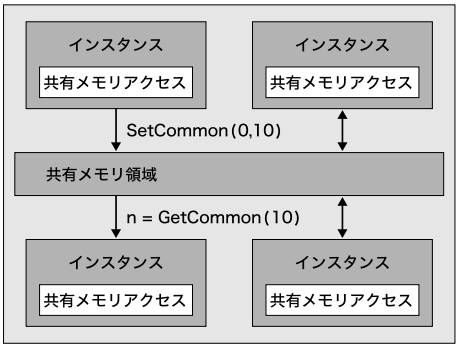
```
j3cc -r event1.j3c
```

リスト1には、この連載で初めて空でないEVENTメソッドが登場しています。EVENTメソッドは、メッセージを受信した場合にだけ、自動的に実行されるメソッドです。RLレジスタ変数にメッセージの内容が設定されます。Receiver1クラスのEVENTメソッドは、RLが1の場合とその他の場合で異なる文字列を表示します。この例のようにEVENTメソッドは、メッセージの種類別に異なる処理ができるように

【図1】クラス間で情報を受け渡す「メッセージ」



【図2】共有メモリ



switch文を使うのが普通です。

Send(インスタンス番号，メッセージ)の形式には例外があります。リスト1のSend(0x7FFF, 10)のように、インスタンス番号に「0x7FFF」(10進数なら32767)を指定すると、「すべて」のインスタンスに同じメッセージを送ることができます。

■共有メモリ

リスト2は、インスタンス番号を設定するために共有メモリを使った例です。共有メモリは、図2に示すように、すべてのインスタンスからアクセスできるメモリ領域です。共有メモリは32ビットの整数の配列で、読み込み時にn=GetCommon(アドレス)とすると、変数nに共有メモリの値が入ります。書き込み時には、SetCommon(値，アドレス)として使いますが、

【リスト1】 event1.j3c

```
// 受信したメッセージを表示するクラス
class Receiver1 {
  int INIT() {
     RQ = RX;              // インスタンスの識別番号を設定
  }
  int RUN() {
     Wait();               // メッセージ受信を待つ
  }
  int EVENT() {
     switch(RL) {          // メッセージ受信時の処理
        case 1 :
           Char('[');
           Number(RQ);  // 識別番号を表示
           String("] recieve 1"); Char(13);
           break;
        default :
           Char('[');
           Number(RQ);
           String("] recieve Unknown Message :");
           Number(RL);  // 受信したメッセージを表示
           Char(13);
     }
  }
}

// キー入力されたら他のインスタンスにメッセージを送信
class main {
  volatile int key;

  int INIT() {
     RX = 1;               // レジスタ変数経由で識別番号を指定
     new(Receiver1, 50); // Receiverのインスタンス生成（1）
     RX = 2;
     new(Receiver1, 50); // Receiverのインスタンス生成（2）
     RX = 3;
     new(Receiver1, 50); // Receiverのインスタンス生成（3）
  }

  int RUN() {
     key = InKey();              // キーコード入力
     ample.tar.gzwitch (key) {
        case 0x1B : Stop();    // ESCキーで終了
           break;
        case 'A'  : Send(1, 1);  // 1番のインスタンスに1を送信
           break;
        case 'B'  : Send(2, 1);  // 2番のインスタンスに1を送信
           break;
        case 'C'  : Send(3, 1);  // 3番のインスタンスに1を送信
           break;
        case '1'  : Send(0x7FFF, 10);   // すべてのインスタンス
           break;                       // に10を送信
        case '2'  : Send(0x7FFF, 20);   // すべてのインスタンス
           break;                       // に20を送信
     }
  }
  int EVENT() {}
}
```

【リスト2】 event2.j3c

```
// 受信したメッセージを表示するクラス
class Receiver2 {
  volatile int i;

  int SetID() {
     i = GetCommon(0);   // 共有メモリ0番地の値を取得
     i = i + 1;
     RQ = i;             // インスタンス識別番号を設定
     SetCommon(i, 0);    // 共有メモリ0番地の値を更新
  }
  int INIT() {
     SetID();
  }
  int RUN() {
     Wait();             // メッセージ受信を待つ
  }
  int EVENT() {
     switch(RL) {        // メッセージ受信時の処理
        case 1 :
           Char('[');
           Number(RQ);  // 識別番号を表示
           String("] recieve 1"); Char(13);
           break;
        default :
           Char('[');
           Number(RQ);
           String("] recieve Unknown Message :");
           Number(RL);  // 受信したメッセージを表示
           Char(13);
        case 1 : break;
     }
  }
}

// キー入力されたら他のインスタンスにメッセージを送信
class main {
  final int N = 20;      // インスタンスの数
  volatile int i, key;

  int INIT() {
     SetCommon(0, 0);       // 共有メモリ0番地に0を設定
     for (i=0; i<N; i=i+1)  // N個のインスタンスを生成
        new(Receiver2, 50); // Receiverのインスタンス生成
  }
  int RUN() {
     key = InKey();              // キーコード入力
     if (key == 0x1B) Stop();    // ESCキーで終了
     if ((key>='A') & (key <= 'Z'))// [A]-[Z]では特定のインスタンス
        Send(key-'@', 1);        // 1を送信
     if ((key>='0') & (key<='9'))  // [1]-[9]ではすべてのインスタンス
        Send(0x7FFF, key);       // keyを送信
  }
  int EVENT() {}
}
```

このときアドレスの指定と書き込む値の順番に注意してください。RXレジスタに書き込む共有メモリの先頭アドレス、データ数をRYに設定してWrite("ファイル名")を実行すると、共有メモリの内容を32ビット符号付き10進数としてファイルに出力できます。RXレジスタに書き込む共有メモリの先頭アドレスを設定してRead("ファイル名")とすると32ビット符号付き10進数を共有メモリに読み込みことができます。読み込んだデータ数はRYに返ります。

リスト2もリスト1と同じように動作しますが、生成されるReceiver2のインスタンスは、共有メモリの0番地に格納されている値を使って自分でインスタンス番号を設定しています。リスト2のmainクラスは、共有メモリの0番地に0を格納した後、連続して20個のReceiver2のインスタンスを生成していますが、Receiver2のインスタンス番号は、異なる値（1から20）で自動的に設定されます。定数Nの値を変えるだけで区別可能なインスタンスを好きなだけ生成できます。

複数のインスタンスが1つのインスタンスに対して同時にメッセージを送る場合も考えられます。複数のメッセージを受信した場合は、最も大きい値のメッセージ以外は廃棄されます。また、メッセージの送出元のインスタンスは識別できません。インスタンス間で同期しながら複雑な動作をする場合は、共有メモリを使用する必要があるでしょう。

3Dアニメーションをリアルタイムに操作するために、「キーに割り当てたコマンドをメッセージとしてインスタンスに送る」という方法をJ3W付属のサンプルで常用しています。キー入力を常に監視して、キー入力があれば、対応するインスタンスにメッセージを送信することを専門に行うクラスを使います。

たくさんのメッセージや送信先のインスタンスを数値で指定すると、プログラムが読みにくくなります。プログラムが長くなったら、この連載の2回目（2001年10月号）で解説したlibraryを使って数値に分かりやすい名前を付けるようにすべきでしょう。

Column

j3wの内部構造の解説

前回（2001年12月号）のコラムでは、J3Wの3Dエンジン部分を使って、回転する三角形を表示するプログラムmy_3d.oppを紹介しました。my_3d.oppは、PCの速度に回転速度が依存するプログラムで、運動する物体は1つだけという簡単なものでした。物体を運動させているforループ部分を変更して、複数の物体が独立していろいろな動きをするようにするのは大変面倒です。さらに、速度の異なるPCでも物体が同じ速さで動作するようにするには、J3Wではどうすればよいのでしょうか？　今回は、複数の物体の動作の管理と、運動に「時間」の概念を追加する部分を見ていきます。

■仮想CPU

j3wコマンドは、仮想CPUのエミュレータとして設計されています。三次元空間で形状を持ち、運動できる特殊なCPUです。3Dエンジンの機能に加えて、時間の管理と与えられた命令の解析と実行が必要です。仮想CPUの命令の実行はTProcessクラスが担当し、TJobクラスが複数の仮想CPUの管理、時間の管理、仮想CPUが実行するコードの格納を担当しています（図A）。

TJobクラス（図A右）は、TProcessのインスタンスへのポインタを配列として持つことによって、TProcessの実行を管理しています。イベントループがTJob::job_scheduler()を呼び出すたびに、仮想CPUを順次実行して画面の更新を行います。TJob::job_scheduler()の呼び出し時には時刻を記録し、前回呼び出しを行った時刻との差の時間を引数として、活動中のTProcess::Runを順に実行していきます。TProcessのインスタンスは、この経過時間分だけ位置などの状態を更新します。そしてすべてのインスタンスの状態が更新された後、画面を更新します（図B）。その他TJobクラスは、命令コードの格納、共有メモリの保持、グラフィック画面に表示する文字列と線分の情報、実行中のプロセス数などシステム全体のリソースを管理しています。

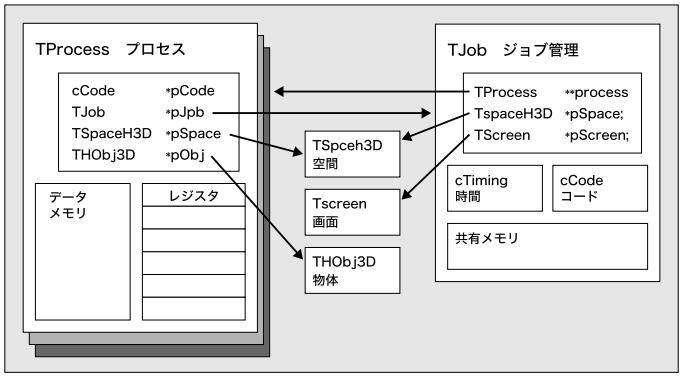
TProcessクラス（図A左）は、J3C言語におけるインスタンスそのものです。J3Cのインスタンスごとに、TProcessクラスのインスタンスが生成されます。TProcessクラスは仮想CPUのレジスタとデータメモリの領域を変数として持っています。仮想CPUの命令コードは、TJobクラスが1つだけ保持していて、TProcessクラスのインスタンスのRUNメソッドによって実行されます。移動や回転などの命令は、「10秒間前進」というように実行時間を指定できますが、TJobsから渡された時間内（PCの速度に依存しますが、10-100msec程度）では命令は終了しません。実行可能時間を消費した時点で、実行中の命令の中間状態を保持したまま制御を他のCPUに渡します。そして、次にCPUへ制御が渡されたときには、保存された中間状態から必要な動作を続行します。すなわち、その命令が終了する場合はプログラムカウンタを次の命令に進め、時間が足りなくて完全に終了する前に制御を返す場合には、前回終了した状態から実行を再開します（図C）。

命令が消費する時間は、移動、回転など時間を渡す必要のある命令の場合は指定した時間を消費しますが、その他の多くの命令の実行時間は0になっています。従って、演算や分岐命令が連続して実行されることが保証されます。

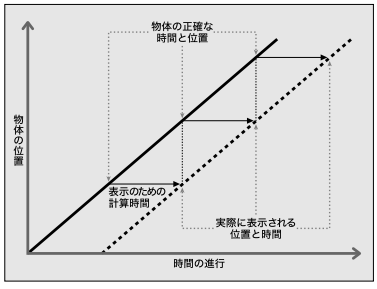
J3Wの内部構造の解説は今回で終了です。次回のコラムはコンパイラであるj3cを解説する予定です。

（水谷純）

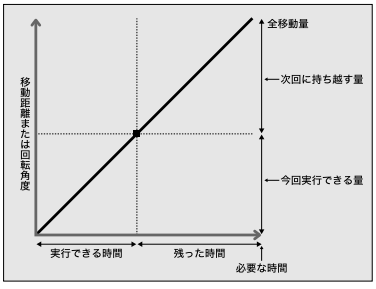
【図A】TJobクラスとTProcessクラス



【図B】TJobクラスによる画面更新のタイミング



【図C】移動距離または回転角度と実行時間の関係



■グラフィック画面への直接描画

フライトシミュレータではヘッドアップディスプレイ(高度や方向、ターゲットの位置を表示する透明なスクリーン)を三次元空間を表示する画面に重ねて文字やグラフィックを表示することで実現しています。J3C言語でも3Dアニメーションに重ねてグラフィックウィンドウに文字や線分を描画する関数を用意しています(表1)。

線分を表示するには、左上隅が(0, 0)、右下が(639, 479)の二次元の座標を使います。グラフィックウィンドウの大きさは幅640ドット、高さ480ドット固定で、変更することはできません。文字は、英数字(ASCII文字)1行80文字で25行の2000文字まで表示できます。日本語は表示できません。文字の表示位置の設定は、左上隅が0、右下隅が1999の一次元の座標で指定します。二次元で指定できないので少し面倒ですが、(80×行)+桁で計算できます。

文字は、gColorで表示色を設定し、gCursorで位置を設定した後に、gString("文字列")、gNumber(数値)、gChar('文字')で表示します。書き込み位置は自動的に更新されるので、書き込みは続けて行えます。文字の位置は、フォントの違いによってWindows版のJ3Wと比べると若干左にずれます。

リスト3は、1秒間あたりの画面の書き換え回数(Frame Per Second)をグラフィックウィンドウの左下に表示するクラスです。INITメソッドでSystemTime()でミリ秒単位の時間、SystemCount()でj3wを起動してからの画面書き換え回数を取得します。RUNメソッドで描画回数/経過時間からFPSを計算して表示しています。桁あふれ(画面書き換え回数が最近のマシンでは大きくなるため)に注意して、小数点以下1桁の精度で求めています。

ウィンドウに直接描画する線分は登録制となっていて、0番から99番の100本までになっています(希望があれば多くします)。登録された線分は位置を書き換えるか、gLineClearで消去されるまで表示されたままになります。線分を表示する場合は、表2のようにレジスタ変数を設定した後、gLine組み込み関数を実行します。

リスト4では、10度おきに36本の線分を放射状に描いています。線分の角度をθ、中心からの距離を始点r1、終点r2としたとき、始点の座標は(r1*cos θ , r1*sin θ)、終点の座標は(r2*cos θ , r2*sin θ)となります(図3)。リスト4では、0番から35番の線分を少しずつ角度を変えて描き直すことによって計器風に回転させています。

リスト3とリスト4のクラスを実行するためのプログラムをリスト5に示します。このプログラムには三次元の物体がないため、筆者の環境(Celeron 450MHz、RivaTNT、XFree86 4.0.3)では、1秒間に450回の描画が行われています(画面1)。CPUの無駄使いですね(^^;。

fps.j3cをインポートして、適当な場所で

【表1】グラフィックウィンドウ描画関数

関数名	機能
gColor	グラフィックウィンドウの文字色を設定
gClear	グラフィックウィンドウの文字列クリア
gCursor	グラフィックウィンドウ文字カーソルの設定(0～1999)
gPosition	グラフィックウィンドウの文字書き込み位置取得
gNumber	レジスタの示す値をグラフィックウィンドウへ出力
gChar	グラフィックウィンドウへ1文字の出力
gStringm	データメモリ文字列をグラフィックウィンドウへ出力
gString	文字列をグラフィックウィンドウへ出力
gLine	グラフィックウィンドウのラインの描画設定
gLineClear	グラフィックウィンドウのラインの消去

【表2】レジスタ変数

変数	意味
RX	始点X座標
RY	始点Y座標
RZ	ライン色
RH	終点X座標
RP	終点Y座標
RB	ライン番号(0～99)

【リスト3】fps.j3c

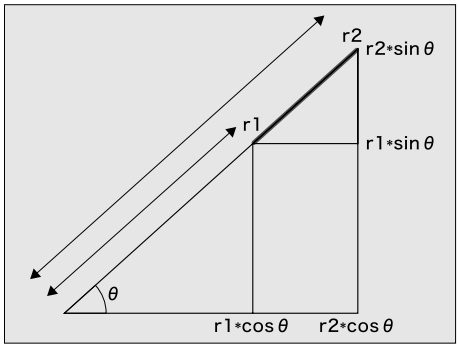
```
class fps {
      volatile int time, time0, time1;
      volatile int count, count0, count1;
      volatile int fps, fps0, fps1;

      int start() {
            time0 = SystemTime();          // 時間を記録
            count0 = SystemCount();        // ループカウントを記録
            Pause(300);
      }

      int update_time() {
            gCursor(2 + 24 * 80);          // 文字位置は(2, 24)から
            time1 = SystemTime();          // 時間を記録
            time = time1 - time0;          // 経過時間を求める
            count1 = SystemCount();
            count = count1 - count0;       // 描画回数を求める
            count = count * 1000;
            fps = count / (time / 10);     // 10msec
            fps0 = fps/10;                 // 整数部
            fps1 = fps - fps0 * 10         // 小数点以下1桁
            gNumber(fps0);                 // 数値を描画
            gString(".");
            gNumber(fps1);
            gString(" FPS ");              // 文字列を描画
      }

      int INIT() {
            start();
      }

      int RUN() {
            update_time();
            Pause(100);
      }
      int EVENT() {}
}
```

new(fps, 50)を実行してインスタンスを起動すれば、どのプログラムにも他の部分に影響することなくFPSを表示する機能が簡単に付加できます。画面2は、先月号（2001年12月号）で使用したサンプルプログラムhierach.j3cにfps.j3cを加えて実行した例です。今月号の付録CD-ROMにhierach2.j3cとして収録しています。500頂点程度の物体を表示しても、90FPS（筆者の環境）と、結構高速に動作していることが確認できます。これはグラフィック周りのベンチマークプログラムになります。

■終わりに

キーボードから複数の物体に情報を伝えたり、文字情報を表示したり、3Dアプリケーションを作成する準備は整いました。次回は3次元空間の見方、「視点」について考えてみる予定です。

【図3】線分の位置の割り出し

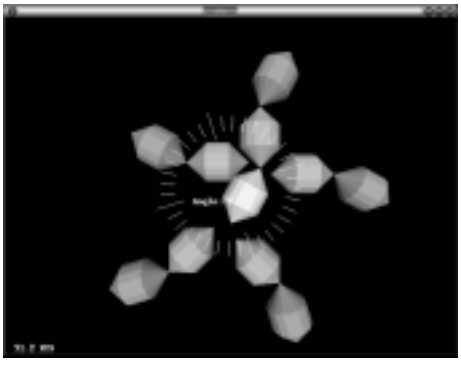


【リスト4】rad_line.j3c

```
class RadialLine {
  volatile int n;
  volatile int m;

  int Line2D(int n, int color, int x1, int y1, int x2, int y2) {
     PushRegisters();                  // レジスタ変数を保存
     RB = n;                           // ライン番号（0～99）
     RZ = color;                       // ラインの色
     RX = x1 + 320; RY = y1 + 240;     // 始点　画面の中心を0とする
     RH = x2 + 320; RP = y2 + 240;     // 終点
     gLine();  // 画面にラインを描画
     PopRegisters();                   // レジスタ変数を復帰
  }

  int RadialLine(int color, int r0, int r1, int d) {
     volatile int i, cosA, sinA;

     for (i=0; i<36; i=i+1) {          // 10度ごとに放射状のライン
        cosA = cos(i * 80 + d * 8);
        sinA = sin(i * 80 + d * 8);
        if (i - i / 9 * 9 == 0)        // 90度の倍数の場合は長く
           Line2D(i, color-1, cosA * (r0-10)/10000, sinA *(r0-10)/10000,
                         cosA * (r1+10)/10000, sinA * (r1+10)/10000)
        else
           Line2D(i, color, cosA * r0 / 10000, sinA * r0 / 10000,
                        cosA * r1 / 10000, sinA * r1 / 10000)
     }
  }

  int INIT() {
     n = 0;
     m = 2;
     gColor(7);   // 文字色は白
  }

  int RUN() {
     RadialLine(7, 80, 100, n);
     gCursor(13*80+38);  // 画面中心付近に文字描画
     gString("Angle:");
     gNumber(n);
     gString("  ");  // 数値の桁数が小さい場合の対策
     if (n >= 180) m = -2;
     if (n < -180) m = 2;
     n = n + m;
     Pause(20);
  }

  int EVENT() { }
}
```

【画面1】リスト5（main_2d.j3c）の実行結果

【画面2】回転体の物体にfps.j3cをインポートした例



【リスト5】main_2d.j3c

```
import "rad_line.j3c";
import "fps.j3c";

class main {
  volatile int key;
  int INIT() {
     new(RadialLine, 50); // インスタンスの生成
     new(fps, 50);
     BackgroundColor(0x000); // 背景色を設定
     GraphMode();// グラフィックウィンドウを開く
  }

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

  int EVENT() {}
}
```