第４章　アニメーション

ポイント：アニメーションの基本的な技術を学ぶ

１．アニメーションの仕組み

２．ワイヤーフレームによる透視図の描き方

３．透視図を描くためのデータ構造

4.1 はじめに

本章では、プレゼンターで最も多用され、有用な情報を使用者に与えるアニメーションについて、その構築技術について学ぶことにする。最初に、アニメーションを行うための仕組みについて勉強する。プレゼンターは、マルチウインドウシステムを用いているため、一つの子ウインドウ内の図形がアニメーションとして動作しているだけでなく、他のウインドウ内の図形やグラフも同期をとって変化させなければならない。これらをどのような仕組みで実現するのか、ここでは、まずこのアニメーションの仕組みについて学ぶことにする。

次に、ワイヤーフレームによる透視図の描き方について勉強する。ここでは、透視図に動きを与えるために描き直しを行わなければならないが、その際、画面がちらつかず滑らかに動作させる方法について勉強する。

4.2 アニメーションの仕組み

本節では、まず、各ウインドウで同期を取りながらアニメーション表示する仕組みについて解説する。各ウインドウでは、異なった図形やグラフが表示されており、これらを全て、同期を取って変化させなければならない。これは結構難しい技術であるが、オブジェクト指向型言語では、この仕組みを容易に構築する事ができる。また、SPACE では、節点座標や変位、応力などは非常に大きなデータ量となる場合があり、そのため、それらを保持する多くの配列はグローバル変数として定義され、安全性よりもメモリーの使用効率を優先している。この２つの特徴をうまく利用して、プレゼンターでは各ウインドウに任意の図形を描画し、アニメーションも効率よく実行する。このアニメーションの仕組みは、ウインドウ管理システムと同様に CSf31View クラスに構築されている。

このアニメーションの仕組みに起動をかける関数は、CSf31View クラスのメンバー関数である OnDynStart()である。この関数は、各種の初

期設定を行った後、pwnd->SetTimer 関数をコールする。VC++ではシステムタイマーを用いて、設定した時間ごとにイベントを発生させる関数を有している。この SetTimer 関数は、イベント発生間隔などの初期設定を行った後、タイマーを起動する。これで、アニメーションの仕組みに起動をかけたことになり、後はアニメーションの仕組みによって自動的に動作することになる。

いよいよ、アニメーションの仕組みについて述べるわけであるが、まず、その仕組みの概略を次図に示す。

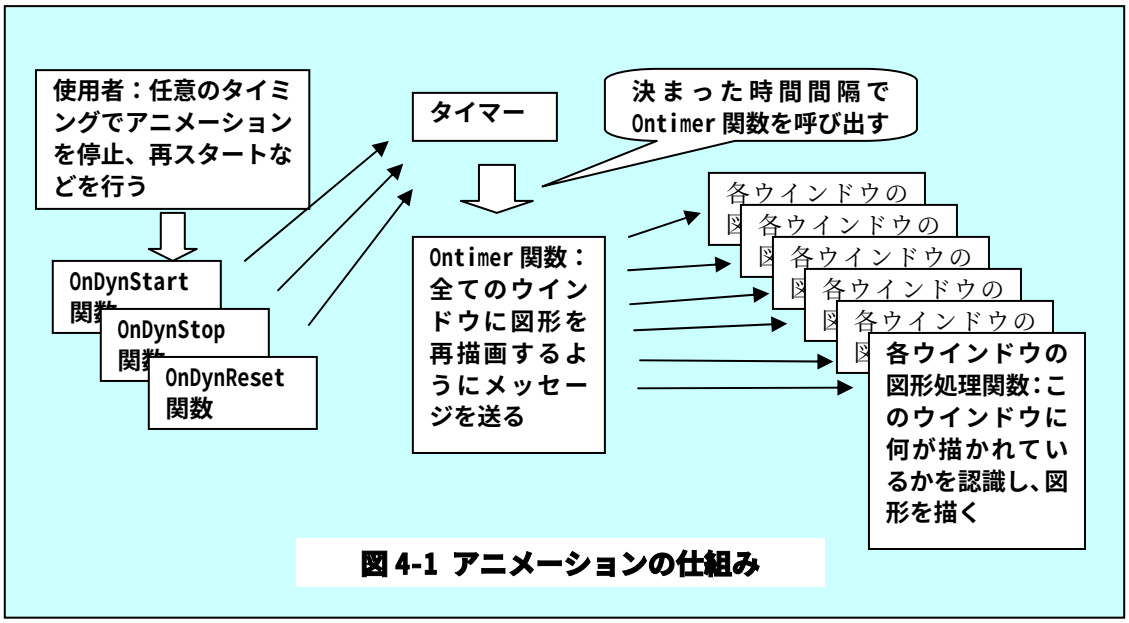

図 4-1 アニメーションの仕組み

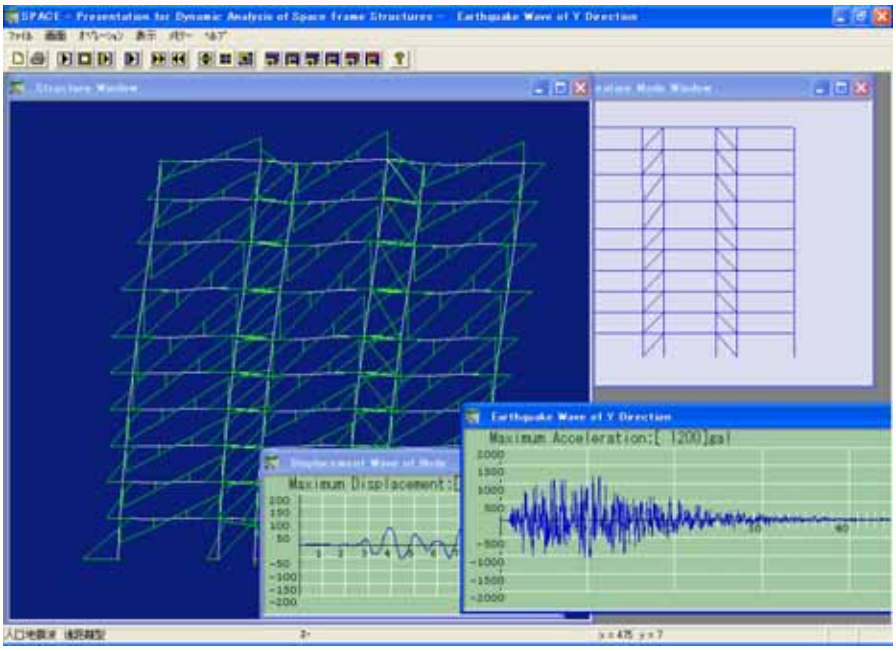

図 4-2 シンクロして各ウインドウ内の図形が動作する

図4-1を参照しながら、アニメーションの仕組みを説明しよう。先に述べた OnDynStart()関数によって起動したタイマーは、設定した秒数ごとに、起動をかけたオブジェクトに対しイベントを発信し続ける。その結果、関数 OnTimer()が呼び出され、設定した秒数間隔で実行されることになる。この関数は、アニメーションの時間管理と表示ウインドウの管理を行っており、このウインドウ管理情報を用いて、現在表示されている子ウインドウの全オブジェクトに、この時刻の図形に書き換えよというメッセージを送る。無論自分自身に対してもメッセージを送ることになる。メッセージを受けた各オブジェクトは、自分が管理している子ウインドウに何が表示されているかを確認し、その時刻に合わせて該当する図形を書き換えるわけである。これが、プレゼンターで用いられているアニメーションの仕組みである。

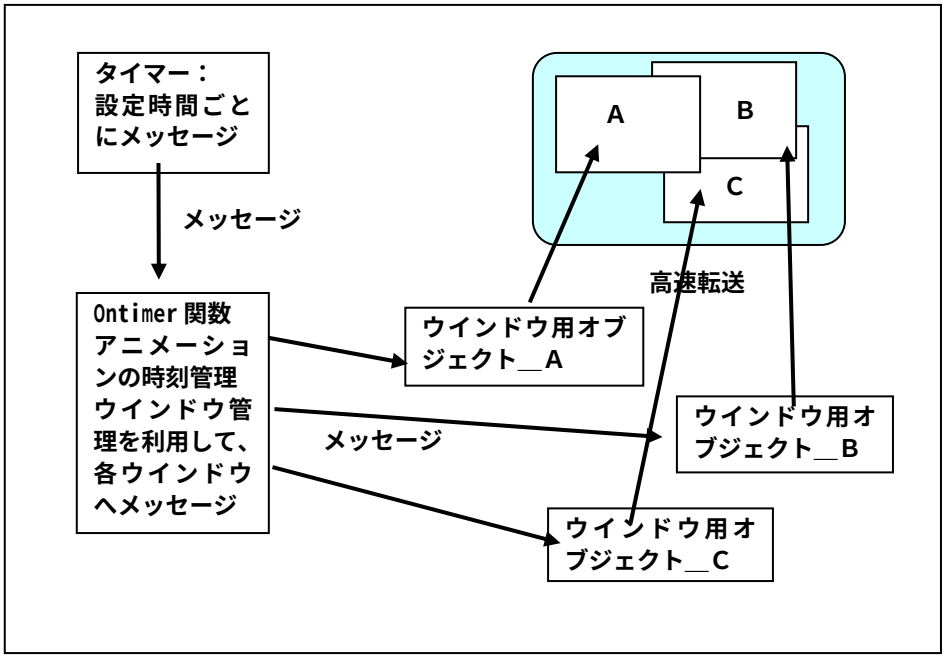

図4-3 アニメーションの仕組みと図形描画

オブジェクト指向型言語の興味深い点は、メッセージを送る関数も、受け取る関数も、一つの CSf31View クラス内のメンバー関数として記述されていることである。ただし、実態のオブジェクトは、一つだけではなく、画面に表示されている子ウインドウ分存在する。そこでは、オブジェクト間でメッセージの交信を行い、各自のオブジェクトが現在表示している図形を頻繁に更新している。

それでは、アニメーションの仕組みに関連する各関数とメッセージマップを以下に示そう。まず、関連するメッセージマップは、以下のようである。

```
BEGIN_MESSAGE_MAP(CSf3stView, CView)
  ON_WM_TIMER()
  ON_COMMAND(ID_DYN_START, OnDynStart)
  ON_COMMAND(ID_DYN_STOP, OnDynStop)
  ON_MESSAGE(IDS_MESSAGE_WIND, OnMessageWind)
   •
   •
END_MESSAGE_MAP()
```

次に、関連する関数を示す。ここに示す関数は動的プレゼンターで使用されている関数であるが、ここでは説明を容易にするため、記述を少し変更している。

```
//     ―――――――――――――――――――――――――――――――
//     ●     アニメーション開始処理
//     ―――――――――――――――――――――――――――――――
//
void CSf31View::OnDynStart()                                          !1
{
if(F_ontimer == 1) return;                                            !2
      F_ontimer = 1;
      F_hwnd_timer = this->GetSafeHwnd();                             !3
      CWnd* pwnd =FromHandle( F_hwnd_timer);
      F_operation=2;                                                  !4
      F_Time=0;
      F_time_ii = 0;
      sf31load. m_data_set_ok = TRUE;
//    ―――――――――――――――――――――――
//     ●    F_Speed = 0.04 秒:既定値スピード
//    ―――――――――――――――――――――――
      F_timer_exe = pwnd->SetTimer(1,F_Speed,NULL);                   !5
      if ( F_timer_exe == 0) MessageBox(" System cannot install timer!!");
}
//    ―――――――――――――――――――――――――――――――――
//     ●    各ウインドウの図形処理を制御
//    ―――――――――――――――――――――――――――――――――
//     ―――――――――――――――――――――――――――――――
//     ●     OnTimer
//     ―――――――――――――――――――――――――――――――
//
void CSf31View::OnTimer(UINT nIDEvent)                                !6
{
//    ―――――――――――――――――――――――
//     ●    時刻計算
//    ―――――――――――――――――――――――
   F_Time =  F_Time + F_delt_cl*FI_Speed;                             !7
   F_time_ii = F_time_ii +FI_Speed;
//    ―――――――――――――――――――――――
//     ●    アニメーション終了処理
//    ―――――――――――――――――――――――
    if(F_Time > F_all_time+ F_delt_cl*0.1) {                           !8
     OnDynStop() ;
     return;
```

```
    }
//    ―――――――――――――――――――――――
//      ●　各ウインドウのコードを取得
//    ―――――――――――――――――――――――
  if(fno_yy != 0){          //          fno_yy:現在描画されているウインドウの数     !9
       for(int i = 0; i<fno_yy; i++){                                              !10
        HWND hWnd;                                                                 !11
        int fno_yx = i+1;
        int fno_xy = getwindx((long)fno_yx);                                       !12
        hWnd =getwindh((long)fno_yx);                                              !13
        LONG lParam = fno_yx;                                                      !14
//    ―――――――――――――――――――――――
//      ●　図形処理を行なうようにメッセージを出す
//    ―――――――――――――――――――――――
        if(fno_xy > 0)::SendMessage(hWnd,IDS_MESSAGE_WIND,0,lParam);               !15
       }
   }
}
CView::OnTimer(nIDEvent);
}
//    ―――――――――――――――――――――――――――――――――
//      ●　メッセージに対応して、ウインドウ自身の図形処理
//    ―――――――――――――――――――――――――――――――――
//    ―――――――――――――――――――――――――――――――――
//      ●    OnMessageWind
//    ―――――――――――――――――――――――――――――――――
//
 LONG CSf31View::OnMessageWind(UINT wParam, LONG lParam)                           !16
 {
//    ―――――――――――――――――――――――
//      ●　ウインドウコードを取得
//    ―――――――――――――――――――――――
    LONG d_han = 1;
    int fno_yx = lParam;                                                           !17
    int ii = fno_yx-1;
    HWND hWnd = this->GetSafeHwnd();                                               !18
    if(checkhwnd(hWnd,ii) == FALSE) return (d_han);
    if(::IsIconic(hWnd)) return (d_han);                                           !19
    CWnd* pWnd = CWnd::FromHandle(hWnd);                                           !20
    CClientDC dc(pWnd);                                                            !21
    int fno_xy = getwindx((long)fno_yx);                                           !22
    int fno_xxx = getwindy((long)fno_yx);                                          !23
    int ihan =0;
    if(F_Time <= F_delt_cl) ihan=-1;                                               !24
//    ―――――――――――――――――――――――
//      ●　ウインドウコードに従って描画関数を呼ぶ
//    ―――――――――――――――――――――――
switch (fno_xy){                                                                   !25
//    ―――――――――――――――――――――――
//      ●　構造物の変形と応力
//    ―――――――――――――――――――――――
case STRUCT:                                                                       !26
    if( fno_xxx == 0) pre_disp_mem_persp(hWnd,ii);
    if( fno_xxx == 1 && F_time_ii <= m_nstep) pre3_disp_mem_persp(hWnd,ii);
```

```
    if( fno_xxx == 2) pre4_disp_mem_persp(hWnd,ii);
    if( fno_xxx == 4 && F_time_ii <= m_nstep) pre5_disp_mem_persp(hWnd,ii);
    if( fno_xxx == 5 && F_time_ii <= m_nstep) pre6_disp_mem_persp(hWnd,ii);
break;
//    ――――――――――――――――――――――
//      ●　地震波及び節点の変位の履歴図
//    ――――――――――――――――――――――
case WAVE:                                                              !27
    if(m_dat_struct == 0) sf31wave.set_wave_dat((CDC*)&dc, hWnd, F_Time,ihan);
    if(m_dat_struct == 2) sf31shear.set_wave_dat((CDC*)&dc, hWnd, F_Time,ihan);
    break;
//    ――――――――――――――――――――――
//      ●　静的荷重の時刻歴
//    ――――――――――――――――――――――
case SLOAD:                                                             !28
    sf31load.set_load_dat((CDC*)&dc, hWnd, F_Time,ihan);
    break;
//    ――――――――――――――――――――――
//      ●　部材の応力状態
//    ――――――――――――――――――――――
case STRESS:                                                            !29
    if(  F_time_ii <= m_nstep) set_member_frame((CDC*)&dc);
    break;
//    ――――――――――――――――――――――
//      ●　断面データの時刻歴
//    ――――――――――――――――――――――
case SECTION:                                                           !30
    if(  F_time_ii <  m_nstep) Section.Section_member_frame((CDC*)&dc, hWnd, F_Time,ihan);
    break;
//    ――――――――――――――――――――――
//      ●　その他
//    ――――――――――――――――――――――
default:                                                                !31
    break;
}
 d_han = 0;
return(d_han);
}
//    ――――――――――――――――――――――――――――――――――
//      ●　タイマーをストップさせる
//    ――――――――――――――――――――――――――――――――――
//
void CSf31View::OnDynStop()                                             !32
{
    if(::KillTimer(F_hwnd_timer,F_timer_exe )){                         !33
          F_ontimer = 0;
    }
}
```

アニメーションの仕組みは、上記の４つの関数によって構築されている。プログラムの内容を読むことで、各関数の役割と動作を十分に理解されたい。

表 4-1　アニメーションの仕組みを構成する関数

関数名	役割	動作
OnDynStart()	アニメーション開始処理	アニメーションの時刻を 0 にする。タイマースピードをセットし、タイマーを起動する。
OnDynStop()	アニメーション終了処理	タイマーを停止させる。
OnTimer()	各ウインドウの図形処理を制御	アニメーションの時刻管理、各ウインドウに図形描画メッセージを送る。
OnMessageWind()	メッセージに対応して、ウインドウ自身の図形処理	メッセージを受信し、各ウインドウの図形を確認した後、時刻を合わせて図形を描画する。

各関数について、コメント番号に従って説明する。

１．関数 OnDynStart()によって、アニメーションが実行される。
２．変数 F_ontimer を用いて、既にタイマーが起動しているかどうかチェックし、起動している場合は、この関数から抜ける。タイマーが未だ起動していない場合は、変数 F_ontimer に１をセットし、以下の処理で起動する。
３．タイマーを起動する子ウインドウのハンドル F_hwnd_timer と、ウインドウポインターpwnd を保存する。これは、アニメーションの動作子ウインドウが消去されないためであり、実際に動作しているそのオブジェクトが消去されるとアニメーションシステムが破綻をきたすからである。
４．アニメーション時刻などの初期設定を行う。
５．タイマーに情報をセットし、起動する。ここで、変数 F_Speed は、タイマーがオブジェクトにメッセージを送る間隔（秒）である。最初は規定値を用いるが、使用者が変更することもできる。変更されると、アニメーションの表示スピードが速まったり、遅くなったりする。
６．OnTimer()は、タイマーによって呼び出される関数であり、アニメーションを制御する。
７．変数 F_Time は、ウインドウに表示する時刻であり、変数 F_time_ii は図形を表示するステップ番号を表す。ここで、変数 F_delt_cl は１ステップの時刻である。アニメーションの動作を制御する変数として FI_Speed があり、この値は使用者が変更できる仕様となっている。
８．動作時刻が解析時間を越えているかどうかチェックを行い、超えて

いる場合は、関数 OnDynStop()をコールして、アニメーションを停止させる。

９．グローバル変数である fno_yy を用いて、現在、図形が描画されているウインドウがあるかどうかチェックを行い、ある場合は、以下の処理を行う。

10. 描画されているウインドウ数 fno_yy 分、以下の処理を行う。

11. ウインドウハンドルの生成と、ウインドウ番号の設定を行う。

12. ウインドウの管理用画面番号 fno_yx を設定する。

13. 管理用画面番号 fno_yx から、そのウインドウのウインドウコードを取得する。

14. 管理用画面番号 fno_yx から、そのウインドウのハンドル hWnd を取得する。また、メッセージ用のパラメータ lParam として、ウインドウ画面番号をセットする。

15. ウインドウコードが 0 以上、つまり、そのウインドウに図形表示されている場合は、そのウインドウオブジェクトにメッセージを送る。

16. ここからは、関数 OnMessageWind()について説明する。この関数は、メッセージ IDS_MESSAGE_WIND によってコールされる。

17. メッセージに含まれるパラメータより、画面番号 fno_yx をセットする。

18. このウインドウのハンドルを取得する。

19. このウインドウハンドルの画面が現在使用されているか、関数 checkhwnd()を用いてチェックする。使用されていない場合は、この関数から抜ける。

20. このウインドウがアイコンになっているかどうか、関数 IsIconic()を用いてチェックする。アイコンになっている場合は、直ちにこの関数から抜ける。

21. このウインドウのクライアント DC を取得する。

22. ウインドウ管理用関数 getwindx()を用いて、ウインドウコード fno_xy を取得する。

23. ウインドウ管理用関数 getwindy()を用いて、構造図出力用コード fno_xxx を取得する。

24. アニメーションが始まっていない場合は、変数 ihan に-1 をセットする。

25. ウインドウコード fno_xy を用いて、switch 文により図形処理を分類する。

26. 構造透視図の時刻歴を表示する。ここでは、構造図出力用コード

fno_xxx を用いて、表示透視図が分類されている。

27. 地震波形あるいは節点の変位波形がグラフ表示される。
28. 動的解析における長期荷重の履歴図が表示される。
29. 部材の応力状態が表示される。
30. 部材断面の応力状態や応力とひずみの関係に関する時刻歴がグラフ表示される。
31. その他の表示であるが、ここでは何もしない。
32. アニメーションを停止させる関数である。
33. タイマーを停止させるために、関数::KillTimer()を実行させる。

以上で、アニメーションの仕組みを構成する基本的な関数の説明は終了である。アニメーションの仕組みが理解できただろうか。アニメーションから効率よく情報を得るためには、他にも各種の操作オプションが必要となる。そのオプションを処理する関数を以下に示すが、その内容は簡単であるので説明を省くことにする。ここでは、内容の理解は読者の課題とする。

```
//     ―――――――――――――――――――――――――――――――――
//     ●     アニメーションのリスタート
//     ―――――――――――――――――――――――――――――――――
void CSf31View::OnDynReset()
{
    if(F_ontimer == 1) return;
      F_ontimer = 1;
      F_hwnd_timer = this->GetSafeHwnd();
      CWnd* pwnd =FromHandle( F_hwnd_timer);
      F_operation=2;
//          F_Speed = 0.04;
      F_timer_exe = pwnd->SetTimer (1,F_Speed,NULL);
      if ( F_timer_exe == 0) MessageBox(" System cannot install timer!!");
}
//     ―――――――――――――――――――――――――――――――――
//     ●     アニメーションの時間 10 ステップ前進処理
//     ―――――――――――――――――――――――――――――――――
void CSf31View::OnDynStepX()
{
    if(F_ontimer == 1) return;
       F_operation=2;
       int deltx = 10;
       if(F_Time + F_delt_cl*deltx> F_all_time+ F_delt_cl*0.01) {
        return;
       }else{
        F_Time =  F_Time + F_delt_cl*deltx;
        F_time_ii = F_time_ii +deltx;
       }
       if(fno_yy != 0){
```

この関数ではアニメーションの再スタートを行う。再び、タイマーに起動をかける。

10 ステップ前進が最終画面に達したかどうかチェックし、最終ステップの場合は、この関数を抜ける。それ以外は、10 ステップ前進させる。
ウインドウ管理システムを用いて、他のウインドウにメッセージを送り、10 ステップ前進させ描画させる。

```
          for(int i = 0; i<fno_yy; i++){
           HWND hWnd;
           int fno_yx = i+1;
           int fno_xy = getwindx((long)fno_yx);
           hWnd =getwindh((long)fno_yx);
           LONG lParam = fno_yx;
           if(fno_xy > 0)::SendMessage(hWnd,IDS_MESSAGE_WIND,0,lParam);
         }
        }
}
//     ―――――――――――――――――――――――――――――――――
//     ●    アニメーションの時間１ステップ前進処理
//     ―――――――――――――――――――――――――――――――――
void CSf31View::OnDynStep()
{
      if(F_ontimer == 1) return;
       F_operation=2;
       if(F_Time + F_delt_cl> F_all_time+ F_delt_cl*0.01) {
        return;
       }else{
        F_Time =  F_Time + F_delt_cl*FI_Speed;
        F_time_ii = F_time_ii + FI_Speed;
       }
       if(fno_yy != 0){
          for(int i = 0; i<fno_yy; i++){
            HWND hWnd;
            int fno_yx = i+1;
            int fno_xy = getwindx((long)fno_yx);
            hWnd =getwindh((long)fno_yx);
            LONG lParam = fno_yx;
            if(fno_xy > 0)::SendMessage(hWnd,IDS_MESSAGE_WIND,0,lParam);
            }
           }
}
//     ―――――――――――――――――――――――――――――――――
//     ●    アニメーションの時間後退処理
//     ―――――――――――――――――――――――――――――――――
//
void CSf31View::OnDynBack()
{
         F_operation=2;
        if(F_ontimer == 1) return;
        F_hwnd_timer = this->GetSafeHwnd();
        CWnd* pwnd =FromHandle( F_hwnd_timer);
        F_operation=2;
        if(F_time_ii == 1){
         return;
        }
        F_Time =  F_Time - 10*F_delt_cl;
        F_time_ii = F_time_ii - 10;
        if(F_time_ii  < 1 ) {
           F_Time =  F_delt_cl;
           F_time_ii =1;
        }
```

1 ステップ前進が最終画面に達したかどうかチェックし、最終ステップの場合は、この関数を抜ける。それ以外は、1 ステップ前進させる。
ウインドウ管理システムを用いて、他のウインドウにメッセージを送り、1 ステップ前進させ描画させる。

10 ステップ後退が最終画面に達したかどうかチェックし、最初のステップの場合は、この関数を抜ける。それ以外は、10 ステップ後退させる。
ウインドウ管理システムを用いて、他のウインドウにメッセージを送り、10 ステップ後退させ描画させる。

```
            if(fno_yy != 0){
               for(int i = 0; i<fno_yy; i++){
                  HWND hWnd;
                  int fno_yx = i+1;
                  int fno_xy = getwindx((long)fno_yx);
                  hWnd =getwindh((long)fno_yx);
                  LONG lParam = fno_yx;
                  if(fno_xy > 0)::SendMessage(hWnd,IDS_MESSAGE_WIND,0,lParam);
               }
            }
   }
```

4.3 アニメーション技法

前節では、アニメーションの仕組みについて解説した。本節では、アニメーション技術、特に、図形描画について述べる。ここでのテクニックは、第２章で学んだラバーバンドやグラフ描画の方法とほぼ同じである。

画面に図形を描画する場合、通常は VRAM に直接描く。VRAM に複雑な図形などを直接描くと、描いている様子が見られる。アニメーションのように少しずつ異なる絵を、描いては消し、描いては消しという処理を何度も繰り返すと、画面がチラつくことになる。これでは、スムーズなアニメーションが得られないことになる。

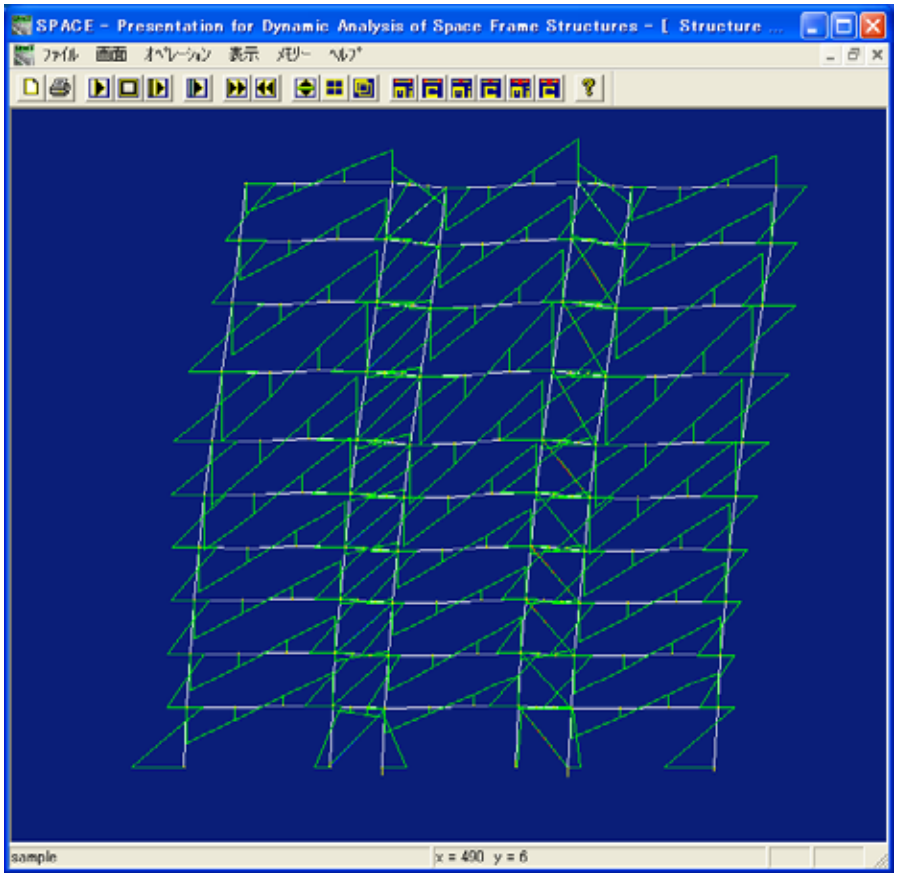

図 4-4 アニメーション技術による図形処理

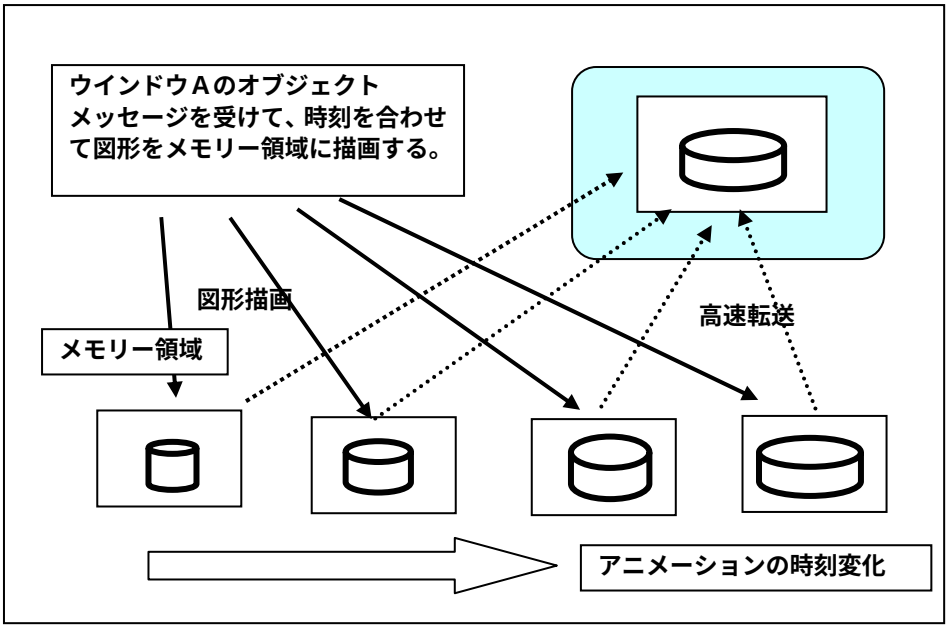

図 4-5 図形描画の仕組み

スムーズなアニメーションを得るために、まず、VRAM と同じ環境を通常の RAM に作り、そこに次ステップの図形を描き、終了後、その RAM から高速に VRAM にデータを転送する。これを次々と繰り返すわけである。具体的に、その仕組みを該当するコードで見てみよう。これに関連する関数を以下に示す。

```
//    ―――――――――――――――――――――――――――――――
//      ●　画面描画設定
//    ―――――――――――――――――――――――――――――――
void CSf31View::pre3_disp_mem_persp(HWND hWnd,int ii)
{
  CWnd* pWnd = CWnd::FromHandle(hWnd);
  CClientDC dc(pWnd);
  disp2_mem_persp((CDC*)&dc);
  return;
}
//    ―――――――――――――――――――――――――――――――
//      ●　画面用メモリー確保と図形のコピー
//    ―――――――――――――――――――――――――――――――
void CSf31View::disp2_mem_persp(CDC* pDC)
{
//    ―――――――――――――――――――――
//      ●　画面用のメモリー領域を確保
//    ―――――――――――――――――――――
    CDC* pMemDC = new CDC;
    HWND hWnd=WindowFromDC(pDC->m_hDC);
         if(IsWindow(hWnd) == FALSE)
         {
          delete pMemDC;
          return;
         }
```

ここでは、子ウインドウのポインターpWnd から、デバイスコンテキストのポインターを取得する。そのポインターを引数として、図形処理関数をコールする。

子ウインドウの VRAM と同じ寸法のメモリーを確保する。

```
    CRect rcFrame;
    ::GetClientRect(hWnd ,&rcFrame);
    pMemDC->CreateCompatibleDC(pDC);
    CBitmap *pMemBitmap  = new CBitmap;
    pMemBitmap -> CreateCompatibleBitmap(pDC,rcFrame.right,rcFrame.bottom);
    CBitmap *poldBitmap =(CBitmap*)pMemDC -> SelectObject(pMemBitmap);
//   ―――――――――――――――――――――
//    ●　ウインドウの大きさをセット
//   ―――――――――――――――――――――
    int iww,iwh;
    iww = rcFrame.right;
    iwh = rcFrame.bottom;
    F_pos[0][0]=0;
    F_pos[1][0]=0;
    F_pos[0][1]=iww;
    F_pos[1][1]=iwh;
//   ―――――――――――――――――――――
//    ●　ウインドウの背景を塗る
//   ―――――――――――――――――――――
    disp_frame_w(pMemDC);
//   ―――――――――――――――――――――
//    ●　構造データの節点位置を透視変換する
//   ―――――――――――――――――――――
    if(m_pst_dis == 0){
        TOSHIP(&F_time_ii,F_disp,&F_all_step,posit,&node,&m_rr[0][0],
               F_bz,&F_pos[0][0],&F_amx,&F_amy,
               iconsb,&mnbsb,imeme,&F_scalep[2],&m_base[0],&m_pst_disp);
    }
    if(m_pst_dis == 1){
        TOSHIP(&F_time_ii,F_vel,&F_all_step,posit,&node,&m_rr[0][0],
               F_bz,&F_pos[0][0],&F_amx,&F_amy,
               iconsb,&mnbsb,imeme,&F_scalep[1],&m_base[0],&m_pst_disp);
    }
    if(m_pst_dis == 2){
        TOSHIP(&F_time_ii,F_acc,&F_all_step,posit,&node,&m_rr[0][0],
               F_bz,&F_pos[0][0],&F_amx,&F_amy,
               iconsb,&mnbsb,imeme,&F_scalep[0],&m_base[0],&m_pst_disp);
    }
    if(m_pst_dis == 3){
        TOSHIP(&F_time_ii,F_accab,&F_all_step,posit,&node,&m_rr[0][0],
               F_bz,&F_pos[0][0],&F_amx,&F_amy,
               iconsb,&mnbsb,imeme,&F_scalep[0],&m_base[0],&m_pst_disp);
    }
//   ―――――――――――――――――――――
//    ●　オプションに従って図形描画する
//   ―――――――――――――――――――――
    if(m_pst_color == 0) disp_graph_1(pMemDC);
    if(m_pst_color != 0) disp_graph_5(pMemDC);
    if(m_pst_hinge == 1) disp_graph_4(pMemDC);
    if(m_pst_graph != 0 || m_pst_arrow != 0) disp_graph_7(pMemDC);
//   ―――――――――――――――――――――
//    ●　メモリー領域から図形を VRAM にコピーする
//   ―――――――――――――――――――――
    pDC->BitBlt(0,0,iww,iwh,pMemDC,0,0,SRCCOPY);
```

ウインドウの大きさを、配列にセットする。これは、背景を塗る際、必要となる。

メモリー領域を背景色で塗りつぶす。

構造データの節点位置にその時刻の変位などを足して、透視変換を行う。

節点に足し込む量は、変位である。

節点に足し込む量は、速度である。

節点に足し込む量は、相対加速度である。

節点に足し込む量は、絶対加速度である。

ユーザーの指定によって、描画の種類を変更する。

描いた図形を VRAM に高速転送する。

```
//    ――――――――――――――――――――――
//      ●　画面用のメモリー領域を解放
//    ――――――――――――――――――――――
      delete pMemDC->SelectObject(poldBitmap);
      delete pMemDC;
}
```

前節で示したように、構造透視図のアニメーションが実施されると、最初に関数 pre3_disp_mem_persp()が呼び出される。さらに、ここで、子ウインドウのポインターを取得した後、次の関数 disp2_mem_persp()が呼ばれる。ここで、上記のアニメーションの技術が使用される。

この関数 disp2_mem_persp()では、最初、当該のウインドウの大きさと同じ状況を RAM に作成する。その手続きが、プログラム内のコメント「画面用のメモリー領域を確保」以降の 13 行で行われている。この領域へのポインターは pMemDC である。次に、ウインドウの大きさをセットするという項で、ウインドウの大きさを取得し、配列 F_pos[]にセットする。

関数 disp_frame_w(pMemDC)で、背景を塗りつぶし、この領域を初期設定する。次に、この領域に図形オプションにしたがって図形を描画する。また、この領域に解析時刻を文字表示する。これで、全ての図形描画が終了する。図形オプションは、m_pst_color（部材応力のカラー表示)、m_pst_hinge（塑性ヒンジ表示オプション)、m_pst_graph（応力の円、図形表示オプション)、m_pst_arrow（部材応力のカラー表示）である。これらの図形処理用オプションは、ダイアログで変更することができる。

図形をメモリー領域に描き終わった後､関数 pDC->BitBlt()によって、高速に VRAM にデータを転送する。これで、画面上の子ウインドウの図形が書き換えられることになる。

最後に、この関数の中で生成したビットマップオブジェクトとメモリー領域を消去する。これで、チラつかないスムーズなアニメーション画像が得られることになる。

SPACE の動的プレゼンターで使用している図形処理用コードは次節で説明する。以下には、構造用データを透視変換する FORTRAN コードを示し、基本的な透視変換処理について解説する。まずは、関連するサブルーチンを以下に示す。

アニメーションの図形描画処理の手続き

1．メモリー領域確保
2．メモリー領域を会い軽食で塗りつぶす
3．節点位置にその時刻の節点変位に倍率を掛けて足し込む。
4．上記で得られた３次元位置情報を透視変換し､全節点について２次元データを得る。
5．全部材について､部材両端の節点番号を用いて両端の２次元座標を求め､その間をライン描画する。
6．メモリー領域のデータを BitBlt 関数を用いて、VRAM に高速転送する。
7．メモリー領域を解放する。

```
C     ――――――――――――――――――――――――――――――――
C     ●　SUBROUTINE /toship
C       ――――――――――――――――――――――――――――――――
C     ●　　構造節点データの透視変換
```

```
C     ―――――――――――――――――――――――――――――――――――
      subroutine toship(itime,fdisp,iallt,bzz,iset,rr,gxy,
     *                  idpos,amx,amy,ij,memb,jic,amp,px,ipdisp)
      dimension bzz(3,iset),igpn(iset),ij(2,memb),jic(memb)
      dimension idpos(2,2),gxy(2,iset),rr(4,4)
      dimension fdisp(3,iset),px(3)
      BX=(idpos(1,1)+idpos(2,1))*0.5+amx                              ! 1
      BY=(idpos(1,2)+idpos(2,2))*0.5+amy
      call dispgx(bzz,iset,rr,gxy,Bx,By,fdisp,amp,px,ipdisp)          ! 2
      return
      end
C     ―――――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE /dispgx
C       ―――――――――――――――――――――――――――――――――――
C     ●   節点変位を含めた透視変換を行う
C     ―――――――――――――――――――――――――――――――――――
      subroutine dispgx(bzz,iset,rr,
     *                  gxy,Bx,By,fdisp,amp,px,ipdisp)
      REAL*4 bzz(3,iset),px(3)
      dimension gxy(2,iset),rr(4,4),fdisp(3,iset),gg(3)
      goto(991,992,993,994,995),ipdisp+1                              ! 3
c-------------------------------------------  ３方向の変位を含めた変換
 991  continue
      do 330 i1=1,iset
      do 195 ki=1,3
      gg(ki)=bzz(ki,i1)-px(ki)+ amp*fdisp(ki,i1)
 195  continue
      sum=rr(4,4)
      do 332 j=1,3
      sum=sum+gg(j)*rr(j,4)
  332 continue
      if (abs(sum) .ge. 0.00001) then
      ch=1./sum
      else
      ch=100000.
       endif
       sum=0.
       summ=0.
       do 334 j=1,3                                                   ! 7
       sum=sum+gg(j)*rr(j,1)
       summ=summ+gg(j)*rr(j,3)
  334  continue
       gxy(1,i1)=sum*ch+BX                                            ! 8
       gxy(2,i1)=-summ*ch+BY
  330  continue
       return
c-------------------------------------------  ｘ方向の変位を含めた変換
 992  continue                                                        ! 9
      do 1330 i1=1,iset
      gg(1)=bzz(1,i1)-px(1)+ amp*fdisp(1,i1)
      gg(2)=bzz(2,i1)-px(2)
      gg(3)=bzz(3,i1)-px(3)
      sum=rr(4,4)
      do 1332 j=1,3
```

3 方向の変位を足し込んだ節点位置を透視変換する。透視変換後の 2 次元情報は、gxy() に保存される。変数 amp は、変位の表示倍率を表す。

ｘ方向の変位を足し込んだ節点位置を透視変換する。透視変換後の 2 次元情報は、gxy()に保存される。従って、ここでは、ｘ方向の変位のみ透視図として表示される。

```
      sum=sum+gg(j)*rr(j,4)
 1332 continue
      if (abs(sum) .ge. 0.00001) then
       ch=1./sum
      else
       ch=100000.
      endif
       sum=0.
       summ=0.
       do 1334 j=1,3
       sum=sum+gg(j)*rr(j,1)
       summ=summ+gg(j)*rr(j,3)
 1334  continue
      gxy(1,i1)=sum*ch+BX
      gxy(2,i1)=-summ*ch+BY
 1330 continue
      return
c-------------------------------------------  y方向の変位を含めた変換
 993  continue                                                    ! 10
      do 2330 i1=1,iset
      gg(1)=bzz(1,i1)-px(1)
      gg(2)=bzz(2,i1)-px(2)+ amp*fdisp(2,i1)
      gg(3)=bzz(3,i1)-px(3)
      sum=rr(4,4)
      do 2332 j=1,3
      sum=sum+gg(j)*rr(j,4)
 2332 continue
      if (abs(sum) .ge. 0.00001) then
      ch=1./sum
     else
      ch=100000.
     endif
      sum=0.
      summ=0.
      do 2334 j=1,3
      sum=sum+gg(j)*rr(j,1)
      summ=summ+gg(j)*rr(j,3)
 2334 continue
      gxy(1,i1)=sum*ch+BX
      gxy(2,i1)=-summ*ch+BY
 2330 continue
      return
c-------------------------------------------  z方向の変位を含めた変換
 994 continue                                                    ! 11
     do 3330 i1=1,iset
     gg(1)=bzz(1,i1)-px(1)
     gg(2)=bzz(2,i1)-px(2)
     gg(3)=bzz(3,i1)-px(3)+ amp*fdisp(3,i1)
      sum=rr(4,4)
      do 3332 j=1,3
      sum=sum+gg(j)*rr(j,4)
 3332 continue
      if (abs(sum) .ge. 0.00001) then
       ch=1./sum
```

y方向の変位を足し込んだ節点位置を透視変換する。透視変換後の 2 次元情報は、gxy()に保存される。従って、ここでは、y方向の変位のみ透視図として表示される。

z方向の変位を足し込んだ節点位置を透視変換する。透視変換後の 2 次元情報は、gxy()に保存される。従って、ここでは、z方向の変位のみ透視図として表示される。

```
      else
       ch=100000.
      endif
      sum=0.
      summ=0.
       do 3334 j=1,3
       sum=sum+gg(j)*rr(j,1)
       summ=summ+gg(j)*rr(j,3)
 3334  continue
       gxy(1,i1)=sum*ch+BX
       gxy(2,i1)=-summ*ch+BY
 3330  continue
      return
c---------------------------------------------  変位を含めない変換
 995  continue                                                    ! 12
      do 4330 i1=1,iset
      gg(1)=bzz(1,i1)-px(1)
      gg(2)=bzz(2,i1)-px(2)
      gg(3)=bzz(3,i1)-px(3)
       sum=rr(4,4)
      do 4332 j=1,3
      sum=sum+gg(j)*rr(j,4)
 4332  continue
       if (abs(sum) .ge. 0.00001) then
       ch=1./sum
       else
       ch=100000.
       endif
       sum=0.
       summ=0.
       do 4334 j=1,3
       sum=sum+gg(j)*rr(j,1)
       summ=summ+gg(j)*rr(j,3)
 4334  continue
       gxy(1,i1)=sum*ch+BX
       gxy(2,i1)=-summ*ch+BY
 4330  continue
       return
       end
```

変位を足しこまない節点位置を透視変換する。透視変換後の 2 次元情報は、gxy()に保存される。従って、ここでは、動きのない構造物の透視図のみ表示される。

上記プログラムの説明を、コード右側に付した番号に従って行う。

1．構造物の原点を計算する。ここで、配列 idpos[]は描画ウインドウの左上と右下の座標であり、amx と amy は、中心位置を移動させる変数である。

2．透視変換を行うサブルーチンをコールする。配列と引数は以下のようである。

```
Bzz(3,iset)   :構造物の３次元座標
px(3)         :３方向の原点移動量
gxy(2,iset)   :透視変換後の２次元座標
```

```
rr(4,4)       :透視変換行列
fdisp(3,iset):節点変位
gg(3)         :ワーク領域
ipdisp        :変位出力コード
               0:全方向　1:x 方向　2:y 方向
               3:z 方向　4:変位なし
amp           :節点変位の倍率
iset          :節点数
```

３．透視変換種別コード ipdisp にしたがって、変換方法を分類する。以下の説明は、最初の変換について行う。

４．ここでは、実際の構造物の座標に原点移動 px とその節点の変位に係数を掛けた値を加える。

５．変位を含めた節点の座標に、透視図変換を行うための分母を計算する。

６．その値が非常に小さい場合は、その値の逆数を 100000.とする。

７．透視変換した２次元座標をセットする。

８．透視変換された 2 次元座標値に、原点移動(BX,BY)を行う。

９．ｘ方向のみ変位を表示する描画処理を行う。透視変換などの処理は、上記の方法と全く同一である。以下の処理も同様である。

10．ｙ方向のみ変位を表示する描画処理を行う。

11．ｚ方向のみ変位を表示する描画処理を行う。

12．変位を表示しない描画処理を行う。

Y=0 面に対する３次元の節点透視変換を行う変換行列は、次式で与えられる４ｘ４の正方行列である。この透視変換については第 2.4 節を参考に、また詳細は文献を参照されたい。

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & g \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & 0 & r_{13} & gr_{12} \\ r_{21} & 0 & r_{23} & gr_{22} \\ r_{31} & 0 & r_{33} & gr_{32} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ここで、Y=0 面における透視図の座標 $\{\bar{x},0,\bar{z}\}$ は、次式で与えられる。

$$\{\bar{x},0,\bar{z},1\} = (x,y,z,1) \begin{bmatrix} r_{11} & 0 & r_{13} & gr_{12} \\ r_{21} & 0 & r_{23} & gr_{22} \\ r_{31} & 0 & r_{33} & gr_{32} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ここで、第４項が成立するためには、次式が成立しなければならず。従って、次の係数 sh を回転行列と節点座標の積に掛ける必要がある。

$$1 = 1 + gr_{21}x + gr_{22}y + gr_{23}z; \quad sh = \frac{1}{1 + gr_{21}x + gr_{22}y + gr_{23}z}$$

4.4 透視図の描き方

本節では、具体的に透視図を描画するプログラムコードを解説する。このプログラムでは、前節で見てきた描画用パラメータを利用して、描画方法が選択されている。動的プレゼンター内には、多数の描画用プログラムコードが存在するが、ここでは、以下に示す代表的な３つの関数について説明する。

```
disp_frame_w()    // 背景色の描画
disp_graph_5()    // 軸力色ありの図形処理
disp_graph_4()    // 塑性ヒンジの描画
```

上記の関数を具体的に以下に示す。

```
//    ―――――――――――――――――――――――――――――――――
//      ●　ウインドウ内の背景色の設定
//    ―――――――――――――――――――――――――――――――――
void CSf31View::disp_frame_w(CDC* pDC)
{
    CRgn theRgn;
    theRgn.CreateRectRgn (F_pos[0][0],F_pos[1][0],
    F_pos[0][1],F_pos[1][1]);
    int i1,i2,i3;
  switch (m_dat_struct){
   case 0:
   i1=20;
   i2=20;
   i3=20;
  break;
   case 1:
   i1=20;
   i2=20;
   i3=22;
 if(m_pst_color != 0){
 if( m_pst_dis == 0){
   i1=10;
   i2=30;
   i3=120;
           }
  •
  •
 case 4:
   i1=30;
   i2=140;
   i3=190;
    break;
 default:
   i1=200;
   i2=200;
   i3=200;
   }
    CBrush MyBrush( RGB(i1,i2,i3));
    CBrush* pOldBrush = pDC->SelectObject(&MyBrush);
```

この関数では、オプションに従って、ウインドウの背景色を決定し、その色でメモリー領域を塗りつぶす。

```
    pDC->FillRgn (&theRgn,&MyBrush);
    pDC->SelectObject(pOldBrush);
    MyBrush.DeleteObject();
    pOldBrush->DeleteObject();
}
//    ――――――――――――――――――――――――――――――――
//      ●　　軸力色ありの場合の図形出力
//    ――――――――――――――――――――――――――――――――
void CSf31View::disp_graph_5(CDC* pDC)
{
    CPen NewPen (PS_SOLID,1,RGB(0,0,255));
    CPen* pOriginalPen = pDC->SelectObject ( &NewPen);
//    ――――――――――――――――――――――――
//      ●　質量の描画
//    ――――――――――――――――――――――――
    if(m_pst_mass != 0){
     int ix_f,iy_f;
     pDC->SelectObject ( &NewPenx_a[107]);
     for(int i = 0; i < node; i++){
     int iik=node;
     if(F_mass[i+iik] != 0){
     ix_f = F_bz[i*2];
     iy_f = F_bz[i*2+1];
     pDC->Arc(ix_f-5,iy_f-5,ix_f+5,iy_f+5,ix_f-5,iy_f,ix_f-5,iy_f);
  }}
   }
//    ――――――――――――――――――――――――
//      ●　カラーオプション：１でオプションなしの描画
//    ――――――――――――――――――――――――
       pDC->SelectObject ( &NewPen);
       if(m_pst_color == 1 && m_pst_options == 0 ){
       int i1,i2,i3,icol,mem,mmx,itype,ichi,mem_line;
       float anp;
       for( int i = 0; i < mnbsb; i++)
       {
        mem = imeme[i]-1;
        itype = mtype[mem];
        ichi=1;
        if(itype == 2 || itype == 4 ) ichi = 2;
        mmx = 5*mem+ichi-1;
        anp = F_snp[mmx];
        STCLSP(&icol,&i,&F_time_ii,F_n_spring,&ichi,&mnbsb,&m_nstep,&anp);
//    ――――――――――――――――――――――――
//      ●　軸力の大きさによってペンの色を変化させる
//    ――――――――――――――――――――――――
       pDC->SelectObject ( &NewPenx_a[icol]);
       i3 = i*2;
       i1 = (iconsb[i3]-1)*2;
       i2 = (iconsb[i3+1]-1)*2;
       pDC->MoveTo ( (int)F_bz[i1], (int)F_bz[i1+1] );
       pDC->LineTo ( (int)F_bz[i2], (int)F_bz[i2+1] );
   }
}
//    ――――――――――――――――――――――――
```

カラーオプション：１で、全部材表示で部材が描かれる。

この関数は、存在軸力と軸方向耐力との比を求める。

部材両端の節点番号 i1 と i2 を取得する。求めた節点の座標を用いて、ラインを描く。ここで、pDC->MoveTo は、ペンを引数の位置に移動する関数、また pDC->LineTo は引数の位置まで線を描く関数である。

```
//     ●  カラーオプション：２でオプションなしの描画
//     ――――――――――――――――――――――――
       if(m_pst_color == 2 && m_pst_options == 0 ){
       int i1,i2,i3,icol,mem,ichi;
       float xx1,yy1,xx2,yy2,x1,x2,x3;
       int mem1,itype,mmx,jj,iix,memm;
       iix =1;
       jj=3;
       for( int i = 0; i < mnbsb; i++)
       {
          memm=i+1;
          mem1 = imeme[i];
          itype = mtype[mem1-1];
          mmx = 5*(mem1-1);
           i3 = i*2;
           i1 = (iconsb[i3]-1)*2;
           i2 = (iconsb[i3+1]-1)*2;
           xx1 = F_bz[i1];
           yy1 = F_bz[i1+1];
           xx2 = F_bz[i2];
           yy2 = F_bz[i2+1] ;
          if(itype == 1 || itype == 2 ){
           SETSP1(&jj,&itype,&x1,&x2,&iix,
                 &F_time_ii, &memm,&mnbsb,&m_nstep,
                 F_n_spring,F_my_spring,F_mz_spring,F_fay,
                 &F_snp[mmx],&F_smyp[mmx],&F_smzp[mmx]);
           pDC->SelectObject ( &NewPen);
         if(x1 <= m_pst_ef2 && x2 <= m_pst_ef2 ) pDC->SelectObject ( &NewPenx_a[101]);
         if(x1 >= m_pst_ef1 || x2 >= m_pst_ef1 ) pDC->SelectObject ( &NewPenx_a[106]);
          }else{
           SETSP2(&jj,&itype,&x1,&x2,&x3,&iix,
                 &F_time_ii, &memm,&mnbsb,&m_nstep,
                 F_n_spring,F_my_spring,F_mz_spring,F_fay,
                 &F_snp[mmx],&F_smyp[mmx],&F_smzp[mmx]);
         pDC->SelectObject ( &NewPen);
         if(x1 <= m_pst_ef2 && x2 <= m_pst_ef2 && x3 <= m_pst_ef2) pDC->SelectObject ( &NewPenx_a[101]);
         if(x1 >= m_pst_ef1 || x2 >= m_pst_ef1 || x3 >= m_pst_ef1) pDC->SelectObject ( &NewPenx_a[106]);
          }
          pDC->MoveTo ( (int)F_bz[i1], (int)F_bz[i1+1] );
          pDC->LineTo ( (int)F_bz[i2], (int)F_bz[i2+1] );
    }
}
//     ――――――――――――――――――――――――
//     ●  カラーオプション：１でオプションありの描画
//     ――――――――――――――――――――――――
  if(m_pst_color == 1 && m_pst_options == 1 ){
    int i1,i2,i3,icol,mem,mmx,itype,ichi,mem_line;
    float anp;
    for( int i = 0; i < mnbsb; i++)
    {
      mem = imeme[i]-1;
      mem_line = imeme_line[i];
//     ――――――――――――――――――――――――
//     ●  部材出力オプションあり
```

```
//    ―――――――――――――――――――――――
      if( m_pst_line[mem_line-1] == 1 ){
      itype = mtype[mem];
      ichi=1;
      if(itype == 2 || itype == 4 ) ichi = 2;
      mmx = 5*mem+ichi-1;
      anp = F_snp[mmx];
      STCLSP(&icol,&i,&F_time_ii,F_n_spring,&ichi,&mnbsb,&m_nstep,&anp);
      pDC->SelectObject ( &NewPenx_a[icol]);
      i3 = i*2;
      i1 = (iconsb[i3]-1)*2;
      i2 = (iconsb[i3+1]-1)*2;
      pDC->MoveTo ( (int)F_bz[i1], (int)F_bz[i1+1] );
      pDC->LineTo ( (int)F_bz[i2], (int)F_bz[i2+1] );
  }}
   }
//    ―――――――――――――――――――――――
//      ●　カラーオプション：２でオプションありの描画
//    ―――――――――――――――――――――――
   if(m_pst_color == 2 && m_pst_options == 1 ){
      int i1,i2,i3,icol,mem,ichi;
      float xx1,yy1,xx2,yy2,x1,x2,x3;
      int mem1,itype,mmx,jj,iix,memm,mem2;
      iix =1;
      jj=3;
      for( int i = 0; i < mnbsb; i++)
      {
      memm=i+1;
      mem1 = imeme[i];
      mem2 = imeme_line[i];
//    ―――――――――――――――――――――――
//      ●　部材出力オプションあり
//    ―――――――――――――――――――――――
      if( m_pst_line[mem2-1] == 1 ){
           itype = mtype[mem1-1];
           mmx = 5*(mem1-1);
          i3 = i*2;
          i1 = (iconsb[i3]-1)*2;
          i2 = (iconsb[i3+1]-1)*2;
          xx1 = F_bz[i1];
          yy1 = F_bz[i1+1];
          xx2 = F_bz[i2];
          yy2 = F_bz[i2+1] ;
          if(itype == 1 || itype == 2 ){
           SETSP1(&jj,&itype,&x1,&x2,&iix,
                  &F_time_ii, &memm,&mnbsb,&m_nstep,
                  F_n_spring,F_my_spring,F_mz_spring,F_fay,
                  &F_snp[mmx],&F_smyp[mmx],&F_smzp[mmx]);
           pDC->SelectObject ( &NewPen);
          if(x1 <= m_pst_ef2 && x2 <= m_pst_ef2 ) pDC->SelectObject ( &NewPenx_a[101]);
          if(x1 >= m_pst_ef1 || x2 >= m_pst_ef1 ) pDC->SelectObject ( &NewPenx_a[106]);
         }else{
           SETSP2(&jj,&itype,&x1,&x2,&x3,&iix,
                  &F_time_ii, &memm,&mnbsb,&m_nstep,
```

```
                F_n_spring,F_my_spring,F_mz_spring,F_fay,
                &F_snp[mmx],&F_smyp[mmx],&F_smzp[mmx]);
          pDC->SelectObject ( &NewPen);
          if(x1 <= m_pst_ef2 && x2 <= m_pst_ef2 && x3 <= m_pst_ef2) pDC->SelectObject ( &NewPenx_a[101]);
          if(x1 >= m_pst_ef1 || x2 >= m_pst_ef1 || x3 >= m_pst_ef1) pDC->SelectObject ( &NewPenx_a[106]);
          }
          pDC->MoveTo ( (int)F_bz[i1], (int)F_bz[i1+1] );
          pDC->LineTo ( (int)F_bz[i2], (int)F_bz[i2+1] );
 }}
 }
   pDC->SelectObject ( pOriginalPen);
}
//    ――――――――――――――――――――――――――――――――
//     ●　塑性ヒンジの描画
//    ――――――――――――――――――――――――――――――――
void CSf31View::disp_graph_4(CDC* pDC)
{
    CPen NewPen (PS_SOLID,2,RGB(255,0,0));
    CPen* pOriginalPen = pDC->SelectObject ( &NewPen);
    int i1,i2,i3,ns,itype;
    float x1,y1,x2,y2;
//    ――――――――――――――――――――――
//     ●　部材出力オプションなしの場合
//    ――――――――――――――――――――――
   if(m_pst_options == 0){
      for( int i = 0; i < mnbsb; i++)
      {
         i3 = i*2;
         i1 = (iconsb[i3]-1)*2;
         i2 = (iconsb[i3+1]-1)*2;
         itype = mtype[imeme[i]-1];
         ns=F_mtype[itype-1];
         for(int j=0; j<ns; j++){
          int nnx=5*i+j;
//    ――――――――――――――――――――――
//     ●　塑性の始まり
//    ――――――――――――――――――――――
         if(F_stat_spring[nnx] == 1){
          SETSPG(&j,&itype,
             &F_bz[i1],&F_bz[i1+1],&F_bz[i2],&F_bz[i2+1],&x1,&y1,&x2,&y2);
          pDC->SelectObject ( &NewPenx_a[103]);
         pDC->MoveTo ( (int)x1, (int)y1 );
         pDC->LineTo ( (int)x2, (int)y2 );
          }
//    ――――――――――――――――――――――
//     ●　塑性ヒンジ出現（断面の 80%塑性）
//    ――――――――――――――――――――――
      if(F_stat_spring[nnx] == 2){
         SETSPG(&j,&itype,
          &F_bz[i1],&F_bz[i1+1],&F_bz[i2],&F_bz[i2+1],&x1,&y1,&x2,&y2);
          pDC->SelectObject ( &NewPenx_a[105]);
         pDC->MoveTo ( (int)x1, (int)y1 );
         pDC->LineTo ( (int)x2, (int)y2 );
          }
```

```
    }}
}
//    ――――――――――――――――――――――
//     ●　部材出力オプションありの場合
//    ――――――――――――――――――――――
       if(m_pst_options == 1){
       int mem,mem_line;
       for( int i = 0; i < mnbsb; i++)
       {
        i3 = i*2;
        i1 = (iconsb[i3]-1)*2;
        i2 = (iconsb[i3+1]-1)*2;
        mem_line = imeme_line[i];
        mem = imeme[i];
//    ――――――――――――――――――――――
//     ●　部材出力オプションあり
//    ――――――――――――――――――――――
        if(m_pst_line[mem_line-1] == 1){
        itype = mtype[mem-1];
        ns=F_mtype[itype-1];
        for(int j=0; j<ns; j++){
        int nnx=5*i+j;
//    ――――――――――――――――――――――
//     ●　塑性の始まり
//    ――――――――――――――――――――――
       if(F_stat_spring[nnx] == 1){
         SETSPG(&j,&itype,
         &F_bz[i1],&F_bz[i1+1],&F_bz[i2],&F_bz[i2+1],&x1,&y1,&x2,&y2);
         pDC->SelectObject ( &NewPenx_a[103]);
         pDC->MoveTo ( (int)x1, (int)y1 );
         pDC->LineTo ( (int)x2, (int)y2 );
        }
//    ――――――――――――――――――――――
//     ●　塑性ヒンジ出現（断面の 80%塑性）
//    ――――――――――――――――――――――
         if(F_stat_spring[nnx] == 2){
          SETSPG(&j,&itype,
           &F_bz[i1],&F_bz[i1+1],&F_bz[i2],&F_bz[i2+1],&x1,&y1,&x2,&y2);
          pDC->SelectObject ( &NewPenx_a[105]);
          pDC->MoveTo ( (int)x1, (int)y1 );
          pDC->LineTo ( (int)x2, (int)y2 );
        }
        }}}}
 pDC->SelectObject ( pOriginalPen);
}
C    ――――――――――――――――――――――――――――――――――
C    ●　 SUBROUTINE /STCLSP
C      ――――――――――――――――――――――――――――――――
C    ●
C    ――――――――――――――――――――――――――――――――――
      Subroutine  STCLSP(icol,ii,itime,spring,ichi,mnbsb,mnstep,an)
      real*4 spring(5,mnbsb)
      i=ii+1
      icol=spring(ichi,i)/an*50.+50
```

```
      if(icol.lt.0) icol=0
      if(icol.gt.100) icol=100
      return
      end
C     ―――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE /SETSP1
C       ―――――――――――――――――――――――――――――――
C     ●
C     ―――――――――――――――――――――――――――――――――
      subroutine  SETSP1(jj,itype,x1,x2,mmemb,itime, mem,
     *           mnbsb,mnstep,Fn,Fmy,Fmz,Fay,Fsnp,Fsmyp,Fsmzp)
      real*4 Fn(5,mnbsb),Fmy(5,mnbsb),Fmz(5,mnbsb),
     *       Fsnp(5),Fsmyp(5),Fsmzp(5),fay(5,mnbsb)
      if(itype.eq.2)goto 100
      if(jj.eq.0)then
      x1=Fn(1,mem )/fsnp(1)
      x2=fn(2,mem )/fsnp(2)
      elseif(jj.eq.1)then
      x1=Fmy(1,mem )/fsmyp(1)
      x2=fmy(2,mem )/fsmyp(2)
      elseif(jj.eq.2)then
      x1=Fmz(1,mem )/fsmzp(1)
      x2=fmz(2,mem )/fsmzp(2)
      else
      x1=Fay(1,mem )
      x2=Fay(2,mem )
      endif
      goto 200
100   continue
      if(jj.eq.0)then
      x1=Fn(2,mem )/fsnp(2)
      x2=fn(3,mem )/fsnp(3)
      elseif(jj.eq.1)then
      x1=Fmy(2,mem )/fsmyp(2)
      x2=fmy(3,mem )/fsmyp(3)
      elseif(jj.eq.2)then
      x1=Fmz(2,mem )/fsmzp(2)
      x2=fmz(3,mem )/fsmzp(3)
      else
      x1=Fay(2,mem )
      x2=Fay(3,mem )
      endif
 200  continue
      if(abs(x1).gt.2.) x1=sign(2.,x1)
      if(abs(x2).gt.2.) x2=sign(2.,x2)
      if(mmemb.gt.0) return
      xx1=x1
      x1=x2
      x2=xx1
      return
      end
```

最初に、ウインドウの背景色を設定する関数 disp_frame_w()について

説明する。この関数は非常に単純であり、理解は容易である。まず、当該ウインドウの大きさを配列 F_pos[]に取得する。次に、m_dat_struct 変数をパラメータにして、switch 文で設定する RGB 値を以下のように分類して設定する。

背景色設定コード：m_dat_struct

```
0: オプション画面
1:構造図
  カラー表示でない場合
  カラー表示の場合
    パラメータ m_pst_dis で分類
2:地震波形
default:
```

カラー値である RGB を設定した後、この値を用いて、CBrush クラスのブラシオブジェクトを作成する。このオブジェクトを用いて、ウインドウ内を塗りつぶし、これをウインドウの背景とする。後は、使用したオブジェクトを消去する。

次に、カラー表示で構造透視図を描く関数 disp_graph_5()について説明する。この関数では、多くのオプションが用意されており、例えば、集中質量位置の表示、曲げモーメント図などである。また、この関数がコールされる前に、構造物の 3 次元座標に節点変位が付加された節点位置が、透視変換されて既に配列 F_bz[]に得られており、この座標情報を元に図形が描かれることになる。

関数 pDC->Arc()は、円を描く VC++の関数である。

まず、パラメータ m_pst_mss をチェックし、その節点の質量有無を確認した後、関数 pDC->Arc()を用いてその位置に円を描く。ユーザーが質量の描画を要求すると、パラメータは m_pst_mass = 1 となる。

次からは、構造物の透視図を描くわけであるが、各種のオプションがあるため、多くの描画コードが記述されている。この中で、まず、部材出力オプション m_pst_options がなしの場合とありの場合とに、大きく 2 つに分けられる。これは、図 4-6 の解析表示選択ダイアログ中のグループ表示使用をチェックすると m_pst_options = 1 となり、その下のデータ入力域で設定したグループ部材が図形出力されることになる。

部材にグループ番号が設定されており、グループ表示とは、指定したグループ番号に合致した部材のみ表示される。

最初のコードは、グループ表示使用せずで、全部材表示する場合について記述されている。部材の色表示オプション m_pst_color = 1 の場合は、軸力を赤から青に 100 分割して表示する。無応力状態は白であり、軸力は軸方向耐力 anp で無次元化されている。

まず、全部材について、for ループを使用して処理を行う。部材数は mnbsb である。部材番号 i より、要素番号 imeme を取り出し、その要素

番号から要素モデル番号を取得する。その要素モデル番号と要素番号からその部材の軸力耐力 anp をセットする。サブルーチン STCLSP()を用いて、部材軸力の 100 分割値 icol を求める。サブルーチン STCLSP()は、上記のプログラムの終わりに添付されており、簡単な処理内容であるので、理解は容易である。この値 icol より関数 pDC->SelectObject()を用いて、ペンの色を変更する。ここで、NewPenx_a[icol]には、青から赤までの色が既に設定されている。次に、部材両端の節点番号を取得し、透視変換された両端の節点座標 F_bz[]を用いて直線を描く。これらの処理を全部材について行うことになる。

次の処理は、m_pst_color == 2 && m_pst_options == 0 の場合についてであり、全部材について描画し、さらに応力の閾値によって色を変えるオプションに対する処理を示す。処理内容は上記とほとんど同じであり、異なる部分は以下のようである。サブルーチン SETSP1()は、変数 jj のコード番号にしたがって、部材両端の耐力に対する応力の割合 x1、x2 を計算する。ただし、ここでは、jj=3 に固定されており、塑性関数に対して処理されている。

```
        SETSP1(&jj,&itype,&x1,&x2,&iix,
              &F_time_ii, &memm,&mnbsb,&m_nstep,
              F_n_spring,F_my_spring,F_mz_spring,F_fay,
              &F_snp[mmx],&F_smyp[mmx],&F_smzp[mmx]);
        pDC->SelectObject ( &NewPen);
      if(x1 <= m_pst_ef2 && x2 <= m_pst_ef2 ) pDC->SelectObject ( &NewPenx_a[101]);
      if(x1 >= m_pst_ef1 || x2 >= m_pst_ef1 ) pDC->SelectObject ( &NewPenx_a[106]);
```

ここでは、サブルーチンで計算された塑性関数値がダイアログで設定した上限と下限の間にあるか、あるいはその間以外にあるかをチェックし、２つの色で分類する。この２種に分類したペンでその部材の両端の節点位置を用いて図形を描く。他の部材モデルについても同様の処理を行い、データ入力した閾値の間か否かを、色で区別して部材を表示する。

次に続く２つの処理は、上記の２つの処理とほとんど同じであり、異なる部分は、以下の部分である。ここでは、表示する部材と表示しない部材をグループ番号で区別する。

```
//    ―――――――――――――――――――――
      mem_line = imeme_line[i];          ! グループ番号
//    ―――――――――――――――――――――
//     ●  部材出力オプションあり
//    ―――――――――――――――――――――
      if( m_pst_line[mem_line-1] == 1 ){
```

まず、部材番号ｉよりグループ番号を取得する。さらにそのグループ番号を有するグループが、ダイアログで指定した表示グループ m_pst_line であるか否かをチェックする。表示グループである場合は、上記の２つの処理と全く同じ処理を行うことになる。

図 4-6　構造図のプロパティダイアログ

上記の処理で基本的な構造物の透視図を描くことができた。図 4-6 のダイアログで各種のオプションが指定されるが、この場合、基本の構造透視図形の上に、指定された曲げモーメントなどを重ね書きをすれば良い。ここでは、それらのオプションのひとつである塑性ヒンジの表示について解説する。この塑性ヒンジの表示は、関数 disp_graph_4()で行われる。

この関数の中でも、構造図の描画部分と同様に、部材出力オプションがなしの場合とありの場合とに、大きく２つに分けられる。まず、ない場合について記述されており、全部材について以下の処理を行う。両端の節点番号を取得し、次に要素番号を、また、その要素に対する塑性チェック位置数 ns を取得する。チェック位置数分以下の処理を行う。塑性状態であるかどうかを、関数 F_stat_spring()を用いてチェックする。関数値が 0 は弾性、1 は断面内に初めて塑性領域が生じた場合、2 は塑性ヒンジの発生もしくは断面の 80%が塑性化した場合を表す。サブルー

チン SETSPG()を用いて、透視図形の中で当該部材のどの位置に生じているかを求め、その位置に直線を描く。最初は、断面内に塑性化が生じたとき、直線を黄色で描く。次は、関数値が２の場合で、つまり、断面内がほぼ塑性化したときは、赤色で直線を引く。

部材出力オプションがある場合も、一部異なるが、ほとんど上記の処理と同じである。その異なる部分は、部材出力オプションをチェックする部分であり、これは、構造図を描く場合と同じである。