７．グラフィックス処理

７－１．OpenGL の初期化・終了処理と Windows バージョンについて

OpenGL を使用する画面は、メイン画面、平面図を表示するダイアログ（配置、視点設定、視点抽出、形状生成・平面、プラグインのいくつか）における編集画面、マテリアル・カラー編集の色玉、型鋼、道路法面・園路法面のマテリアル編集の小さな四角、グラフィックなマテリアル編集、カラー編集等）がある。これらは全て固定数であるが、その他にまた、景観データベースの検索結果の表示用の小さな窓（検索結果に応じて不定数）がある。

メイン画面、及び編集のために一時的に開く編集画面のいくつかは、その一部に OpenGL 描画のための子ウィンドウを持っている。例えば、配置、平面、視点移動、可視範囲解析などは、オルソ系（平面図表示）の OpenGL 子ウィンドウを有する。OpenGL の描画のための子ウィンドウに関しては、全て、専用のクラス(CWnd の派生クラス)を用意し、これを用いて描画処理を行っている。

OpenGL 画面の描画のためには、デバイス・コンテキスト（DC）と OpenGL 用のレンダリング・コンテキスト(HGLRC)の二つのコンテキストが必要とされる。この内、Windows の描画として OS で用意される DC には、各 Window 固有のデバイス・コンテキストと、共通リソースから一時的に取得されるデバイス・コンテキストの２種類があり、後者の場合、同時に最大５以内という制約がある。共通リソースのデバイス・コンテキストを、ひとつのアプリケーションが DC を取得後、描画が終わった後にも解放せずに保持しつづけると、別のアプリケーションの表示に障害をもたらす。過去のバージョンを振り返ると、WindowsME において、リソースの制約が最も厳しく、描画が終了した後に DC を解放しないと、すぐにリソース不足による描画の異常を生じた。

WindowsXP、2000 においてはプログラム修正を要する変化はなかったが、その後 WindowsVISTA において、DC の管理方法が若干変化したと見られ、描画が本来行われる窓の中だけでなく、デスクトップや他のウィンドウの中に描画が行われる場合が見られた。解放された DC に HGLRC が結び付いたままであると、その DC を取得した別のウィンドウ（例えばデスクトップ）に、想定外の再描画が行われる現象が生じた。

この障害は、DC を開放する前に、DC->ReleaseOutputDC()を実行し、HDC を無効とすることにより解決する。しかし、共通の関数で不特定多数の OpenGL ウィンドウを扱っている景観データベース検索結果表示部分など、一部の OpenGL ウィンドウに関しては、この方法で処理することができなかったため、内部的なビットマップウィンドウをメモリ上に構築し、視点位置や光源やモデルに対する修正が行われた場合にはこのビットマップへの描画を行っておき、外から見えるウィンドウに対しては、OnPaint()関数の中で、ビットマップのコピー(BitBlt())を行う方法で解決した。この方法は、グラフィックス・カードのハードウェアによる高速描画の機能が利用できない可能性がある一方、上記の内容変更が

なく、単に他のウィンドウとの位置関係が変化しただけの理由による再描画は速くなる。

OpenGL ウィンドウを管理するライブラリ関数は、二系統用意してある。一つは、wg3XXX、g3YYY という名称の、視点移動等を伴う多機能な OpenGL ウィンドウを処理するためのライブラリ関数である。もう一つは、wg3sXXX、g3sYYY という名称の、主として画像やカラー等を小さなウィンドウで表示するためのライブラリ関数である。OpenGL ウィンドウは、構築とともにリストに登録され、除却に際してリストから抹消される。wg3、g3 系列と、wg3s、g3s 系列とでは、別のリストで管理している。

各ウィンドウのハンドラ・ルーチン(cpp)は、描画に用いる DC のハンドル HDC をポイントする固有のメンバ変数をもっている。上記のリストには、DC そのものを登録することはできないが、このメンバ変数のアドレスを登録するようになっている。DC は、描画が必要となる都度取得し、このポインタに関連づける。一方、HDLRC は、初期化段階で取得されたものが、このリストに登録され、ウィンドウが除却されるまで保持される。

リスト７－１：標準的な OpenGL ウィンドウの初期化

```
int Cxxx::OnCreate(LPCREATESTRUCT lpCreateStruct){
  m_HDC = new HDC;    //クラスのメンバ変数(ポインタ)にメモリブロックを割り当てる
  pmCDC = GetDC();                              //共有 DC を取得
  *m_HDC = pmCDC->GetSafeHdc();                 //HDC をメモリブロックに格納
  set_pixel(*m_HDC);                            //ピクセル・フォーマットを設定
  m_HGLRC = wglCreateContext(*m_HDC); //OpenGL レンダリングコンテキスト設定
  wg3EntryDrawarea((void*)&m_HDC,(void*)&m_HGLRC);  //設定内容を登録
  status = wg3AssignDrawarea((void*)&m_HDC);  //作業対象をこの表示画面に
  status = g3InitializeWindow();                //初期化処理
/*****************以下、その他の初期化（中略）******************/
  wg3AssignDrawarea(NULL);                      //作業対象をこの表示画面から外す
  pmCDC->ReleaseOutputDC();                     //DC を返す前に、出力との関係を切る
  ReleaseDC(pmCDC);                             //DC を共有プールに返す
  return 0;
}
```

リスト７－２：標準的な OpenGL ウィンドウの除却

```
BOOL Cxxx::DestroyWindow()
{
  // 031106 OnDestroy より移動（理由：OnDestroy では wglMakeCurrent が失敗する
  CClientDC dc(this);
  *m_HDC = dc.m_hDC;
  int rc = wg3AssignDrawarea((void*)&m_HDC);
  if(!rc) z3Message( 1383, "¥nCxxx::OnDestroy()" );

  wg3DeleteDrawarea((void*)&m_HDC);
  wglDeleteContext(m_HGLRC);

  return CWnd::DestroyWindow();
}

void Cxxx::OnDestroy(){
```

```
   CWnd::OnDestroy();
   delete m_HDC;
}
```

リスト７－３：標準的な OnPaint コールバック

```
void CDrawFrm::OnPaint()
{
   CPaintDC dc(this); // 描画用のデバイス コンテキスト。このスコープの中でのみ有効
   *m_HDC = dc.m_hDC;
   if(!wg3AssignDrawarea((void*)&m_HDC)) z3Message(5377);
   if(!wg3Redraw((void*)&m_HDC)) z3Message(1425);
   wg3AssignDrawarea(NULL);
}
```

リスト７－４：メモリ・デバイスを用いる場合

（メモリ・デバイス上に描画しておき、表示内容、視点あるいはウィンドウサイズ等に変更がない OnPaint では、画面の無効領域にメモリ・デバイスから画像コピーするだけとする構成）

```
int CMyOrthoVW::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CWnd::OnCreate(lpCreateStruct) == -1)   return -1;

   // TODO: この位置に特殊な生成用のコードを追加してください。
   CPaintDC dc(this);
   mDC.CreateCompatibleDC(&dc);
   mBM.CreateCompatibleBitmap(&dc,1,1); //サイズ１×１のビットマップを仮に作る
   mDC.SelectObject(mBM);

   m_sHDC = new HDC;
   *m_sHDC = mDC.m_hDC;//dsb.
   zet_pixel(*m_sHDC);
   m_sHGLRC = wglCreateContext(*m_sHDC);
   wg3EntryDrawarea((void*)&m_sHDC,(void*)&m_sHGLRC);
   wg3AssignDrawarea((void*)&m_sHDC);
   g3InitializeWindow();
   return 0;
}

BOOL CMyOrthoVW::DestroyWindow()
{
   // TODO: この位置に固有の処理を追加するか、または基本クラスを呼び出してください
   TRACE("CMyOrthoVW::DestroyWindow() ¥n");
   CClientDC dc(this);
   dc.AssertValid();
   *m_sHDC = dc.m_hDC;//ME マシンで、次行で失敗する。
   if(!wg3AssignDrawarea((void*)&m_sHDC)) z3Message(5000,"Ortho::OnDestroy1");
   g3ClearHilight();
   g3SetLightGroup(NULL);//引数 NULL で、メモリの解放を行う。
   g3SetCameraPers(NULL);//同上
   g3ReleaseGroundPixel();
   wg3DeleteDrawarea((void*)&m_sHDC);
   wglDeleteContext(m_sHGLRC);
```

```
   return CWnd::DestroyWindow();
}

void CMyOrthoVW::OnDestroy()
{
   CWnd::OnDestroy();
   // TODO: この位置にメッセージ ハンドラのコードを追加してください。
   delete m_sHDC;
}

void CMyOrthoVW::OnSize(UINT nType, int cx, int cy)
{
   CWnd::OnSize(nType, cx, cy);

   // TODO: この位置にメッセージ ハンドラのコードを追加してください。

   CClientDC clientDC(this);
   *m_sHDC = clientDC.m_hDC;//dsb.
   if(cx == 0 || cy == 0) {
      ;
   } else {
      wg3AssignDrawarea((void*)&m_sHDC);
      g3Resize( cx, cy );
   }
   mBM.DeleteObject(); //画面サイズが変わった場合には、メモリビットマップを更新
   mBM.CreateCompatibleBitmap(&clientDC,cx,cy);
   mDC.SelectObject(mBM);
   RedrawWindow();//新しい空のビットマップに再描画
}

void CMyOrthoVW::OnPaint()
{
   TRACE("MyOrthoVW::OnPaint[%d]¥n",giFirst);
   //081004 合理化実験
   RECT Rect;
   if(1){
      int rusak;
      rusak = GetUpdateRect(NULL,FALSE);
      if(!rusak){//ここには来ない
         MessageBeep(MB_ICONASTERISK);//0x40 ピンポン
         return;
      }else{//被さっているウィンドウが静止した場合
//          MessageBeep(MB_OK);//0:トン
         rusak = GetUpdateRect(&Rect,FALSE);
      }
   }

   CPaintDC dc(this); // 描画用のデバイス コンテキスト。
   //CPaintDC の構築は、BeginPaint を意味する
```

```
   // TODO: この位置にメッセージ ハンドラのコードを追加してください。

   *m_sHDC = dc.m_hDC;//dsb.

   int            GrpCnt;
   s3LightGroup   *lg;
   int            status, kind;
   g3Ortho        ortho;
   int            val;
   int            sts1, sts2;
   int            MATATETAP=-1;
   float          time;

//          giFirst が-1 のとき、常にオルソ全体視界の初期化を行い、１に変更する。。
//          giFirst が０のとき、前回最後の状態に初期化を行い、１に変更する。
//          giFirst が１のとき、視点の初期化を行わず、通常の GL 描画を行う。
//          giFirst が２のとき、無効領域を補修する
   if(giFirst == -1) {//初期化を必要とする場合
        /***************固有の初期化を行う(略)***************/
        giFirst = 1; //再描画を指定
      }
   }   // giFirst<0 最初に平面描画を行った場合の初期化はここまで。

   wg3AssignDrawarea((void*)&m_sHDC);

   if(giFirst == 0){
      g3SetCameraOrtho(kindsave,&orthosave);
      giFirst = 1;
      // 20000504 DR.H.K.メイン側から光源/時間を取得する必要がある
      wg3AssignDrawarea((void*)&m_pView->m_drawFrm->m_HDC);
      g3GetLightGroup(&lg);
      g3GetTime(&time);
      wg3AssignDrawarea((void*)&m_sHDC);
      g3SetLightGroup(lg);
      g3SetTime(time);
   } else{
      g3GetCameraOrtho(&kindsave,&orthosave);//視点移動等の結果を取得・保存
   }

   if(giFirst == 1){ //表示内容を、メモリ・デバイス上に再描画する
      int x,y;
      m_pParkRoadWnd->BringWindowToTop();
      int rc = wg3Redraw((void*)&m_sHDC);
      g3GetSize(&x,&y);
      rc = dc.BitBlt(0,0,x,y,&mDC,0,0,SRCCOPY); //表示画面全体に反映する
      giFirst = 2;
   }else{//giFirst == 2; メモリ・デバイス上の描画内容を修正する必要なし
      int x,y,top,bottom,left,right,width,height,rc;
      g3GetSize(&x,&y);
      top = Rect.top, bottom = Rect.bottom;
```

```
      left = Rect.left, right = Rect.right;
      width = right - left;
      height = bottom - top;
      rc = dc.BitBlt(left,top,width,height,
         &mDC,left,top,SRCCOPY);  //表示画面の内、無効領域のみを修復
      if(!rc) MessageBeep(MB_ICONHAND);//ジャン
   }
} //OnPaint selesai

//編集・視点移動等の後、RedrawWindow()を実行する
BOOL CMyOrthoVW::RedrawWindow
(LPCRECT lpRectUpdate,CRgn* prgnUpdate, UINT flags){
   if( 1 < giFirst ) giFirst = 1; //メモリ・デバイス上に再描画するフラグを設定する
   return CWnd::RedrawWindow(lpRectUpdate, prgnUpdate,   flags);
};
```

メモリ・デバイスを用いた処理では、表示内容の更新の有無に関してフラグを使用し、地物の編集、視点移動、画面サイズ変更が行われた場合には、メモリ・デバイス上への再描画を行った上で、表示画面に画像をコピーする。表示内容の更新がない、単なるウィンドウの無効領域の更新（例えば上に被さっていた別のウィンドウが移動したような場合）に関しては、メモリ・デバイスから表示画面への画像コピー（無効領域を含む最小限の領域のみ）を行うのみである。

更に、一つのクラスで多数の OpenGL ウィンドウの制御を一括して行っている CChildFrm クラスにおいては、メモリ・デバイスのみを保持し、OnPaint の中で OpenGL コンテキストを生成し、描画し、メモリ・デバイスにイメージを残すだけで、OpenGL コンテキストを除却して処理を終了している。

リスト７－５：多数の OpenGL 子ウィンドウを一括処理する場合

```
void CChildFrm::OnPaint()
{
        int status;
        void zet_pixel(HDC);

        CPaintDC dc(this); // 描画用のデバイス コンテキスト。
        if(!GLFLAG){ //無効領域の再描画の場合、画像をコピーするのみ
                RECT rect;
                rect = dc.m_ps.rcPaint;
                dc.BitBlt(rect.left,rect.top,rect.right-rect.left,rect.bottom-rect.top,
                        &mDC,rect.left,rect.top,SRCCOPY);
                return;
        }
        dbImageData *img = NULL;
        dbDataBase*     pDBtab = dbGetDataBaseAddr();
        int DBno = dbGetDB();
        int recNo,recHeadNo;
        int rc;
        CRect clientrect;
        GetClientRect( &clientrect );
```

```
        /*-----------------データベースより選択画像の情報取得----------------*/
         （中略）
        /*--------------------ＤＣ処理-----------------------*/
        m_hDC = new HDC;
        mBM.DeleteObject();
        mBM.CreateCompatibleBitmap(&dc,m_size.cx,m_size.cy);
        mDC.SelectObject(mBM);
        *m_hDC = mDC.m_hDC;//dsb.
        zet_pixel(*m_hDC);
        m_hGLRC = wglCreateContext(*m_hDC);
        wg3EntryDrawarea((void*)&m_hDC, (void*)&m_hGLRC);
        wg3AssignDrawarea((void*)&m_hDC);
        status = g3InitializeWindow();//980520 DR.H.K. これはＧＬの初期化
        /*--------------------画像の設定-----------------------*/
         （中略）
        /*--------------------縮小イメージの作成------------*/
         （中略）
        /*---------------------表示処理---------------------*/
        g3GetBackImage(&old_img);/*990930 DR.H.K.*/
        if(old_img) d3Free(old_img);
        g3SetBackImage(sml_img);
        wg3Redraw((void*)&m_hDC);
Painted: //以下終了処理。OpenGL のコンテキストを除却する
        d3Free( sml_img );
        g3SetBackImage( NULL );
        rc = wg3AssignDrawarea( NULL );
        wg3DeleteDrawarea( &m_hDC );
        rc = wglDeleteContext( m_hGLRC );
        m_hGLRC = wglGetCurrentContext();//確認のため、取得して見る。(NULL)
        status = dc.BitBlt(0,0,m_size.cx,m_size.cy,&mDC,0,0,SRCCOPY);
        GLFLAG = 0;
        delete m_hDC;
}
```

Windows7 では、描画処理終了後、wg3AssignDrawarea(NULL)を実行していない個所で障害を生じた。なお、OS のバージョンによる違いについて、第 13 章も参照されたい。

旧版に見られた各ウィンドウ作成に際してのプログラマによる不統一も調整した。

（1）ウィンドウの重なり

ウィンドウには親子関係が存在する。例えば、一つのダイアログを構成するボタンやエディットボックス等は、子ウィンドウとして定義されている。子ウィンドウは、親ウィンドウをアクティブにしても、その裏側に隠れることができない。従って、ある編集画面を新たに追加する場合に、メイン画面の子ウィンドウではない、メイン画面と対等のウィンドウとして作成する必要がある。このことを明示的に行うためには、すべてのウィンドウの親ウィンドウである、デスクトップ・ウィンドウの子ウィンドウとして作成する。

（2）ウィンドウのアイコン

ウィンドウの左上にユニークなアイコンを付けるためには、ウィンドウ・クラスを構築

する際に、SetIcon を実行する。

上の（1）と（2）のプログラム例を下に示す。

リスト７－６：ウィンドウの重なりとアイコンの処理例

```
CDispKeinen::CDispKeinen(CWnd* pParent /*=NULL*/)
        : CDialog(CDispKeinen::IDD, pParent)
{
        //{{AFX_DATA_INIT(CDispKeinen)
        m_date = 0.0f;
        //}}AFX_DATA_INIT
        BOOL rc = Create(IDD,GetDesktopWindow());    //デスクトップを親に指定
        SetIcon(AfxGetApp()->LoadIcon(MAKEINTRESOURCE(IDI_SIM)),1);
                                //ウィンドウの左上隅に表示するアイコンを指定する
}
```

７－２．メイン画面の表示処理

メイン画面の表示処理は、CDrawFrm::OnPaint() 関数の中で起動される wg3Redraw() (wg3.c)が入口となって実行される。

wg3Redraw() 関数は、g3draw() (g3drl.c)を呼び出し、この関数の中で描画を行う。

リスト７－７：wg3Redraw 関数

```
int wg3Redraw(void*w)
{
  g3Clear();                    表示のリセット
  g3Draw();                     再描画
  wg3Swapbuffers(w);            ダブル・バッファを表示に反映させる
  return(1);
}
```

g3Draw() 関数は、以下の処理を行っている。

リスト７－８：g3Draw 関数

```
void g3Draw()
{
  draw3dObjects();              三次元オブジェクトの表示
  drawCameramark();             カメラ記号（視点位置を示す）の表示
  drawMeasure();                縮尺の表示
  drawRect();                   指定範囲を示す長方形の表示
  glFlush();                    表示操作
}
```

draw3dObjects() 関数は、以下の処理を行っている。

リスト７－９：draw3dObjects 関数

```
static void draw3dObjects()
{
  int jitter;
  jitter_point *jb;

  cameraProjection();
  cameraViewing();
```

```
  if(da[cd].draws[G3_DRA_LIGHT]){                    光源が定義されている場合、設定
    light();
    lightEnable();
  }
  if(da[cd].draws[G3_DRA_ANTIALIAS] && da[cd].aaCount > 0){ アンチエリアシング
    jb = getJitter(da[cd].aaCount);
    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < da[cd].aaCount; jitter++) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      if(da[cd].backImage && da[cd].draws[G3_DRA_BACKIMAGE]){ /*1996.5.10*/
        drawImage(da[cd].backImage->width,
                  da[cd].backImage->height,
                  da[cd].backImage->pixels);
      }
      accCamera(jb[jitter].x, jb[jitter].y);
      cameraViewing();
      display3dObjects();
      if(da[cd].frontImage && da[cd].draws[G3_DRA_FRONTIMAGE]){ /*1996.5.10*/
        drawImage(da[cd].frontImage->width,
                  da[cd].frontImage->height,
                  da[cd].frontImage->pixels);
      }
      glAccum(GL_ACCUM, 1.0/(GLfloat)da[cd].aaCount);
    }
    glAccum(GL_RETURN, 1.0);
  }else{                                             通常の場合
    if(da[cd].backImage && da[cd].draws[G3_DRA_BACKIMAGE]){ /*1996.5.10*/
      drawImage(da[cd].backImage->width,          背景があれば表示する
                da[cd].backImage->height,
                da[cd].backImage->pixels);
    }
    display3dObjects();                              三次元要素を表示する
    if(da[cd].frontImage && da[cd].draws[G3_DRA_FRONTIMAGE]){ /*1996.5.10*/
      drawImage(da[cd].frontImage->width,          前景があれば表示する
                da[cd].frontImage->height,
                da[cd].frontImage->pixels);
    }
  }
}
```

display3dObjects() 関数は、以下の処理を行っている。

リスト７－１０：display3dObjects 関数

```
static void display3dObjects()
{
  drawAxis();                                      座標軸の表示
  drawGrid();                                      グリッドの表示
  if(da[cd].draws[G3_DRA_BUMI] == TRUE) drawAllGround();      地面のみ表示
  else {
        drawAllGroupEtc();                         全てのグループを表示
        if(g3GetDrawable(G3_DRA_BUMITEX)){   地面だけテクスチャ表示する場合
              if(!g3GetDrawable(G3_DRA_TEXTURE)){
```

```
                        int ingat;
                        ingat = g3GetDrawable(G3_DRA_WIRE);
                        if(ingat) g3Disable(G3_DRA_WIRE);
                        g3Enable(G3_DRA_TEXTURE);
                        drawAllGround();
                        g3Disable(G3_DRA_TEXTURE);
                        if(ingat) g3Enable(G3_DRA_WIRE);
                }else;
        }else;
  }
  drawHilight(); /*1996.3.1*/          選択されたオブジェクトの強調表示
  drawOrbit();                         軌道の表示
  drawVisual(); /*daerah terlihat*/    可視範囲の表示
}
```

drawAllGroupEtc() 関数は、以下の処理を行っている。

リスト７－１１：drawAllGroupEtc 関数

```
drawAllGroupEtc(){
  if( g3GetDrawable(G3_DRA_SHADOW)){
    g3MakeShadowVolume();              影の場合、影立体を作成する
    drawAllGroupAndShadow();           影付きで全グループ表示を行う
  }else{
    drawAllGroup();                    通常の全グループ表示
  }
}
```

drawGroup() 関数は、以下の処理を行っている。

リスト７－１２：drawGroup 関数

```
static int materialsekarang=0, tekturesekarang=0;
    マテリアルとテクスチャの現在の設定状況を保持するスタティック変数

static void drawGroup(d3Group *g)
{
  d3Face *f;
  d3Link *l;
  d3Group *child;
  d3Matrix linkMat;
  int tree3;
/*980508 DR.H.K.*/
  int bahaningat;  /*マテリアルとテクスチャを一時退避するオート変数*/
  int teksilingat;
  int bahan;       /*当面のマテリアルとテクスチャ*/
  int teksil;
/*dsb*/

  if(!g) return;
  tree3 = i3IsAttribute(g,I3_ATTR_TREE3);

/*マテリアルとテクスチャの設定*/
  bahan = d3GetGroupMaterial(g);       グループに定義されたマテリアルを取得
  teksil = d3GetGroupTexture(g);       グループに定義されたテクスチャを取得
```

```
   if(bahan || teksil ||g->lup==NULL   ){
         /*両方ともない場合、明示的にないのではなく、単に未定義と考える*/
         /*ルートグループで無い場合、明示的にないことにする*/
     if(!da[cd].draws[G3_DRA_WIRE]){
       setMaterialTexture(bahan,teksil);
/*この中で、マテリアルにテクスチャがあれば teksilsekarang はそれに変更される*/
     }else{
       setMaterialTextureLine(bahan,teksil);
     }
     bahaningat = materialsekarang;        /*auto 変数をスタックとして用いる*/
     teksilingat = texturesekarang;        /*面、子グループの処理後、戻すため*/
   if(bahan+teksil){ /*グループにいずれかが定義されていた場合、それを設定する*/
     materialsekarang = bahan;
     texturesekarang = teksil;
   }/*この処理は、そのグループにマテリアルかテクスチャの
    いずれかが定義されていれば、親から継承せずに独自の属性を用い、
    かつその属性を面と下位グループに継承する、というロジックを具体化する*/

/*マテリアルとテクスチャの設定完了*/

   以下、グループに属する面を表示する
   for(f=NULL;f=d3GetGroupFace(g,f);){
     int hasil;
     d3Group *Gsakit;
       if(hasil=drawFace(f,tree3)){
         for( Gsakit=g; Gsakit->lup; Gsakit = Gsakit->lup->up->parent){
           if(Gsakit->type == I3_GTYPE_FILE){
             z3Message( 5372, Gsakit->kind );
             break;
           }
         }
      }
   }
/*  以下、子グループを表示*/
   for(l=NULL;l=d3GetGroupDLink(g,l);){
     child = d3GetLinkDGroup(l);
     d3GetLinkMatrix(l,linkMat);
     if(berapa(linkMat)){
         glPushMatrix();
         glMultMatrixd(linkMat);
         drawGroup(child);
         glPopMatrix();
     }else{
         drawGroup(child);
     }
   }
/*待避したマテリアル、テクスチャ情報の復帰*/
   if(!da[cd].draws[G3_DRA_WIRE]){
    /*テクスチャ表示およびシェーディング表示の場合、処理前の状態に戻す*/
     setMaterialTexture(bahaningat,teksilingat);
```

```
  }else{
    setMaterialTextureLine(bahaningat,teksilingat);
  }
  materialsekarang = bahaningat; /*親から継承されたものに戻す*/
  texturesekarang = teksilingat;
}
```

グループには、マテリアル、及びテクスチャの情報を付けることができる。この情報がグループに定義されていない場合には、親グループの属性が継承される。

また、グループにマテリアルやテクスチャの情報が無い場合であっても、そのグループに直接属する面にマテリアル、テクスチャ、またはカラーの情報を付けることができる。

７－３．面の表示処理

面の表示は、drawFace(*F)関数によって実行される。

面が凹ポリゴン(D3_SHP_CONCAVE)のフラグを有し、頂点が４以上あって、表示モードがワイヤーフレーム以外（テクスチャ、シェーディング）の場合には、面を三角形に分割して、drawMuka() 関数を通じて、OpenGL に出力する。

凹ポリゴンのフラグが無い場合には、そのまま drawMuka()関数に渡される。

CONCAVE(ON); の状態においては、凹ポリゴンの検査を行い、drawMuka(*F)関数で、凹ポリゴンを正しく表示する。

CONCAVE(OFF); の状態においては、上記の検査は省略され、無責任モードで表示が行われる。凹ポリゴンは正しく表示されない。

マテリアル、テクスチャ、カラー

マテリアル、およびテクスチャは、グループに対して定義することができる。

グループに帰属する面に対して、個々に定義することもできる。その場合、グループに対して定義されたマテリアルもしくはテクスチャは、デフォルト値のように扱われ、別の値が定義された面はそれを用いて表示するのに対して、未定義の面に関してはグループに定義された値を表示に用いる。

この他、面に対してはカラーを定義することができる。カラーは、頂点に対しても定義することができ、面に定義されたカラーは、上記と同様に、頂点に定義されたカラーに対するデフォルト値のように参照される。

同一の面に対して、マテリアルとカラーが同時に定義されていて、マテリアルにカラー情報が含まれている場合、カラー設定値が上書きされる。

最終的な表示においては、OpenGL の表示系に対しては、

Ambient[4]

Diffuse[4]

Specular[4]

Shininess[1]

Emission[4]

マテリアルで定義されている属性は、最終的な OpenGL での表示に使用される属性に対応したものとなっている。

リスト７－１３：dbMaterial 構造体定義

```
Typedef struct _dbMaterial{
  float ambient[4];        環境光との関係で表示色を決定する物体色
  float diffuse[4];        拡散光との関係で表示色を決定する物体色
  float specular[4];       反射光との関係で表示色を決定する物体色
  float shniness;          鏡面指数
  float emission[4];       物体独自の放射光
  char *texture;           マテリアルの中で定義されるテクスチャ名称
}
```

カラー(r,g,b,a)が設定されている場合には、その値を、ambient 及び diffuse に反映する。その際に、ambient は、r,g,b,a 値に DB_AMBIENT_RATE (dbms.h で 0.3f と定義)を乗じた値を与え、diffuse は r,g,b,a 値をそのまま代入している。

マテリアルの属性は、マテリアル・ファイルに保存されている。この中では、以下の項目を定義している。

Lab：カラー値（LAB 表色系で表現　●実装されていない）

RGB: カラー値（diffuse[0-2]に値を代入）

SPECULAR: 反射率（specular[0-2]に共通値を代入, specular[3]は 1.0）

EMISSION: 輝度(diffuse[0-2]に共通値を乗じた値を emission[0-2]に代入)

TRANSPARENCY: 透明度（diffuse[3]に値を代入）

TEXTURE: テクスチャ(テクスチャ名称を取得し代入)

７－４．ステレオ表示機能

人間の目は、三次元の地物の透視図を網膜において二次元画像として入力している。その際に、左右の目の視点位置の違いによる画像の差異を認識することにより遠近感を得ている。景観シミュレータにおいてこのことを再現するためには、左右の目に対応する、異なる視点からの画像を同時に生成するプログラムを作成すると共に、二つの画像を、ユーザーの左右の目に分離して送り込むためのハードウェアを必要としている。

ヘッドマウンド・ディスプレイのように、左右の目のために二つのディスプレイ装置を用意するのが最も直接的である。しかし、機材のは高価であり、また複数のユーザーが同時に同じ映像を見て討論するためには多くの機材を用意しなければならない。

そこで、プロジェクタによりスクリーンに投影された画像、またはモニタ画面に表示された画像を時分割または面分割し、異なる画像を左右の目に出力する方法が多く採用されている。時分割するためには、左右の画像を交互に表示すると共に、シャッター機能のあるメガネをユーザーに装着してもらい、表示のリフレッシュとシンクロして、左右の目の

シャッターを交互に開くような装置を使用する。一部のゲームマシン等に用いられている。もう一つは、画面に左右の画像で異なる偏光をかけ、ユーザーは、左右で異なる向きに取り付けられた偏光フィルタを装着してもらい、これを眺める方法である。

プロジェクタの場合には、２台のプロジェクタを用意しておき、それぞれに、異なる向きの偏光フィルタを取り付けることで同一のスクリーンに重複した画像を投影する。偏光メガネをかけてこれを眺めることにより、画像を分離する。スクリーン面で反射する際に偏光が変わることを避けるために、立体投影のためには通常のスクリーンではなく、アルミ粉などを用いた銀色塗料を施したスクリーンを使用する。このための偏光フィルタを用いたメガネは、フレームに紙などの安価な材料を用いることにより配布可能な単価で製造できるため、この方法はイベント会場などでよく用いられている。

カラー表示の必要がなければ、例えば青色フィルタと赤色フィルタを用いた紙メガネを観客に配り、これで眺めると左右が分離するような配色で、例えば山岳の空中写真を重複印刷したパネルを展示するようなイベントは古くから行われていた。

偏光フィルタによる紙メガネを用いてステレオ表示を行うためにはいくつかの方法がある。複数のプロジェクタを用いてスクリーンに投影する方法は、仮想現実としてしばしば採用されている。複数のコンピュータ上に複数のプログラムを動作させ、左右の画像がシンクロするように視点移動を行う方法が、最も処理速度の高いハイエンドの方法である。この左右プロジェクタの組を正面のみならず左右上下のスクリーンにも適用することにより全方位の立体画像を生成する、没入感の高い仮想現実装置が製品化されている。

景観シミュレータにおいては、ステレオ表示を、①画面走査線の奇数ラインと偶数ラインに右目視点と左目視点の画像を作成すること、②画面マトリクスの行単位でストライプ状に偏光の向きを替えるマイクロポール・シートを組み込んだ液晶ディスプレイにより、奇数ラインと偶数ラインに異なる偏光を与えること、③偏光メガネにより、右目に奇数ライン、左目に偶数ラインだけを見せること、の３点により実現している。

奇数ラインと偶数ラインに異なる視点位置の画像を表示するために、g3Draw()関数の中で、分離する処理を行っている。

リスト７－１４：ステレオ表示を行わない g3Draw 関数

```
void g3Draw()
{
  if(cd<0) return;
  draw3dObjects();
  drawCameramark();
  drawMeasure();
  drawRect();
  glFlush();
}
```

表示モードが「パース」で、StereoMode が NULL でない場合に、ステレオ表示を行い、それ以外の場合では、従来と同様の表示を行う。

リスト７－１５：条件によりステレオ表示を行う g3Draw 関数

```
void g3Draw()
{
  static unsigned char MaskBitmap
```

```
                                    [MAX_SCREEN_WIDTH* MAX_SCREEN_HEIGHT / 8 ];
static int MaskWidth = 0;
static s3Camera PersSave;          // Pers infomation save area from da[cd].pers

static s3Camera rightPers, leftPers;              // perspective information area
s3Camera *prightPers=NULL,*pleftPers=NULL,*pPersSave=NULL;
GLint i;
GLfloat    map[] = { 0.0, 1.0 };

int       w = 1600;
FILE    *fp=NULL;

if( cd < 0 )    return;

//w = ( ( da[cd].width + 15 ) >> 4 ) << 4;
w = ( ( da[cd].width + 31 ) >> 5 ) << 5;
if( w != MaskWidth )
{
   for( i = 0; i < MAX_SCREEN_HEIGHT; i += 2 )
      memset( MaskBitmap + w / 8 * i, 0xff, w / 8 );
   for( i = 1; i < MAX_SCREEN_HEIGHT; i += 2 )
      memset( MaskBitmap + w / 8 * i, 0x00, w / 8 );
}

if(da[cd].cameraKind == G3_CAM_PERS && StereoMode)    // Check stereo mode
{
   PersSave = da[cd].pers;    // Save current perspective information

   pPersSave=&PersSave;
   pleftPers=&leftPers;
   prightPers=&rightPers;
            //Note:For standard C could not return by &,so changed it to *
   g3GetStereoPers( StereoEye, pPersSave, pleftPers, prightPers );


   glPixelMapfv( GL_PIXEL_MAP_S_TO_S, 2, map );
   glPixelMapfv( GL_PIXEL_MAP_I_TO_R, 2, map );
   glPixelMapfv( GL_PIXEL_MAP_I_TO_G, 2, map );
   glPixelMapfv( GL_PIXEL_MAP_I_TO_B, 2, map );
   glPixelMapfv( GL_PIXEL_MAP_I_TO_A, 2, map );

   glEnable( GL_STENCIL_TEST );
   glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );
   glClear( GL_STENCIL_BUFFER_BIT );

   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity( );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity( );

   glRasterPos2i( -1, -1 );
   glDrawPixels(w,da[cd].height, GL_STENCIL_INDEX, GL_BITMAP, MaskBitmap );

   glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
   glStencilMask(0x7fff); // investigating--prohibit wrinting first bit

   glStencilFunc( StereoSwap ? GL_EQUAL : GL_NOTEQUAL, 0x0, 0x1);
   da[cd].pers = rightPers;
   draw3dObjects();            // draw objects to odd lines
   drawCameramark();          // draw cameramark to odd lines of back buffer
   drawMeasure();              // draw measure to odd lines of back buffer
   drawRect();                 // draw rect to odd lines of back buffer

   glStencilFunc( StereoSwap ? GL_NOTEQUAL : GL_EQUAL, 0x0, 0x1);
   da[cd].pers = leftPers;
   draw3dObjects();
   drawCameramark();
   drawMeasure();
```

```
      drawRect();
      glFlush();

      glDisable( GL_STENCIL_TEST );
      da[cd].pers = PersSave; // Restore perspective information
    }
    else
    {
//    Original g3Draw is as follows
      draw3dObjects();
      drawCameramark();
      drawMeasure();
      drawRect();
      glFlush();
    }
}
```

ステレオ表示における右目・左目の視点位置を計算する、g3GetStereoPers 関数が、新たに追加されたソース StereoSight.c に定義されている。

左右の目のために共通の表示画面を用いるため、左右の目と注視点を結ぶ線は平行ではなく、また表示画面と斜交する。このため、左右の目の事成る視点位置の違いを表示に反映させるために、単に視点位置を左右にずらすのではなく、全体を glTanslated 関数で平行移動した上で、注視点付近が画面の同じ位置となるように表示範囲を変えるために glFrustum 関数を用いて左右に「アオリ」をかけている(g3drl.c の cameraProjection 関数)。

スクリーンで、ステレオ表示を行っている景観シミュレータのウィンドウが縦方向に移動された場合には、偏光フィルタとの関係において奇数行と偶数行に対する左右の画像の出力を反転しなければならない。従来の StereoSwap フラグは、スクリーン上のウィンドウ位置に無関係に固定的に左右画像と奇数ライン・偶数ラインを関係付けるものであったが、Ver.2.09 においては、表示画面（その最上行）が現在存在する位置が、画面全体の中の奇数行か偶数行かを検出した上で、それに対応したステレオ画像を生成・出力するように改良した。従来のバージョンでは、画面サイズの変更が無い移動の場合には、他のウィンドウとのオーバーラップの変化などが無い限り、OpenGL の再描画を省略していたが、ステレオ表示で上記の対応を実現するためには、移動に際しても再描画を行うこととした。但し、移動の途中で、OnMove イベントが発生する度に再描画を行うと、画面操作自体の粘度が非常に高くなるため、OnMove イベントの際に 0.2 秒のタイマーをセットし、OnTimer イベントの中で再描画要求を行っている。移動操作が継続している間は、タイマーが再セットされるため、OnTimer イベントは発生しない。この処理は、マテリアル編集画面でスライドバーによりカラー編集を行う場合に、選択したオブジェクトに選択したカラーを適用し、メイン画面を再描画する処理とほぼ同様である。

リスト７－１６：ユーザーによるステレオ表示ウィンドウの移動への対応

```
① CmainFrame::OnMove,  OnTimer
② CdrawFrm::OnPaint
③ G3 ライブラリ G3drl.c の g3draw()関数、FromFirst 変数
```

このようにして作成した左右の目のための画像を、左右それぞれの目に相互干渉することなく出力するための具体的製品においても、近年技術開発・社会普及が進みつつある。

ステレオ表示機能を開発した 2001 年当時には、市販の液晶ディスプレイの表面に特殊な偏光フィルタを貼り付けて、奇数行と偶数行の偏光の向きを変える方法により改造された二次的な製品と、液晶プロジェクタの内部に同様の偏光フィルタを追加した製品が存在していた。本解説書をとりまとめた 2009 年時点では、既にステレオ画面を放送するサービスも開始されており、前者の液晶ディスプレイに関しては、製造段階で偏光フィルタを最初から組み込んだ製品が販売されている。一方液晶プロジェクタに関しては、偏光フィルタを組み込む方法は発展せず、製造中止となっている。これに代わって、二つのプロジェクタに単純な偏光フィルタを装着し、同一のスクリーンに重ねて投影する方法が一般的となっている。複数の sim.exe の視点移動を、立体視のためにシンクロさせるような機構、あるいは OpenGL によるステレオ表示機能を提供するハードウェア（グラフィックス・チップセット）に対応した拡張などは、簡単に行えるため、今後、ステレオ表示機能を含め、表示機能を高度化するためのプラグイン DLL を開発することの価値があると考えられる。

７－５．影の表示

7-5-1.動作原理

光源から見て、影の原因となる物体から、影を落とす対象までの間の部分（影の立体）を求める。この影の立体の側面（複数の四角形）をレンダリングする際に、ステンシルバッファを用いて、各ピクセルに描かれる側面に関して表と裏のカウントを行う。

このカウントが正となる（表側が裏側よりも一つ多い）ピクセルは、影の部分である。

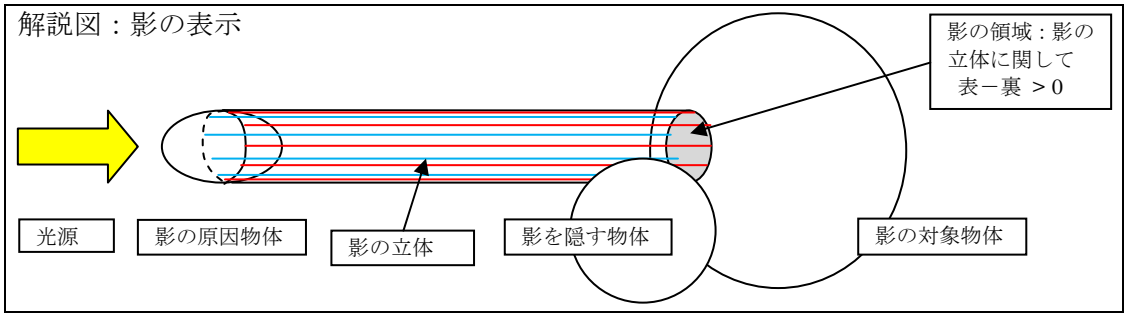

図７－１：影の表示

7-5-2. オペレーション

メイン画面のメニュー[表示][影]の下に、［影なし］［地面以外の物体（通常）］［全物体］［選択した物体］の４選択肢がある。また、［設定］により、影の長さを指定することができる。

関連するプログラムの構成

ShadowDlg.cpp において、影の長さを指定するパラメータの設定ダイアログを処理する。

Mainfrm.cpp　において、影に関するメニュー選択、及びパラメータ設定ダイアログを開く処理等を行っている。

DrawFrm.cpp において、現在選択されている影表示モードに従った表示を行う。

g3drl.c の中で、影に対応した表示処理を行っている。

新たに追加した関数は、

g3MakeShadowVolume() 影の立体を求め、ディスプレイリストに格納する。

drawAllGroupAndShadow() 全グループと、影を表示する。

である。

また、既存の関数に増補し、フラグにより影の機能を選択的に実行している。

drawFace()

MakeShadowFlg のフラグが立っていた場合（g3MakeShadowVolume()からの起動)、影の立体の側面を生成し、ディスプレイリストに追加する。

display3dObjects()

影表示モードの場合には、drawAllGroup の代わりに drawAllGroupAndShadow を実行する。

なお、影の表示段階では、ステンシルバッファを使用しているため、ステレオ表示と共存することはできない。

ステンシルバッファに関しては以下のような OpenGL 関数を使用している。

glStencilFunc(GLenum func, GLint ref, GLuint mask)

func：評価方法

GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, and GL_ALWAYS.

初期値は GL_ALWAYS

ref：評価基準値

ステンシルバッファの値を、この基準値と比較評価する。評価が成功した場合にのみ、描画が行われる。

mask：マスク

ステンシルバッファのマスクした値に関して、評価を行う。初期値は 1

glStencilOp(GLenum fail, GLenum zfail, GLenum zpass)

評価結果の各場合につき、ステンシルバッファに結果をどう反映させるかを指定する。（ステンシル・テスト失敗、デプス・テスト失敗、成功のそれぞれの場合）

指定できる反映方法の選択肢は以下の通り：

GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, and GL_INVERT.

初期値は、 GL_KEEP

これらを用いて、以下の工程で影の表示を実現している。

①まず全ての地物を描画し、デプスバッファを構築する

②影の立体の側面の内、視点から見た表面を描画し、描画が成立したピクセルに関して、ステンシルバッファの値に 1 を足す

③影の立体の側面の内、視点から見た裏面を描画し、描画が成立したピクセルに関して、

ステンシルバッファの値から１を減じる

この処理により、各ピクセルについて、ステンシルバッファの値が１以上となる領域が、影となる地物の表面に相当するピクセルの領域として得られる。なお、影の立体の描画に際してはデプスバッファを変更しない。

④光源を消灯し、影となる領域について、全ての地物を描画する。

メニュー選択で、主光源だけを消すか、全光源を消すかを指定できる。

⑤光源を点灯し、影以外の領域について、全ての物体を描画する。

実際の処理においては、処理速度を向上するために、計算処理のための描画に際して、不必要な表示画面への反映は抑止すると共に、マテリアル、テクスチャ、法線ベクトルなどの描画も省略する。複数回描画される影の立体に関しては、OpenGL の機能を用いて予めディスプレイ・リストを作成し使用する。地物や光源位置が不変の間は有効である。

影の立体は、原因となる物体の各面について作成している。このため、一つの面に関して、影の領域におけるステンシルバッファの差引値は１となるが、原因となる物体が閉多面体であるような場合には、通常、あるピクセルを影とする面は、原因物体の日向側と日陰側に二つ存在することとなり、物体全体では、影の領域におけるステンシルバッファの差引値は２となる。面の境界（稜）から伸びる表向きと裏向きの同一面は相殺可能である。

7-3-3. 影の表示に関する実際のプログラム

影の表示モードは、メイン画面のメニューで設定する。選択されたメニュー項目のコールバックの中で、g3SetShadowMode(G3_SHADOW_XXX);　関数が呼び出される。引数として選択される影の表示モードは、g3drl.h に定義されている。

リスト７－１７：影の表示モード

```
#define G3_SHADOW_ALL                    1   /* All Objects *//* すべての物体 */
#define G3_SHADOW_OTHER_GROUND           2   /* Other Ground Objects *//* 地面以外の物体 */
#define G3_SHADOW_ONE                    3   /* One Object *//* 1 つの物体 */
#define G3_SHADOW_OTHERLIGHT_OFF        10   /* Other LIHGT0 Lights OFF *//* LIGHT0 以外の光源を OFF */
#define G3_SHADOW_OTHERLIGHT_ON         11   /* Other LIHGT0 Lights ON *//* LIGHT0 以外の光源を ON */
```

影の表示を行わない場合には、引数は０である。

影の表示を行うモードが設定されている場合には、g3drl.c における、disp3dObjects() 関数から起動される、drawAllGroupEtc()関数の中で、通常の drawAllGroup の代わりに、

```
g3MakeShadowVolume();
drawAllGroupAndShadow();
```

の処理が行われる。

g3MakeShadowVolume()関数の中では、選択された影の原因物体（選択された物体、全ての物体）に関して、影の立体の側面に関するディスプレイリストを作成する。

次に、drawAllGroupAndShadow() 関数の中で、影の付いた表示を行う。

リスト７－１８：影の表示処理

```
/* シャドウとグループの描画
** drawAllGroupAndShadow( void )
*/
static void drawAllGroupAndShadow( void )
```

```
{
   int res,bits;//080716 DR.H.K.
/*1: 地物のデプスバッファ値を求める*/
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
   res = GetLastError();
   glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE );//表示を止める
   drawAllGroup(); /* render all group in depthbuffer
      影の領域を求めるために下地の地物のデプスを出しておく*/
   /*080715 DR.H.K. drawAllGroup が 3 カ所ある。
      影の形状は、ディスプレイリストに保存されている。*/
/*2: 影の立体の側面の見栄をステンシルバッファに出す*/
   glEnable( GL_STENCIL_TEST );
   glDepthMask( GL_FALSE );//デプスバッファへの追記は止める


   glGetIntegerv( GL_STENCIL_BITS, &res);
   if( res < 1 ){
            MessageBox(NULL, "Stencil Buffer is not available. Please check pixel.c", NULL, MB_OK);
   }
   /* bits = (2 << res -1 ) - 1;  例 : res が 8 なら bits は 255 */
   bits = (1 << res) - 1;
   glClearStencil( 1 ); /*STENCIL BUFFER クリア値を 1 とする*/
   glClear( GL_STENCIL_BUFFER_BIT );  /*それを使ってクリア*/
   /*ステンシルの初期値をゼロとし、1 以上を影とすることで単純化は可能
      ここでは、ステンシルの用例（特に不等号判定）の意味を例示する*/
   glStencilFunc( GL_ALWAYS, 0, bits );

   glEnable( GL_CULL_FACE ); /////
/*2-1: 影の立体の側面の表向きの面をカウントアップする*/
   glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
   glCullFace( GL_BACK ); /* 裏面を表示しない */
   glCallList( G3_DLIST_SHADOWVOL );//DL 化し準備してある影を描画
/*2-2: 影の立体の側面の裏向きの面をカウントダウンする*/
   glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );
   glCullFace( GL_FRONT ); /* 表面を表示しない */
   glCallList( G3_DLIST_SHADOWVOL );
/*3: 地物の表面から影の部分だけを切り出して描画*/
   glDepthMask( GL_TRUE );/*デプスバッファ書込を復活*/
   glClear( GL_DEPTH_BUFFER_BIT );/*デプスバッファをクリア*/
   glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );

   glCullFace( GL_FRONT );//表面を表示しない
   glDepthFunc( GL_LEQUAL );
   glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );/*ステンシル値を固定*/

   glStencilFunc( GL_LEQUAL, 2, bits ); /*  2 がステンシル値以下なら表示する */
   /*即ち、ステンシル値が 2 以上の部分だけ、影と判定して表示する*/
   glEnable( GL_STENCIL_TEST );
   glDisable( GL_LIGHT0 );      //主照明を消灯
   glDisable( GL_CULL_FACE ); //これを実行しないと、裏面だけが表示される
   drawAllGroup();//ステンシルバッファの影の領域を、立体表示
/*4: 影以外の部分を描画する*/
   glStencilFunc( GL_GREATER, 2, bits ); /*  2 がステンシル値を超えるなら表示する */
   /*即ち、ステンシル値が 2 未満の部分だけ、影以外と判定して表示する*/
//参考（同一結果）  glStencilFunc( GL_EQUAL, 1, bits );
                                 /*  1 がステンシル値に等しいなら表示する */
   glEnable( GL_LIGHT0 );       //主照明を点灯
   drawAllGroup();//影以外の地物を再描画する

   glDepthFunc( GL_LESS );
   glClearStencil( 0 );//STENCIL BUFFER クリア値を 0 に戻す
   glClear( GL_STENCIL_BITS );  //それを使ってクリア
   glDisable( GL_STENCIL_TEST );
   glDeleteLists( G3_DLIST_SHADOWVOL, 1 );
}
```

影の原因物体を構成する個々のポリゴンから光源とは反対の向きに、十分な長さ（ユーザーにより設定可能、初期値 1,000m）だけ伸びる影の立体に内包される、影の対象物体の表面上の領域が、影が落ちる領域となる。この部分のピクセルに関しては、地物全体のデプスバッファ値に対して、影の原因物体を構成するポリゴンの内、このピクセルに影を落とす立体の各面が生成する影の立体の前後関係を、影の対象物体を含む全地物のデプスバッファ値を用いて判定（但し、デプスバッファ値は比較結果をもって変更しない）した時に、影の立体の表面の数が裏面の数よりも１多くなる。但し、別の物体により隠され、見えなくなる影の対象物体は、表面も隠されるため、この限りではない。

影の原因物体が閉多面体である場合、影の対象物体の同じ領域に関して、原因物体の日向側の面と日陰側の二つの面から影が落ちる。日陰側の面から生成する影の立体が、外側を裏側とする面で構成されると、これらが表面の数と裏面の数を相殺し、差がゼロとなってしまい判定できない。そこで、影の立体の側面を作成する再に、原因物体を構成する各面に関して、光源に対して日向側と日陰側かを判定し、いずれの場合もポリゴン毎に作成する影の立体を構成する側面の外側が表側になるように、影の立体の面を作成しておく。

なお、表示に際して差引値の判定に用いる glStencilFunc(func, ref, mask); 関数において、第１引数で指定する判定条件により、第 2 引数と、ステンシルバッファ値の比較を指定しているが、例えば GL_GREATER という比較条件においては、「ref > ステンシル値」（逆ではない）という条件設定となる点には注意を要する。そこで、誤解を避ける目的で、表面と裏面の集計判定のために、ステンシルバッファに初期値として１をセットしておき、まず表側を描画し、デプス・バッファテストに合格した点（見える影の立体の表面）について、バッファ値をインクリメントしておく。次に裏側を描画し、デプス・バッファテストに合格した点（見える影の立体の裏面）について、バッファ値をデクリメントする。結果が１よりも大きい場合に、そのピクセルを影が表示される領域と判定している。

次に、このステンシルの判定条件を用いて、まず影の領域に関して主照明または全照明を消して描画を行い、最後に影以外の領域に関して、全照明を当てて描画を行っている。

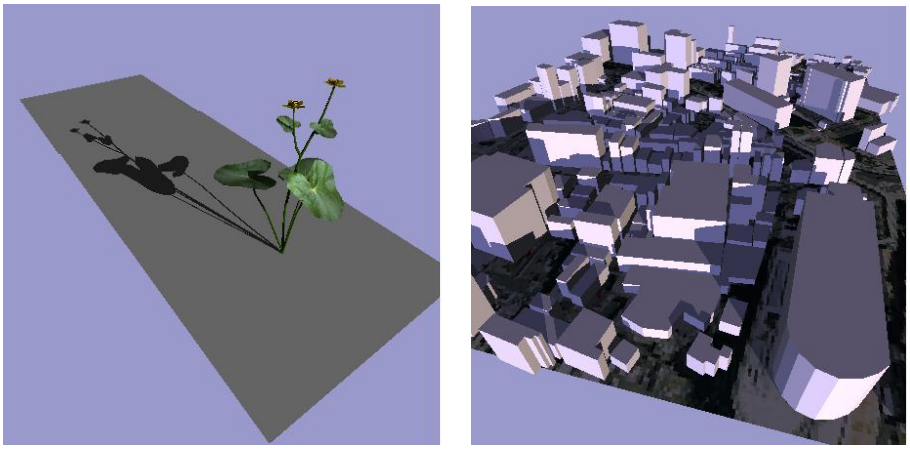

図７－２：影の表示例

７－６．高速表示処理

高速表示処理を行うために、以下の三つの処理機能を追加した。

(1) ディスプレイリストの使用

(2) 連続する三角形、四角形に関する、glBegin～glEnd の省略

(3) マウス・ドラッグによる視点移動

この内、(1)及び(2)は、メインメニューの[表示][高速表示]で開く高速表示設定ダイアログで設定する（4-4(62)）。

また、(3) マウス・ドラッグによる視点移動は、操作ボタンだけではなく、メイン画面上でのマウス右クリックのドラッグ操作によっても指定できるインターフェースを追加した。

7-6-1. ディスプレイリストの使用

ディスプレイリストを用いることにより、データを OpenGL に送出するための準備段階の処理が省略されるため、視点移動を高速化する。特に、CPU の処理速度が遅いマシンに、高速のグラフィックス・ボードが装着されているような場合に改善効果が顕著である。

オブジェクトの編集が行われた場合には、ディスプレイリストを再構築しなければならないため、再描画に時間がかかるが、視点移動のみの場合には、高速で表示される。

同じ物体が多数配置されているような地物構成の場合には、ディスプレイリストに展開した場合には、表示に必要な全ての面に展開されるため、メモリを大量に消費する。

drawAllGroup()の中で、

```
glNewList(dispList, CL_COMPILE_AND_EXECUTE);
    ・・・・・・
glEndList();
```

により、処理している。二回目以降の標示においては、ここで作成したディスプレイリストを用いた表示のみを行う。

glCallList

7-6-2. glBegin と glEnd の省略

三角形、四角形等が連続する場合に、面と面の間にある glEnd, glBegin を省略する。

drawMuka()の中で処理を行う。

数値地図や、ステレオ空中写真・衛星画像から作成した地形データのように、三角形が連続するようなデータでは、この処理による改善効果が高く、ディスプレイリストと併用すると、10 倍以上の速度に改善される場合がある。

WindowsVISTA においては、面に対して設定されるカラー、テクスチャおよびこれらを含んだマテリアルが変わる場合には、一度 glEnd→変化した表面光学特性の設定→glBegin という処理を行わないと正常に表示されない（設定内容が、次の頂点にしか反映されない）。

7-6-3. マウス・ドラッグによる視点移動

CDrawFrm::OnMouseMove() 関数により実行する。この関数は、マウスが移動された場合に起動する。マウスの右ボタンの状態、シフトキー押し下げの状態、及び Ctrl キーの押し下げの状態は、それぞれのイベントに応じてフラグに設定しておく。表示がパースで、右ボタンが押し下げられている場合以外は何もしない。次にマウスカーソルの位置を記憶しているスタティック変数と現在のマウスカーソルの位置を比較し、差分に応じて視点を移動する。左右の移動は、注視点を中心とする視点の左右回転に、マウスの上下の移動は視点の上下の回転に反映させる。これは、メイン画面下の「回転」の操作ボタンに対応する。シフトキーが押されている場合には、視点と注視点を平行移動する。これは、「シフト」の操作ボタンに対応する。コントロールキーが押されている場合には、マウスカーソルの上方移動に応じて接近、下方移動に応じて後退する。これは「接近」「後退」の操作ボタンに対応した動きである。（リスト７－１９）。

視点座標ダイアログ(CEditShit)が開いている場合には、G3DRL ライブラリの関数を用いて再描画に先立って、これらの視点移動に伴い変化する座標値等を表示部に設定する。これにより更新された視点座標・注視点座標が動的に数値として確認することができる。

リスト７－１９：マウス・ドラッグによる視点移動

```
static CPoint savePoint;//マウス移動時更新

void CDrawFrm::OnMouseMove(UINT nFlags, CPoint point) {
 if((nFlags & 2) == 0)//外でRIGHT_BUTTON=2 が外されたまま戻ってきた場合
    if(nSpinFlag && GetViewType() == G3_CAM_PERS) nSpinFlag = 0;
 if(nSpinFlag && GetViewType() == G3_CAM_PERS) {
    CMainFrame* pMainFrame = (CMainFrame*)AfxGetMainWnd();
    ASSERT(pMainFrame->IsKindOf(RUNTIME_CLASS(CMainFrame)));
    CPoint move;

    CClientDC dc(this);//080801
    *m_HDC = dc.m_hDC;//080801
    wg3AssignDrawarea( (void*)&m_HDC );

    move = savePoint - point;
    savePoint = point;
    if(nShiftKeyFlag) {
       g3SetCameraHorizontalShift(0.001f * (double)move.x);
       g3SetCameraVerticalShift(0.001f * (double)-move.y);
    }else if(nCtrlKeyFlag) {
       if(move.y > 0)
          g3SetCameraZoom(1.0 + (K_ZOOM_UP_RATE - 1.0) * 0.1);
       else if(move.y < 0)
          g3SetCameraZoom(1.0 / ( 1.0 + (1.0 / (K_ZOOM_DOWN_RATE) - 1.0 ) * 0.1 ));
    }else {
       g3SetCameraHorizontalRound(0.1f * (double)move.x);
       g3SetCameraVerticalRound(0.1f * (double)move.y);
    }
    pMainFrame->m_editShit->SetEyeWinParam();
    RedrawWindow();
  }
  CWnd::OnMouseMove(nFlags, point);
}

void CDrawFrm::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags) {
  if(nSpinFlag && GetViewType() == G3_CAM_PERS) {
    if(nChar == 16)
       nShiftKeyFlag = TRUE;
```

```
    else if(nChar == 17)
      nCtrlKeyFlag = TRUE;
    else {
      nShiftKeyFlag = FALSE;
      nCtrlKeyFlag = FALSE;
    }
  }
  CWnd::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CDrawFrm::OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  nShiftKeyFlag = FALSE;
  nCtrlKeyFlag = FALSE;

  CWnd::OnKeyUp(nChar, nRepCnt, nFlags);
}
```

７－７．画像のファイル保存、動画保存、印刷

（１）表示画面のファイル保存

景観シミュレータ sim.exe のメニュー[ファイル]の下にある、[JPEG 形式で出力]メニューから起動する、OnMenuImageSave()関数において、画像の保存を行っている。ファイル選択ダイアログを開いて、ユーザーが保存ファイル名を指定し OK で終了した場合に、拡張子に従い、以下の処理を行っている。拡張子はデフォルトで「.jpg」であるが、ユーザーにより、保存ファイル名の拡張子に「.sgi」が指定された場合には SGI 形式での保存を行う。

①JPEG 形式の場合、画面から OpenGL 描画領域を取り出し、ピクセル情報を配列にコピーした上で、JPEG ライブラリ関数を用いて圧縮を行い、ユーザーが指定したファイルに保存する。

②SGI 形式の場合、画面から OpenGL 描画領域を配列に取り出し、アプリケーション・ライブラリ ISAVE.C の SaveImageData 関数を用いてファイルに保存している。

なお、この処理の中には、指定されたファイル名本体部の末尾が「_small」または「_s」が指定された場合に景観データベース見出し用の小さな画像を作成する処理を行っている。

（２）動画のファイル保存

景観シミュレータのメニュー[編集][視点設定][移動経路設定]により起動する、移動経路ダイアログ(CKeiroWnd)においては、ユーザーが設定した経路に従って、視点をユーザーが指定した刻みで移動するたびに、メイン画面の表示を更新することにより、メイン画面で動画として表示する機能を提供している。

この移動経路ダイアログで、設定した移動経路に従って移動（ウォークスルー）が実行できる状態がセットされている状態において、ダイアログ下部にあるスタートボタンの代わりに、メニューの[ファイル][動画保存]を選択して動画を保存する AVI ファイル名を指定すると、下記の処理が実行され、設定されている移動を実行すると同時に動画ファイルを作成する。

具体的には、以下の処理を行っている：

① CKeiroWnd::OnMyAviSave() において、まず保存するファイル名の入力をユーザーに求め、OK で終了すると、動画保存処理を開始する。

②この時、CMyEditAvi クラス(MyAvi.cpp)の変数 avi を作成し、視点移動の開始時点で、指定されたファイル名で avi.Create 関数を実行する。

③視点位置が変更されるステップ毎に、メイン画面の OpenGL 描画領域(CDrawFrm)の表示結果をビットマップとして取得し、これを avi.Append 関数を用いて動画ファイルに追加する。

④ 移動経路の終点に到達すると、avi.Close 関数を実行し終了する。

（３）画像のプリンタ出力

従来の方法(sim.exe Ver.2.07 まで)においては、画像保存は以下の方法に従っている。

① プリンタを、環境設定ファイルに定義された文字列から識別する。

② 表示画像を、スクリーンから取得し、プリンタに送出する。

これを、具体的には以下の処理プログラムで実現している。

メイン画面のメニュー構成において、カラー印刷は、ID_FILE_PRINT で、これが指定されると、以下の関数が起動される。

(ソース：Mainfrm.cpp)

void CMainFrame::mainColorPrintCB()

メイン画面の OpenGL 描画エリアを RECT として取得し、ColorPrintDIB 関数を起動する。

(ソース：B3bitmap.c)

int　ColorPrintDIB(RECT *rect)

引数として渡された画面の部分をコピーし、DIB 構造体を作成する。

これを引数として、PrintDIB 関数を起動する

成功なら TRUE、失敗なら FALSE を返す

(ソース：print.c)

WORD　PrintDIB(HDIB hDib, WORD fPrintOpt, WORD wXScale, WORD wYScale,
LPSTR szJobName)

汎用の開発用キットに含まれるソースコードに追記し、環境設定ファイルに指定されたプリンタ名称から印刷用のデバイス・コンテキストを作成し、ここに引数として渡された DIB 構造体を出力する。この関数は、

成功なら０、失敗ならエラーコードを返す

この処理プロセスは、メニューの[ファイル][画面印刷]の機能として Ver.2.09 に残してある。

一方、最近の各種アプリケーションでは、通常、メイン画面の[ファイル]メニューの下に、

プリンタの設定、印刷プレビュー、印刷の３のメニュー項目がある。これによりユーザーはアプリケーションの中からプリンタを選択・変更できる。これらの機能は、Document-View アーキテクチャの Cview の中に標準で含まれており、プログラマは、アプリケーションの中で定義する派生クラス（景観シミュレータ sim.exe の場合には、CSimView）の中で CView クラスにおいて予め用意されている OnPrint 関数をオーバーライドし、その中で、引数として渡された印刷用のデバイス・コンテキストを利用して、アプリケーション独自の印刷機能をプログラムする。

CSimView クラスは、Document-View アーキテクチャにおける CView の派生クラスとして定義されている。

CView クラスには、印刷に関して、以下の関数が提供されている。

リスト７－２０：印刷のための CView クラスのメンバ関数

OnPrint 印刷を実行する。 DoPrintPreview　印刷プレビューを実行する。 OnFilePrintPreview 印刷プレビューのコールバック OnBeginPrinting OnEndPrinting OnPreparePrinting

プリンタの選択は、CWinApp クラスで提供されているため、派生クラスである CSimAPp (sim.cpp)にオーバーライドされている。

上記の通り、「画面印刷」として残してある従来のカラー印刷機能で、環境設定ファイルにより指定されたプリンタが使用不能である場合には、CPrintDialog を直接開いて、利用可能な印刷機の選択をユーザーに要求するように修正した。

ユーザーが適切な印刷機を選択して OK で終了した場合には、これを環境変数(E3_COLOR_PRINTER)に設定した上で、再度印刷処理を行う（但し、ユーザーが環境編集ダイアログから保存操作を行わない限り、kdbms.set ファイルの修正は行わない)。

メニューで印刷機の選択が行われた場合には、CWinApp::OnFilePrintSetup()が標準で起動される(appprnt.cpp)。実際の印刷の実行は、CSimView::OnPrint 関数の中で実行している。従来のカラー印刷では、画面と同じ解像度の画像をプリンタに出力するが、メニューの印刷が選択された場合には、関数

void CSimView::OnPrint(CDC* pDC, CPrintInfo* pInfo)

により印刷機が提供する高い解像度での印刷を実行する。この関数には、引数としてプリンタのデバイス・コンテキストが渡される。

この関数の中では、印刷用のビットマップを作成して、この上に OpenGL による描画を行い、プリンタに出力する。この時、まず通常の方法で、プリンタの解像度のビットマップを作成する。

リスト７－２１：印刷用のビットマップの生成１

```
CBitmap mBM;
CDC mDC;
```

```
double BX = (double)Prect.right/(double)Srect.right;
double BY = (double)Prect.bottom/(double)Srect.bottom;
if(BX<BY){
        Prect.bottom = (LONG)((double)Srect.bottom * BX);
}else{
        Prect.right = (LONG)((double)Srect.right * BY);
}
hasil = mBM.CreateCompatibleBitmap(pDC,Prect.Width(),Prect.Height());
        if(hasil) mDC.SelectObject(mBM);
```

しかし、上記の mBM.CreateCompatibleBitmap 関数は、利用可能なメモリの限界よりも手前で、メモリ不足エラーを返す場合がある。そこで、エラーとなった場合には、以下のより原始的な方法でリトライしている。

リスト７－２２：印刷用のビットマップの生成２

```
BITMAPINFO bmpinfo;
HBITMAP hNewBmp;
HGDIOBJ hG;
ZeroMemory( &bmpinfo, sizeof(bmpinfo));
bmpinfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
bmpinfo.bmiHeader.biWidth = Prect.right;
bmpinfo.bmiHeader.biHeight = Prect.bottom;
bmpinfo.bmiHeader.biPlanes = 1;
bmpinfo.bmiHeader.biBitCount = 32;
bmpinfo.bmiHeader.biCompression = BI_RGB;
hNewBmp=CreateDIBSection(NULL,&bmpinfo,DIB_RGB_COLORS,NULL,NULL,0);
hG = SelectObject( mDC.m_hDC, hNewBmp);
hasil = (int) hG;
```

この方法により、メモリの限界まで解像度を上げて、印刷を行うことができる。

この二番目の方法でも、メモリ不足エラーを起こす場合には、画面の解像度で印刷を行う。

リスト７－２３：印刷用のビットマップの生成３

```
Prect = Srect;
hasil = mBM.CreateCompatibleBitmap(pDC,Prect.Width(),Prect.Height());
mDC.SelectObject(mBM);
```

なお、CSimView::OnPrint 関数は、印刷プレビューと、印刷の両方から共用されており、印刷プレビューの場合には、

CView::DoPrintPreview→・・・→CView::OnPaint →CView::OnDraw

→CSimView::Onprint

の経路で、また印刷の場合には、

CView::OnFilePrint→CSimView::OnPrint

の経路で呼び出される。