１３．OS と開発環境の更新へ対応

１３－１．概要

第 1 章ヒストリーでも触れたように、開発着手後１５年以上にわたって改良・メンテナンスを行ってきた本システムは、数度にわたるＯＳの更新及び開発環境の更新を経験してきた。OS を更新することにより、従来問題なく動作していた機能に障害が生じることは、現状維持を目的とする限り、大きなオーバーヘッドとなる。何も新しい機能を付け加えようとしないにもかかわらず、新たな OS の上で正常に動作させるために、多くのマンパワーを投入しなければならない。とりわけ、グラフィックス処理の規約が変更されたような場合には、同じ機能を維持するために、修正が必要となったことがある（１３－３）。

しかしながら、長期的に見ると、OS や開発環境の更新に伴い、潜在バグが具体的障害化する場合がある。Windows98 から ME に更新した際には、多くの障害が生じた。しかしながら、原因を特定すると、結果的に潜在バグが顕在化したことによる障害がほとんどであった。従って、長期的に見ると、OS の変更への対応は、無駄な努力ではなく、より信頼性の高いシステムを実現し、かつ将来のより高度な機能開発や応用への有効な布石となっていると考えている。

Window98 までは、解放済みのメモリブロックや、スタック上に構築されるオート変数へのポインタを使い続けても、そのメモリブロックの内容が書き換えられるまでは一見正常な動作を続けるような場合があり、物理的なメモリが別の目的に割り当てられ、内容が書き換えられてから、初めて異常な現象が生じるようなケースがあった。デバッグに際しては、これを発見するために、メモリブロックの解放やデータの除却に際して、明らかに異常なデータに書き換えてから解放する方法や、デバッグ用実行形式の中で、メモリブロックのリスト管理を独自に行う方法により、終了時点で検出されたリークの発生源を特定するための工夫を行った。最近の開発環境では、このような機能が次第に充実してきている。その後 OS や開発環境の進化に伴い、OS が提供するメモリ･ブロックの解放処理等が変更され、無効を示すデータが解放時に OS により書き込まれているように見える。また、オート変数やメモリブロックの、取得・定義時点でのメモリの物的初期状態も OS により変化してきたため、例えば変数を初期化しないまま使用しても、偶々メモリの内容が初期状態で 0x00 であったために一見正常に動作しているような場合に、OS の変化に伴い顕在化するような場合があった。

Windows XP から Vista への更新に際しても、いくつかの修正が必要となったが、グラフィックス関連では本質的な修正が 2 個所にとどまったことは、一種の完成度を示していると考える。

１３－２．VS2005 への対応

開発環境は、Windows 系に関しては Visual Studio 2.0→4.0→6.0→8.0(VS2005)と更新を行ってきた。

とりわけ、8.0 への更新に際しては、従来の C 言語で開発されていたライブラリ関数のソースコードの多くにおいて、多数の警告が発せられた。最も修正の手間を要した部分は、最初に UNIX マシンで開発されていたソースコードにおけるコメント等において日本語表記が EUC コードで行われていた部分の扱いである。

更に、ANSI-C の基本的な関数であった、fprintf 等の基本的な関数の内、バッファ・オーバーフローを起こす可能性のある関数に関して、使用を避けるような警告が多く生じた。

これらは、セキュリティを高めるための対応を要求するものである。

この他、各実行形式のコンパイル、リンクおよびデバッグ等の動作を指示するために、より多くの詳細な設定を行う必要があった。

現在までに、新たな開発環境のバグに起因する障害や、解決不能な障害は生じていない。しかし、プロパティの設定などを試行錯誤した結果解決した障害であっても、なぜ解決したのかが必ずしも明快ではないような障害は存在している。

１３－３．Windows Vista への対応

ＯＳの更新に伴い、潜在バグが具体的障害化する場合がある。Windows XP から Vista への移行に際しては、OpenGL の画像表示に関する部分が問題となった。

Windows における各ウィンドウは階層的に構成されている。子ウィンドウは、親ウィンドウの上に表示され、同じ親に帰属するウィンドウ相互は、アクティブなものがそれ以外のものの上に表示される。そして最上位にデスクトップ・ウィンドウが位置する。ひとつのダイアログの中に配置された OpenGL 画面やボタンやエディットボックスなどのコントロールも、同じ階層の子ウィンドウである。

OpenGL の画面は、初期化の手続きによって、ウィンドウに関連づけられる。その際に、図形描画の出力機能を担うデバイス・コンテキストが使用される。デバイス・コンテキストは、各ウィンドウに固有のものと、全てのウィンドウに共通に使用されるものがある。描画に際して、共通に使用するものを描画が必要な時点で取得し、描画が終了すると解放するようにプログラムを構成した方が、資源節約になる。更に共通のデバイス・コンテキストは、最大５まで、という制約があるため、描画終了後に解放しないまま停止するようなスレッドが存在すると、別の画面における描画が失敗する場合がある。

さらに、OpenGL をデバイス・コンテキストと結びつけるために、HGLRC というもう一つのコンテキストが必要とされる。こちらには数の上限がない。さらに、個々の OpenGL 画面の描画条件設定には多くのパラメータが必要であり、複数の画面のそれぞれに関して、一度設定したパラメータは、アプリケーションが終了するまで記憶しておく必要がある。そこで、景観シミュレータにおいては、HGLRC を OpenGL 画面作成時から除却時まで保持するとともに、各 OpenGL 画面に関して、ウィンドウ作成時に、初期化と共に DA 構造体を生成し、パラメータを保存し、ウィンドウ除却時にこれを解放している。個々のウィンドウ描画時には、DA 構造体のパラメータを用いて描画を行っている。

Windows　VISTA においては、初期化処理の終了時点で、OpenGL とデバイス・コンテ

キストの関連づけの解消をより厳密に実行しておかないと、再描画に際して、意図するウィンドウ以外の画面に描画が行われてしまう、という障害が発生した。多くの場合、最上位にあるデスクトップ・ウィンドウに表示が行われてしまう。

これに対して、最も簡単な解決方法は、従来の方法を保持しつつ、初期化の終了時点で、デバイス・コンテキストの解放をより厳密化することである。具体的には、以下のように１行を挿入する。

リスト１３－１：OpenGL 初期化部分の修正

```
pCDC = GetDC();
m_HDC = new HDC;
*m_HDC = pCDC->m_hDC;
set_pixel( *m_HDC);
m_HGLRC = wglCreateContext(*m_HDC);
wg3EntryDrawarea(((void*)&m_HDC(void*)&m_HGLRC);
wg3AssignDrawarea((void*)&m_HDC);
g3InitializeWindow();
(各種初期化)
wg3AssignDrawarea(NULL);
pCDC->ReleaseOutputDC();  //●この１行を追加
ReleaseDC(pCDC);
```

推察するに、Windows　Vista の内部処理においては、OpenGL のコンテキストと関連づけられたまま、デバイス・コンテキストが解放され共用のプールに返されると、次に描画しようとした時に、先刻解放したデバイス・コンテキストを新たに取得している別のウィンドウに描画されてしまう、という事のようである。なお、ReleaseOutputDC を、ウィンドウのメンバに関連づけられたデバイス・コンテキスト（例えば、CClientDC dc(this); 等の命令で取得したもの）に適用すると、エラーとなるため、OpenGL 画面の初期化には GetDC()等の命令で共通のプールから取得した一時的なものを使用する。

もうひとつの方法は、描画をメモリ上のデバイスに行うことにより、トリプル・バッファリングを実現する方法であり、メモリ上に最終的な表示画面と互換のデバイス・コンテキストを作成してこれに描画を行う。OpenGL のダブル・バッファが、描画途上の画面をユーザーに見せずに、描画終了後に瞬間的に切り替えることを目的としているのに対して、メモリ上のデバイスの使用は、描画コンテキストを、他のウィンドウ等との重なり具合といった外部環境から切り離した環境で行うことにある。最終的な表示は、メモリから画面へのデータ転送のみであり、三次元のレンダリングより一般に処理は速い。このメモリ上のコンテキストは、ウィンドウの作成から除却までの間保持していて構わない。

この方法においては、表示内容の変化（地物の変化、視点や光源などの環境の変化）が生じた際の OpenGL の再描画が少し遅くなる一方、単に表示画面の上を覆っていた別のウ

ィンドウが移動した際に無効領域を再描画する処理は、単にメモリのコピーだけの処理となり、高速化する。従って、視点移動アニメーションなど、最上位のウィンドウに高速で再描画を繰り返すような処理には前者が適しており、様々な画面を併用しながらモデリングを行うような場合には、後者の処理が適している。

リスト１３－２：メモリ上のデバイスを用いる方法

```
①OnCreate における処理
  CPaintDC dc(this);
  mDC.CreateCompatibleDC(&dc);
  mBM.CreateCompatibleBitmap(&dc,1,1); //とりあえず１ドットのビットマップ
  mDC.SelectObject(mBM);
  m_sHDC = new HDC;
  *m_sHDC = mDC.m_hDC;//メモリ上のデバイスを用いて初期化
  （この m_sHDC は、OnSize, OnPaint で再初期化せずに、維持する）
  set_pixel(*m_sHDC);
  m_sHGLRC = wglCreateContext(*m_sHDC);
  status = wg3EntryDrawarea((void*)&m_sHDC,(void*)&m_sHGLRC);
  if(!status) エラー処理へ
  status = wg3AssignDrawarea((void*)&m_sHDC);
  if(!status) エラー処理へ
  status = g3InitializeWindow();
  if(!status) エラー処理へ
        各種初期化処理

②OnSize(UINT nType, int cx, int cy) における処理
  CClientDC dc(this);
  *m_sHDC = dc.m_hDC;
  wg3AssignDrawarea((void*)&m_sHDC);
  g3Resize(cx,cy);
  mBM.DeleteObject();
  mBM.CreateCompatibleBitmap(&clientDC,cx,cy);
  mDC.SelectObject(mBM);
  RedrawWindow();

③OnPaint における処理
  RECT Rect;
  int rc, x, y;
```

```
   GetUpdateRect(&Rect,FALSE);
   CPaintDC dc(this);
(モデルや視点が変更されていて、画面全体を更新する必要がある場合のみ)
   g3GetSize(&x,&y);
   wg3RedrawWindow(&m_sHDC);
   rc = dc.BitBlt(0,0,x,y,&mDC,0,0,SRCCOPY);
(被さっているウィンドウが静止し、無効領域だけを更新する場合)
   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);
④OnDestroy における処理
   wg3AssignDrawarea((void*)&m_sHDC);
   g3ReleaseGroundPixel()
   wg3DeleteDrawarea(&m_sHDC);
   wglDeleteContext(m_sHGLRC)
   mBM.DeleteObject();
```

初期のグラフィクス・ワークステーションは、言わば専用のグラフィック・カードが巨大化したような構成であり、直接ハードに GL のコマンドを送ることにより高速描画を実現していた。これに対し、最近の PC では、マザーボードに OpenGL を含むグラフィクス機能が搭載され、転送速度が向上される傾向にある。このため、メモリ上のデバイスに対する間接描画も、直接出力に遜色のない描画速度が実現されているように感じられる。

景観データベースの検索結果を表示するための childfrm.cpp においては、多数のウィンドウを処理するために多大のリソースを消費することを避け、初期化、ペイント、終了処理で、より深い処理を行っている。即ち、HGLRC を保持するのではなく、表示用の画像をイメージとして作成した後は、画像以外のデータを解放している。

リスト１３－３：HGLRC を毎回解放する方法

```
①OnCreate の処理
     CPaintDC dc(this);
     mDC.CreateCompatibleDC(&dc);
     GLFLAG = 1; //OnPaint で GL 描画を要求
```

②OnPaint の処理

```
CPaintDC dc(this);
if(!GLFLAG){ //画像が既にある場合
  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;
}else{ //OpenGL 描画が必要な場合
  (前処理)
  m_hDC = new HDC;
  mBM.DeleteObject();
  mBM.CreateCompatibleBitmap(&dc,m_size.cx,m_size.cy);
  mDC.SelectObject(mBM);
  *m_hDC = mDC.m_hDC;
  zet_pixel(*m_hDC);
  m_hGLRC = wglCreateContext(*m_hDC);
  wg3EnryDrawarea((void*)&m_hDC, (void*)&m_hGLRC);
  wg3AssignDrawrea((void*)&m_hDC);
  g3InitializeWindow();
  (画像の描画)
  wg3Redraw((void*)&m_hDC);

  wg3AssignDrawarea(NULL);
  wg3DeleteDrawarea(&m_hDC);
  m_hGLRC = wglGetCurrentContext();
  dc.BitBlt(0,0,m_size.cx,m_size.cy,&mDC,0,0,SRCCOPY);
  GLFLAG = 0;
  delete m_hDC;
}
```

③デストラクタの処理

```
mBM.DeleteObject();
```