Section 1

Chapter1　開発環境の準備

多くの同種解説書籍で開発環境の説明やゲーム開発パラダイムの歴史等に関する記述がありますが、全くの初心者を除く多くの読者にとって冗長なものであると筆者は感じていました。ゲーム開発（規模の大小に関わらず）の経験があれば環境が既に準備されている訳でありこの章を読む必要はありません。もっともそのような読者には本章が読み飛ばされ、この文章も読まれていないかもしれませんが。本章に限らずこの本全体について当てはまることですが、冗長な事柄は排除していきます。

しかしながら、本書はゲーム開発をゼロから始めようとしている読者も対象としているためこの章を設けました。

準備するものをソフトウェアに絞って解説すると次の通りです。なお、ハードウェアの解説に関しては、パソコンの一般家庭への普及率が６０％に届くような現状で必要だろうか、また、ハードウェアの準備なしに本書を読む読者がはたしているのだろうかと疑問に思うところであり解説はしません。

OS

Windows　98,Me,2000,XP の何れかのウィンドウズ上で開発します。(ＮＴ・Windows ＣＥは SDK の対象外。また、DirectX9 から Windows95 が対象外となった。)

コンパイラー

コンパイラーはマイクロソフト Microsoft、ボーランド Borland、ワトコム Watcom が有名ですが、ウィンドウズ環境でのアプリケーション、特に DirectX を利用したゲームアプリケーションの開発となると選択肢は一つしかありません、つまり選択の余地が無いということであり、それはマイクロソフトの VisualC++ です。筆者は別に同社の回し者ではありませんし、特定の会社のコンパイラーを「これしかありません」と言って薦めるのは（薦めなければならないことは）少々寂しいですが、現実がそうなので仕方がありません。このように言い切るのは理由がありますが、それはあえて割愛いたします。今後も紙面で「言い切る」部分があろうかと思います、いささか乱暴であり、また、反論の可能性があるのは分っています、しかし、冗長あるいは歯切れの悪い説明は極力無くしてゆきたいということが筆者のこの本を書く上でのコンセプトとしているものであり、言い切ることが出来ない事柄でもあえて「言い切り」ます。とにかく、私は最短のプロセスでいちはやくゲーム開発ができるようになっていただきたいと願っているだけであり、余計なところで時間を費やしてもらいたくないと思っています。私を信じてください。

さて、VisualC++ の最新バージョンは 2003 年 7 月現在で 7.1 です。7.0 バージョンからの VisualC++ は ".NET（ドットネット）" とも呼ばれています。ドットネットバージョンからは、バージョンを番号ではなくリリース年度で表すようになり、あまりバージョン 7.0 とか 7.1 とは呼ばないようになりました。バージョン 7.0 はバージョン 2002、また、バージョン 7.1 はバージョン 2003 と呼ばれることのほうが多いようです。バージョン 4 から 6 まではインターフェイスはさほど変わりはありませんがバージョン 7.0 以降 (.NET 以降) はインターフェイスが大きく変わっています。

筆者の個人的な意見としては .NET に慣れるともうそれ以前のバージョンのインターフェイスは物足りなく感じてしまいます。より新しいバージョンということで使い勝手がいいのは当たり前かもしれません。

また、バージョン 6 以前の VisualC++ は、ウィンドウズ 9x 系上で使用できますが、.NET バージョンは OS が Windows2000 か WindowsXP である必要があります。Windows9x 系上では起動できません。.NET を新規購入するときは使っているＯＳを確認してください。（当然これから新規購入するという方は最新バージョンである .NET を購入することと思いますので、OS を確認してください）。

DirectX SDK

SDK とは Software Development Kit の略であり開発に必要なソフトウェアの一連のセットという意味です。DirectXSDK には DirectX 実行時に必要である DirectX ランタイムとプログラミング時に必要なヘッダーファイル、ライブラリファイル、サンプルプログラム、ヘルプファイルが含まれています。DirectX ゲームをプログラミングする際にはこの「プログラミング時に必要な」ファイル等が必要になります。マイクロソフトのウェブサイトから無料でダウンロードすることが出来ますのでダウンロードしてください。

http://www.microsoft.com/japan/msdn/directx/downloads.asp

なお、ダウンロードサイトには DirectX の「ランタイムバージョン」と「SDK フルバージョン」を分けて置いてありことが多いですが、ランタイムは DirectX そのものであり、

Windows98 以降はその時点でのバージョンの DirectX ランタイムは最初から OS に入っているものです、プログラミングに必要なヘッダーファイル等のファイルは含まれていませんので、開発を行うにはフルバージョンの方をダウンロードする必要があります。

ダウンロードした後 DirectXSDK をパソコンにインストールします。インストール時に Retail モード（製品モード）か Debug モード（開発モード）のどちらでインストールするかをインストーラーが聞いてきますので、Debug モードでインストールしてください。

そして、DirectXSDK を機能させるコンパイラーラの設定をしなければなりませんので , これから説明します。

DirectX を利用したソースコードをビルドするためには、DirectX SDK のヘッダーファイルとライブラリファイルの場所（パス）およびライブラリファイルのファイル名をコンパイラーに登録する方法が最も簡単です。

DirectX をインストールした際にパスをデフォルトから変更していなければインストールルートパスは C:\DXSDK となっているはずです。そしてヘッダーファイルは C:\DXSDK\Include、ライブラリファイルは C:\DXSDK\Lib ディレクトリにそれぞれあることになります。インストールした際にインストールパスを変更した場合は "C:\DXSDK" の部分が変更したパスになります。例えば D ドライブに DXSDK9 というフォルダを作成しそこにインストールした場合にパスはそれぞれ D:\DXSDK9（ルートパス）D:\DXSDK9\Include（ヘッダーファイルのパス）D:\DXSDK9\Lib(ﾗｲﾌﾞﾗﾘﾌｧｲﾙのﾊﾟｽ) となるわけです。

VisualC++ のバージョンによりパスの登録仕様が異なりますので、6.0 バージョンと .NET バージョンに分けて説明します。

コンパイラーの設定

【VisualC++6.0 の場合】

メインメニュー「ツール」→「オプション」でオプションダイアログを表示します。
オプションダイアログの「ディレクトリ」タブをクリックした状態が図 1-1 です。

〈ヘッダーファイルパス（インクルードファイルパス）の登録〉

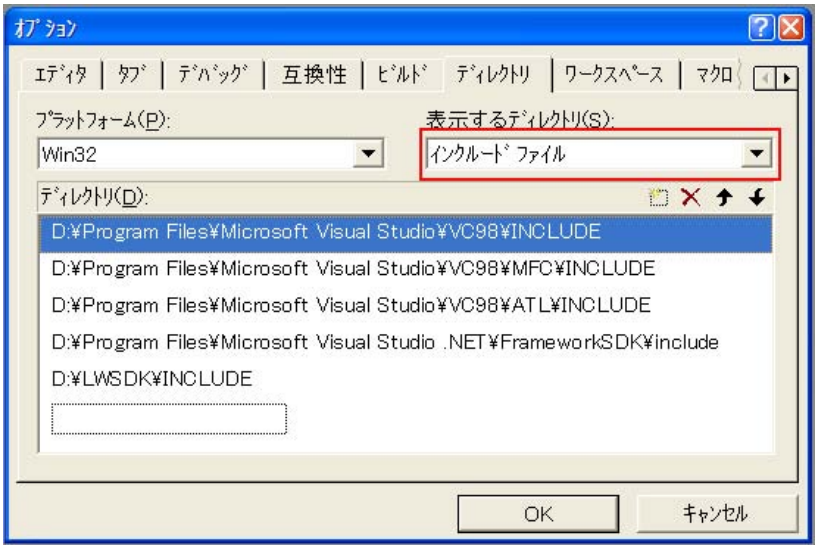


図 1 － 1

「表示するディレクトリ（S)」コンボボックス内を「インクルードファイル」にします。
これでヘッダーファイルパス（インクルードファイルパス）を登録する準備ができましたので、次に実際にヘッダーファイルのパス（デフォルトで C:\DXSDK\Include）を入力します。（図 1-2 参照）

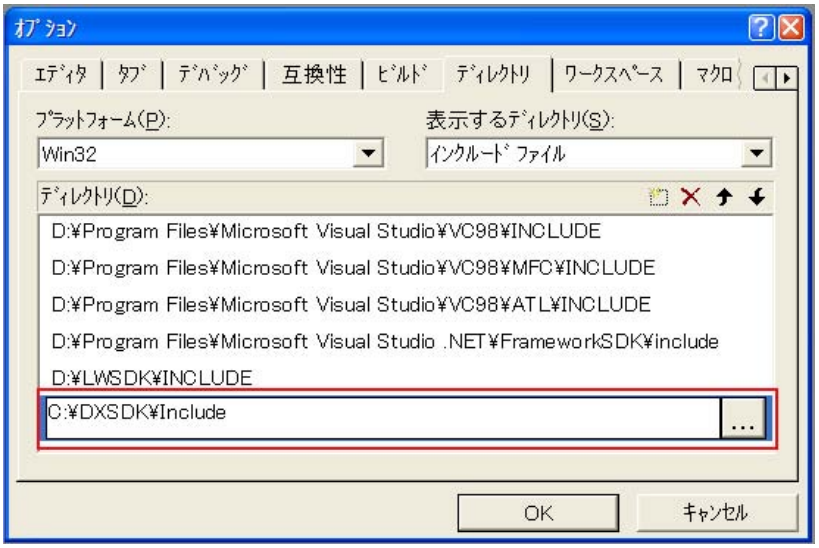


図 1-2
そして、（必ずしも必要ではありませんが）入力したパスを一番上に移動します。（図 1-3 参照）

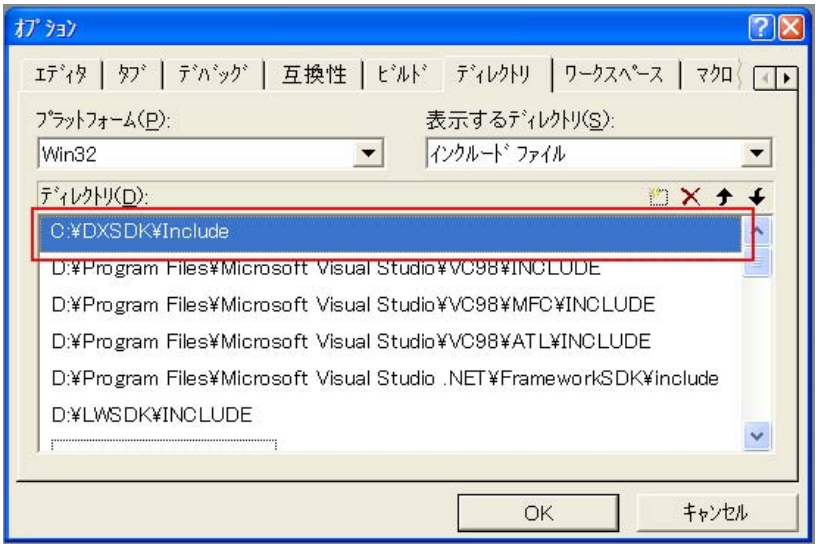


図 1-3

〈ライブラリファイルパスの登録〉

表示するディレクトリ（S)」コンボボックス内を「ライブラリファイル」に切り替えます。
これでライブラリファイルパスを登録する準備ができましたので、ヘッダーファイルパスと同様の手順で図 1-4 のようにします。

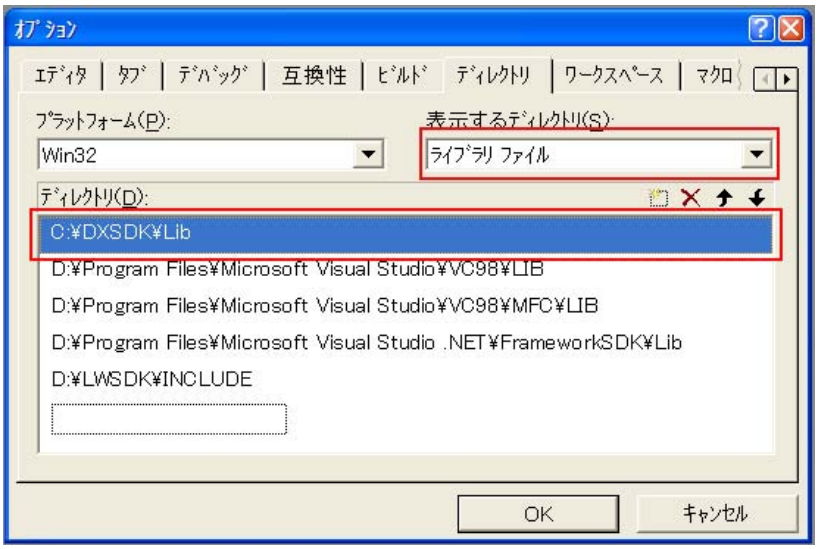


図 1-4
ここまで終えた段階で OK を押せば、DirectX ソースコードをコンパイルできるようになります。

【VisualC++.NET の場合】

メインメニュー「ツール」→「オプション」でオプションダイアログを表示します。
オプションダイアログの左にあるフォルダリストの「プロジェクト」下を「VC ++ディレクトリ」にします。

〈ヘッダーファイルパス（インクルードファイルパス）の登録〉

右上「ディレクトリを表示するプロジェクト（S)」コンボボックス内を「インクルードファイル」にした状態が図 1-5 です。

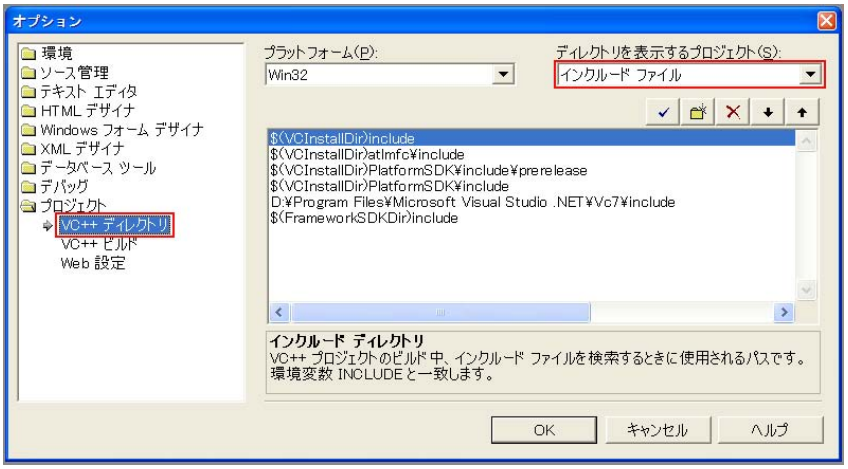


図 1-5

これでヘッダーファイルパス（インクルードファイルパス）を登録する準備ができましたので、次に実際にヘッダーファイルのパスを入力します。
右上の「新しい行」ボタンを押して新しいパスを入力するエディットボックスを表示して、インクルードファイルパスを入力します。（図 1-6 参照）

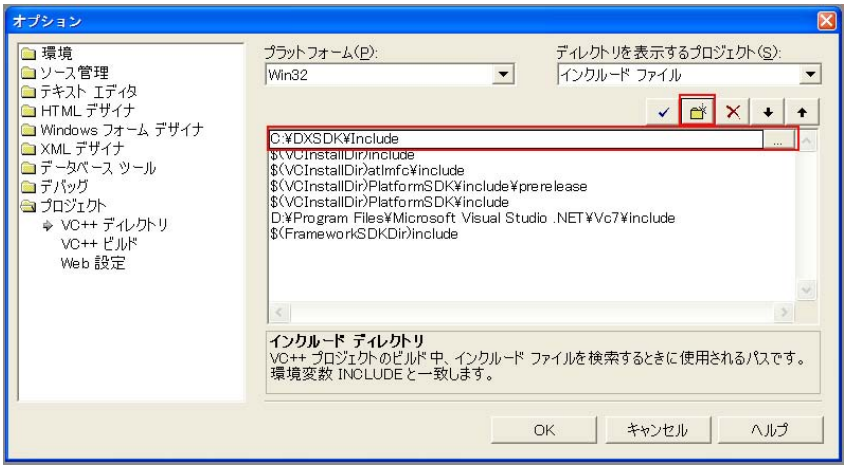


図 1-6

〈ライブラリファイルパスの登録〉

右上「ディレクトリを表示するプロジェクト（S)」コンボボックス内を「ライブラリファイル」に切り替えます。
〈ライブラリファイル名の登録〉

これでライブラリファイルパスを登録する準備ができましたので、ヘッダーファイルパスと同様の手順で図 1- 7 のようにします。

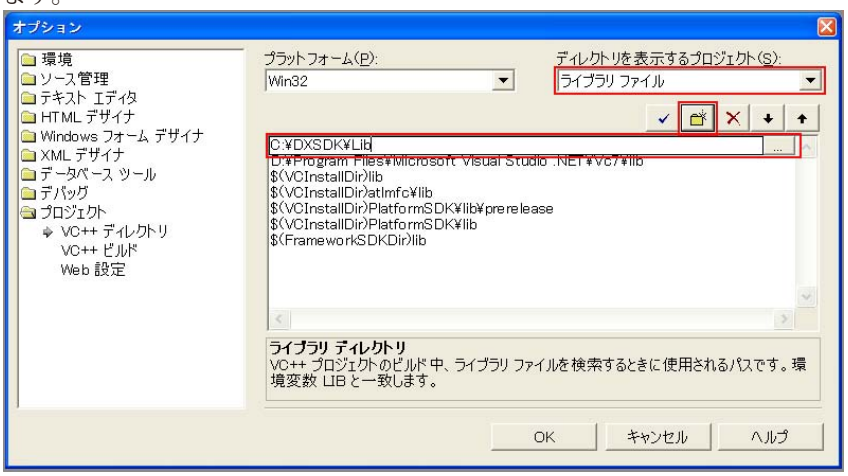


図 1-7
ここまで終えた段階でＯＫを押せば、DirectX ソースコードをコンパイルできるようになります。

〈ライブラリファイル名の登録〉

この設定は、開発するアプリケーションのプロジェクト毎に行う必要があります。
この設定をしないとコンパイルはできてもリンクが失敗しリンカエラーが出ます。（つまり、ビルドが完了しません）
なお、本書サンプルプロジェクトはプロジェクトファイル内で既に設定済みなのでサンプルプロジェクトのビルドにこのプロセスは必要ありません。
コンパイラーに登録が必要なライブラリの数およびライブラリファイル名は使用しようとする DirectX コンポーネントの種類によって異なります。
例えば、第２章のプロジェクトでは数ある DirectX コンポーネント (Direct3D,DirectSound,DirectMusic,DirectPlay…etc) の中の Direct3D コンポーネントしか使用していませんので、ライブラリファイルは d3d9.lib と d3dx9dt.lib があれば足りますし、プロジェクト設定を見てもらえば、その２つしか設定していないことが分かると思います。
しかし、最初のうちはどのコンポーネントにどのライブラリファイルが対応しているのか分からないと思いますし（SDK ヘルプファイルを見れば分かりますが）、分かっていてもプロジェクトごとにいちいち異なる設定をするというのも面倒だと思います。
そこで、最も簡単な方法として、各 DirectX コンポーネントの主要ライブラリファイル名の全てをテキストファイルに保存しておき、そのファイル名をコピーアンドペーストするという方法があります。実際、コンパイラーにとって必要なのはアプリケーションで使用している DirectX コンポーネントに対応したライブラリだけであり、不要なライブラリまで登録することになりますが実害はありません。
d3dxof.lib d3dx9dt.lib ddraw.lib d3d9.lib dinput.lib dplayx.lib dsound.lib dsetup.lib dxguid.lib comctl32.lib winmm.lib　これらのライブラリを追加すれば当面は十分でしょう、特殊なコンポーネントを使用する際にこれら以外のインポートライブラリが必要な場合があればヘルプファイルの当該コンポーネントのページの最後に必要なインポートライブラリが記述されていますので参照して適宜追加してください。
設定方法は VisualC++6.0 の場合は図 1-8 と図 1-9、VisualC++.NET の場合は図 1-10 と図 1-11 のようになります。

【VisualC++6.0】

設定するプロジェクトを VisualC++ に読み込んだ状態でメインメニュー「プロジェクト」→「設定」でプロジェクトの設定ダイアログを表示します。
ダイアログの「リンク」タブをクリックすると図 1-8 のようになります。

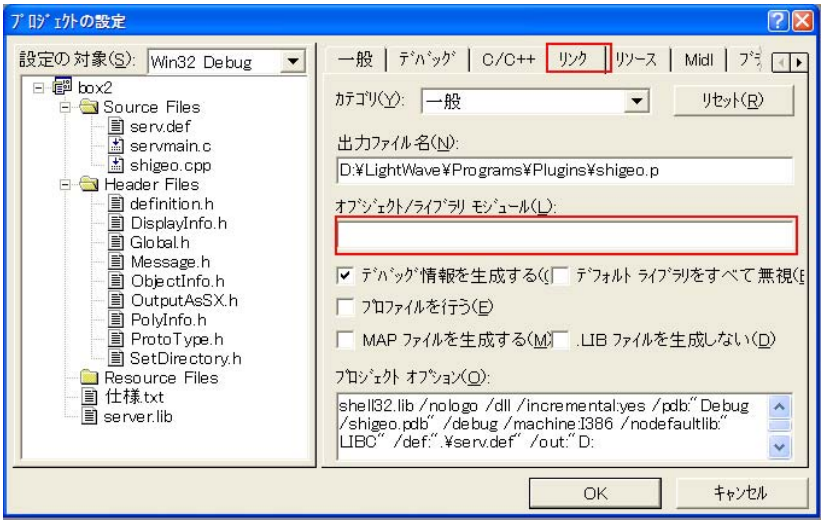


図 1-8

「オブジェクト / ライブラリモジュール（L）」エディットボックス内にライブラリファイル名を貼り付けてください。（図 1-9 参照）

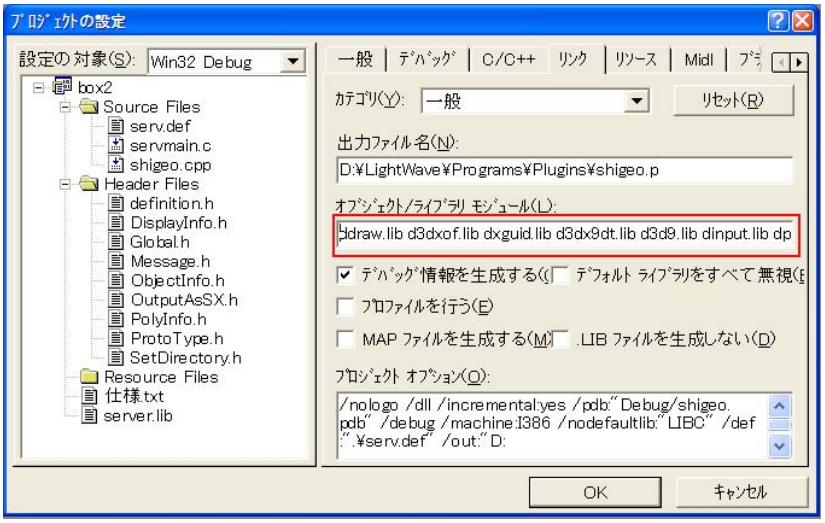


図 1-9

【VisualC++.NET】

設定するプロジェクトを VisualC++ に読み込んだ状態で、メインメニュー「プロジェクト」→「プロパティ」で当該プロジェクトのプロパティダイアログを表示します。
(「プロパティ」がゴースト表示の場合（灰色で選択できない場合）は、ソリューションウィンドウ内のプロジェクト名をクリックして選択状態にするとゴースト表示
が解除されて「プロパティ」を選択できるようになります)
ダイアログの左側のリスト内で「リンカ」→「入力」と選択すると図 1-11 のようになります。

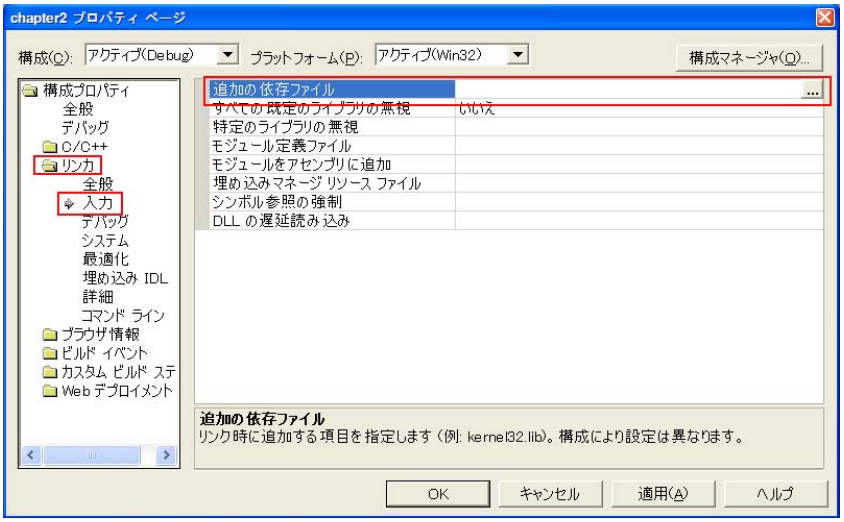


図 1-10
この状態で「追加の依存ファイル」エディットボックス内にライブラリファイル名を貼り付けてください。(図 1-11 参照)

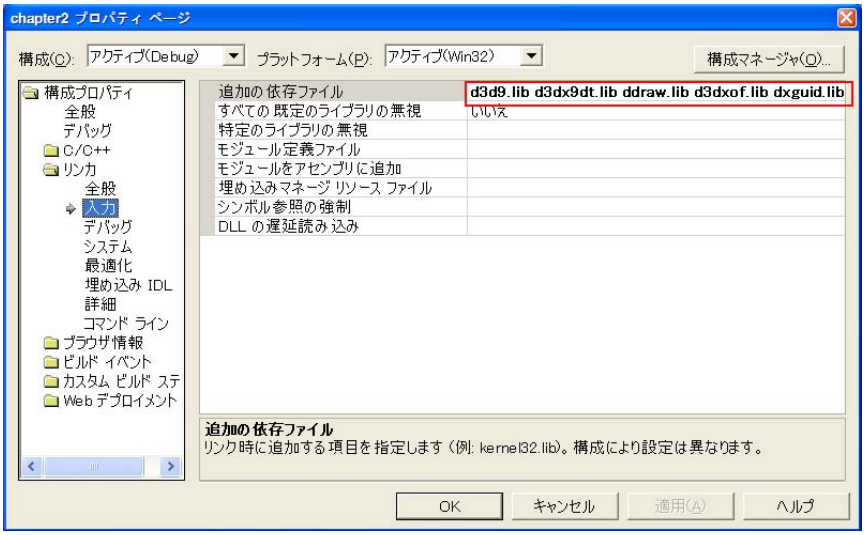


図 1-11

ここまでで全て設定完了です。DIRECTX のソースコードをビルドできるようになりました。

画像（グラフィック）作成ソフト

ゲームを作るにあたって画像作成は大きなウェイトを占めます。どうやって画像を作成するのか？ドット絵が主流だったのは２０年から１５年前の話で、当時は専用ソフトが無くても画像作成できましたし、画像作成用のソフトを自作することも比較的簡単に可能でした。しかし、現在はドロー系ソフト（Adobe Illustrator 等）でイラストを書いたり、３ＤＣＧソフト（Maya, SoftImage,3DStudioMAX,LightWave 等）で３Ｄモデルをレンダリングし、必要であればペイント系ソフト（Adobe Photoshop 等）で仕上げをするというのが一般的かと思われ、筆者も実際この方法で画像作成しています。特にムービーを作成する場合や３Ｄゲームを作成する場合において３ＤＣＧソフトが必要なのは言うまでもありません。
これらのソフト無しに画像（グラフィック）を作成することは、（理論上）全くの不可能ではないにしても実務上は不可能・非現実と言えます。

効果音及びバックグラウンドミュージック作成ソフト

まず「音」の作成について効果音 SoundEffect(SE) とバックグラウンドミュージック BackGroundMusic(BGM) に分けて説明します。
効果音（ＳＥ）を作成・編集するにはデファクトスタンダードである SoundForge をお薦めします。他にシェアウェア形式でありながら注目すべきソフトである CoolEdit もあります。CoolEdit はインターフェイスが SDI ☒で筆者としてはワークフロー上多少ネックではありましたが十分な機能があり外観もクールで嫌いではありません。なお、CoolEdit の最上位バージョンである CoolEditPro は今後（2003 年 5 月以降）に adobe 社から Audition という名前でリリースされるようです。余談にはなりますが、つい先日ある 3DCG ソフトの製品イベントの中で Adobe 社の新製品紹介もあったのですが、そのとき Audition のパンフレットを見たとき「あれっ？これって、いつも見慣れている CoolEdit にそっくり」と一瞬、パンフレットの製品名を２，３回見直しました。似ているのは当たり前で、CoolEdit がベースになっているものでした。
本格的に効果音を作るなら上記ソフトにより、自分で録音した音を編集したり、波形生成機能により文字通り音をゼロから生成する必要がありますが、ロイヤリティーフリーの素材を使うのも一つの方法です。
バックグラウンドミュージック (BGM) の作成はシーケンスソフトいわゆるシーケンサーと呼ばれているソフトで行います。DirectX SDK に付属している DirectMusicProducer というソフトがあります、このソフトは DirectMusic 形式のファイルコンバーターであると同時にシーケンサーでもあります。このソフトのみで Midi 等のミュージックファイルを作成することは可能ですが、画像作成と同様やはり別の専用シーケンスソフト上でミュージックファイルを作成するほうが作業効率的にもいいでしょう。シーケンサーは数多くのメーカーのものがあり有名なソフトは機能的には（ゲームミュージックという観点からみて）さほど差はありません、となると大切なのはインターフェイス、操作しやすいかどうかということだと思いますが、これはまったくの個人的な意見ですが国産のシーケンサーは独自の操作仕様のものが多く操作に慣れるまで時間がかかりました。シーケンサーに限らずどのようなソフトでもそうでインターフェイスが一般的なウィンドウアプリケーションの操作に準拠しているソフトはなんとく感覚で操作できてしまうものです（デファクトスタンダードと呼ばれるソフトはこの「すぐに操作できる」ものが多いと筆者は考えています）、国産シーケンサーではすぐに作業できず、まずそのソフト固有（独自）の操作を覚えなければならかったのがけっこうストレスになりました。
また別の方法として音に関してはロイヤリティーフリーの素材をそのまま、あるいは加工して利用するという方法もあります。時間がなかったり音以外の部分に集中したい場合にはいいでしょう。素材も数社から数多く販売されています。音の収録数・クオリティーから考えるとハリウッド映画の SE にも多く使用されているサウンドアイディア社（アメリカ）の製品がダントツです。（その分価格もダントツですが…）
筆者は国産の素材集は基本的にホビーレベルで製品レベルの使用には考えていません。
なにか国産批判のような感じになってきましたが、事実を述べるとそうなってしまいます。筆者制作の市販ゲーム第 1 作目において SE には某国産メーカーの音素材を数多く使用したのですが（個人で開発していて上記サウンドアイディア社の素材を購入する資金がなかったのです）、ある事件（？）がありました（事件という大それたことではなく筆者が気付いただけのことですが）、一つの効果音が上記サウンドアイディア社の販売するシリーズの中に収録されている音とまったく同一だったのです！ロイヤリティーフリーとはいっても再販は禁止しているはずです。これには正直驚きました、（と同時に怒りを覚えました、なぜならもうその音はゲームの中に組み込んでいたからです）しかも、当該国産メーカーは国内ではけっして無名ではなくむしろ大々的に販売展開しています。技術力・開発スタッフ・知名度等から考えてもサウンドアイディア社が盗用したと考えるにはクエスチョンマークが３つくらい付きます。となれば…もちろんどちらが盗用したかまた天文学的な偶然性で同じ音を作ってしまったのかは筆者に分かるはずもなく、ひたすら虚しい気分になったのを覚えています。その国産メーカーにメールで連絡したところ類似音ということでしたが、類似という言葉の許容範囲をはるかに超えた同一性でした。
冗長な説明の排除がコンセプトの一つである本書籍でいささか話がわき道に入ってしまいました。ではここで開発環境の準備の説明を終えて、次の章から実際のゲーム開発へ入りましょう。

Chapter2 画面に何かを表示する

このセクションのサンプル全般について言えますが、既に開発経験がある読者にとっては、「そこまで説明する必要があるのか」と思うほど基本的なことを全て解説しています。不要と思われる冗長な解説は避けますが、基本的事項の細かい解説は初心者にとっては冗長どころかむしろ重要なことと考えます。
全くゲームプログラミングの経験がない方、またはゲームプログラミングの経験があっても DirectX によるプログラミング経験がない方（はじめ筆者は後者でした）が、最初に直面する問題が画面表示の方法だと思います。とにかく何かをスクリーンに表示できないことには話になりません。逆に言えば画面表示が出来ればある程度のゲーム（ゲームと呼べないようなテストプログラムでも）は作れることになります。理解を容易にするため最もシンプルに説明できる「スクリーンにスプライト※を表示する」ということから始めたいと思います。
このレベルであればまだソースコード全体を記載することが可能なのでまずはソースコードの全体を載せて、順次解説していきます。
なお、極力シンプルにするためコード体系はクラス形式ではなく単一モジュール形式※で書きました。章が進むにつれてこの書式では無理が出できますのでクラス書式になりますが現段階では理解が容易なこの書式を採用しました。ソースコードの全体なのでこれだけでそのままビルド・実行できます。（もちろん、コンパイラーの設定※が終了していることが前提です）

サンプルプロジェクト名：Chapter2

```
#include <windows.h>
#include <d3dx9.h>

LPDIRECT3D9 pD3d;
LPDIRECT3DDEVICE9 pDevice;
LPDIRECT3DTEXTURE9 pTexture;
LPD3DXSPRITE pSprite;

LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
HRESULT InitD3d(HWND);
VOID DrawSprite();

//
//INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
// アプリケーションのエントリー関数
INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
{
    HWND hWnd=NULL;
    MSG msg;
    // ウィンドウの初期化
    static char szAppName[] = "Chapter2" ;
   WNDCLASSEX  wndclass ;

   wndclass.cbSize         = sizeof (wndclass) ;
   wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
   wndclass.lpfnWndProc    = WndProc ;
   wndclass.cbClsExtra     = 0 ;
   wndclass.cbWndExtra     = 0 ;
   wndclass.hInstance      = hInst ;
   wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
   wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
   wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
   wndclass.lpszMenuName = NULL ;
   wndclass.lpszClassName = szAppName ;
   wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;

   RegisterClassEx (&wndclass) ;

   hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
                0,0,640,480,NULL,NULL,hInst,NULL) ;
  ShowWindow (hWnd,SW_SHOW) ;
  UpdateWindow (hWnd) ;
    // ダイレクト3D の初期化関数を呼ぶ
    if(FAILED(InitD3d(hWnd)))
    {
                return 0;
    }
    // メッセージループ
  ZeroMemory( &msg, sizeof(msg) );
  while( msg.message!=WM_QUIT )
```

```
51:     {
52:                     if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
53:       {
54:                               TranslateMessage( &msg );
55:           DispatchMessage( &msg );
56:       }
57:       else
58:                   {
59:                               DrawSprite();
60:                   }
61:       }
62:     return (INT)msg.wParam ;
63: }
64:
65: //
66: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
67: // ウィンドウプロシージャ関数
68: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
69: {
70:       switch(iMsg)
71:       {
72:                   case WM_DESTROY:
73:                               PostQuitMessage(0);
74:                               break;
75:                   case WM_KEYDOWN:
76:                   switch((CHAR)wParam)
77:                   {
78:                               case VK_ESCAPE:
79:                                           PostQuitMessage(0);
80:                                           break;
81:                   }
82:                   break;
83:       }
84:       return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
85: }
86:
87: //
88: //HRESULT InitD3d(HWND hWnd)
89: // ダイレクト 3D の初期化関数
90: HRESULT InitD3d(HWND hWnd)
91: {
92:       //「Direct3D」オブジェクトの作成
93:       if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
94:       {
95:                   MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
96:        return E_FAIL;
97:       }
98:       //「DIRECT3D デバイス」オブジェクトの作成
99:       D3DPRESENT_PARAMETERS d3dpp;
100:      ZeroMemory( &d3dpp, sizeof(d3dpp) );
101:
102:      d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
103:      d3dpp.BackBufferCount=1;
104:      d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
105:      d3dpp.Windowed = TRUE;
106:
107:       if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
108:                                   D3DCREATE_MIXED_VERTEXPROCESSING,
109:                                   &d3dpp, &pDevice ) ) )
110:    {
111:                   MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
112:                   if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
113:                                   D3DCREATE_MIXED_VERTEXPROCESSING,
114:                                   &d3dpp, &pDevice ) ) )
115:                   {
116:                               MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
117:                                return E_FAIL;
118:                   }
119:    }
120:      //「テクスチャオブジェクト」の作成
```

```
121:      if(FAILED(D3DXCreateTextureFromFileEx(pDevice,"Sprite.bmp",100,100,0,0,      D3DFMT_UNKNOWN,
122:                                                                                    D3DPOOL_DEFAULT,D3DX_
FILTER_NONE,D3DX_DEFAULT,
123:                                                                                    0xff000000,NULL,NULL,&pTexture)))
124:      {
125:                MessageBox(0," テクスチャの作成に失敗しました ","",MB_OK);
126:                return E_FAIL;
127:      }
128:      //「スプライトオブジェクト」 の作成
129:      if(FAILED(D3DXCreateSprite(pDevice,&pSprite)))
130:      {
131:                MessageBox(0," スプライトの作成に失敗しました ","",MB_OK);
132:                return E_FAIL;
133:      }
134:      return S_OK;
135: }
136:
137: //
138: //VOID DrawSprite()
139: // スプライトを描画する関数
140: VOID DrawSprite()
141: {
142:      pDevice->Clear( 0, NULL, D3DCLEAR_TARGET,D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
143:      if( SUCCEEDED( pDevice->BeginScene() ) )
144:      {
145:                RECT rect={0,0,100,100};
146:                D3DXVECTOR2 vec2Scale(1.0,1.0);
147:                D3DXVECTOR2 vec2RotationCenter(1.0,1.0);
148:                D3DXVECTOR2 vec2Position(270,180);
149:                pSprite->Draw(pTexture,&rect,&vec2Scale,&vec2RotationCenter,0,&vec2Position,0xffffffff);
150:                pDevice->EndScene();
151:      }
152:      pDevice->Present( NULL, NULL, NULL, NULL );
153: }
```

DirectX に関連するステートメントは 87 行目以降です。それ以外はウィンドウプロシージャーコールバック関数の定義とウィンドウの初期化及びメッセージループの実装という最低限必要なコードであり、ウィンドウベースのアプリケーションでは決まり文句のようなものです。

ヘッダーファイルのインクルード (1 行目と 2 行目)

1 行目
　普通 DirectX ゲームアプリケーションはウィンドウベースアプリケーションです。ウィンドウベースのコードには最低限このヘッダーファイルをインクルード※しなくてはなりません。ウィンドウベースのコードではない場合 (コンソールアプリ等) でも INT,DWORD などマイクロソフト定義の表記に慣れている開発者はインクルードします。
2 行目
　Direct3D を使用しているコードなのでこのヘッダーをインクルードする必要があります。

Direct3D 関連のポインター変数を宣言 (4 行目～ 7 行目)

LPDIRECT3D9　ダイレクト 3D オブジェクトへのポインター宣言
LPDIRECT3DDEVICE9 ダイレクト 3D デバイスオブジェクトへのポインター宣言
LPDIRECT3DTEXTURE9 ダイレクト 3D テクスチャオブジェクトへのポインター宣言
LPD3DXSPRITE ダイレクト 3D スプライトオブジェクトへのポインター宣言

オブジェクトとは DirectX インターフェイスを実際にメモリーにロードし、実体化させたものであり、インスタンスとも呼びます。C++ におけるクラスとオブジェクトの関係と同じように考えて当面は差し支えないでしょう。これら 4 つの変数はオブジェクト (インスタンス) へのポインターです。DirectX では、このようにほとんどの要素がインターフェイスで用意されていて、それらを使用するにはインターフェイスのオブジェクト (インスタンス) を作成してオブジェクトへのポインターによりアクセスするという方法をとります。

関数プロトタイプ宣言 (9 行目～ 11 行目)

LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
後述するウィンドウプロシージャー関数のプロトタイプ宣言。
HRESULT InitD3d(HWND);　Direct3D の初期化関数のプロトタイプ宣言。
VOID DrawSprite();　スプライト描画関数のプロトタイプ宣言。

全ての始まり WinMain 関数（とウィンドウプロシージャ関数）。それはウィンドウズアプリケーションのお決まりコードです (16 行目～ 85 行目)

当初は WinMain 関数やウィンドウプロシージャー関数の解説をすべきか悩みましたが、DirectX ゲームも Windows アプリケーションである以上、関連する事項は全て解説する必要があると考えました。
WinMain 関数と WndProc 関数は DirectX 関連ではありませんが、ウィンドウズアプリケーションにおいて、必ず無ければならないコードです。
WinMain 関数はエントリー関数とか単にエントリーポイントなどと呼ばれています。
WndProc 関数はウィンドウプロシージャー関数と言います。
WinMain 関数でやっている事は、アプリケーションウィンドウを作成し、メッセージループを実装するという処理です。
WndProc 関数でやっていることは、OS が発行したイベントメッセージを受け取り、それを処理するというものです。
MFC や雛形クラスライブラリを利用してコードを書き始める場合では、これら WinMain 関数やウィンドウプロシージャー関数のコードが隠蔽されていて、どこにもコードが見当たらないように感じることもあるかもしれません。しかし、それはただ単に（いい意味で）隠蔽・カプセル化されているだけでありコードのどこかに（カプセル内部に）必ず存在するはずです。時にはソースファイルレベルではなくライブラリファイル内にバイナリレベルで存在することもあるでしょう。いずれにしても繰り返しになりますが、これらの処理はウィンドウズアプリケーションでは必ず存在するものであり、存在しなければならないものです。単純なメッセージボックスのみのプログラムでもメッセージループは、メッセージボックスのダイナミックリンクライブラリ内に存在します。
これらのコードは文字通り「お決まり」のコードでありアプリケーションが変わってもほとんど変わらないコードであるがゆえに一旦理解してしまえば楽です。
「ほとんど変わらない」ということから、賢明なクリエイターであれば、新たなアプリケーションのソースコードを書き始めるにあたって、クラス等に最初から組み込むことができるような処理を一から新たに書くことはしないでしょう。おそらく自作または既存の雛形を利用したりライブラリをリンクして全く同じ内容の処理を書くという無駄を回避するはずです。そもそも MFC やライブラリに最初から実装されているのも、そのような無駄を排除するためなのですから。本書では MFC の利用は想定していませんし、読者が既存のクラスを利用することも想定していませんので、ウィンドウズプログラミングの経験がなければ当面このコードをコピーして自身のコードにペーストしてもいいでしょう。コードの再利用においてコピーアンドペーストはプログラマーとしてはあまりエレガントなスタイルとは言えないのですが今は十分です、なにも問題はありません。読者が開発者としてコードを書く経験を積んでいくにつれて自分でクラスライブラリを作りたいと思ったり、必要に迫られて自作したりしてプログラミングスタイルを少しずつ進化させていくものです。最初からエレガントに作業できるプログラマーはいません。私ごとですが筆者の初心者時代においてコピーアンドペーストでコードを再利用することはメインテクニックでした。（笑）また、筆者が非常勤講師をさせてもらっている某専門学校ゲーム学科の生徒たちにとってコピーアンドペーストは常套手段ですし、それでいいと筆者は思っています。最初はシンプルな方法でいきましょう。
では、コードを順に解説していきます。
ますは WinMain 関数から解説します。
WinMain 関数はすべての始まりと書きました。つまりアプリケーションが起動されて CPU が最初に実行するコードです。厳密に OS レベルで言うと WinMain 関数の前に呼ばれるスタートアップコードルーチンがありますが、開発者にとっては見えないものですし、スタートアップはコンパイラーが勝手（?）に EXE ファイルに挿入するものであり EXE ファイルの仕様の範疇です。したがって WinMain 関数が最初に実行されると考えてもらって支障はありません。（OS の内部動作を理解してプログラミングする必要があるでしょうか、少なくとも今はありません。）
このようにアプリケーション内で起動後最初に実行される関数をエントリーポイントあるいはエントリー関数と呼びます。
ユーザーがプログラムを起動して最初に実行されるエントリーポイントの WinMain 関数ですが、関数名は必ず「WinMain」でなければなりません。WINMAIN や Main あるいは WindowsMain などと自由に関数名を決めることは出来ません。コンソールアプリケーションにおいて main() 関数がエントリー関数であり、必ず main という関数名でなければならないのと同様です。これは自由な関数命名規則において唯一の例外です。WinMain 関数以外、すなわちエントリーポイント以外は関数名を自由に決めることが出来ます。そして型も INT WINAPI 型でなくてはならず、引数も (HINSTANCE 型 , HINSTANCE 型 , LPSTR 型 , INT 型) でなくてはいけません。つまり必ず INT WINAPI WinMain(HINSTANCE , HINSTANCE, LPSTR, INT) という形式でなければいけないということです。自由に決めることが出来るのは仮引数の変数名くらいです。仮引数とは本章コード内でいうと NT WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR szStr,INT iCmdShow) の hInstance、hPrevInstance、szStr、iCmdShow のことです。
18 行目
HWND 型の変数を宣言します。ウィンドウアプリケーションでは、最低 1 つのウィンドウを作成し、OS 側はハンドルによって当該アプリケーションのウィンドウを管理します。
HWND 型の変数はそのウィンドウハンドルの値を格納する変数です。
19 行目
MSG 型の構造体を宣言します。MSG 型の構造体は OS から受け取るメッセージを格納するために必要なものです。「OS からメッセージを受け取る」ということは重要な概念であり、後で解説するウィンドウプロシージャー関数（本章では WndProc 関数 68 行～ 85 行目）およびメッセージループ (本章では 49 行～ 61 行目) とも密接に関連する概念であり、ウィンドウズプログラムでは最も重要で基本的な概念だということをまず頭に留めておいてください。すぐ後で説明します。
21 行目～ 42 行目
サイズは 640 × 480 でタイトルバー、拡大縮小ボタン、閉じるボタンがあるだけのシンプルなアプリケーションウィンドウを作成し、ウィンドウを画面に表示するところまでの処理をしています
44 行目～ 47 行目
Direct3D の初期化関数をコールします。この時点で DirectX が初期化され利用できるようになります。念のため FAILED マクロにより Init3D 関数が何らかの原因で失敗した場合にはアプリケーションを終了するようにしています。WinMain 関数はアプリケーション内の他のどの関数から呼ばれるものではなく OS から呼ばれている関数なので、エントリー関数から return するということはアプリケーションを終了することを意味します。

49 行目～ 61 行目
この部分がメッセージループと呼ばれる処理です。
先ほど「OS からメッセージを受け取る」という概念が重要だと書きました、メッセージループとは OS からの何らかのメッセージがあるかどうか調べ（52 行目）、メッセージがある場合には、それをウィンドウプロシージャー関数に渡し（54 行目と 55 行目）、メッセージが来ていない時間にアプリケーションの処理を実行する（59 行目）という処理です。先ほど宣言しておいた MSG 型の構造体にメッセージを格納しているのがわかると思います。
「メッセージ」とか「メッセージが届く」とかウィンドウズプログラミング経験の無い読者にとっては分からない事柄でしょうから、説明します。
Windows はマルチタスク OS です。Windows 上で動くウィンドウズアプリケーションもマルチタスクアーキテクチャになります。これに対して MS-DOS プログラムの場合はシングルタスクアーキテクチャでありメッセージ処理の概念はありません。いままで MS-DOS のようなシングルタスク環境でプログラミングした経験がある人には分かりづらい概念でしょう。
マルチタスク環境では複数のアプリケーションが同時に実行でき、それがマルチタスクということなのですが、普通パソコンに搭載されている CPU は一つです。それぞれのアプリケーションは OS により適切に割り振られたタイミングのみ CPU を占有できます。OS が各アプリケーションの CPU 占有時間（CPU 時間と言います）を高速に切り替えることにより見かけ上同時に実行しているように見えるわけです。したがって、同時に実行するアプリケーションが多すぎたり、極端に重たい処理をするアプリケーションが 1 つでもあるとパソコン全体のパフォーマンスが重くなります。そのような現象を経験した人は多いのではないでしょうか。OS と CPU は一生懸命次から次へとアプリケーション間を行き来しながらなんとかユーザーにアプリケーションが同時に動いているように影で必死になっているわけです、しかし物理的には所詮一つの CPU ですから限界があります。ちなみに、Intel Xeon プロセッサーなどでは 1 台のマシンに複数の CPU を搭載することが出来ますが（マルチプロセッサーと言います）、マルチプロセッサーマシンの動作が重くなりにくいのはこのためです。筆者はマルチプロセッサーマシンを仕事に使っていますが、感覚としては「かなり快適」です。（スタートメニューを開くのに 10 数秒近くかかるなんて現象も無くなりました）
このようにマルチタスク環境の場合、OS が各アプリケーションを管理・制御する必要があり、そして、それぞれのアプリケーションが排他的にマシンを占有するようなアーキテクチャ (DOS プログラムのような) では困るわけです。
マルチタスクを実現するためには先に書いた「メッセージ」を OS から受け取るメッセージアーキテクチャである必要があるのです。マルチタスクに関する説明をざっと簡単に（本当に簡単に）書いたのは、そのためです。
具体的なメッセージ処理の例を本章のコードにそって説明すると、例えば、エスケープキーをユーザーが押したとします。OS はキーボードが押されたということと、押されたキーの処理をメッセージとして発行します。アプリケーション側はメッセージがあるかどうかを調べて、メッセージがある場合にウィンドウプロシージャーにそのメッセージを転送します。「関数コール」ではなく「転送」するというのは、ウィンドウプロシージャーをコールするのは OS であってアプリケーションではないからです。これがコード中でウィンドウプロシージャーをコールしているステートメントが無い理由です。（このウィンドウプロシージャー関数のような関数をコールバック関数と呼びます。）ウィンドウプロシージャー内ではまず、メッセージの種類を switch,case 文で調べています。キーボード押下メッセージは WM_KEYDOWN という定数が対応付けされていますので、処理は、75 行目の case 文に飛びます。そして、さらにキーの種類を調べる switch,case 文があります。（本章では簡単のためエスケープキー VK_ESCAPE のみの処理しかありません）今の例ではまさにエスケープキーを押したということなで 79 行目が実行されます。PostQuitMessage(0) とはアプリケーションを終了するという API ですので、これでアプリケーションが終了することになります。
キーボードだけではなくマウス操作やウィンドウの移動やサイズの変更等ユーザーの操作のありとあらゆるイベントがメッセージとして発行されアプリケーションは発行されたメッセージを調べて、それをウィンドウプロシージャーに転送するというのがメッセージアーキテクチャの基本構造です。コンソールアプリケーション（DOS アプリケーションのような）が能動的なのに対し、ある意味、ウィンドウズアプリケーションは OS に対して受動的と言えるかもしれません。
マルチタスク、メッセージアーキテクチャについては数ページで解説できるものではありませんが、あえてざっと説明したのは、ウィンドウズアプリケーションである DirectX アプリケーションの解説を始めるにあたってどうしても触れておかなければならなかったからです。（本当に触れただけです）
読者には、「ああそうゆうものなんだなぁ」となんとなく気に留めてもらえればいいですし、それが狙いです。言うまでも無くそれ以上のことはこの数ページからは不可能です。
メッセージループがメッセージアーキテクチャに基づくもので、メッセージループによりメッセージ処理とアプリケーション処理を振り分けて処理していることがなんとなく分かったところで、本章メッセージループについて補足します。
49 行目
念のため ZeroMemory により、msg 構造体をゼロで初期化します。
50 行目
msg 構造体の message メンバーが WM_QUIT ではない限りループを繰り返すようにしています。WM_QUIT は PostQuitMessage(0) が実行された場合に発行されるメッセージであり、PostQuitMessage(0) はアプリケーションを終了する場合に使用する API です。本章ウィンドウプロシージャー内でエスケープキーが押された場合及びウィンドウが閉じられた場合に PostQuitMessage(0) が実行されています。
WM_QUIT がメッセージとして届くと制御が while ブロックから抜けてその下の return (INT)msg.wParam により WinMain 関数から return すなわちアプリケーションを終了するという仕組みです。
52 行目
PeekMessage はメッセージキュー（OS がメッセージを発行するとメッセージキューにメッセージがポストされます）にメッセージがあるかどうか調べて引数の MSG 構造体にメッセージを格納します。メッセージがある場合、TRUE になるので 54 行目と 55 行目が実行されます。
54 行目と 55 行目
TranslateMessage と DispatchMessage は対で使用するようにします。ウィンドウプロシージャーにメッセージを転送しているのは DispatchMessage の方です。
59 行目
メッセージが無い場合すなわち PeekMessage が FALSE の場合、このブロックが実行されます。一般的に、このようにメッセ

ージが無い場合にアプリケーションの処理を書きます。今回は DrawSprite 関数によりスプライトの描画を行っています。
62 行目
アプリケーションの終了メッセージによりメッセージループから制御が抜けた場合にこの行に制御が到達します。WinMain 関数からリターンしアプリケーションを終了しています。
WinMain 関数とメッセージループの説明はこれで終わります。次は、今までも文中で何回か登場したウィンドウプロシージャーについてコードを追いながら解説します。
とは言っても、既にウィンドウプロシージャーの存在理由とおおまかな機能については説明していますのでそれ以外のことについて補足します。
68 行目
ウィドウプロシージャは WinMain 関数とは違いその関数名は自由に決められます。大抵のアプリケーションでは WndProc とか MsgProc などという名前にしているようです、自由に命名できてもやはり分かりやすい名前にするべきなので似たような関数名になるのでしょう。関数名は自由に命名できても関数の型と引数の型は決まっていますので本章コードと同様に書く必要があります。関数の型は LRESULT CALLBACK であり引数の型は (HWND,UINT,WPARAM,LPARAM) です。仮引数の名前は自由に命名できます。
そして、関数内でることは switch,case 文でメッセージを調べてメッセージ毎の処理を書くということです。
最後 84 行目にある return DefWindowProc (hWnd, iMsg, wParam, lParam) ですが、これはアプリケーションが処理しなかったメッセージをシステムに処理してもらうという処理です。アプリケーションに対し発行されウィンドウプロシージャーに転送されてくるメッセージは 200 種類を超えます、その全てのメッセージに対応する処理をウィンドウプロシージャーに書くことはありませんし、その必要があるアプリケーションはまず無いと言えます。アプリケーションにとって関連するメッセージ、言い換えればアプリケーションにとって必要なメッセージ（本章ではキーボードの押下メッセージ）だけの処理を書けばいいわけです。そして、アプリケーションにとって不要なメッセージは全てシステムに処理してもらうように DefWindowProc にメッセージを渡します。このように、DefWindowProc に未処理のメッセージを渡さないと予期せぬ挙動をしますので、これは鉄則と覚えておいてください。例えば、本章ウィンドウプロシージャーの最後で return 0 などと直接リターン※してしまうとキーボード押下メッセージ以外はなにも処理されないということになり確実にアプリケーションは予測不可能な挙動をします。
最後は必ず return DefWindowProc() という形にしてください。このように全てのメッセージにたいする処理を許可されているウィンドウプロシージャーは強力な関数であると同時に不適切な処置をすると怖い関数であるとも言えます。

本章で必要な Direct3D の初期化やスプライトの初期化などを処理する関数 (90 〜 135 行目)

93 行目
まず、最初にすることは Direct3D オブジェクトを作成することです。Direct3D オブジェクトは全ての Direct3D 関連インターフェイスの根源となるもっとも重要なオブジェクトです。本章でこの後に作成する Direct3D デバイスオブジェクトは Direct3D オブジェクトから作成され、テクスチャオブジェクト、スプライトオブジェクトは direct3D デバイスオブジェクトが存在しなければ作成することができません、したがって全てのオブジェクト作成には Direct3D オブジェクトの存在が必要となり、Direct3D オブジェクトを作成しなければ他の３つのオブジェクトは作成できません。
Direct3DCreate9 は、DirectXSDK で用意されているグローバルな関数です。この関数により Direct3D オブジェクトを作成しそのポインターを得ます。引数は必ず D3D_SDK_VERSION となります。 1 種類に固定されている引数を渡すというのも奇妙に思われるかもしれませんが、仕様なので単純にそのまま覚えておきましょう。
99 〜 119 行目
Direct3D デバイスオブジェクトを作成します。デバイスオブジェクトを作成する手順は、まず D3DPRESENT_PARAMETERS 型の変数を用意し、各メンバーを初期化した後、IDirect3D9::CreateDevice メソッドに D3DPRESENT_PARAMETERS 変数のアドレスを渡します。以下に各メンバーの意味を説明します。

BackBufferFormat
　画面表示は普通、フロントバッファとバックバッファの２つ（３つ以上のバッファーを使うこともある）を交互に切り替えることにより滑らかな画面更新を実現します。これはバックバッファのフォーマットつまり表示形式を意味します。D3DFMT_UNKNOWN を指定すれば現在のディスプレイモード（デスクトップの表示モード）が自動的に設定されるので便利です。
BackBufferCount
バックバッファの数。本章ではバックバッファは一つなので 1 を指定します。
SwapEffect
BackBufferFormat メンバーの説明で述べたフロントバッファとバックバッファの切り替えを「フリップ」と呼びます。SwapEffect(スワップエフェクト) とはこのフリップの形態を言います。
当面は D3DSWAPEFFECT_DISCARD を指定すると覚えていても差し支えないでしょう。

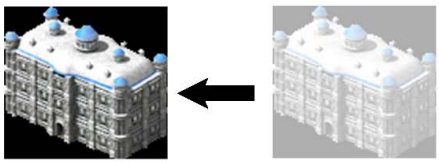
Windowed
　アプリケーションの表示形式がウィンドウモードかフルスクリーンモードかを指定します。TRUE の場合はウィンドウモードで表示され、FALSE の場合はフルスクリーンモードで表示されます。本章ではウィンドウモードを指定しています。
D3DPRESENT_PARAMETERS 変数を初期化してようやく Direct3D デバイスオブジェクトを作成する準備ができました。作成には IDirect3D9::CreateDevice メソッドを使用します。先に作成しておいた Direct3D オブジェクトへのポインターを pD3d 変数に格納していますのでコードでの表記は pD3d-> CreateDevice となります。（オブジェクトへの「ポインター」なのでポインター演算子は “->” 演算子になります。）CreateDevice の引数の意味は次のとおりです。
D3DADAPTER_DEFAULT
パソコンにビデオカードが複数搭載されていない限りこの序数を使用してください。
D3DDEVTYPE_HAL

表示はハードウェアかソフトウェアのどちらかで実現しますが、ハードウェアで実現するほうがかなり高速です。本章ではハードウェア機能を使用するよう指定します。
HWnd
アプリケーションのウィンドウハンドルです。(WinMain 関数内で初期化されるので InitD3 関数が呼ばれる前に既に初期化されています。)
D3DCREATE_MIXED_VERTEXPROCESSING
この引数の説明は本章のレベルを超えているので割愛します。当面は単純にこの定数を使うものだと考えてください。
&d3dpp
先に初期化しておいた D3DPRESENT_PARAMETERS 変数のアドレスです。
&pDevice
CreateDevice 関数をコールする前の時点では内容が空のデバイスオブジェクトへのポインター変数のアドレスです。CreateDevice 関数をコールした後にはじめて中身にデバイスオブジェクトのアドレスが格納され、以降使用することができるようになります。DirectX ではこのように初期化したいポインターのアドレス(ポインタのポインタ)を引数で渡して DirectX 関数側でポインター変数を初期化するという方法が多く使用されています。(DirectX に限らず広く使用されるテクニックですが)
121 行目～ 127 行目
テクスチャオブジェクトを作成します。テクスチャオブジェクトは次のスプライトオブジェクトを作成するために必要なものであり、スプライト作成のための事前準備のようなものです。
D3DXCreateTextureFromFileEx 関数の引数を説明します。
pDevice
デバイスオブジェクトのポインター。
"Sprite.bmp"
ビットマップファイル名。本章では Sprite.bmp というファイルから読み込みます。
100,100
ビットマップのサイズです。本章では Sprite.bmp が縦・横共に 100 ピクセルなので 100,100,0,0 と指定しています。
今のところは 0,0 と指定すると覚えてください。
D3DFMT_UNKNOWN,D3DPOOL_DEFAULT, D3DX_FILTER_NONE,D3DX_DEFAULT
これについても、深入りせずにこのように指定すると考えてください。
0xff000000
これは、カラーキーあるいは「抜け色」と呼ばれているカラー値です。(図 2-1 参照) ビットマップ自体は四角形であり、四角形でしか有り得ません。しかし、ビットマップの内容(絵)は必ずしも四角形ではなく、複雑な形をしている場合の方が多いわけです。そのまま表示すると図 2-1 のようになり絵の回りに不自然な枠のように表示されてしまいます。絵以外の部分は表示したくない部分です。そこで図 2-2 のようにその部分の色値を指定し、その色値の部分は表示しないようにします。この色値が「抜け色」と呼ばれるものです。本章では Sprite.bmp の絵以外の領域の色値が黒です。黒とは RGB 値で表現すると 0x000000(Red:00 Green:00 Blue:00) となります (先頭の 0x は 16 進数表記という意味です)。この RGB 値の左にアルファ値(透明度)を付けた値を指定します。本章では絵以外の部分は完全に透明にしたいのでアルファ値は ff にします。その結果抜け色が 0xff000000 となるわけです。別に抜け色は黒である必要はありません。例えばビットマップの表示したくない部分が青である場合、青の RGB 値は 0xff0000ff(Red:00 Green:00 Blue:ff) なので抜け色は 0xff0000ff となります。

ディスプレイに
既に描画されているイメージ

重ねて描画したいイメージ
（建物の周りの色は黒）

描画結果
イメージの全てのピクセル
が描画されてしまう



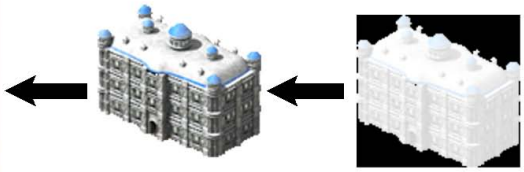
カラーキーを設定しないと、
イメージの全てのピクセルを
転送してしまう。

図 2-1

ディスプレイに
既に描画されているイメージ

重ねて描画したいイメージ
（建物の周りの色は黒）

描画結果
カラーキー領域が転送されないので、自然な仕上がりになる



カラーキーを（この場合）黒に設定すると、その色は転送されない
つまり、この場合、建物のピクセルのみが転送される

図 2-2

NULL,NULL
ファイル内のデータの記述を格納する必要はいまのところ無く、また、本章サンプルは 256 色モードではないのでこれらに NULL を指定します。
&pTexture
テクスチャオブジェクトのポインター変数のアドレスです。（ポインターのポインターです）
129 行目～ 133 行目
スプライトオブジェクトを作成しています。事前準備であるテクスチャオブジェクトを既に作成しているのでコードは簡単です。
129 行目
D3DXCreateSprite 関数にデバイスオブジェクトのポインター（これはポインター自体を、つまりデバイスオブジェクトのアドレスを引数にします）と初期化前のスプライトオブジェクトのポインター変数のアドレス（こちらはポインタのポインタです）を引数として渡すだけのシンプルなコードです。

以上で本章用の Direct3D の初期化は全て完了です。
次に実際に画面に描画する関数を解説します。

スプライトを画面に表示する関数 (140 ～ 153 行目)

142 行目
まず画面に描画する前に画面をクリア（消去）するというよくある処理です。
IDirect3DDevice9::Clear メソッドの各引数は今後もそのまま変更無しに使えると思います。ひとつだけ D3DCLEAR_TARGET,D3DCOLOR_XRGB(100,100,100) の部分だけ説明すると、これは何色でクリアするかという意味です。言い換えれば何色で画面全体を塗りつぶすかということです。実は「クリア（消去）」処理は「単一の意で画面全体を塗りつぶす処理」なのです。したがって、クリアする色に黒を使えば画面を黒でクリアできるし、例えば青を指定すれば画面が真っ青にクリアされるということです。本章では灰色 (R:100 G:100 B:100) によりクリアしています。
（pDevice は Direct3D デバイスオブジェクトへの「ポインター」なのでポインター演算子は “->” 演算子になります。Direct3D に限らず DirectX 全般的に、インターフェイスオブジェクトへのアクセスはポインターを介して行うので、ほとんど全て ”->” 算子を使用することになります。）
143 行目
IDirect3DDevice9::BeginScene メソッドはすべての描画処理の前に必ずコールする必要があります。そして BeginScene メソッドが成功した場合のみ、その後のすべての描画処理を実行します。

if(SUCCEEDED…の SUCCEEDED は「もし…が成功したならば」というマクロです。ここでは「もし BeginScene が成功したならば」という意味になります。
全ての描画処理はこの if 文の成功ブロック (if 文の中括弧の中) の中に含まれているのがわかると思います。
145 行目
RECT とは四角形領域（「矩形」と呼びます）を格納するための構造体です。左、上、右、下の 4 つの座標情報から成ります。0,0 座標を左上とし、100,100 を右下とする四角形（矩形）を作成しています。この矩形はスプライト用のテクスチャーとして読み込むビットマップのどの部分を読み込むかを指定するために必要なもので、今回はビットマップのサイズと完全に一致していますのでビットマップ全体を読み込むということになります。
146 行目～ 148 行目
D3DXVECTOR2 とはベクトル※を格納するための構造体です。型名からベクトルに関係する型であるということが分かった読者もいるのではないでしょうか。最後に付いている数字の 2 は 2 次元ベクトルという意味です。ちなみに第 4 章で解説する 3D 物体の表示では D3DXVECTOR3 という 3 次元ベクトルの構造体を使用します。
ここでは 3 つの 2 次元ベクトル (vec2Scale, vec2RotationCenter, vec2Position）を作成しています。左からスケール情報用の 2 次元ベクトル、回転の中心情報用の 2 次元ベクトル、位置情報用の 2 次元ベクトルです。
149 行目
矩形及び 3 つのベクトルを作成したのはすべてこの ID3DXSprite::Draw メソッドを呼び出す際に引数として必要だったからです。
第 1 引数 pTexture は先に作成しておいたテクスチャオブジェクトのポインターです。
第 2 引数 &rect も先に作成した矩形のアドレスです。
第 3 引数 &vec2Scale
第 4 引数 &vec2RotationCenter
第 5 引数 回転をラジアン単位で指定します。今回はスプライトを回転させないので 0 とします。
第 6 引数
第 7 引数　元のビットマップそのままの色合いで描画したい場合は 0xffffffff を指定します。この値は先ほど IDirect3DDevice9::Clear メソッドの解説で出てきたクリアする色の値と全く同じで RGB 値の先頭にアルファ値（透明度）を付けた値です。今回は不透明、RGB 値はビットマップのものをそのまま使うという設定です。試しに、アルファ値や RGB 値を変更してビルド・実行してみてください。ビットマップが半透明に描画されたり色合いが変化するのが確認できます。
150 行目
IDirect3DDevice9::EndScene メソッドは全ての描画処理を終了する箇所でコールします。なお、IDirect3DDevice9::BeginScene が成功した場合に実行する必要がありますので BeginScene 成功判断の if 文ブロック内に入れています。
152 行目
いよいよ最後の処理が、pDevice->Present メソッドです。これはバックバッファとフロントバッファを切り替える（フリップさせる）処理です（図 2-2 参照）、いままでの描画はバックバッファに絵を書き込んでいたのでフロントバッファとバックバッファを反転させて絵を画面に出す必要があります。言い換えると画面の裏（目に見えない）に描画していた絵を画面の表（目に見える）に出す処理であるとでも言えばいいでしょうか、要するに画面を更新するものだと覚えておいてください。この処理をしないとせっかく描画処理が正常でも画面には何も表示されません。

図 2-3

Chapter3 表示したものを移動させる・操作する

前章にて、画面に絵を表示することが出来ました。これはゲームプログラミングを開始するにあたって大変大きな第一歩でした。本章では、キーボード上の対応したキーを押すことによりその絵を移動させる、すなわち操作するということを実践します。他の同種書籍で、この処理の解説をしているものは見たことはありません。思うにその理由は、あまりにも基本的な処理ということで、独立した章を設けてまで解説せずともソースコードを見れば理解できる事柄だという判断ではないかと思います。それに対する筆者の見解はそうかもしれませんし、「そうではないかもしれない」というところです。「そうではないかもしれない」に該当する読者が居た場合、本章が役に立つことを想定し本章を設けました。本書のセクション 1 はセクション冒頭で書いたように全くのゲームプログラミング初心者も「確実」にゲームプログラミングが出来るようになっていただきたいというのが筆者の願いです。
画面に絵を表示し、さらに、それを操作するという処理が実装されればそれだけでゲームとして成り立つものであり、ゲームにおいて、表示した絵を移動させるというのは表示処理と同じくらい必須処理であり重要な処理です。余談ですが、今から 21 年前、筆者が中学生の頃、親に買ってもらった＊ JR-100 のモニター上でキャラクターを動かす簡単なプログラムの作成に成功したときの感動はその後、どんなにすごいマシン上でどんなに高度な処理やグラフィックのプログラムが成功した時の感動よりも大きく劇的なものでした。
では、まずは「移動」から始めましょう。

移動表示処理

「移動」という現象をモニター上でどのように実現しているのでしょうか？
現実にある物で格好の例えがあります。それは、営業宣伝に使用される「電工掲示板」やパチンコ屋さんや飲食店（特に夜系の）によく設置されている「ネオンサイン」です。
電光掲示板やネオンサインは、たくさんの小さな電球 (あるいは LED 素子) から成っていて、それらの電球の点灯や色変化により絵や文字を表示しているのは言うまでもない事柄でしょう。
当たり前ですが、ネオンサインに表示される文字や絵が動いている（ように見える）のは、一つ一つの電球が物理的に動いているわけではありません。電球は点いたり消えたり、時には色が変化するだけです。では、なぜ動いているように見えるのでしょうか？これは人間の目の錯覚を利用したテクニックというかトリックです。今、電光掲示板が図 3-1 のような状態だとします。

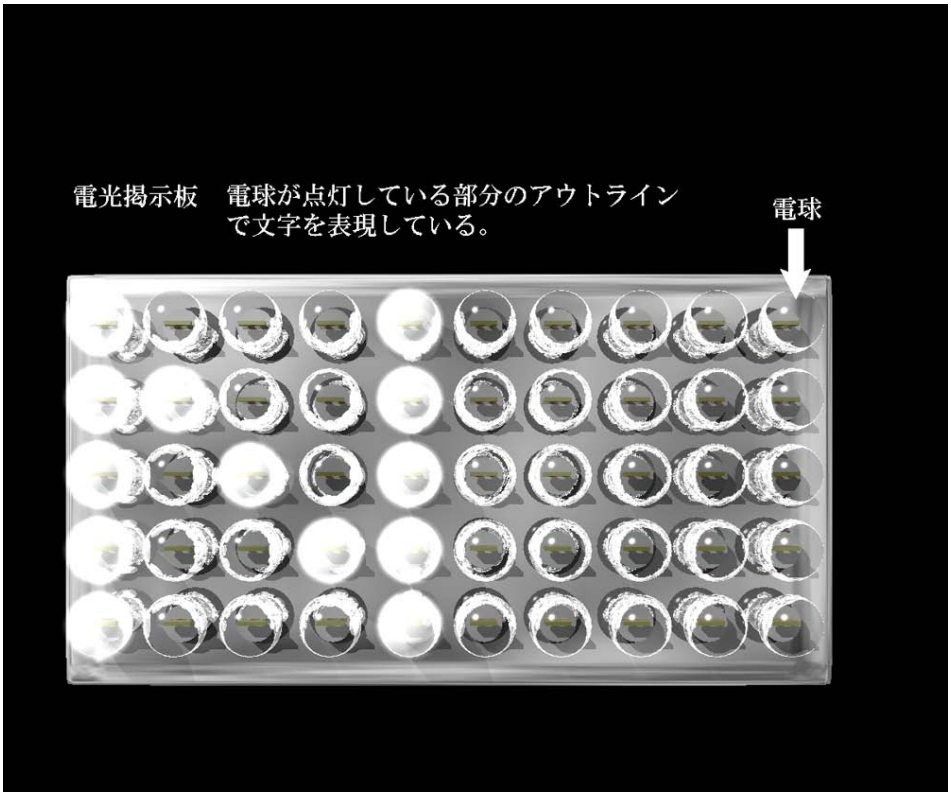


図 3-1

この文字を右に移動するように見せるために、まず、点灯している電球より右側の電球を点灯させると “ 同時に ” 今まで点灯していた電球を消します。図 3-2 参照

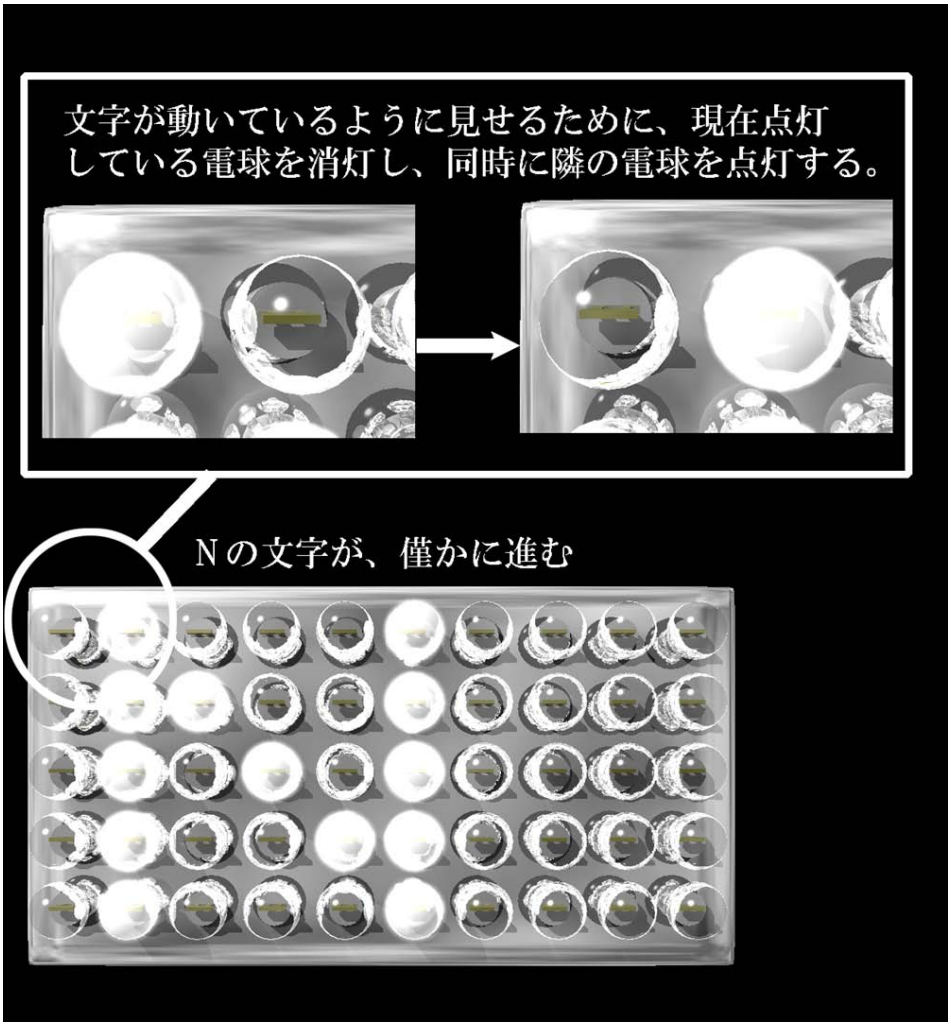


図 3-2 を挿入

人間の目が識別できる時間的感覚は優れている人で最高 60 分の 1 秒あたりと言われています。そして 60 分の 1 秒や数 10 分の 1 秒というような速さでこの一連の処理が行われると人間の目には図 3-3 のように見えます、つまり、移動しているように見えるわけです。

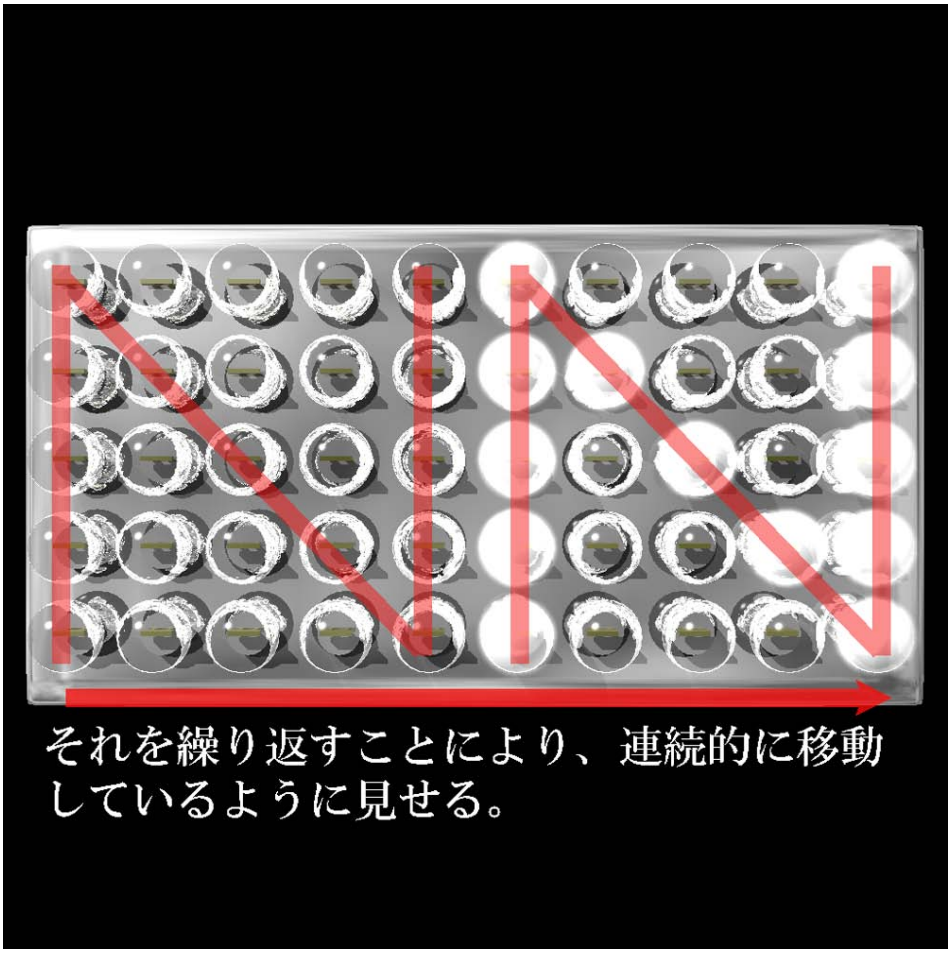


図 3-3 を挿入

パソコンモニター上で移動を実現する仕組みは、まさにこれと同様であり、電球がモニターのピクセルに置き換わっただけのことです。（モニターのピクセルは電球より遥かに多いですが）これが格好の例えと書いた理由です。
パソコンモニター上で移動を実現するには、現在表示さている絵を消すと同時に数ピクセル分ズレた座標に表示するということ高速かつ連続して行います。それらは、人間の目の錯覚により移動しているように見えます。

次は、「操作」処理について解説します。

操作処理

キーボードのキーを上下左右の移動に対応させて、キーを押せばその方向に表示している絵を移動させるという処理を考えます。本章サンプルではキーボードの矢印キー 4 つを上下左右の移動に使用しています。
ユーザーのキー入力を何らかの形でプログラムに知らせる、言い換えればプログラムが何らかの形でキー入力を認識する必要があるという事は想像できるでしょう。プログラムがキー入力を知ることが出来ればそれに対応して絵を移動表示させればいいわけです。
キー入力を知らせる・認識させる方法は 2 つあります。1 つは前章でも触れた Win32 メッセージを利用する方法。2 つめは DirectInput を利用する方法です。Win32 メッセージを利用する方法は前章で使用していますので何となく分かるかもしれません。
では、どちらを利用すべきなのでしょうか？あるいはどちらの手法でもいいのでしょうか？
答えは、「極度にスピードを要求される場面では DirectInput、それ以外の場面ではどちらの手法でもいい」です。なぜ、スピードを要求される状況において DirectInput を使用すべきなのか、その理由を書きます。
Win32 メッセージ発行の時間的間隔は DirectInput に比べて遅いのです。遅いといっても数ミリ～数十ミリ秒オーダーですが、DirectInput と比べると入力情報受信の分解能に 2 倍弱の開きがあります。○○ミリ秒と書いてもピンとこないかも知れませんので、実例を挙げましょう。
入力情報には、キーボードの押下、マウスボタンの押下、マウスの移動があります。これら 3 つの中で最も高速に情報を得る必要があるのは “ マウスの移動 ” です。なぜかと言うと、キーボードを数十ミリ秒より速い間隔で打てる人はそう居ないでしょうし、マウスボタンの押下についても同じでしょう。この 2 つについて Win32 メッセージと DirectInput の速度比較をしようとしても差を体感できるような操作をすること自体困難です。しかし、マウスの移動となると話が違います。マウスをゆっくり動かしたとしても移動軌跡の座標は理論上無限に存在するので、情報の分解能が高いことに越したことはない部分です。例えばソフトウェアカーソル※をプログラムする場合、ユーザーがマウスを移動したとき、移動軌跡上の座標を出来るだけ多く、しかも高速に（高分解能・高解像度で）得る必要がありますが、メッセージから座標を得た場合、その間隔

が DirectInput と比べて遅いためカーソルは飛び飛びの表示になってしまい、特にユーザーが素早くマウスを動かした場合などカーソルが瞬間移動したかのように見えてしまいます。（この現象は非力なマシン環境において顕著に見られますが、最近のマシンでは差がはっきりしないかもしれません）一方 DirectInput の場合は十分な速さで入力情報（この場合はマウス座標）を得ることが出来るのでソフトウェアカーソルもハードウェアカーソルのように見せることが出来ます。このように、メッセージ方式は時間的解像度が劣っているため高速性が必要な状況においては使い物にならないことがあります。
しかし、スピードをさほど要求されない場面では、メッセージから入力情報を得ても問題ありません。例えばゲーム開始前の設定画面では Win32 メッセージでも十分ですし、ゲーム中でもゲーム自体がリアルタイムではない場合には問題ないかもしれません。また、キーボード入力情報をテキストとして扱う場面では Win32 メッセージを利用した方が有利な場合があります。とは言え、明らかに遅い方式を積極的に使用する理由はないと思われ、入力は全て DirectInput を利用するというスタイルでもいいでしょうし、実際筆者の場合、製品の最終コードではそのスタイルをとっています。メッセージ方式は実装が楽なため（ウィンドウプロシージャー関数内で数行のコードで実装できるため）簡単なテストプログラムの作成時で DirectInput の初期化を書くのが面倒な場合にのみ使用しています。テキストを扱うプログラムで DirectInput を使用している場合、キーリピートやキーが押されているのか離されているのかの判断はアプリケーションが自前のコードで実装しなくてはなりませんがこれは開発者側の問題であり、どうとでもなることです。
ここで、参考のために筆者が所有するマシンで行った Win32 メッセージと DirectInput の性能比較を掲載します。
このテストはマウス移動に伴う座標変化を取得する速さを計測したものですので、値が小さいほど高速にデータを取得出来ているということです。
図 3-4

			所要時間（単位 ミリ秒）
Xeon 2400Mhz Dual	Windowメッセージ		8.1
	DirectInput排他モード	直接データ	5.8
		バッファリングデータ	4.8
	DirectInput非排他モード	直接データ	7.8
		バッファリングデータ	4.8
Pentuam4 1700Mhz	Windowメッセージ		8.7
	DirectInput排他モード	直接データ	8
		バッファリングデータ	4.5
	DirectInput非排他モード	直接データ	8
		バッファリングデータ	4.6
Duron 1000Mhz	Windowメッセージ		14.6
	DirectInput排他モード	直接データ	11.8
		バッファリングデータ	6.4
	DirectInput非排他モード	直接データ	13.1
		バッファリングデータ	6.5
Pentium2 400Mhz	Windowメッセージ		17.4
	DirectInput排他モード	直接データ	17.4
		バッファリングデータ	8.9
	DirectInput非排他モード	直接データ	17.3
		バッファリングデータ	9

このテストを終えてまず感じたことは、これほどきれいに予想通りの値になるとは思わなかったということです。筆者が予想していた速さは、速い順に
1 DirectInput 排他モード バッファリングデータ
2 DirectInput 非排他モード バッファリングデータ
3 DirectInput 排他モード 直接データ
4 DirectInput 非排他モード 直接データ
5 Windows メッセージ形式
というものであり、結果は例外なく予想通りになっています。
この結果から解るように、同じ DirectInput でも、直接データよりバッファリングデータのほうが速く、非排他モードより排他モードのほうが速い、また Win32 メッセージはどの DirectInput モードよりも遅いということです。一番速い DirectInput のモードと比べ 2 倍弱の時間を要しています。
あと、マシン環境が優れているほうが分解能は高いというのは敢えて言うまでもないでしょう。同じ土俵であれば（マシン環境が同じであれば）例外なく DirectInput の方が高速です。このことだけは覚えておいてください。

本章では、入力処理以外は全く同じである 2 つのプログラム（メッセージ方式と DirectInput 方式の両方のコード）を掲載します。初心者にはメッセージ方式の方がシンプルで覚えることも少ないと思うからです。

ウィンドウメッセージを利用した入力処理の場合

サンプルプロジェクト名：Chapter3-1

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5:
 6: LPDIRECT3D9 pD3d;
 7: LPDIRECT3DDEVICE9 pDevice;
 8: LPDIRECT3DTEXTURE9 pTexture;
 9: LPD3DXSPRITE pSprite;
10: FLOAT fPosX=270,fPosY=180;
11:
12: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
13: HRESULT InitD3d(HWND);
14: VOID DrawSprite();
15: VOID FreeDx();
16:
17: //
18: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
19: // アプリケーションのエントリー関数
20: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
21: {
22:       HWND hWnd=NULL;
23:       MSG msg;
24:       // ウィンドウの初期化
25:       static char szAppName[] = "Chapter3-1" ;
26:     WNDCLASSEX  wndclass ;
27:
28:     wndclass.cbSize         = sizeof (wndclass) ;
29:     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
30:     wndclass.lpfnWndProc    = WndProc ;
31:     wndclass.cbClsExtra     = 0 ;
32:     wndclass.cbWndExtra     = 0 ;
33:     wndclass.hInstance      = hInst ;
34:     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
35:     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
36:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
37:     wndclass.lpszMenuName = NULL ;
38:     wndclass.lpszClassName = szAppName ;
39:     wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;
40:
41:     RegisterClassEx (&wndclass) ;
42:
43:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
44:                 0,0,640,480,NULL,NULL,hInst,NULL) ;
45:    ShowWindow (hWnd,SW_SHOW) ;
46:    UpdateWindow (hWnd) ;
47:       // ダイレクト3D の初期化関数を呼ぶ
48:       if(FAILED(InitD3d(hWnd)))
49:       {
50:                 return 0;
51:       }
52:       // メッセージループ
53:    ZeroMemory( &msg, sizeof(msg) );
54:    while( msg.message!=WM_QUIT )
55:     {
56:                 if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
57:        {
58:                         TranslateMessage( &msg );
59:            DispatchMessage( &msg );
60:        }
61:        else
62:                 {
63:                         DrawSprite();
64:                 }
65:       }
66:       // メッセージループから抜けたらオブジェクトを全て開放する
67:       FreeDx();
68:       // OS に戻る （アプリケーションを終了する）
69:     return (INT)msg.wParam ;
70: }
```

```
71:
72: //
73: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
74: // ウィンドウプロシージャ関数
75: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
76: {
77:       switch(iMsg)
78:       {
79:                 case WM_DESTROY:
80:                           PostQuitMessage(0);
81:                 break;
82:                 case WM_KEYDOWN:
83:                 switch((CHAR)wParam)
84:                 {
85:                           case VK_ESCAPE:
86:                                     PostQuitMessage(0);
87:                           break;
88:                           // スプライトを操作するための入力処理
89:                           case VK_LEFT:
90:                                     fPosX-=4;
91:                           break;
92:                           case VK_RIGHT:
93:                                     fPosX+=4;
94:                           break;
95:                           case VK_UP:
96:                                     fPosY-=4;
97:                           break;
98:                           case VK_DOWN:
99:                                     fPosY+=4;
100:                          break;
101:                }
102:                break;
103:      }
104:      return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
105: }
106:
107: //
108: //HRESULT InitD3d(HWND hWnd)
109: // ダイレクト 3D の初期化関数
110: HRESULT InitD3d(HWND hWnd)
111: {
112:      //「Direct3D」 オブジェクトの作成
113:      if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
114:      {
115:                MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
116:       return E_FAIL;
117:      }
118:      //「DIRECT3D デバイス」 オブジェクトの作成
119:      D3DPRESENT_PARAMETERS d3dpp;
120:      ZeroMemory( &d3dpp, sizeof(d3dpp) );
121:
122:      d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
123:      d3dpp.BackBufferCount=1;
124:      d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
125:      d3dpp.Windowed = TRUE;
126:
127:       if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
128:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
129:                                 &d3dpp, &pDevice ) ) )
130:    {
131:                MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
132:                if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
133:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
134:                                 &d3dpp, &pDevice ) ) )
135:                {
136:                          MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
137:                           return E_FAIL;
138:                }
139:    }
140:      //「テクスチャオブジェクト」 の作成
```

```
141:      if(FAILED(D3DXCreateTextureFromFileEx(pDevice,"Sprite.bmp",100,100,0,0,D3DFMT_UNKNOWN,
142:                                                                                    D3DPOOL_DEFAULT,D3DX_
FILTER_NONE,D3DX_DEFAULT,
143:                                                                                    0xff000000,NULL,NULL,&pTexture)))
144:      {
145:                MessageBox(0," テクスチャの作成に失敗しました ","",MB_OK);
146:                return E_FAIL;
147:      }
148:      //「スプライトオブジェクト」 の作成
149:      if(FAILED(D3DXCreateSprite(pDevice,&pSprite)))
150:      {
151:                MessageBox(0," スプライトの作成に失敗しました ","",MB_OK);
152:                return E_FAIL;
153:      }
154:      return S_OK;
155: }
156:
157: //
158: //VOID DrawSprite()
159: // スプライトを描画する関数
160: VOID DrawSprite()
161: {
162:      pDevice->Clear( 0, NULL, D3DCLEAR_TARGET,D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
163:      if( SUCCEEDED( pDevice->BeginScene() ) )
164:      {
165:                RECT rect={0,0,100,100};
166:                D3DXVECTOR2 vec2Scale(1.0,1.0);
167:                D3DXVECTOR2 vec2RotationCenter(1.0,1.0);
168:                D3DXVECTOR2 vec2Position(fPosX,fPosY);
169:                pSprite->Draw(pTexture,&rect,&vec2Scale,&vec2RotationCenter,0,&vec2Position,0xffffffff);
170:                pDevice->EndScene();
171:      }
172:      pDevice->Present( NULL, NULL, NULL, NULL );
173: }
174: //
175: //VOID FreeDx()
176: // 作成した DirectX オブジェクトの開放
177: VOID FreeDx()
178: {
179:      SAFE_RELEASE( pTexture );
180:      SAFE_RELEASE( pSprite );
181:      SAFE_RELEASE( pDevice );
182:      SAFE_RELEASE( pD3d );
183:
184: }
```

4 行目
#define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
今までは（第 2 章では）、生成したダイレクト X のオブジェクトを開放せずにそのままアプリケーションから抜けていました。メモリーにロードされた DirectX オブジェクトはアプリケーションが終了した後もそのままメモリー空間に存在し続けてしまうという現象の原因に繋がります。この現象をメモリリークと呼びます。メモリリークは一般的にプログラムのバグやプログラマーの不注意により起こる現象であり、開発者はメモリリークを起こすようなアプリケーションを作成してはいけません。メモリーにロードしたものは不要になった時点でアンロード（開放）するべきです。2 章で開放処理を行わなかったのはコードを極限までシンプルにしたかったのと、あの程度のオブジェクトを開放しないからといってマシンに悪影響がでるとは思えなかったからです。本章ではプログラムの終了時に FreeDx 関数により全ての DirectX オブジェクトを一括開放しています。この SAFE_RELEASE マクロを定義しておくと開放時に便利です。開放は SAFE_RELEASE(オブジェクトのポインター) とすれば簡単しかも安全に行えます。このマクロはまず、ポインターが空すなわち NULL でないかどうかを判断し、NULL でない場合にのみ Release メソッドを実行します。Release メソッドは DirectX オブジェクト全てに実装されている開放メソッドです。そして開放した後にそのポインターを空 (NULL) にします。オブジェクトが既に開放されていてポインターが NULL の時、その NULL ポインターにより Release メソッドをコールしたりすると、宝くじではずれるより高確率（？笑）でプログラムがクラッシュし強制終了します。これは強烈なバグです。このマクロはこのようなことがないようにチェックしてくれるので安全に開放が出来るわけです。
6 ～ 15 行目
グローバル変数と関数プロトタイプの宣言です。
FLOAT fPosX と fPosY はスプライトの座標用に用意した変数です。

20 行目～ 155 行目
WinMain 関数、および InitD3D 関数は 2 章と殆ど同一なので解説は省きます。
WinProc 関数も殆ど変化ありませんが、キー入力を調査する処理を追加しています。どの部分かというと 89 行目～ 99 行目で、

キーボードの 4 つの矢印キーが押されたときに fPosX または fPosY を増減させています。fPosX と fPosY はスプライトの座標です。

160 行目～ 173 行目
DrawSprite 関数も 2 章とさほど変わりませんが、1 箇所だけ変更しています。それは 168 行目　D3DXVECTOR2 vec2Position(fPosX,fPosY); の部分です。vec2Position はスプライトの表示座標を格納する 2 次元ベクトルです。2 章では定数を引数にしていました、それではスプライトを移動させることは不可能なので変数を引数として動的に座標を変更・更新できるようにします。スプライトを移動させたいときは、vec2Position を新たな値で更新します、この場合は fPosX,fPosY の値を変化させるということです。
FposX の値を増やせばスプライトは右に移動、減らせば左に移動します。同様に fPosY の値を増やせばスプライトは下に移動、減らせば上に移動します。
なぜ、増やせば右や上に移動するのかというと、DirectX におけるモニター画面の座標は左上が座標 (0,0) と定義されているからです。

以上がスプライト移動のメカニズムです。簡単ですよね。
あと、先に書いた DirectX オブジェクトの開放を行う関数 FreeDx 関数はただ単に SAFE_RELEASE マクロにより全オブジェクトを開放しているだけです。これも問題ないでしょう。

DirectInput を利用した入力処理の場合

サンプルプロジェクト名：Chapter3-2

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3: #include <dinput.h>
 4:
 5: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 6:
 7: LPDIRECT3D9 pD3d;
 8: LPDIRECT3DDEVICE9 pDevice;
 9: LPDIRECT3DTEXTURE9 pTexture;
10: LPD3DXSPRITE pSprite;
11: LPDIRECTINPUT8 pDinput=NULL;
12: LPDIRECTINPUTDEVICE8 pKeyDevice=NULL;
13: LPD3DXFONT m_pFont;
14: FLOAT fPosX=270,fPosY=180;
15:
16: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
17: HRESULT InitD3d(HWND);
18: HRESULT InitDinput(HWND);
19: VOID AppProcess();
20: VOID FreeDx();
21:
22: //
23: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
24: // アプリケーションのエントリー関数
25: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
26: {
27:       HWND hWnd=NULL;
28:       MSG msg;
29:       // ウィンドウの初期化
30:       static char szAppName[] = "Chapter3-2" ;
31:     WNDCLASSEX  wndclass ;
32:
33:     wndclass.cbSize          = sizeof (wndclass) ;
34:     wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
35:     wndclass.lpfnWndProc     = WndProc ;
36:     wndclass.cbClsExtra      = 0 ;
37:     wndclass.cbWndExtra      = 0 ;
38:     wndclass.hInstance       = hInst ;
39:     wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
40:     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
41:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
42:     wndclass.lpszMenuName = NULL ;
43:     wndclass.lpszClassName = szAppName ;
44:     wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
45:
46:     RegisterClassEx (&wndclass) ;
```

```
47:
48:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
49:                     0,0,640,480,NULL,NULL,hInst,NULL) ;
50:     ShowWindow (hWnd,SW_SHOW) ;
51:     UpdateWindow (hWnd) ;
52:       // ダイレクト3D の初期化関数を呼ぶ
53:       if(FAILED(InitD3d(hWnd)))
54:       {
55:                     return 0;
56:       }
57:       // ダイレクトインプットの初期化関数を呼ぶ
58:       if(FAILED(InitDinput(hWnd)))
59:       {
60:                     return 0;
61:       }
62:       // メッセージループ
63:     ZeroMemory( &msg, sizeof(msg) );
64:     while( msg.message!=WM_QUIT )
65:     {
66:                      if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
67:         {
68:                                 TranslateMessage( &msg );
69:             DispatchMessage( &msg );
70:         }
71:         else
72:                     {
73:                                 AppProcess();
74:                     }
75:       }
76:       // メッセージループから抜けたらオブジェクトを全て開放する
77:       FreeDx();
78:       // OS に戻る （アプリケーションを終了する）
79:     return (INT)msg.wParam;
80: }
81:
82: //
83: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
84: // ウィンドウプロシージャ関数
85: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
86: {
87:       switch(iMsg)
88:       {
89:                     case WM_DESTROY:
90:                                 PostQuitMessage(0);
91:                     break;
92:                     case WM_KEYDOWN:
93:                     switch((CHAR)wParam)
94:                     {
95:                                 case VK_ESCAPE:
96:                                             PostQuitMessage(0);
97:                                 break;
98:                     }
99:                     break;
100:      }
101:      return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
102: }
103:
104: //
105: //HRESULT InitD3d(HWND hWnd)
106: // ダイレクト 3D の初期化関数
107: HRESULT InitD3d(HWND hWnd)
108: {
109:      //「Direct3D」 オブジェクトの作成
110:      if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
111:      {
112:                     MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
113:       return E_FAIL;
114:      }
115:      //「DIRECT3D デバイス」 オブジェクトの作成
116:      D3DPRESENT_PARAMETERS d3dpp;
```

```
    ZeroMemory( &d3dpp, sizeof(d3dpp) );

    d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
    d3dpp.BackBufferCount=1;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.Windowed = TRUE;

     if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                D3DCREATE_MIXED_VERTEXPROCESSING,
                                &d3dpp, &pDevice ) ) )
  {
            MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
            if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
                                D3DCREATE_MIXED_VERTEXPROCESSING,
                                &d3dpp, &pDevice ) ) )
            {
                    MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
                     return E_FAIL;
            }
  }
    //「テクスチャオブジェクト」の作成
    if(FAILED(D3DXCreateTextureFromFileEx(pDevice,"Sprite.bmp",100,100,0,0,     D3DFMT_UNKNOWN,
                                                                     D3DPOOL_DEFAULT,D3DX_FILTER_NONE,D3DX_DEFAULT,
                                                            0xff000000,NULL,NULL,&pTexture)))
    {
            MessageBox(0," テクスチャの作成に失敗しました ","",MB_OK);
            return E_FAIL;
    }
    //「スプライトオブジェクト」の作成
    if(FAILED(D3DXCreateSprite(pDevice,&pSprite)))
    {
            MessageBox(0," スプライトの作成に失敗しました ","",MB_OK);
            return E_FAIL;
    }
    return S_OK;
}
//
//HRESULT InitDinput(HWND hWnd)
// ダイレクトインプットの初期化関数
HRESULT InitDinput(HWND hWnd)
{
    HRESULT hr;
    //「DirectInput」オブジェクトの作成
     if( FAILED( hr = DirectInput8Create( GetModuleHandle(NULL),
             DIRECTINPUT_VERSION,IID_IDirectInput8, (VOID**)&pDinput, NULL ) ) )
     {
             return hr;
     }
  //「DirectInput デバイス」オブジェクトの作成
  if( FAILED( hr = pDinput->CreateDevice( GUID_SysKeyboard,
            &pKeyDevice, NULL ) ) )
    {
            return hr;
    }
    // デバイスをキーボードに設定
  if( FAILED( hr = pKeyDevice->SetDataFormat( &c_dfDIKeyboard ) ) )
    {
            return hr;
    }
    // 協調レベルの設定
  if( FAILED(hr= pKeyDevice->SetCooperativeLevel(
            hWnd,DISCL_NONEXCLUSIVE | DISCL_BACKGROUND )) )
    {
            return hr;
    }
    // デバイスを「取得」する
  pKeyDevice->Acquire();
    return S_OK;
}
```

```
186:
187: //
188: //VOID AppProcess()
189: // アプリケーション処理関数
190: VOID AppProcess()
191: {
192:       // スプライトの描画
193:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET,D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
194:       if( SUCCEEDED( pDevice->BeginScene() ) )
195:       {
196:                   RECT rect={0,0,100,100};
197:                   D3DXVECTOR2 vec2Scale(1.0,1.0);
198:                   D3DXVECTOR2 vec2RotationCenter(1.0,1.0);
199:                   D3DXVECTOR2 vec2Position(fPosX,fPosY);
200:                   pSprite->Draw(pTexture,&rect,&vec2Scale,&vec2RotationCenter,0,&vec2Position,0xffffffff);
201:                   pDevice->EndScene();
202:       }
203:       //  キーボードで押されているキーを調べ、対応する方向に移動させる
204:       HRESULT hr=pKeyDevice->Acquire();
205:       if((hr==DI_OK) || (hr==S_FALSE))
206:       {
207:                   BYTE diks[256];
208:                   pKeyDevice->GetDeviceState(sizeof(diks),&diks);
209:
210:                   if(diks[DIK_LEFT] & 0x80)
211:                   {
212:                               fPosX-=4;
213:                   }
214:                   if(diks[DIK_RIGHT] & 0x80)
215:                   {
216:                               fPosX+=4;
217:                   }
218:                   if(diks[DIK_UP] & 0x80)
219:                   {
220:                               fPosY-=4;
221:                   }
222:                   if(diks[DIK_DOWN] & 0x80)
223:                   {
224:                               fPosY+=4;
225:                   }
226:       }
227:       pDevice->Present( NULL, NULL, NULL, NULL );
228: }
229: //
230: //VOID FreeDx()
231: // 作成した DirectX オブジェクトの開放
232: VOID FreeDx()
233: {
234:     if(pKeyDevice)
235:       {
236:                   pKeyDevice->Unacquire();
237:       }
238:       SAFE_RELEASE( pKeyDevice );
239:     SAFE_RELEASE( pDinput );
240:       SAFE_RELEASE( pTexture );
241:       SAFE_RELEASE( pSprite );
242:       SAFE_RELEASE( pDevice );
243:       SAFE_RELEASE( pD3d );
244: }
245:
```

ウィンドウメッセージのみを入力情報とすると、先に述べたようにレスポンスが悪く、まともなアクションゲームを作ろうとした場合には、パワー不足を感じる時があるかもしれません。
DirectInput であれば、十分な速さで入力情報を提供してくれます。まさに、DirectX がゲーム API として存在する理由を感じることができる局面の一つでしょう。
本サンプルは Chapter3-1.exe と入力処理部分のみ異なるので、異なる部分のみ解説していきます。
追加された主な箇所は、156 行～ 185 行目の InitDinput 関数です。あとの変更点は DirectInput を実装したことに伴う、DirectInput 関連のインターフェイスポインタの定義や初期化関数呼び出し等です。

11 行目と 12 行目　インターフェイスポインタを定義
LPDIRECTINPUT8 型の変数 pDinput を定義。これは DirectInput オブジェクトのポインターです。DirectInput オブジェクトはすべての DirectInput 関連のインターフェイスの元になるルートのオブジェクトです。ちょうど Direct3D オブジェクトが Direct3D 関連のインターフェイスのルート（元）になっていたのと同様です。
LPDIRECTINPUTDEVICE8 型の変数 pKeyDevice を定義。これは DirectInput デバイスオブジェクトのポインターです。入力情報を取得するにはマウスまたはキーボード用のデバイスを作成する必要があります。

156 行～ 185 行目　InitDinput 関数。ダイレクトインプットの初期化関数です。
160 行目と 161 行目
IDirectInputDevice8::CreateDevice メソッドにより DirectInput オブジェクトを作成します。これは、このまま覚えても他のコードでも使用できるでしょう。第 3 引数にポインターのアドレスを書きます。
166 行目と 167 行目
DirectInput デバイスオブジェクトを作成します。本サンプルはキーボードのみを使用しますので、キーボード用のデバイスを作成しています。同時にマウスも使用する場合はマウス用に別のデバイスオブジェクトを作成することになります。
キーボードの場合は、第 1 引数は GUID_SysKeyboard となります。（ちなみにマウスの場合は GUID_SysMouse です）第 2 引数にはポインターのアドレスを書きます。
172 行目
IDirectInputDevice8::SetDataFormat を実行します。
177 行目と 178 行目
IDirectInputDevice8::SetCooperativeLevel メソッドにより協調レベルを設定します。
デバイスには「協調レベル」という属性を設定できます。協調レベルとは何かというと、一言で言えばデバイス（本サンプルの場合はキーボード）を当該アプリケーションがどの程度占有するかということです。
協調レベルのフラグは、排他的か非排他的かということと、フォアグラウンドかバックグラウンドかという 2 つの動作について 2 つずつ、計 4 種類があり、したかがって組み合わせは全部で 4 種類あります。（4 種類しかありません）
具体的に書くと、
1 DISCL_NONEXCLUSIVE | DISCL_BACKGROUND　非排他でバックグラウンド
2 DISCL_NONEXCLUSIVE | DISCL_FOREGROUND　非排他でフォアグラウンド
3 DISCL_EXCLUSIVE | DISCL_BACKGROUND　　排他でバッググランド
4 DISCL_EXCLUSIVE | DISCL_FOREGROUND　　排他でフォアグラウンド
の 4 通りになります。
実はあと一つ DISCL_NOWINKEY というフラグがあります。これはキー入力に関するウィンドウメッセージを無効にするものですが、あまり用途がないと思われるので説明から除きました。
排他・非排他、フォアグラウンド・バックグランド…はっきり言って（今は）あまり気にすることではありません、なぜなら、排他にしたからといって他のアプリケーションが同じデバイスを取得できないとは限らず、また、バックグラウンド（これはフォアグラウンド、バックグラウンドにかかわらず常にデバイスへアクセス出来るというフラグです）にしてもデバイスへのアクセスを簡単に失うからです。フラグは絶対的ではなく、「出来ればそのようなモードにしたい」ということを DirectInput に通知しているにすぎません。
敢えて言うなら、最初のうちは排他・非排他は「非排他」にしたほうがいいかもしれない、ということぐらいでしょうか。アプリケーションがウィンドウモードである場合に排他モードにするとクライアント領域の外ではカーソルが消えることがあるからです。
最初は常に DISCL_NONEXCLUSIVE | DISCL_BACKGROUND（非排他でバックグラウンド）と指定してもいいくらいです。本サンプルでもこのフラグ組み合わせを使用しています、また、この組み合わせは SetCooperativeLevel メソッドのデフォルト設定になっています。
183 行目
IDirectInputDevice8::Acquire メソッドにより、デバイスを取得しています。
デバイスの「取得」とはなんのことなのでしょう？
DirectInput においては、デバイスの「作成」と「取得」という 2 つの動作が定義されています。「作成」は先に書いた IDirectInputDevice8::CreateDevice により行いました。デバイスを作成しただけではデバイスにアクセスできず入力情報を得ることができません。作成したデバイスへのアクセス権を得る必要があります。取得とはアクセス権を取得することです。当該デバイスへのアクセス権を取得する手段が Acquire メソッドです。
DirectInput では、アクセス権は、常にあるものではなく、一般的にアプリケーションは実行中に何度もアクセス権を失います。アプリケーションはアクセス権を失ったときはこの Acquire メソッドを実行してデバイスを再取得します。
204 行目～ 236 行目
この部分が、入力情報によりスプライトの座標を更新している部分です。
まず、デバイスへのアクセス権を失っている場合を想定して Acquire を実行します。これで確実にデバイスへアクセスできます。Acquire は取得に成功すれば、DI_OK を既に取得済みであれば S_FALSE を返します、すでに取得済みである場合に Acquire を実行しても問題はありません。
BYTE diks[256]　デバイスがキーボードの場合、入力データの受け皿（DirectInput に入力情報を格納させる変数・構造体・配列）は BYTE 型で要素数 256 の配列となります。
あとは、IDirectInputDevice8::GetDeviceState メソッドにその配列の（先頭）ポインターを渡してコールすれば入力情報をその配列に入れてくれます。
ここまでで、diks 配列には、キーボードの押下情報が入っています。ここで情報の仕様の説明をする必要があるでしょう。あるキーが押されている場合、そのキーに対応した要素に値が入ります。キーボードの 0 キーなら 0 番目の要素に、A キーなら 10 番目の要素という具合です。DirectInput では開発者の便宜のために要素に対してあらかじめ序数が定義されています。DIK_ というプレフィックスではじまる序数は 0 キーが DIK_0　A キーが DIK_A　左矢印キーが DIK_LEFT という風に単なる数字よりわかり易いよう配慮されていますので、それを利用してもいいでしょう。本サンプルでは DIK_LEFT DIK_RIGHT DIK_

UP DIK_DOWN の 4 つの矢印キーに対応した序数を使用しています。
そして、キーが押されている場合は値の上位ビットが 1 になります、上位ビットが 1 か 0 を判断するにはビット演算子を使い本サンプルのように (値) & 0x80　とします。0x80　は BYTE 値最上位ビットのマスク値であるので 0x80 とのビット論理積が真であれば上位ビットが 1 すなわちキーが押されているということになります。なお、0x80 は 16 進数ですが、10 進数で書くと 128 ですので（値）& 128 としても結果は全く同じです。
if(diks[DIK_LEFT] & 0x80){　　fPosX-=4;　} もし左矢印キーが押されている場合はスプライトの X 座標を左に 4 ピクセル進めます。その他 3 つのキーに対しても同様の処理を行い上下左右に移動させています。

以上、Win32 メッセージと DirectInput それぞれによる操作処理を紹介しました。
Win32 メッセージと DirectInput のレスポンスの違いについても述べました。もっとも Win32 メッセージを利用してもマシンが十分に速ければ問題が露呈しない場合も考えられます（こうゆうことが言えるようなマシン環境を筆者は 2 , 3 年前には想像できませんでしたが…）、しかし、“ レスポンス差 ” 自体はマシン環境がよくなっても常に存在しますので、入力情報を高速に得たいのであれば DirectInput を利用するということになるでしょう。

Chapter4　3D 物体の表示

これまでの章ではスプライトすなわち 2 次元物体を扱っていました。2D の場合は表示するまでのコード及びコードの背景となる概念は、3D に比べると単純で覚えやすく、感覚的に解りやすいものと言えます。3D 物体の表示は、表示処理に辿り着くまでのコードが 2D に比べると複雑で、また、コードを書く上で必要となる背景理論にも多少の数学的理論が含まれます。数学的理論とは、主に線形代数理論のことです。高校でベクトル・行列を履修した覚えのある読者も多いと思いますが、それも線形代数のカテゴリーに含まれます。具体的に言えば、「表示」するだけであれば、最低限、ベクトルと行列の知識、さらに、「物体間の衝突検出」（当たり判定と書いた方が、馴染みがあるでしょうか）まで実装するのであれば、さらに最低限、ベクトルの内積・外積、点・直線・平面方程式の導出とその利用…等の知識が不可欠になります。ゲームの場合は表示するだけではなく物体間の相互作用を何らかの方法で検出する場合が殆どかと思われます。物体間の相互作用について具体例を挙げると、自由落下した物体が地面と衝突しているかどうか、さらに地面に衝突した後の跳ね返り方といったようなものであり、この場合、簡単な物理法則も使用することになります。物体間の衝突検出や物理法則による相互作用はダイナミクスと言います。Direct3D はダイナミクスまでは提供していません、Direct3D が提供しているのはあくまで「表示」までであり、その表示した物のダイナミクスに関しては開発者が自分で実装しなくてはなりません。
本章では表示までを目的としています。
ダイナミクスは本章及び本セクションの目的ではありませんし、初心者がいきなりダイナミクスと言えるようなシステムを実装するのは困難かと思われます。しかし、ゲームを開発する上でダイナミクスは避けては通れないものであり避けて通ってはならないものと筆者は考えますので、ダイナミクス構築に必要な数学理論の利用例および、それらを用いたサンプルプログラムはセクション 3 で重点的に解説します。
さて、3D 物体の表示を実現するわけですが、Direct3D における 3D 物体の表示手続きには大きく分けて 2 種類あり、実装方法も異なります。本章では両方の実装を解説します。
1 つ目の手法は頂点を直接作成・設定して、平面や立方体を作成し表示するというものです。
2 つ目の手法は 3D 物体の情報を記録した X ファイルというファイルを読み込んで表示するというものです。
厳密に言うと 2 つ目の手法は 1 つ目の手法を高度に自動化・ライブラリ化しただけのものであり内部では 1 つ目の手法を使用しているのですが、実務上、異なる手法と考えてもいいほど手続き（コード）は異なります。
まずは、1 つ目の手法から解説します。

頂点を直接的に設定して 3D 物体を表示する手法

サンプルプロジェクト名：Chapter4-1

```
#include <windows.h>
#include <d3dx9.h>

#define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)

struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;
};

LPDIRECT3D9 pD3d;
LPDIRECT3DDEVICE9 pDevice;
LPDIRECT3DVERTEXBUFFER9 pVB = NULL;

LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
HRESULT InitD3d(HWND);
VOID Render();
VOID FreeDx();

//
//INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
// アプリケーションのエントリー関数
INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
{
    HWND hWnd=NULL;
    MSG msg;
    // ウィンドウの初期化
    static char szAppName[] = "Chapter4-1" ;
    WNDCLASSEX  wndclass ;

    wndclass.cbSize         = sizeof (wndclass) ;
    wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc    = WndProc ;
```

```
36:     wndclass.cbClsExtra      = 0 ;
37:     wndclass.cbWndExtra      = 0 ;
38:     wndclass.hInstance       = hInst ;
39:     wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
40:     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
41:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
42:     wndclass.lpszMenuName = NULL ;
43:     wndclass.lpszClassName = szAppName ;
44:     wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
45:
46:     RegisterClassEx (&wndclass) ;
47:
48:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
49:                 0,0,640,480,NULL,NULL,hInst,NULL) ;
50:    ShowWindow (hWnd,SW_SHOW) ;
51:    UpdateWindow (hWnd) ;
52:      // ダイレクト3D の初期化関数を呼ぶ
53:      if(FAILED(InitD3d(hWnd)))
54:      {
55:                  return 0;
56:      }
57:      // メッセージループ
58:    ZeroMemory( &msg, sizeof(msg) );
59:    while( msg.message!=WM_QUIT )
60:    {
61:                   if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
62:       {
63:                               TranslateMessage( &msg );
64:           DispatchMessage( &msg );
65:       }
66:       else
67:                   {
68:                               Render();
69:                   }
70:       }
71:       // メッセージループから抜けたらオブジェクトを全て開放する
72:       FreeDx();
73:       // OS に戻る （アプリケーションを終了する）
74:    return (INT)msg.wParam ;
75: }
76:
77: //
78: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
79: // ウィンドウプロシージャ関数
80: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
81: {
82:       switch(iMsg)
83:       {
84:                   case WM_DESTROY:
85:                               PostQuitMessage(0);
86:                   return 0;
87:                   case WM_KEYDOWN:
88:                   switch((CHAR)wParam)
89:                   {
90:                               case VK_ESCAPE:
91:                                           PostQuitMessage(0);
92:                               return 0;
93:                   }
94:                   break;
95:       }
96:       return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
97: }
98:
99: //
100: //HRESULT InitD3d(HWND hWnd)
101: // ダイレクト 3D の初期化関数
102: HRESULT InitD3d(HWND hWnd)
103: {
104:       //「Direct3D」 オブジェクトの作成
105:       if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
```

```
106:     {
107:             MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
108:       return E_FAIL;
109:     }
110:     //「DIRECT3D デバイス」オブジェクトの作成
111:     D3DPRESENT_PARAMETERS d3dpp;
112:     ZeroMemory( &d3dpp, sizeof(d3dpp) );
113:
114:     d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
115:     d3dpp.BackBufferCount=1;
116:     d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
117:     d3dpp.Windowed = TRUE;
118:
119:      if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
120:                         D3DCREATE_MIXED_VERTEXPROCESSING,
121:                         &d3dpp, &pDevice ) ) )
122:   {
123:             MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
124:             if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
125:                         D3DCREATE_MIXED_VERTEXPROCESSING,
126:                         &d3dpp, &pDevice ) ) )
127:             {
128:                   MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
129:                    return E_FAIL;
130:             }
131:   }
132:
133:     // 頂点の初期値を設定
134:     CUSTOMVERTEX vertices[] =
135:   {
136:       { -1.0f,-1.0f, 0.0f, 0xffff0000, },
137:       {  1.0f,-1.0f, 0.0f, 0xff00ff00, },
138:       {  0.0f, 1.0f, 0.0f, 0xff0000ff, },
139:   };
140:     // 頂点バッファーを作成
141:   if( FAILED( pDevice->CreateVertexBuffer( sizeof(vertices),
142:                                  0, D3DFVF_CUSTOMVERTEX,
143:                                  D3DPOOL_DEFAULT, &pVB, NULL ) ) )
144:   {
145:             MessageBox(0," 頂点バッファーの作成に失敗しました ","",MB_OK);
146:     return E_FAIL;
147:   }
148:     // 頂点バッファーに頂点を設定 （コピー） する
149:   VOID* pVertices;
150:   if( FAILED(pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 ) ) )
151:     {
152:             MessageBox(0," 頂点バッファーのロックに失敗しました ","",MB_OK);
153:       return E_FAIL;
154:     }
155:   memcpy( pVertices, vertices, sizeof(vertices) );
156:   pVB->Unlock();
157:     // カリングはオフにする （つまり、背面もレンダリングするということ）
158:     pDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
159:     // ライトをオフにする
160:   pDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
161:
162:     return S_OK;
163: }
164:
165: //
166: //VOID Render()
167: // 頂点バッファーの頂点をレンダリングする関数
168: VOID Render()
169: {
170:
171:     // ワールドトランスフォーム （絶対座標変換）
172:      D3DXMATRIXA16 matWorld;
173:      D3DXMatrixIdentity(&matWorld);
174:    pDevice->SetTransform( D3DTS_WORLD, &matWorld );
175:      // ビュートランスフォーム （視点座標変換）
```

```
176:
177:      D3DXVECTOR3 vecEyePt( 0.0f, 1.0f,-3.0f ); // カメラ（視点） 位置
178:      D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
179:      D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
180:      D3DXMATRIXA16 matView;
181:      D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
182:      pDevice->SetTransform( D3DTS_VIEW, &matView );
183:        // プロジェクショントランスフォーム（射影変換）
184:      D3DXMATRIXA16 matProj;
185:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
186:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
187:        // レンダリング
188:        pDevice->Clear( 0, NULL, D3DCLEAR_TARGET,D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
189:        if( SUCCEEDED( pDevice->BeginScene() ) )
190:        {
191:                    pDevice->SetStreamSource( 0, pVB, 0, sizeof(CUSTOMVERTEX) );
192:          pDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
193:          pDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
194:
195:                    pDevice->EndScene();
196:        }
197:        pDevice->Present( NULL, NULL, NULL, NULL );
198: }
199:
200: //
201: //VOID FreeDx()
202: // 作成した DirectX オブジェクトの開放
203: VOID FreeDx()
204: {
205:      SAFE_RELEASE( pVB );
206:      SAFE_RELEASE( pDevice );
207:      SAFE_RELEASE( pD3d );
208: }
```

頂点による物体の表現は Direct3D のみならず OpenGL 環境でも基本原理であり、低レベル※な操作です。一般的に低レベルな操作は基本的な処理単位から成るため、自分で考案したユニークな処理を柔軟に実装することができる可能性がある半面、そのような処理を実装しようとする場合、多くのコードを自前で書かなくてはならないという側面があります。
この場合も例外ではなく、頂点を直接設定しての 3D 物体の表示は低レベルな処理であり、例えば人体をレンダリングしようとした場合に多くの頂点全てを設定して表示しなくてはならず、更にそれをアニメーションさせようとした場合はより複雑なコードを自分で実装する必要があります。Chapter4-1 サンプルは人体のような頂点数が多い複雑な形状の物体ではなく、頂点数がたった 3 つの三角形をレンダリングするというものなのでコードはシンプルです。なお、人体のような頂点数の比較的多い物体の頂点データは特に「メッシュ」※と呼ばれ、メッシュのレンダリングをするのが 2 つ目の手法であり、それはその部分で解説します。
ではサンプルコードを追いながら解説していきましょう。
まず、このサンプルの出力結果は単純な三角形を描画するだけのものです。三角形の 3 つの頂点にはそれぞれ別々の色を設定しています。なぜ、3 角形かと言うと、3 角形が最も短いコードでシンプルに解説できるからです。この 3 角形はポリゴンです、本サンプルはたった 1 枚のポリゴンをレンダリングしているプログラムとも言えます。人体等のモデルをレンダリングするには多くのポリゴンを必要としますが、おおもとの原理は本サンプルであり、理論的には本サンプルの手法が元になります。この手法を元に人体をレンダリングするために言うまでも無くコードはより複雑になり圧倒的にコード量は増えることが容易に想像されますが、それはボリュームの問題であり、原理が変わるわけではありません。
本セクションはステップアップ形式なのでコードはそれまでのものに追加する形になっています。WinMain 関数及び WndProc 関数については今まで何回も登場してきていますし、同一なので問題ないでしょう。InitD3d 関数もデバイスの作成までは同一です。今までと異なる部分または追加された部分のみ解説します。

5 行目　頂点の性質を定義
D3DFVF_CUSTOMVERTEX という定数を定義します。今回はローカル座標とディフューズ（色）という 2 つの性質だけを頂点に持たせるので D3DFVF_XYZ と D3DFVF_DIFFUSE 定数を組み合わせたものをマクロ定義しています。本サンプルのように極限まで単純化したコードでは、別個に定義するまでもなく直接 D3DFVF_XYZ「ローカル座標」と D3DFVF_DIFFUSE「ディフューズ（色）」を使用すればいいのですが、後々の章との整合性も考え一応定義することにしました。ローカル座標、ディフューズ、頂点の性質や設定に関しては本章後の部分で説明しますので今は取りあえずさらっと流してください。
6 行目～ 10 行目　独自の頂点構造体を定義
　ローカル座標とディフューズ色をメンバーに持つ構造体を定義しています。
15 行目　頂点バッファーポインタの宣言
　頂点は頂点バッファーというバッファーに格納します。これは、その頂点バッファーへのポインターです。
134 行目～ 156 行目　頂点関係の初期設定

頂点の初期値を設定

先に定義しておいた独自の頂点用構造体を vertices として宣言（実体化）し、それに値を設定します。3 つの頂点についてそれぞれ座標と色を、一番上から順に赤、緑、青として設定しています。

頂点バッファーを作成

IDirect3Ddevice9::CreateVertexBuffer メソッドは、頂点バッファーを作成します。バッファーとは一定に確保されたメモリー領域のことで、一般的には、なんらかの値を格納するための単純な " 倉庫 " です。頂点バッファーも頂点情報を格納するための単純な倉庫であり、単なるメモリー領域です。

第 1 引数
バイト単位の必要バッファーサイズを指定します。今回は vertices 構造体のサイズを渡します。

第 2 引数
当面はゼロを指定します。

第 3 引数
FVF 定数という定数が Direct3D であらかじめ定義されています。FVF 定数とは何かというと、頂点の性質・属性を現す個々の定数のことであり、先ほどローカル座標及びディフューズ情報を頂点にバインドするということに触れましたが、それらの性質を指定する際に使用する定数が FVF 定数です。具体的には、例えば D3DFVF_DIFFUSE はディフューズを意味し、この定数が指定されると頂点はディフューズ特性を持った頂点ということになります。他に、例えば D3DFVF_SPECULAR を指定すれば頂点はスペキュラ（反射光）情報を持つことを意味するといった具合です。定数はそれぞれ論理和により組み合わせることが出来ます。本サンプルでも D3DFVF_XYZ | D3DFVF_DIFFUSE もように 2 つの定数を組み合わせて指定しています。（D3DFVF_CUSTOMVERTEX は D3DFVF_XYZ | D3DFVF_DIFFUSE を意味する単なるマクロであることを思い出してください）

第 4 引数
当面はこのままにしておきましょう。

第 5 引数
先に宣言しておいた頂点バッファーへのポインターのアドレスを渡します。

第 6 引数
Direct3D に予約されている引数で、常に NULL(0) とします。

頂点バッファーに頂点を設定（コピー）する

頂点バッファーの作成に成功した直後はバッファーにはデータはなにも入っていません。つまり空の倉庫です。バッファーに実際の頂点データを入れる必要があります。具体的には vertices 構造体のデータ（頂点データ）を全て入れます。まず、データを格納する前に、バッファーをロックしなければなりません。ロックとはバッファーへのアクセス権を得ることであり、ロックしなければバッファー内データの読み書きが出来ません。
ロックは IDirect3DVertexBuffer9::Lock メソッドにより行います。

第 1 引数
ロックを部分的に行う場合以外は、ゼロを指定します。この引数はロックを部分的に行う場合、任意の頂点データへのオフセット（バッファー先頭からその頂点データまでのバイト単位の距離）を指定するものです。

第 2 引数
頂点データの全体サイズ。全体をロックするのであればゼロを指定することも出来ます。今回は vertices 構造体のサイズを渡していますが、ゼロを渡しても同じです。

第 3 引数
バッファーへの適正なポインターを入れる void 型の変数のアドレスを渡します。メソッドが成功すればその変数はバッファーを指し示すポインターとなります。

第 4 引数
ロック形態のフラグを指定します。当面はゼロを指定し、デフォルトでロックしましょう。
ロックが成功したら、バッファーに頂点データを格納します。memcpy 関数によりバッファー (pVertices はバッファーを指し示すポインターです) に頂点データ (vertices 構造体) をコピー（格納）しています。ロックの目的が終了したら必ず IDirect3DVertexBuffer9::Unlock メソッドによりバッファーをアンロックします。

157 行目～ 160 行目 本サンプル用にレンダリングステートを調整
IDirect3Ddevice::SetRenderState メソッドは初めて登場した関数ですので軽く説明します。
この関数はレンダリングに関する様々な動作を変更するものです。引数により実に多くの動作を変更・制御します。第 1 引数により " 何を "、第 2 引数により " どのように " 変更するかを Direct3D に通知します。
SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
D3DRS_CULLMODE「カリング」を D3DCULL_NONE「オフにする」という意味です。カリングとはジオメトリ（この場合はポリゴン）の背面をレンダリング " しない " ことによりパフォーマンスを上げることです。したがって " カリングをしない " ということはポリゴンの " 背面もレンダリングする " ということになります。
SetRenderState(D3DRS_LIGHTING, FALSE);
D3DRS_LIGHTING「ライト」を FALSE「オフ」にするという意味です。本サンプルの頂点は頂点自身が発光色を持っている為、ライトがオンになっているとかえって不具合（真っ黒にレンダリングされる）が起きますので、事前にオフにしておきます。

165 行目～ 204 行目　頂点バッファーの頂点をレンダリングする関数
画面表示関数を Draw ではなく Render と名付けたのは、2D の描画と区別するためですが、深い意味合いはありません。
筆者は日常的に、2D は "Draw「描画」"3D は "Render「レンダー」" と意識的に言葉を分けています。特にこれといって意味は無く、もちろん分けなくても不具合はありませんし、正しい分けなのかすら解りませんが、なんとなく分けています。おそ

らく、2D の場合は “ ただ単純にピクセルを画面に表示する ” というイメージがあり、一方、3D は “ 幾何学計算をし、投影変換した後でようやく画面にラスタライズできる ” というイメージから 2D と同じ “ 描画 ” とい言葉を使うより、分けたほうがいいという曖昧な気持ちが理由だと思います。最終的なラスタライズ過程だけを見ると “ 描画 ” と表現してもいいのですが、そこにたどり着くまでの多くの計算が存在するという意味を含めて “ レンダリング ” と表現しているような気がします。何れにしても、どうでもいい事柄であり、筆者の個人的なことなので、コードの解説に戻ります。
さて、これから述べることは大変重要な基本原理であり、コード的側面で言うと必ず実装されるものであると同時に頻繁にソースコード内に登場するものです。
Direct3D では (OpenGL でもそうです) ジオメトリをレンダリングする際には、３つのトランスフォームを行います。３つのトランスフォームとは何かと言うと、

1. ワールドトランスフォーム
2. ビュートランスフォーム
3. プロジェクショントランスフォーム

の３つです。日本語で言うと１絶対座標変換、２視点座標変換、３射影変換と言えばいいでしょうか。トランスフォームは「変換」するという意味です。では、“ 何を ” 変換するというのでしょうか？ 2D のように単純にビットマップ等を画面に描画できないのでしょうか？
これら疑問の答えを得るために、３つの “ トランスフォーム「変換」” について詳しく解説しましょう。

１．ワールドトランスフォーム

ワールドトランスフォームとは、ジオメトリ※が個々に持つローカル座標をワールド（絶対）座標に変換することを言います。ローカル座標と絶対座標の意味及びローカル座標を絶対座標に変換しなければならない理由は次の通りです。

ローカル座標とは

ローカル座標とはジオメトリが個々に持つ座標系及び座標を意味します。普通、ゲーム中には、複数のジオメトリ（人間、建物、武器、アイテム…等）が存在しますが、その場合、ゲーム内にはジオメトリの数だけローカル座標系が複数存在することになります。ローカル座標の原点は、普通、ジオメトリの中心に置きますが、意図的に原点を中心からずらした場所に設定する場合もあります。ローカル座標の原点は 3DCG ソフトのオブジェクトデータの観点から見ればピボットポイントとも呼ばれます。正 6 面体を例にすると図 4- 1 のようになります。図 4-2 は視点を XY 平面と平行にした時のものです。

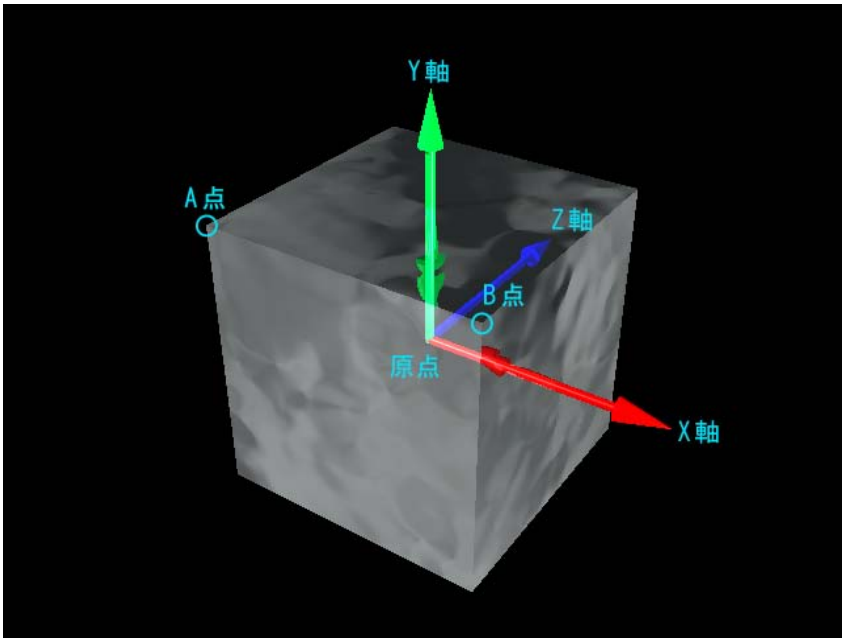


図 4-1

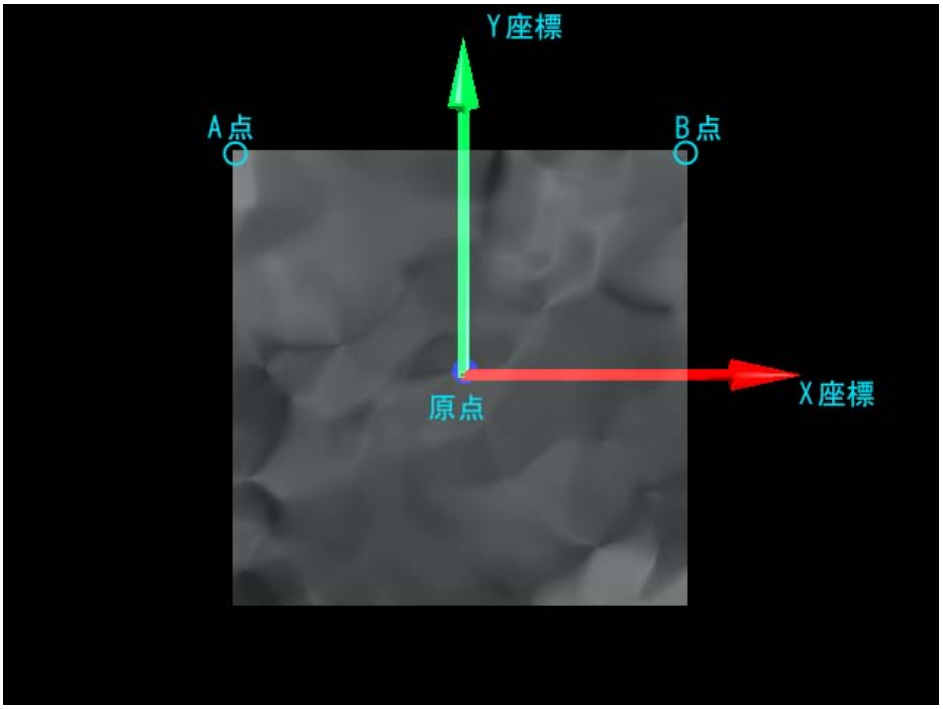
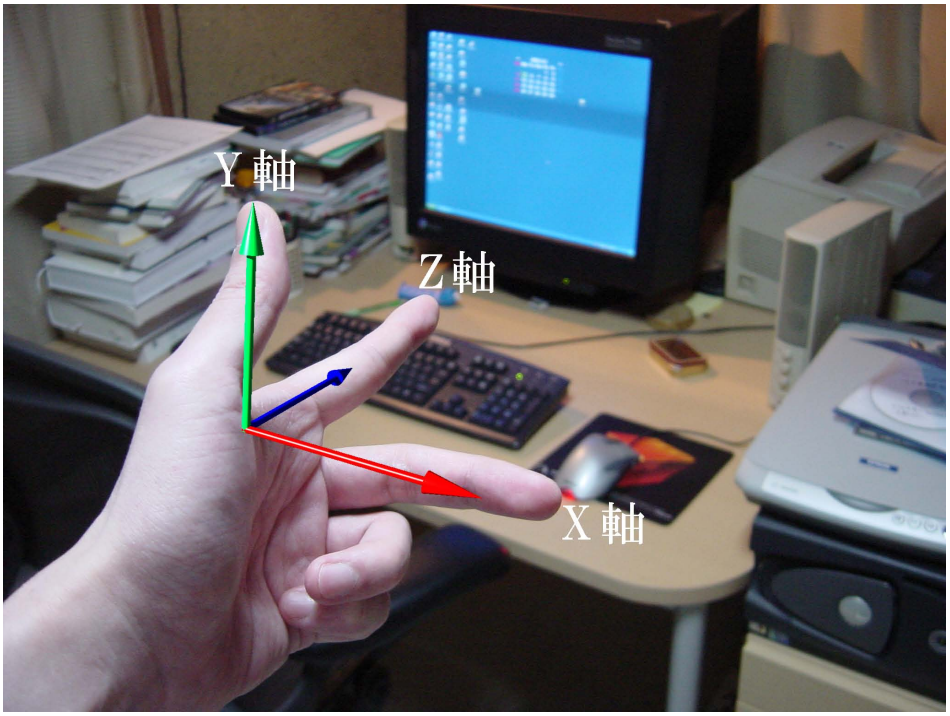


図 4-2

このジオメトリのサイズは縦、横、奥行きそれぞれ全てが 2 です。2m と考えてもいいですし、2km と考えても構いません（単位は無関係です）、とにかく全ての軸について長さが 2 の正 6 面体です。そして、中心はローカル座標（0,0,0）にあります。したがって、A 点のローカル座標は（-1,1,-1）で、B 点のローカル座標は（1,1,-1）となります。
なお、Direct3D 環境は左手座標系なので Z 軸は奥の方に行くにしたがってプラスになります。A 点 B 点共に手前の頂点なので Z 座標は－ 1 です。左手座標系とは左手を図 4 － 3 のように突き出した時に薬指が X 軸、親指が Y 軸、人差し指が Z 軸に対応し、指先方向がプラス方向を意味する座標系のことです。（ちなみに OpenGL や他の多くの環境は右手座標系です）



（筆者の仕事机前にて近影）
図 4 － 3

ワールド（絶対）座標とは

絶対座標とは、ゲーム内にただ一つのみ存在する座標系であり、全てのローカル座標の基準になります。ローカル座標が個々のジオメトリに附帯し、ジオメトリの数だけ存在していたのとは対照的です。絶対座標という概念が無いと、ジオメトリの配

置や移動さらには衝突判定等がまったく不可能になります。ローカル座標系だけではどうにもなりません。読者も絶対的な基準が存在する必要があることは容易に想像できるのでないでしょうか。例えば、あるジオメトリを（何でもいいのですが、例えば戦車を）ある地点に移動させたいと思った時はこう考えるはずです、“座標（10,12,13）に移動させよう”と、何も難しいことはありません、読者が座標としてすぐに想像できるものです。この座標（10,12,13）が絶対座標なのですから。
念のため図４－４に絶対座標系に３つのジオメトリが存在している様子を示します。図４－５はXY平面のみを見た様子です。

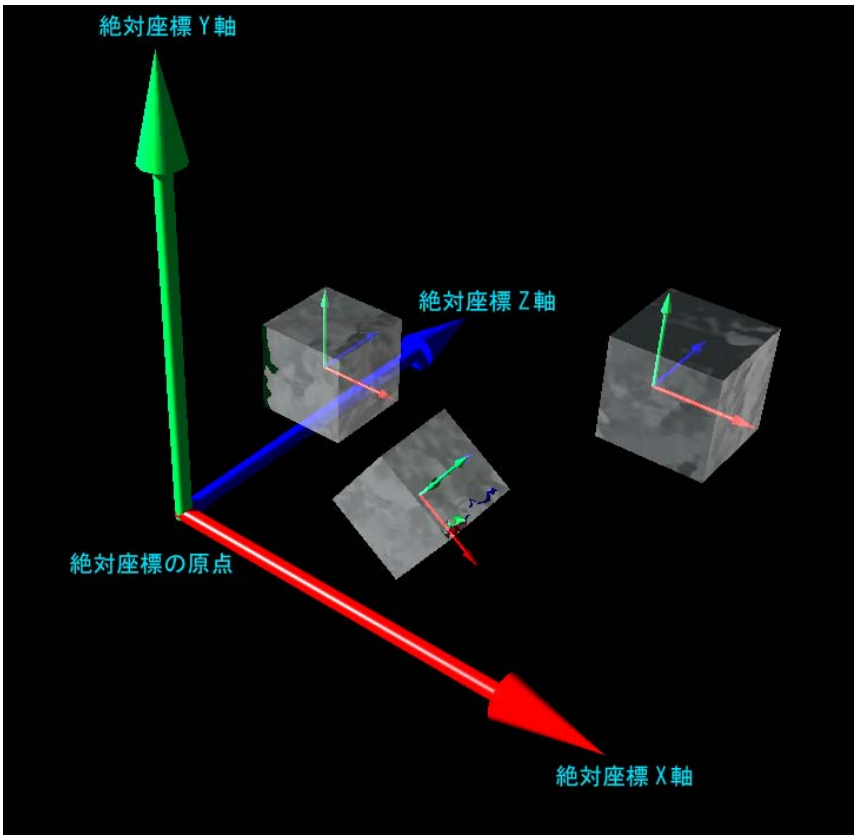


図４－４

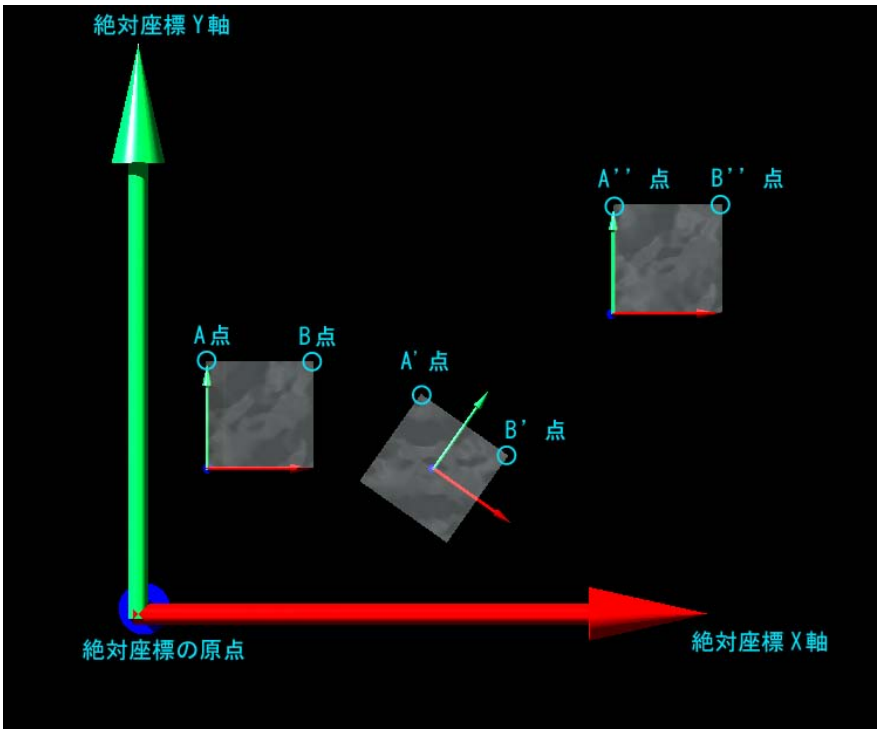


図４－５
大きい軸が絶対座標軸で、ジオメトリ毎に附帯している小さな軸はローカル座標軸です。

ジオメトリは図４－１、図４－２と同一とします。３つのジオメトリのローカル座標系における A 点 A’ 点、A” 点及び B 点、B’ 点、B” 点の座標は全て同一です（同じジオメトリなので当然です）。しかし、絶対座標系で見ると個々のジオメトリのローカル中心点や頂点は、図から明らかに、全て異なる座標になります。

なぜ絶対座標に変換しなければならないのか？

ここまでの説明を読んで、絶対座標に変換する必要があることを既に予想あるいは理解できた読者もいるのではないでしょうか？先ほど絶対座標系でなければ配置や移動が不可能であると述べました、同一のジオメトリは全て同じローカル座標であり、異なる位置に配置しレンダリングするには異なる “ 絶対座標 ” を指定しなければなりません。考えてみると当たり前であり、当たり前のことを長々と説明することによって逆に読者を混乱させるのではないかとう不安さえ覚えていますが、説明しないわけにはいかないのでその辺はご勘弁願います。

コード上での変換方法

まず、３つの変換全てについて言える事は、変換は “ 行列 ” を媒介して行うということです。行列については Chapter4-2 で詳しく説明します。
172 行目
D3DXMATRIXA16 は行列データ型です。MatWorld という行列をまず用意します。
174 行目
SetTransform(D3DTS_WORLD, &matWorld);　D3DTS_WORLD「ワールドトランスフォーム」を「matWorld 行列を係数として」行うという事を Direct3D に通知します。

実は本サンプルではワールド変換を行う必要はありません。なぜなら、ジオメトリは移動せずに最初から最後までずっと同じ位置だからです。プログラム全体を通してジオメトリの動きがまったく無かったり、複数のジオメトリを異なる配置でレンダリングする必要が無い場合にはワールド変換は不要です。しかし、それは本サンプルに限ったことであり、実際のゲームでは有り得ないこと、また、今後のサンプルとの整合性のために３つの変換すべてをコードに書きました。

２．ビュートランスフォーム

ワールドトランスフォームが、個々の “ ジオメトリ ” を絶対座標上に配置したり移動させたりする目的であるのに対して、ビュートランスフォームは “ ワールド座標 ” を “ ビュー（視点）座標 ” に変換します。流れとしてはワールドトランスフォームした後にビュートランスフォームします。ビュートランスフォームによって “ 視点 ” を配置したり移動させることが出来ます。ビュートランスフォームにより視点を移動させた時に、画面にレンダリングされている像が全て一斉に動き、あたかも撮影カメラ自体を移動させたかのような効果があることからカメラ座標という言葉を好んで使用する人もいます。
具体的な使用例を挙げると、例えば一人称視点のシューティングゲーム（Quake や Half Life、Unreal 等）ではマウス操作で主人公の視点を移動・回転させることが出来、その際には画面全体が移動・回転します。まさにそれはビュートランスフォームにより視点を移動・回転させることによる効果なのです。
実は、ビュートランスフォームも本サンプルでは不要です。なぜなら、視点位置はずっと同じでいいからです。カメラを固定して撮影していると言えばわかりやすいでしょうか、そのような場合に、ビュートランスフォームは不要です。

４．プロジェクショントランスフォーム

プロジェクショントランスフォームを一言で表現すると、“ カメラのレンズを設定する ” ことと言えます。プロジェクションの設定値の考え方は現実のカメラの広角レンズや望遠レンズの原理と全く同一であり、レンダリング結果もレンズと写真の関係と全く同じになります。広角レンズのように設定すれば最終的なレンダリングはパースが強く、遠くのものはより小さく見えます。対して望遠レンズのように設定すればパースが殆どかからず、距離感が出ないときもあります。要するにプロジェクショントランスフォームは視点をカメラに例えたときにズームイン・ズームアウトをする変更と言ってもいいでしょう。具体的な例では、また一人称シューティングゲームを例にすると、例えばライフルのスコープを覗いている時は普通の画面とは違う遠近感で見えます。スコープの倍率を上げたときはパースが弱くなるという効果は、まさにプロジェクショントランスフォームによるプロジェクション（レンズ）変更の効果です。
また、プロジェクショントランスフォームは射影変換でもあり、射影とは 3D の物体を画面に描画出来るように 2D データに変換することです。（画面は物理的に 2D デバイスであり 3D 物体はそのまま表示できません）
もう、予想された読者もいるかもしれませんが、プロジェクショントランスフォームも本サンプルでは不要です。理由はおわかりですよね？ズームイン・ズームアップを行っていないからです。

さて、３つのトランスフォームをさらりと解説しました。結局本サンプルでは３つのトランスフォームは無ければ無いでもいいのですが、それは “ ソースコード上 ” での話しであり、Direct3D 内部、レンダリングパイプラインという場所で毎回３つの変換を行っているのです。順番で言うとまず、ワールド、次にビュー、最後にプロジェクションという変換過程を行って最終的に画面にラスタライズ※するということを毎フレーム繰り返しています。そうゆう意味で敢えてソースコードに書きました。しかし、やはりソースコード及びその出力に沿って解説したほうが、インパクトがあり、理解しやすいと思いますので、第５章では、本章のように “ 実は XXX トランスフォームは不要です ” ということはなく、再度、詳しい解説をその時点で行います。第５章では、３つのトランスフォーム全ての効果を体験していただく目的で、3D 物体の操作とカメラ移動、ズームイン・ズームアウトを行います。本章ではレンダリングには３つのトランスフォームが必要であるということを頭に留めておいていただければと思います。

レンダリング

188 行目～ 197 行目
レンダリングに関しては、解説することは殆ど無く、シンプルです。
本サンプルのように頂点バッファーによりレンダリングする時は、まず SetStreamSource メソッドにより当該頂点バッファーの位置（pVB）と頂点のサイズ（sizeof(CUSTOMVERTEX)）を Direct3D に通知します。
次に、SetFVF メソッドにより、頂点の性質を Direct3D に通知します。頂点の性質を D3DFVF_CUSTOMVERTEX にマクロ定義したことを思い出してください。
そして、DrawPrimitive メソッドによりレンダリングします。D3DPT_TRIANGLELIST の意味は、“ 連続した三角形を描画する ” ということです。三角形の頂点は頂点バッファーに格納されている頂点座標です。Direct3D での頂点バッファーによるジオメトリは 3 角形が最小単位ですので、もし 4 角形をレンダリングしたいのであれば 3 角形 2 個分のデータをバッファーに格納しておき第 3 引数に 2 を渡して 4 角形になるように 3 角形を隣り合わせて 2 個レンダリングすればいいわけです。もちろん、その場合は頂点バッファーの三角形データは連続してレンダリングした時に 4 角形になるような座標を格納しておかなければなりません。

ファイルからメッシュを読み込んでレンダリングする手法

サンプルプロジェクト名：Chapter4-2

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5:
 6: LPDIRECT3D9 pD3d;
 7: LPDIRECT3DDEVICE9 pDevice;
 8: LPD3DXMESH pMesh = NULL;
 9: D3DMATERIAL9* pMeshMaterials = NULL;
10: LPDIRECT3DTEXTURE9* pMeshTextures  = NULL;
11: DWORD dwNumMaterials = 0;
12:
13: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
14: HRESULT InitD3d(HWND);
15: VOID Render();
16: VOID FreeDx();
17:
18: //
19: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
20: // アプリケーションのエントリー関数
21: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
22: {
23:        HWND hWnd=NULL;
24:        MSG msg;
25:        // ウィンドウの初期化
26:        static char szAppName[] = "Chapter4-2" ;
27:      WNDCLASSEX  wndclass ;
28:
29:      wndclass.cbSize          = sizeof (wndclass) ;
30:      wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
31:      wndclass.lpfnWndProc     = WndProc ;
32:      wndclass.cbClsExtra      = 0 ;
33:      wndclass.cbWndExtra      = 0 ;
34:      wndclass.hInstance       = hInst ;
35:      wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
36:      wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
37:      wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
38:      wndclass.lpszMenuName = NULL ;
39:      wndclass.lpszClassName = szAppName ;
40:      wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
41:
42:      RegisterClassEx (&wndclass) ;
43:
44:      hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
45:                  0,0,800,600,NULL,NULL,hInst,NULL) ;
46:
47:     ShowWindow (hWnd,SW_SHOW) ;
48:     UpdateWindow (hWnd) ;
```

```
49:       // ダイレクト3D の初期化関数を呼ぶ
50:       if(FAILED(InitD3d(hWnd)))
51:       {
52:                 return 0;
53:       }
54:       // メッセージループ
55:     ZeroMemory( &msg, sizeof(msg) );
56:     while( msg.message!=WM_QUIT )
57:      {
58:                  if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
59:        {
60:                           TranslateMessage( &msg );
61:             DispatchMessage( &msg );
62:         }
63:         else
64:                 {
65:                           Render();
66:                 }
67:        }
68:        // メッセージループから抜けたらオブジェクトを全て開放する
69:        FreeDx();
70:        // OS に戻る （アプリケーションを終了する）
71:      return (INT)msg.wParam ;
72: }
73:
74: //
75: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
76: // ウィンドウプロシージャ関数
77: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
78: {
79:       switch(iMsg)
80:       {
81:                 case WM_DESTROY:
82:                           PostQuitMessage(0);
83:                 return 0;
84:                 case WM_KEYDOWN:
85:                 switch((CHAR)wParam)
86:                 {
87:                           case VK_ESCAPE:
88:                                     PostQuitMessage(0);
89:                           return 0;
90:                 }
91:                 break;
92:       }
93:       return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
94: }
95:
96: //
97: //HRESULT InitD3d(HWND hWnd)
98: // ダイレクト 3D の初期化関数
99: HRESULT InitD3d(HWND hWnd)
100: {
101:       //「Direct3D」 オブジェクトの作成
102:       if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
103:       {
104:                 MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
105:        return E_FAIL;
106:       }
107:       //「DIRECT3D デバイス」 オブジェクトの作成
108:       D3DPRESENT_PARAMETERS d3dpp;
109:       ZeroMemory( &d3dpp, sizeof(d3dpp) );
110:       d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
111:       d3dpp.BackBufferCount=1;
112:       d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
113:       d3dpp.Windowed = TRUE;
114:     d3dpp.EnableAutoDepthStencil = TRUE;
115:     d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
116:
117:        if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
118:                                  D3DCREATE_MIXED_VERTEXPROCESSING,
```

```
119:                                &d3dpp, &pDevice ) ) )
120:     {
121:                 MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
122:                 if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
123:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
124:                                 &d3dpp, &pDevice ) ) )
125:                 {
126:                             MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
127:                              return E_FAIL;
128:                 }
129:     }
130:       // X ファイルからメッシュをロードする
131:       LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
132:
133:       if( FAILED( D3DXLoadMeshFromX( "Chips.x", D3DXMESH_SYSTEMMEM,
134:             pDevice, NULL, &pD3DXMtrlBuffer, NULL,
135:                             &dwNumMaterials, &pMesh ) ) )
136:     {
137:             MessageBox(NULL, "X ファイルの読み込みに失敗しました ",NULL, MB_OK);
138:             return E_FAIL;
139:     }
140:       D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
141:     pMeshMaterials = new D3DMATERIAL9[dwNumMaterials];
142:     pMeshTextures  = new LPDIRECT3DTEXTURE9[dwNumMaterials];
143:
144:       for( DWORD i=0; i<dwNumMaterials; i++ )
145:       {
146:                   pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
147:         pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
148:         pMeshTextures[i] = NULL;
149:         if( d3dxMaterials[i].pTextureFilename != NULL &&
150:             lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
151:         {
152:             if( FAILED( D3DXCreateTextureFromFile( pDevice,
153:                                         d3dxMaterials[i].pTextureFilename,
154:                                         &pMeshTextures[i] ) ) )
155:             {
156:                 MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
157:                             }
158:         }
159:     }
160:       pD3DXMtrlBuffer->Release();
161:       // Z バッファー処理を有効にする
162:     pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
163:       // ライトを有効にする
164:       pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
165:       // アンビエントライト（環境光）を設定する
166:        pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
167:        // スペキュラ（鏡面反射）を有効にする
168:        pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
169:       return S_OK;
170: }
171:
172: //
173: //VOID Render()
174: //X ファイルから読み込んだメッシュをレンダリングする関数
175: VOID Render()
176: {
177:       // ワールドトランスフォーム（絶対座標変換）
178:        D3DXMATRIXA16 matWorld,matRotation;
179:        D3DXMatrixRotationY( &matWorld, timeGetTime()/3000.0f );
180:        D3DXMatrixRotationX( &matRotation, 0.5f );
181:        D3DXMatrixMultiply(&matWorld,&matWorld,&matRotation);
182:      pDevice->SetTransform( D3DTS_WORLD, &matWorld );
183:        // ビュートランスフォーム（視点座標変換）
184:
185:      D3DXVECTOR3 vecEyePt( 0.0f, 1.0f,-3.0f ); // カメラ（視点） 位置
186:      D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
187:      D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
188:      D3DXMATRIXA16 matView;
```

```
189:      D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
190:      pDevice->SetTransform( D3DTS_VIEW, &matView );
191:        // プロジェクショントランスフォーム（射影変換）
192:      D3DXMATRIXA16 matProj;
193:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
194:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
195:        // ライトをあてる 白色で鏡面反射ありに設定
196:        D3DXVECTOR3 vecDirection(0,0,1);
197:      D3DLIGHT9 light;
198:      ZeroMemory( &light, sizeof(D3DLIGHT9) );
199:      light.Type       = D3DLIGHT_DIRECTIONAL;
200:      light.Diffuse.r  = 1.0f;
201:      light.Diffuse.g  = 1.0f;
202:      light.Diffuse.b  = 1.0f;
203:        light.Specular.r=1.0f;
204:        light.Specular.g=1.0f;
205:        light.Specular.b=1.0f;
206:      D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
207:      light.Range      = 200.0f;
208:      pDevice->SetLight( 0, &light );
209:      pDevice->LightEnable( 0, TRUE );
210:        // レンダリング
211:        pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
212:                        D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
213:
214:        if( SUCCEEDED( pDevice->BeginScene() ) )
215:        {
216:                        for( DWORD i=0; i<dwNumMaterials; i++ )
217:        {
218:            pDevice->SetMaterial( &pMeshMaterials[i] );
219:            pDevice->SetTexture( 0, pMeshTextures[i] );
220:            pMesh->DrawSubset( i );
221:        }
222:                        pDevice->EndScene();
223:        }
224:        pDevice->Present( NULL, NULL, NULL, NULL );
225: }
226:
227: //
228: //VOID FreeDx()
229: // 作成した DirectX オブジェクトの開放
230: VOID FreeDx()
231: {
232:       SAFE_RELEASE( pMesh );
233:       SAFE_RELEASE( pDevice );
234:       SAFE_RELEASE( pD3d );
235: }
```

頂点バッファーを使用し直接的に頂点を操作する手法において、複雑な形のものをレンダリングしようとすればコードを自前で書かなくてはなりませんが、それはかなり面倒な作業です。そのため、表現できるジオメトリの形状が限られてきます。
一般的にゲームで使用するジオメトリは 3 角形や 4 角形ではなく、より頂点数、ポリゴン数を多く使用しますが、サンプル 4－ 1 の手法だと膨大な頂点を設定しなくてはなりません。
そのようなことをしなくても、複雑な形状で見栄えが良く 3DCG ソフトのオブジェクトのような形状のジオメトリを簡単に管理し、レンダリングすることができる方法がここで解説するものです。
仕組みは、次のようになります。

頂点データを「ファイル」として別に作成する。
↓
プログラム側では、そのファイルを読み込むことによりジオメトリを形成し、レンダリングする。

頂点情報を記録したファイルは “X ファイル ” と呼ばれています。X ファイルには頂点情報の他にテクスチャー情報やアニメーション情報も記録することが出来ます。X ファイルを例えるとすれば、3DCG ソフトのオブジェクトファイルをイメージすればいいでしょう。両者のファイルフォーマットは酷似しています。ただ、オブジェクトファイルのように無制限に頂点を記録するとうよりも、ある程度頂点数を制限して記録するところが違いです。3DCG ソフトのような非常に細かいジオメトリをリアルタイムでレンダリングするのは困難であるのは容易に想像できるでしょう。筆者は 3DCG の作成には LightWave を使用しているのですが、以前 LightWave のオリジナルプラグインを製作した際にオブジェクトファイルの中身を調べたことがあります。オブジェクトファイルの仕様・フォーマットは「これは X ファイルか」と思えるほど殆ど同じでした、実際には

LightWave の方が X ファイルより歴史が古いので言い方が逆かもしれません。X ファイルがオブジェクトファイルと殆ど同じと言うべきでしょうか。X ファイルの読み込みやその後の管理は、専用の関数が Direct3D に用意されていますので、本サンプルで示します。
X ファイルから読み込まれるジオメトリは、普通は " メッシュ " です。メッシュとは多数のポリゴンから成るものです。ポリゴンを単体で読み込んだり、ましてや 2,3 個の頂点を読み込むようなことはありません。そもそも X ファイルを使用する目的は大量の頂点、ポリゴンから成るメッシュを一括して読み込むことだからです。
本サンプルでの重要箇所は、X ファイルの読み込み部分と X ファイルから読み込んだメッシュのレンダリング部分です。
まずは、X ファイルからメッシュを読み込む部分をコードに沿って見ていきましょう。

X ファイルからメッシュを読み込む

本サンプルでは、筆者が製作したテクスチャー及びポテトチップス（のパッケージ）オブジェクトを X ファイルにしたものの読み込みを行います。オブジェクトは LightWave7.5 で製作し、LightWave の ExportDirectX というプラグインにより X ファイルに変換しました。ファイルネームは Chips.x です。X ファイルは拡張子が .x となりますが、決まりではありません。ファイル名を指定して読み込むコードを書くのは開発者であり、ソースコード上の拡張子と一致すればどのような拡張子でも構いませんが、まあ、普通はそのままの拡張子で使用するのがいいでしょう。
130 行目～ 160 行目
LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
マテリアルバッファーのポインターを宣言します。マテリアルとは、頂点やポリゴンの色、光沢等及びテクスチャーのことで、要するにポリゴンの表面的質感のことです。テクスチャーも質感を意味する概念ですが、マテリアルはより広い意味での質感概念です。3DCG ソフトを使用した経験のある読者にはオブジェクトのサーフェイスのことであると言えば、直ぐに理解できるでしょう。したがって、そのオブジェクトが持つサーフェイスの数だけマテリアルが作成され X ファイルに記録されることになります。
マテリアルバッファーは文字通り、マテリアルのバッファーすなわちマテリアルを格納するメモリー領域です。X ファイルからメッシュを読み込む際には、マテリアルバッファー " のポインター " を用意しておきます。マテリアルバッファー自体は読み込み時に自動的に作成されマテリアルデータが自動的にバッファーに格納されます。開発者はそのポインターだけを事前に用意します。ここで少々注意が必要です、LPD3DXBUFFER はマテリアルバッファー専用のポインター型ではなく汎用バッファーのポインター型です。単純なポインター型ではなく、それ自体が、意図するバッファーアドレスを指しているわけではありません。それゆえに、例えば 134 行目を見てください、
D3DXMATERIL*d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
意図するポインターを参照するには自分自身の GetBufferPointer メソッドをコールしなければなりません、単純なポインターに比べ、もうひと手間必要になるわけです。

D3DXLoadMeshFromX 関数
これが実際に X ファイルを読み込む関数です。関数が成功すると、3 つのパラメーターを適正な値で初期化してくれます。
1 つ目は先に用意しておいたマテリアルバッファーポインターをバッファーの適正なアドレスで初期化してくれます（pD3DXMtrlBuffer）、
2 つ目はマテリアルの数を返します（dwNumMaterials）、マテリアルの数はレンダリング関数でも参照するので 9 行目でグローバルに宣言しています。
3 つ目はメッシュへのポインター変数をメッシュの適正なアドレスで初期化してくれます（pMesh）。マテリアルバッファーポインターと同じようにメッシュ自体は自動的に作成されますが、そのメッシュへのポインターだけは事前に用意しておきます。メッシュへのポインターはグローバルに宣言する必要があったので 6 行目で宣言しています。
140 行目
一時的なマテリアルバッファーへのポインターを宣言し、適正なアドレスで初期化します。
このポインターはすぐ下の for 文ブロック内でのみ使用します。
141 行目
new 命令により、マテリアルの数の分だけマテリアルのメモリー領域を確保します。本サンプルの Chips.x はサーフェイスが２つだけなのでマテリアルも２つになります。したがって必要なメモリーはマテリアル 2 個分になります。そのメモリーへのアドレスをグローバルに宣言していた pMeshMaterials に代入します。これで、これ以降 pMeshMaterials はメッシュ (Chips.x) のマテリアルを指す適正なポインターになりました。
142 行目
メッシュにテクスチャーを使用している場合もあるので、テクスチャーに必要なメモリー領域も確保し、ここでもグローバルでに宣言されている pMeshTextures に確保されたアドレスを代入します。Chips.x の２つのサーフェイスは共にテクスチャーを使用しているのでテクスチャーも２つ作成する必要があります。なお、Chips.x のテクスチャーは２つとも同じビットマップを使用していますが、テクスチャー座標等が異なるため別のテクスチャーとして作成する必要があります。サーフェイスの数だけ、言い換えればマテリアルの数だけテクスチャーを作成するようにすればいいでしょう。
144 行目～ 159 行目
この for 文ブロック以前の段階において、実際に適正なメッシュのマテリアルを指しているのは d3dxMaterials だけです。（134 行目でローカルに宣言した一時的なポインターです）
pMeshMaterials と pMeshTextures は for 文直前ではそれぞれの型に必要なメモリー領域は確保したものの、中身はまだ空です。
そこで、この for 文ブロック内で中身、すなわち、マテリアルデータを代入します。
まず、pMeshMaterials[n] にマテリアルデータを代入します。
次に、pMeshMaterials[n] の Ambient メンバーを Diffuse メンバーと同じ値にします。これは筆者が奇妙に思ったことなのですが、なぜか X ファイルから読み込むとアンビエントがゼロのまま、つまり、アンビエントが初期化されないので必要な処理です。
最後にテクスチャーのファイルネームを基に、テクスチャオブジェクトを作成します。
ここまでで、グローバルに宣言されていた全てのマテリアルオブジェクト、テクスチャオブジェクトが適正に初期化され、よ

うやくレンダリングできる体制が整ったことになります。
160 行目
この関数内でローカルに用意しておいた pD3DXMtrlBuffer 及び、それが示すバッファーは、読み込みが完了した段階で不用になるものなので、ここでリリース、つまりそのメモリーを開放します。pD3DXMtrlBuffer 及びそれが指し示すバッファーは X ファイルから目的のオブジェクトのメモリー (pMeshMaterials の指すメモリー領域 , pMeshTextures の指すメモリー領) にデータをコピーする際の媒介役に過ぎず、読み込み処理が一旦終わってしまえば、存在理由がなくなるからです。
162 行目
SetRenderState(D3DRS_ZENABLE, TRUE);
D3DRS_ZENABLE「Z バッファー有効」を TRUE「オン」にします。
メッシュを描画する時は、Z バッファーを有効にします。Z バッファーとは「陰面処理」のことであり、厳密に言うと「Z バッファー処理」と言うべきですが、簡単に Z バッファーと呼ばれています。
陰面処理とは、視点から見て、あるメッシュが別のメッシュの手前に位置している場合、後ろのメッシュの隠れている部分はレンダリングしないようにする処理です。隠れているはずのメッシュをレンダリングしてしまうと、不自然なのは明らかです。Z バッファーはメッシュとメッシュ、同じメッシュ内でも、そのメッシュを形成している各ポリゴンについても同様に後ろに隠れているポリゴンはレンダリングしないようにします。
164 行目
SetRenderState(D3DRS_LIGHTING, TRUE);
D3DRS_LIGHTING「ライト」を TRUE「有効」にします。
本サンプルのメッシュ Chips.x は、頂点に色情報がありませんので、ライトがあたった時にはじめてその色が見えます。ですので、ライトを有効にしています。対して Chapter4-1 サンプルでは、頂点自体が色を持っていたのでライトをオフにする必要があったのを思い出してください。
166 行目
SetRenderState(D3DRS_AMBIENT, 0x00eeeeee);
D3DRS_AMBIENT「アンビエント光」を 0x00eeeeee に設定します。
アンビエント光は環境光とも呼ばれ、シーン内のジオメトリに均一に照射される光であり、また、影を作らない光のことです。アンビエント光は、ライトの向きや位置を考えなくてもジオメトリを確実に明るくする効果があり、その意味では便利ではありますが、文字通り均一に照射されるため、時としてライト特有の明暗演出効果を打ち消してしまい、平面的にレンダリングされてしまうという副作用があります。特に 3DCG ソフトの場合、アンビエント光は 10% 程度に抑えて他のライトを複数個組み合わせることによって、ライト効果を演出し、見栄えを良くします。Direct3D によるリアルタイムレンダリングでも、考え方は変わりませんが、3DCG ソフトのように多くのライトを使用しての演出はかなり重たい処理のために、スピードを要求されるシーンではアンビエント光に頼るという妥協も必要になります。

メッシュをレンダリングする

X ファイルを読み込み、無事にメッシュを生成することが出来たので、あとはメッシュをレンダリングするだけです。レンダリングのコードはいたって簡単です。しかも、ワールド、ビュー、プロジェクショントランスフォームについては前のサンプルで解説しているので解説する箇所も、ごく僅かです。
ワールドトランスフォームについてのみ、若干、前サンプルと違いますので解説します。
177 行目～ 181 行目
ここでやっている事は、チップスメッシュを回転させてから、やや上下に回転をかけるということを行っています。3D 空間での移動や回転には行列を媒介させます。行列の基本概念及び使用方法は Chapter5 で詳しく解説しますので今は眺める程度にしていてください。
210 行目～ 224 行目
この部分が実際のレンダリング処理です。
まず、マテリアルの数だけ、3DCG ソフトで言うサーフェイスの数だけループさせます。
Direct3D におけるメッシュのレンダリングはマテリアルを基準にしています。
SetMaterial(&pMeshMaterials[i]);　マテリアルを設定します。
マテリアルが設定され、次に設定されるまでは（つまり、変更されるまでは）、同じマテリアル設定が適用されます。マテリアル設定が適用されるという意味は、メッシュが当該マテリアルの色・輝度等によりレンダリングされるということです。例えば、黄色いマテリアルと赤いマテリアルがあり、最初に黄色のマテリアルが設定した場合、それ以降のレンダリングは赤のマテリアルが適用されるまで、すべてのメッシュは黄色でレンダリングされます
SetTexture(0, pMeshTextures[i]);　テクスチャーを設定します。
テクスチャーの設定についても全く同様です。テクスチャーが変更されない限り、いかなるメッシュも同じテクスチャーでレンダリングされます。（されてしまいます。）
したがって、本サンプルコードのように、ゲームループで当該メッシュのレンダリング直前に、毎回設定する必要があるのです。
最後に ID3DXMesh::DrawSubset メソッドによりレンダリングします。

Chapter5　3D 物体の操作と見せ方

3D 物体を表示出来たら、今度はそれを移動させたり回転させたりしましょう。移動や回転まで出来れば、ゲームとしての基本要素（最低限ではありますが）を習得したと言えます。
それと、全章ではジオメトリがシーンにたった一つしか存在していなかったので、ジオメトリの初期位置の設定、つまり配置について深く考える必要はなく、また、解説もしていませんでしたが、本章では配置についても解説します。配置は移動・回転と密接に関連しています。と言うよりも移動・回転は配置の概念そのものです。なぜなら、移動・回転はある時点の配置のパラメーターを変更させ、再配置することを繰る返す動作（連続した再配置）と言えるからであり、コードもそのようになります。したがって、配置についての背景理論を理解することは、移動や回転の背景理論を理解することです。、

行列とは？

行列と言っても、人気のラーメン屋さんの昼時の光景ではありません。同じ文字ですが、3D グラフィックにおける行列は別の意味です。
3D グラフィックにおける行列（Matrix「マトリクス」）という言葉は、数学から来ている数学用語です。高校数学の授業で、行列が登場したことを憶えている読者も多いと思います。「行列とは？」というあまりにも漠然とした問いに対して、数学の１つの分野にまで成熟している行列概念が持つ全ての意味を簡潔に説明することは至難の技です。しかしながら、こと 3D グラフィック、Direct3D においては、行列の中でも基礎的なものしか使用しません。そうなると、それを一言で表現することは、さほど難しくありません。
行列とは、縦横、長方形状に羅列した数字それぞれを、互いに加算・減算・乗算（除算はありません）する際の演算順序を規定した計算体系であり、縦横に羅列した数字の書式そのものを指す場合もあります。
具体的に数字を当てはめて見ていきましょう、まずは次を見てください。

1　，　2　第 1 行

3　，　4　第 2 行
第 1 列　第 2 列

これが、行列の書式です。数値は、取り敢えず適当に 1,2,3,4 としました。このように、行列は縦横に数字を羅列する書式になります。この行列は、横のラインが 2 行、縦のラインが 2 列あるので 22 行列と呼ぶという決まりがあります。このことから容易に連想されると思いますが、m 行、n 列の行列を mn 行列と呼びます。
この行列のように行数と列数が同じである行列を特に “n 次の正方行列 ” とも呼びます。
同様に、12 行列、13 行列、33 行列、44 行列は次の通りです。

1　，　2　第 1 行のみ　　1　，　2　，　3　第 1 行のみ
第 1 列　第 2 列　　第 1 列　第 2 列　第 3 列

1 ，　2 ，　3　第 1 行　1 ，　2 ，　3　第 1 行
4 ，　5 ，　6　第 2 行　4 ，　5 ，　6　第 2 行
7 ，　8 ，　9　第 3 行　7 ，　8 ，　9　第 3 行
第 1 列　第 2 列　第 3 列
1 行 n 列の行列（行のみの行列）のことを特に、n 次の行ベクトルとも呼びます。したがって、12 行列は 2 次の行ベクトル、13 行列は 3 次の行ベクトルとも呼ぶことができます。このように行列はベクトルのスーパーセット※、ベクトルは行列のサブセットという関係でもあります。ベクトルは行列の一部、行列はベクトルの発展形とも考えることができます。
Direct3D で使用する行列は、
2 次元物体の移動・回転・変形であれば、“2 次の行ベクトル ” と “44 行列 ” です。
3 次元物体の移動・回転・変形であれば “3 次の行ベクトル ” と “44 行列 ” です。
今までに、コード上で何回も登場してきた D3DXVECTOR3 が “3 次の行ベクトル ” であり、D3DXMATRIXA16 が “44 行列 ” です。おそらく D3DXVECTOR3 は、数値が 3 個だから 3 を最後に付け、D3DXMATRIXA16 は数値が 16 個（4x4）なので最後に 16 を付けているのでしょう。
さて、行列の定義は分かったところで、次に行列の足し算、引き算、掛け算について解説します。行列の加算・減算・乗算は、Direct3D におけるトランスフォーム関連で行う演算であるため理解する必要あります。退屈かもしれませんが、少々お付き合いください。

行列同士の加算・減算

加算と減算は、簡単です。素直にそれぞれの数値を加減すればいいのです。
そして、1 つ決まりがあります。それは、行列の加減算は “ 全く同じ型 ” の行列同士でなければならないということです。例えば 22 行列と 33 行列同士を加算・減算することはできません。これは加減算の計算手順から当然導かれる事であり、実際に計算してみれば、計算の途中で、式そのものが作成出来ないことから、すぐ分かります。

1 ，2　　5 ，6　　1+5 ，2+6　　6 ，8
　　+　　=　　=
3 ，4　　7 ，8　　3+7 ，4+8　　10 . 12

1 ，2　　5 ，6　　1-5 ，2-6　　-4 ，-4
　　-　　=　　=
3 ，4　　7 ，8　　3-7 ，4-8　　-4 . -4

行列に定数を掛ける

定数 x 行列も簡単で、全ての数値にその定数を書ければいいのです。
なお、単純な定数はスカラー値※であるため、“ 行列のスカラー倍 ” などと呼ぶこともあります。

　　1 ，2　　3x1　　，3x2　　3 ，6
3　x　　=　　=
　　3 ，4　　3x3　，3x4　　9 ，12

行列同士の乗算

乗算は、単純ではなく少々複雑です。
そして、乗算の場合、加算や減算のように互いの行列の型は“全く同じ”である必要はありませんが、“左の行列の列数と右の行列の行数が一致した行列同士”でなければなりません。例えば 12 行列と 22 行列を掛けることが出来ますし、13 行列と 33 行列を掛けることが出来ます。
行と列が一致しなければならないという条件は、乗算の計算手順から当然導かれる事であり、もし、行と列が一致しない行列同士を掛けようとした時に、計算過程で中間式自体が書けないことから、すぐに分かります。

計算手順）

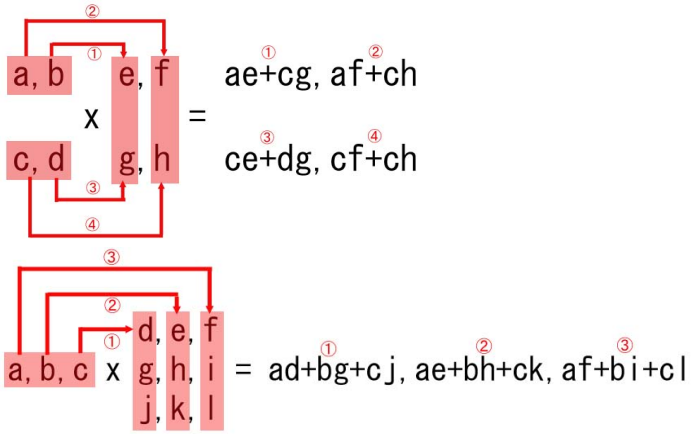


図 5-1

考え方としては、“左の行列の行“と”右の行列の列“を単位として掛け合わせていくということでしょうか。
また、これも計算手順から当然導かれる事項ですが、計算結果の行列は左の行列の行、右の行列の列数を持つ行列となります。図を見れば分かると思いますが、22 行列と 22 行列の計算結果の行列は 22 行列になり、13 行列と 33 行列の計算結果の行列は 13 行列になります。
計算例）
22 行列同士の乗算
1 ， 2　 5 ， 6　 1x5+2x7 ， 1x6+2x8　 19 ， 22
　　x　　=　　=
3 ， 4　 7 ， 8　 3x5+4x7 ， 3x6+4x8　 43 ． 50

33 行列同士の乗算
1,2,3　10,11,12　1x10+2x13+3x16,1x11+2x14+3x17,1x12+2x15+3x18　84,90,96
4,5,6 x 13,14,15= 4x10+5x13+6x16,4x11+5x14+6x17,4x12+5x15+6x18 =201,216,231
7,8,9　16,17,18　7x10+8x13+9x16,7x11+8x14+9x17,7x12+8x15+9x18　318,342,366

12 行列と 22 行列の乗算

　　3 , 4
1 , 2 x　　= 1x3+2x5 , 1x4+2x6 = 13 , 16
　　5 , 6

13 行列と 33 行列の乗算

　　4 , 5, 6
1,2,3 x 7 , 8, 9 = 1x4+2x7+3x10,1x5+2x8+3x11,1x6+2x9+3x12 =
　　10,11,12

どうですか？計算法則自体はさほど複雑ではありませんが、いざ計算してみると、かなり面倒くさそうですよね。33 行列でさえこれだけ数字を書かなくてはなりません。（本書を編集して下さっている工学社の方がチェックするにも骨が折れることと心配さえしてしまいます。）44 行列にいたっては手書きで計算することを臆してしまいます。ちなみに、筆者の場合、行列を手作業で計算することは、まずありません。Excel に行列計算式を作成し、利用しています。

Direct3D の行列

ここでちょっと、立ち止まってみましょう。ここまで、読んで何か疑問に感じたことはありませんか？先ほど、Direct3D で使用する行列は、“2 次の行ベクトル”と“44 行列”、“3 次の行ベクトル”と“44 行列”…と述べたました。それぞれ、同じ型で無いばかりか、行と列すら一致していません。これは誤植ではありません。Direct3D は全て 44 行列を使用します。
開発者のソースコード上では 2 次や 3 次のベクトルでも、Direct3D 内部で自動的にダミーの次元を追加して 4 次のベクトルにしてから計算しています。ですので、なにも Direct3D が特別な行列を持っている訳ではありません。具体的には、次のよ

うな処理をしています。
2 次ベクトルの場合　(a , b) →　(a , b , 0 , 1)
3 次ベクトルの場合　(a , b , c) →　(a , b , c , 1)
次元を追加しても、追加した次元が 1 つであればその要素を 1 に、複数の次元を追加した場合は、最後の次元要素を 1 に、他の追加次元の要素をゼロにすると上手い具合に適正な結果が得られるのです。このことは、すぐ後で同次座標と関連させて、再度解説します。

なぜ、行列を利用するのか？

もしも、行列を初めて目にした読者なら、なぜこのような、ややこしい計算手順が必要なのかが分からないことでしょう。そして、高校の数学で行列を履修したことのある読者でも、理系の大学等に進んでいない限り、高校時代に行ったはずの行列計算など、恐ろしい速さで頭の中から消え去っているかもしれません。
ここでは、実際に行列を用いてジオメトリを操作して、どのように行列が機能しているかを解説します。行列がどのように利用され、それが機能している様を具体的に見た時、初めて行列のパワー・行列の存在意義を見出すかもしれません。そしてそれは、筆者が期待していることでもあります。

行列を用いたジオメトリの回転

2 次元平面に図のような三角形ジオメトリがあるとします。頂点の座標は A(2,2) B(2,0) C(0,0) であるとします。

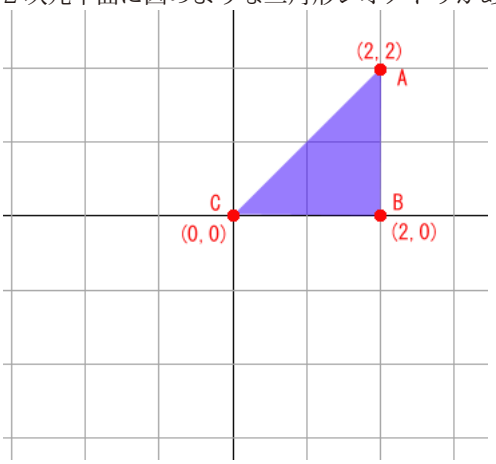


図 5-2

この三角形の頂点全てに次の行列
0 , 1
-1 , 0
を掛けると、
頂点はそれぞれ A(2,-2) B(0,-2) C(0,0) となり、図形は図のようになります。（実際に計算してみると、頂点が変換されることが確認できます。）

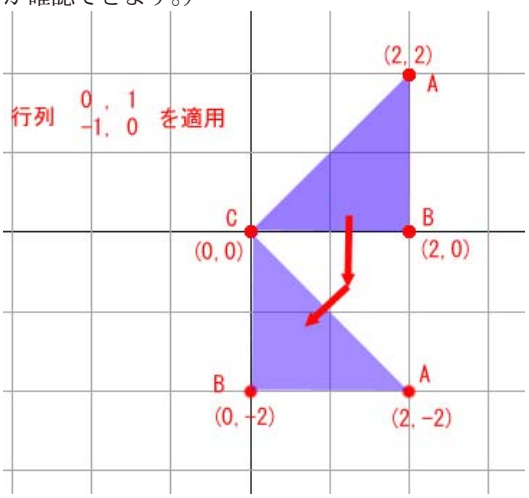


図 5-3
三角形は幾何的に 90 度回転しました。このことから、この行列が回転行列であったことが分かると思います。もちろん、この行列は三角形に限らずあらゆる 2 次元上のジオメトリを 90 度回転させます。
90 度ではなく、もっと任意の角度で自由回転させる場合の行列は、回転角度を θ とした時、
$\cos\theta$　$\sin\theta$

-sin θ cos θ　　となります。

行列を用いたジオメトリの変形

次は変形です。変形には拡大・縮小も含まれます。
ジオメトリの初期状態は回転の説明でのジオメトリと同一とします。
頂点全てに次の行列、
2 , 0
 0 , 2 を掛けると、
頂点はそれぞれ A(4,4) B(4,0) C(0,0) となり、図形は図のようになります。（実際に計算してみると、頂点が変換されることが確認できます。）

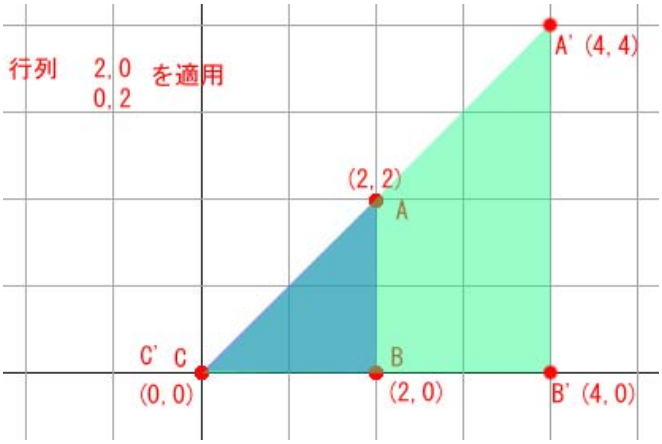


図 5-4

また、次の行列
0.5 , 0
0 , 0.5 を掛けると ,
頂点はそれぞれ A(1,1) B(1,0) C(0,0) となり、図形は図のようになります。

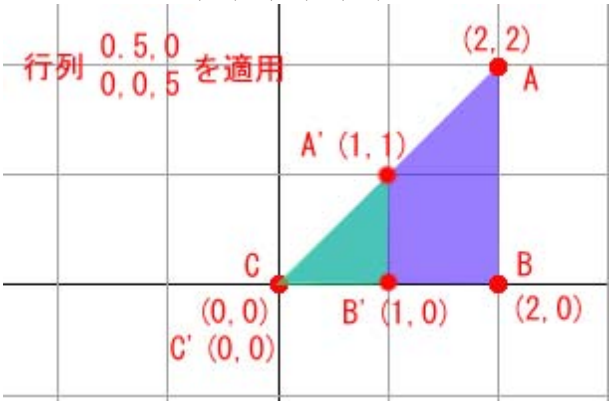


図 5-5

三角形は、拡大・縮小しました。これらの行列は拡大・縮小行列だったわけです。

行列を用いたジオメトリの平行移動

ジオメトリの平行移動を最後に解説するのには理由があります。というのは、平行移動は回転やスケーリングでの座標系及び系と同次元の行列では実現不可能であり、平行移動のためだけに、もう一歩数学的世界に踏み込まなければならないからです。
数学的に言うと、回転・スケーリングは 1 次変換で可能だが平行移動はベクトルの加算でしか実現できないということです。
なぜ不可能かということは、瞬時に分かります。ジオメトリの初期の状態を図と同じとします。そこに如何なる行列を掛けても C 点は変化しません。（ゼロに何も掛けてもゼロです。）よって、平行移動は不可能です。実は、ジオメトリの頂点に原点（C 点）を含めたのは、平行移動が出来ないということを瞬時に理解してもらうためだったのです。頂点が全て非ゼロであったとしても、（これまでの方法では）平行移動が不可能であることに変わりはありません。
しかし、上手いこと行列で平行移動する方法があります。それは “ 同次座標系 ” の概念を取り入れることにより可能にするものです。どのようなものなのか、具体的に説明しましょう。
まずは、X 軸プラス方向に 2、Y 軸プラス方向に 3 の平行移動を例にするし、図ではなく数式によりそのプロセスを示すと、次のようになります。
頂点を全て、1 次元多い 3 次元座標にします。追加した次元の要素は 1 にします。
A(2,2) A(2,2,1)

B(2,0) B(2,0,1)
C(0,0) C(0,0,1)
そして、行列も 1 次元多い 33 行列を掛けます
1 , 0 , 0
0 , 1 , 0
2 , 3 , 1
結果はそれぞれ、A(4,5,1) B(4,3,1) C(2,3,1) になります。ここで、第 3 次元の要素を無視して、2 次元座標のみに着目すると A(4,5) B(4,3) C(2,3) というふうに綺麗に平行移動していることがわかります。この次元を追加した座標を“同次座標”と呼び、同次座標系により平行移動行列を作ることが出来ました。
なお、回転、スケーリング、平行移動のような座標の線形変換をアフィン変換とも呼びます。

同次座標について

同次座標（homogeneous　coordinate「ホモジーニアス　コーオーディネイト」）とは、一言で言うと、“本来の次元より次元を増やした座標”です。斉次座標とも呼ばれています。
二言三言で言うと、“行列による平行移動を可能にすることを目的に、次元を追加し、その追加した次元の要素が移動倍率を表している座標”となります。
平行移動の説明の中で、頂点座標の次元を一つ追加し、その要素を 1 にしましたよね、それを図にすると図になります。

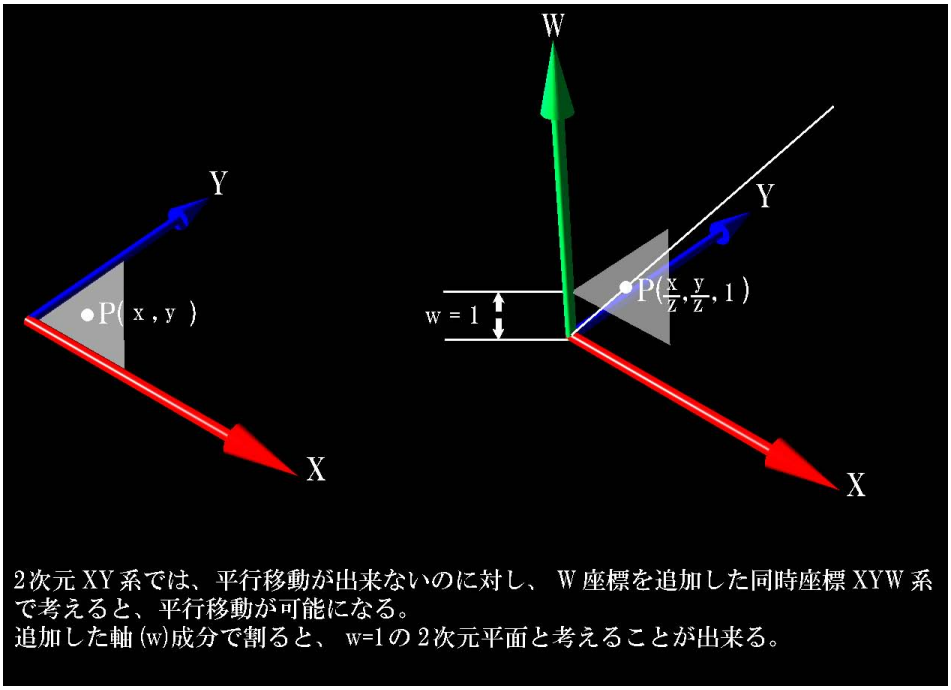


図 5-6

追加した第 3 の座標軸を w とすると、ちょうど w 軸方向に 1 だけ浮いたイメージになります。（但し、見かけは 3 次元空間に見えますが、これはあくまでの 2 次元の同次座標系です。）このように 2 次元の座標を 2 次同次座標系で考えると移動行列を掛けることができるようになり、上手く移動できることはすでに説明したとおりです。

行列の結合（変換の合成）

各種変換は合成することができます。変換を合成するとはどうゆう事かというと、変換行列同士を前もって掛け合わせて、複数の変換を一つの行列で行うというものです。個々の変換行列全てを掛け合わせた出来上がった新たな行列は、全ての変換の性質を持った行列となり、座標にその行列を 1 度掛けるだけで、全ての変換が反映されます。
例えば、あるジオメトリを①時計回りの 90 度回転させた後、② 2 倍に拡大し、③最後に、X 軸方向に 2、Y 軸方向に 3 平行移動させるという 3 つの変換を連続して行う場合を考えてみましょう。
今、ジオメトリの 1 つの頂点が (1,2) にあるとします、その頂点に対する計算式と結果は次のようになります。
回転変換

　　0,1,0
(1,2,1) x -1,0,0 = (-2,1,1)
　　0,0,1
拡大変換
　　2,0,0

(-2,1,1) x 0,2,0 = (-4,2,1)
0,0,1
平行移動変換
1,0,0
(-4,2,1) x 0,1,0 = (-2,5,1)
2,3,1

最終的にこの頂点は　2 次の同次座標で (-2,5,1)、すなわち 2 次元座標 (-2,5) に移動しました。この座標を憶えておいてください。

今度は、まず、最初に全ての変換行列同士を掛け合わせます。（結合します。）
0,1,0　2,0,0　0,2,0
-1,0,0 x 0,2,0 = -2,0,0
0,0,1　0,0,1　0,0,1

0,2,0　1,0,0　0,2,0
-2,0,0 x 0,1,0 = -2,0,0
0,0,1　2,3,1　2,3,1

全ての変換行列を 1 つにまとめた行列を頂点座標に掛けます。
0,2,0
(1,2,1) x -2,0,0 = (-2,5,1)
2,3,1

最終的にこの頂点は　2 次の同次座標で (-2,5,1)、すなわち 2 次元座標 (-2,5) に移動しました。これは、先の結果と同じです。

各変換をひとつずつ別々に行ったので頂点に対して計 3 回の計算をしましたが、行列の結合を前もって行うことにより、頂点と行列の計算を一回で行うことが出来ます。

Direct3D における変換行列

勘の良い読者ならもう予想しているかもしれませんが、3 次元空間において平行移動する場合も 2 次元の場合と全く同様の理由から、1 次元多い系で変換します。つまり、3 次元より 1 つ次元の多い 4 次元＊系上での変換を行います。
Direct3D では、他の回転やスケーリング変換もそれに統一して 1 次元多い系で変換します。なぜかというと、そのほうが、統一されてすっきりするし、一番大きな理由は、変換の種類に関係なく行列の結合を画一的に行えるからです。もしも、回転・スケーリング変換に 33 行列を使用して、一方で平行移動の変換に 44 行列を使用したとしたら、変換行列同士の結合は出来ません。
そのような理由により Direct3D では最も次元数の多い系に合わせています。つまり、Direct3D において全ての変換は同次座標系で変換しているということになります。
これらが、先に述べた通り Direct3D の行列が 44 行列の理由です。
なお、Direct3D は 2 次元も扱いますが、2 次元の変換にも 44 行列を使います、同次座標は、追加する次元数が 1 つだけと決まっているわけではありません。見かけは 4 次元の座標でも 2 次の同次座標として考えることが出来ます。例えば点 P(2,3) という 2 次元座標の同次座標として点 Q(2,3,0,1) を考え、それに 44 行列を掛けことにより 2 次元平面系での変換をすることが出来ます。（Z 成分はゼロにします）

ジオメトリを動かす（ワールドトランスフォーム）

移動を考えるにあたり理解しなければならないことがあります。それは “ 配置 ” です。配置とはローカル座標を絶対座標に変換することです。配置方法を知らなければ移動はできません、そして、方法だではなく配置の背景理論を知っておくことが好ましいでしょう。背景理論の理解なしにプログラミングすることは可能ですが、何かのサンプル通りの出力しか得られず、コードの発展性は極めて限定されることでしょう。これはプログラミング全般について言えることですので、是非、3D ジオメトリ、ここではメッシュの配置についてコードとともに背景理論を理解するようにしてください。
メッシュを思い通りに配置することができれば、移動することは簡単です。というよりも移動は “ 連続した配置 ” に他ならず、両者は本質的に同一の概念です。先にも触れましたが、移動は “ 連続した再配置 ” 以外のなにものでもありません。どうゆうことかと言うと、ある位置に配置したジオメトリを 30 分の 1 秒後に横に 1 単位ずらして再配置していくことをイメージしてください。それは “ 移動 ” です。連続して行うともっとよく移動していることを確認できることとなります。これは回転、スケーリングについても全く同様の考え方です。
さて、なぜ配置ということを改めて理解しなければならないのでしょうか？ 2D の場合、配置は簡単な概念です。座標（x,y）に絵を描画すればいいだけのことです。改めて配置という概念を理解する必要はなく、そもそも意識せずに配置を行うことができました。
しかし、3D となると話は違います。配置するには少々の数学知識が必要なのです。それは先に解説したベクトルと行列です。Direct3D はもちろん、ビデオカードのハードウェアは行列を利用することを想定しハードウェア的に行列処理が実装されているので、行列及び行列を使用する意味を理解することは必須です。

同じメッシュを別々の位置に配置する

サンプルプロジェクト名：Chapter5-1

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5:
 6: LPDIRECT3D9 pD3d;
 7: LPDIRECT3DDEVICE9 pDevice;
 8: LPD3DXMESH pMesh = NULL;
 9: D3DMATERIAL9* pMeshMaterials = NULL;
10: LPDIRECT3DTEXTURE9* pMeshTextures  = NULL;
11: DWORD dwNumMaterials = 0;
12: D3DXVECTOR3 vecChips[3+1];
13: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
14: HRESULT InitD3d(HWND);
15: VOID Render();
16: VOID RefreshMatrices(D3DXVECTOR3*);
17: VOID FreeDx();
18:
19: //
20: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
21: // アプリケーションのエントリー関数
22: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
23: {
24:       HWND hWnd=NULL;
25:       MSG msg;
26:       // ウィンドウの初期化
27:       static char szAppName[] = "Chapter5-1" ;
28:     WNDCLASSEX  wndclass ;
29:
30:     wndclass.cbSize          = sizeof (wndclass) ;
31:     wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
32:     wndclass.lpfnWndProc     = WndProc ;
33:     wndclass.cbClsExtra      = 0 ;
34:     wndclass.cbWndExtra      = 0 ;
35:     wndclass.hInstance       = hInst ;
36:     wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
37:     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
38:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
39:     wndclass.lpszMenuName = NULL ;
40:     wndclass.lpszClassName = szAppName ;
41:     wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
42:
43:     RegisterClassEx (&wndclass) ;
44:
45:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
46:                 0,0,800,600,NULL,NULL,hInst,NULL) ;
47:
48:    ShowWindow (hWnd,SW_SHOW) ;
49:    UpdateWindow (hWnd) ;
50:       // ダイレクト3D の初期化関数を呼ぶ
51:       if(FAILED(InitD3d(hWnd)))
52:       {
53:                 return 0;
54:       }
55:       // Chips メッシュ用の、異なる3つの位置 （座標） を初期化する
56:       vecChips[0].x=0;
57:       vecChips[0].y=0;
58:       vecChips[0].z=2;
59:
60:       vecChips[1].x=-2;
61:       vecChips[1].y=2;
62:       vecChips[1].z=0;
```

```
63:
64:       vecChips[2].x=2;
65:       vecChips[2].y=0;
66:       vecChips[2].z=-2;
67:       // メッセージループ
68:     ZeroMemory( &msg, sizeof(msg) );
69:     while( msg.message!=WM_QUIT )
70:      {
71:                       if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
72:        {
73:                                TranslateMessage( &msg );
74:             DispatchMessage( &msg );
75:        }
76:        else
77:                       {
78:                                Render();
79:                       }
80:       }
81:       // メッセージループから抜けたらオブジェクトを全て開放する
82:       FreeDx();
83:       // OS に戻る （アプリケーションを終了する）
84:      return (INT)msg.wParam ;
85: }
86:
87: //
88: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
89: // ウィンドウプロシージャ関数
90: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
91: {
92:       switch(iMsg)
93:       {
94:                       case WM_DESTROY:
95:                                PostQuitMessage(0);
96:                       break;
97:                       case WM_KEYDOWN:
98:                       switch((CHAR)wParam)
99:                       {
100:                               case VK_ESCAPE:
101:                                        PostQuitMessage(0);
102:                               break;
103:                      }
104:                      break;
105:      }
106:      return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
107: }
108:
109: //
110: //HRESULT InitD3d(HWND hWnd)
111: // ダイレクト 3D の初期化関数
112: HRESULT InitD3d(HWND hWnd)
113: {
114:      //「Direct3D」 オブジェクトの作成
115:      if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
116:      {
117:                       MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
118:       return E_FAIL;
119:      }
120:      //「DIRECT3D デバイス」 オブジェクトの作成
121:      D3DPRESENT_PARAMETERS d3dpp;
122:      ZeroMemory( &d3dpp, sizeof(d3dpp) );
123:      d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
124:      d3dpp.BackBufferCount=1;
125:      d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
126:      d3dpp.Windowed = TRUE;
127:     d3dpp.EnableAutoDepthStencil = TRUE;
128:     d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
129:
130:       if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
131:                                D3DCREATE_MIXED_VERTEXPROCESSING,
132:                                &d3dpp, &pDevice ) ) )
```

```
133:     {
134:                 MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
135:                 if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
136:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
137:                                 &d3dpp, &pDevice ) ) )
138:                 {
139:                             MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
140:                              return E_FAIL;
141:                 }
142:     }
143:       // X ファイルからメッシュをロードする
144:       LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
145:
146:       if( FAILED( D3DXLoadMeshFromX( "Chips.x", D3DXMESH_SYSTEMMEM,
147:             pDevice, NULL, &pD3DXMtrlBuffer, NULL,
148:                             &dwNumMaterials, &pMesh ) ) )
149:     {
150:             MessageBox(NULL, "X ファイルの読み込みに失敗しました ",NULL, MB_OK);
151:             return E_FAIL;
152:     }
153:       D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
154:     pMeshMaterials = new D3DMATERIAL9[dwNumMaterials];
155:     pMeshTextures  = new LPDIRECT3DTEXTURE9[dwNumMaterials];
156:
157:       for( DWORD i=0; i<dwNumMaterials; i++ )
158:       {
159:                     pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
160:         pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
161:         pMeshTextures[i] = NULL;
162:         if( d3dxMaterials[i].pTextureFilename != NULL &&
163:             lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
164:         {
165:             if( FAILED( D3DXCreateTextureFromFile( pDevice,
166:                                         d3dxMaterials[i].pTextureFilename,
167:                                         &pMeshTextures[i] ) ) )
168:             {
169:                 MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
170:                             }
171:         }
172:     }
173:       pD3DXMtrlBuffer->Release();
174:       // Z バッファー処理を有効にする
175:     pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
176:       // ライトを有効にする
177:       pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
178:       // アンビエントライト（環境光）を設定する
179:        pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
180:        // スペキュラ（鏡面反射）を有効にする
181:        pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
182:       return S_OK;
183: }
184:
185: //
186: //VOID Render()
187: //X ファイルから読み込んだメッシュをレンダリングする関数
188: VOID Render()
189: {
190:       // ライトをあてる 白色ライト、鏡面反射有りに設定
191:        D3DXVECTOR3 vecDirection(1,1,1);
192:       D3DLIGHT9 light;
193:       ZeroMemory( &light, sizeof(D3DLIGHT9) );
194:       light.Type       = D3DLIGHT_DIRECTIONAL;
195:       light.Diffuse.r  = 1.0f;
196:       light.Diffuse.g  = 1.0f;
197:       light.Diffuse.b  = 1.0f;
198:         light.Specular.r=1.0f;
199:         light.Specular.g=1.0f;
200:         light.Specular.b=1.0f;
201:       D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
202:       light.Range        = 200.0f;
```

```
203:     pDevice->SetLight( 0, &light );
204:     pDevice->LightEnable( 0, TRUE );
205:       // レンダリング
206:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
207:                   D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
208:
209:       if( SUCCEEDED( pDevice->BeginScene() ) )
210:       {
211:                   for(DWORD j=0;j<3;j++)
212:                   {
213:                               RefreshMatrices(&vecChips[j]);
214:                               for( DWORD i=0; i<dwNumMaterials; i++ )
215:                               {
216:                                           pDevice->SetMaterial( &pMeshMaterials[i] );
217:                                  pDevice->SetTexture( 0, pMeshTextures[i] );
218:                         pMesh->DrawSubset( i );
219:                               }
220:                   }
221:                   pDevice->EndScene();
222:       }
223:       pDevice->Present( NULL, NULL, NULL, NULL );
224: }
225: //
226: //
227: //
228: VOID RefreshMatrices(D3DXVECTOR3* pvecChips)
229: {
230:       // ワールドトランスフォーム（絶対座標変換）
231:       D3DXMATRIXA16 matWorld,matPosition;
232:       D3DXMatrixIdentity(&matWorld);
233:       D3DXMatrixTranslation(&matPosition,pvecChips->x,pvecChips->y,pvecChips->z);
234:       D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
235:     pDevice->SetTransform( D3DTS_WORLD, &matWorld );
236:       // ビュートランスフォーム（視点座標変換）
237:
238:     D3DXVECTOR3 vecEyePt( 0.0f, 1.0f,-10.0f ); // カメラ（視点） 位置
239:     D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
240:     D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
241:     D3DXMATRIXA16 matView;
242:     D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
243:     pDevice->SetTransform( D3DTS_VIEW, &matView );
244:       // プロジェクショントランスフォーム（射影変換）
245:     D3DXMATRIXA16 matProj;
246:     D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
247:     pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
248: }
249:
250: //
251: //VOID FreeDx()
252: // 作成した DirectX オブジェクトの開放
253: VOID FreeDx()
254: {
255:       SAFE_RELEASE( pMesh );
256:       SAFE_RELEASE( pDevice );
257:       SAFE_RELEASE( pD3d );
258: }
```

仮に、前章 Chapter4-2 のコードそのままで、同じ Chips.x を基にしたメッシュを 3 つ作成し、レンダリングするコードだけを追加したとしましょう。結果はどうなるでしょうか？
3 つのメッシュは同じ位置、同じ向き、同じスケールでピッタリ寸分のズレも無くレンダリングされることとなり、それが 3 つなのか 10 個なのか、見た目で判断するのは不可能です。見た目はただ 1 つのメッシュなのですから（フレームレートが落ちるくらいしか、確認できないでしょう）。「それは、3 個のメッシュが重なっているんだよ」、と誰かに教えられて初めて、「あぁ、そうなんだぁ」、と気付くしかありません。
同じ X ファイルから読み込んだメッシュ、つまりは、ジオメトリ的に同一のメッシュを複数作成してレンダリングしたり、あるいは別々にメッシュを作成することすらせずに、まったく 1 つのメッシュを異なる位置にレンダリングして複数の物体として表現することは、頻繁に行う処理です。やや言葉は悪いですが、このような “ メッシュの使い回し ” をしないゲームの

ほうが、むしろ珍しいのではないでしょうか。
その時に、それぞれのメッシュを、それぞれ固有の位置にレンダリングする必要があります。つまりそれぞれの位置に“配置”するとういことです。
本章サンプル Chapter5-1 では、ひとつのメッシュに、3 つの座標（座標だけ）を用意し、その座標をもとにワールドトランスフォームして、異なる位置にレンダリングします。
前章の Chapter4-2 サンプルのコードから変更している部分は、ワールドトランスフォーム部分と、3 つの異なる座標に表示するために、レンダリング関数を多少変更しているだけなので、解説はその部分のみ行います。
また、前章ではコード上で行列を利用しているにも関わらず解説が無かったので、トランスフォーム部分の理解が曖昧なままになっていることと思います。本章では本サンプルのコードに沿いつつ、Direct3D における行列の使用方法を解説します。
では、コードを、順を追って見ていきましょう。
読み下すコードは、関数 VOID RefreshMatrices(D3DXVECTOR3*) だけです。他の部分は今までと同様です。
関数 RefreshMatrices 内で行っていることは、次の 3 つの変換を行っています。3 つの変換とはワールド、ビュー、プロジェクションです。ここでは、“配置”を理解するのが目的なのでワールド変換にもに着目します。ビュー及びプロジェクショントランスフォームに関しては次節で行います。
この関数自体、短いコードあり、ワールドトランスフォームに関しても 5 行しかありませんので、じっくり見ていきましょう。
D3DXMATRIXA16 matWorld,matPosition;
matWorld は最終的なワールドトランスフォーム行列を、matPosition は平行移動行列を格納するための行列です。
D3DXMatrixIdentity(&matWorld);
D3DXMatrixIdentity は単位行列を作成する関数です。ここでは、簡便のため、ワールドトランスフォーム行列を最初に単位行列で初期化します。単位行列は数字の 1 のように掛け合わせたものを変化させないという性質があります。もし、ジオメトリにこの時点の matWorld でワールドトランスフォームを掛けると、ジオメトリのローカル座標そのままでレンダリングされることとなります。
D3DXMatrixTranslation(&matPosition,pvecChips->x,pvecChips->y,pvecChips->z);
D3DXMatrixTranslation は平行移動行列を作成する関数です。pvecChips 引数で与えられた位置ベクトルを基に matPosition を平行移動行列として初期化しています。
D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
D3DXMatrixMultiply は行列を結合する関数です。第 2 引数に第 3 引数を掛け合わせ、その結果を第 1 引数に格納します。ここでは平行移動行列として初期化された matPosition を matWorld に結合しています。
pDevice->SetTransform(D3DTS_WORLD, &matWorld);
SetTransform は IDirect3DDevice9 インターフェイスのメソッド（関数）であり、様々な種類の変換をレンダリングパイプラインに指示ものです。ここでは、「D3DTS_WORLD ワールドトランスフォームを matWorld 行列により行いなさい」という指示を出していることになります。なお、レンダリングパイプラインがこの指示を実行するのは SetTransform をコールした時ではなく、レンダリング時です。そういう意味で、SetTransform により“（レンダリング仕様を）登録しておく”と言ったほうが良いかもしれません。
ワールドトランスフォームの最低限のコードについての解説は以上です。

異なるメッシュを異なる位置に配置する

サンプルプロジェクト名：Chapter5-1-b

同じメッシュの使い回しではなく、全て異なるメッシュを読み込むサンプルも念のため作りました。このサンプルに関しては、とりわけ解説しなければならないことは無いと考えます。というのは、ワールドトランスフォームに関して chapter 5-1 と全く同様だからです。
THING という構造体を定義して、その構造体にメッシュ情報を詰め込むようにしています。そして異なるメッシュの読み込みとレンダリング関数にそれぞれ、一階層下の関数をそれぞれ作成し (InitD3d 関数には InitThing 関数を、Render 関数には RenderThing 関数を)、コードが肥大しないようにしています。これは、特別なことではないと思います。また、Direct3D 固有の仕組みという訳でもないので敢えて解説はしません。
さて、配置の概要が理解できたところで、平行移動してみましょう。配置ができていれば移動の準備が出来ていることになります。対象ジオメトリの位置（位置ベクトル）を時間とともに好きなように変化させてやればジオメトリは移動します。

サンプルプロジェクト名：Chapter5-2

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5:
 6: LPDIRECT3D9 pD3d;
 7: LPDIRECT3DDEVICE9 pDevice;
 8: LPD3DXMESH pMesh = NULL;
 9: D3DMATERIAL9* pMeshMaterials = NULL;
10: LPDIRECT3DTEXTURE9* pMeshTextures  = NULL;
11: DWORD dwNumMaterials = 0;
```

```
12: FLOAT fPosX=0,fPosY=0,fPosZ=0;
13:
14: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
15: HRESULT InitD3d(HWND);
16: VOID Render();
17: VOID FreeDx();
18:
19: //
20: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
21: // アプリケーションのエントリー関数
22: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
23: {
24:         HWND hWnd=NULL;
25:         MSG msg;
26:         // ウィンドウの初期化
27:         static char szAppName[] = "Chapter5-2" ;
28:       WNDCLASSEX  wndclass ;
29:
30:       wndclass.cbSize            = sizeof (wndclass) ;
31:       wndclass.style             = CS_HREDRAW | CS_VREDRAW ;
32:       wndclass.lpfnWndProc     = WndProc ;
33:       wndclass.cbClsExtra      = 0 ;
34:       wndclass.cbWndExtra      = 0 ;
35:       wndclass.hInstance         = hInst ;
36:       wndclass.hIcon             = LoadIcon (NULL, IDI_APPLICATION) ;
37:       wndclass.hCursor           = LoadCursor (NULL, IDC_ARROW) ;
38:       wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
39:       wndclass.lpszMenuName = NULL ;
40:       wndclass.lpszClassName = szAppName ;
41:       wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
42:
43:       RegisterClassEx (&wndclass) ;
44:
45:       hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
46:                   0,0,800,600,NULL,NULL,hInst,NULL) ;
47:
48:      ShowWindow (hWnd,SW_SHOW) ;
49:      UpdateWindow (hWnd) ;
50:        // ダイレクト3D の初期化関数を呼ぶ
51:        if(FAILED(InitD3d(hWnd)))
52:        {
53:                    return 0;
54:        }
55:        // メッセージループ
56:      ZeroMemory( &msg, sizeof(msg) );
57:      while( msg.message!=WM_QUIT )
58:      {
59:                    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
60:          {
61:                                TranslateMessage( &msg );
62:              DispatchMessage( &msg );
63:          }
64:          else
65:                    {
66:                                Render();
67:                    }
68:        }
69:        // メッセージループから抜けたらオブジェクトを全て開放する
70:        FreeDx();
71:        // OS に戻る （アプリケーションを終了する）
72:      return (INT)msg.wParam ;
73: }
74:
75: //
76: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
77: // ウィンドウプロシージャ関数
78: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
79: {
80:        switch(iMsg)
81:        {
```

```
 82:                     case WM_DESTROY:
 83:                             PostQuitMessage(0);
 84:                     break;
 85:                     case WM_KEYDOWN:
 86:                     switch((CHAR)wParam)
 87:                     {
 88:                             case VK_ESCAPE:
 89:                                     PostQuitMessage(0);
 90:                             break;
 91:                             case VK_LEFT:
 92:                fPosX-=0.01f;
 93:                             break;
 94:                             case VK_RIGHT:
 95:                fPosX+=0.01f;
 96:                             break;
 97:                             case VK_DOWN:
 98:                fPosY-=0.01f;
 99:                             break;
100:                             case VK_UP:
101:                fPosY+=0.01f;
102:                             break;
103:                             case 'Z':
104:                fPosZ-=0.01f;
105:                             break;
106:                             case 'X':
107:                fPosZ+=0.01f;
108:                             break;;
109:                     }
110:                     break;
111:     }
112:     return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
113: }
114:
115: //
116: //HRESULT InitD3d(HWND hWnd)
117: // ダイレクト 3D の初期化関数
118: HRESULT InitD3d(HWND hWnd)
119: {
120:     //「Direct3D」 オブジェクトの作成
121:     if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
122:     {
123:             MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
124:       return E_FAIL;
125:     }
126:     //「DIRECT3D デバイス」 オブジェクトの作成
127:     D3DPRESENT_PARAMETERS d3dpp;
128:     ZeroMemory( &d3dpp, sizeof(d3dpp) );
129:     d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
130:     d3dpp.BackBufferCount=1;
131:     d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
132:     d3dpp.Windowed = TRUE;
133:   d3dpp.EnableAutoDepthStencil = TRUE;
134:   d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
135:
136:      if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
137:                                      D3DCREATE_MIXED_VERTEXPROCESSING,
138:                                      &d3dpp, &pDevice ) ) )
139:   {
140:             MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
141:             if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
142:                                      D3DCREATE_MIXED_VERTEXPROCESSING,
143:                                      &d3dpp, &pDevice ) ) )
144:             {
145:                             MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
146:                              return E_FAIL;
147:             }
148:   }
149:     // X ファイルからメッシュをロードする
150:     LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
151:
```

```
152:     if( FAILED( D3DXLoadMeshFromX( "Corn.x", D3DXMESH_SYSTEMMEM,
153:         pDevice, NULL, &pD3DXMtrlBuffer, NULL,
154:                         &dwNumMaterials, &pMesh ) ) )
155:   {
156:         MessageBox(NULL, "X ファイルの読み込みに失敗しました ",NULL, MB_OK);
157:         return E_FAIL;
158:   }
159:     D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
160:   pMeshMaterials = new D3DMATERIAL9[dwNumMaterials];
161:   pMeshTextures  = new LPDIRECT3DTEXTURE9[dwNumMaterials];
162:
163:     for( DWORD i=0; i<dwNumMaterials; i++ )
164:     {
165:                 pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
166:       pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
167:       pMeshTextures[i] = NULL;
168:       if( d3dxMaterials[i].pTextureFilename != NULL &&
169:         lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
170:       {
171:         if( FAILED( D3DXCreateTextureFromFile( pDevice,
172:                                     d3dxMaterials[i].pTextureFilename,
173:                                     &pMeshTextures[i] ) ) )
174:         {
175:             MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
176:                         }
177:       }
178:   }
179:     pD3DXMtrlBuffer->Release();
180:     // Z バッファー処理を有効にする
181:   pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
182:     // ライトを有効にする
183:     pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
184:     // アンビエントライト（環境光）を設定する
185:      pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
186:      // スペキュラ（鏡面反射）を有効にする
187:      pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
188:     return S_OK;
189: }
190:
191: //
192: //VOID Render()
193: //X ファイルから読み込んだメッシュをレンダリングする関数
194: VOID Render()
195: {
196:     // ワールドトランスフォーム（絶対座標変換）
197:      D3DXMATRIXA16 matWorld,matPosition,matRotation;
198:      D3DXMatrixIdentity(&matWorld);
199:      D3DXMatrixTranslation(&matPosition,fPosX,fPosY,fPosZ);
200:      D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
201:   pDevice->SetTransform( D3DTS_WORLD, &matWorld );
202:     // ビュートランスフォーム（視点座標変換）
203:   D3DXVECTOR3 vecEyePt( 0.0f, 1.0f,-3.0f ); // カメラ（視点） 位置
204:   D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
205:   D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
206:   D3DXMATRIXA16 matView;
207:   D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
208:   pDevice->SetTransform( D3DTS_VIEW, &matView );
209:     // プロジェクショントランスフォーム（射影変換）
210:   D3DXMATRIXA16 matProj;
211:   D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
212:   pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
213:     // ライトをあてる 白色ライト、鏡面反射有りに設定
214:     D3DXVECTOR3 vecDirection(1,1,1);
215:   D3DLIGHT9 light;
216:   ZeroMemory( &light, sizeof(D3DLIGHT9) );
217:   light.Type       = D3DLIGHT_DIRECTIONAL;
218:   light.Diffuse.r  = 1.0f;
219:   light.Diffuse.g  = 1.0f;
220:   light.Diffuse.b  = 1.0f;
221:     light.Specular.r=1.0f;
```

```
222:        light.Specular.g=1.0f;
223:        light.Specular.b=1.0f;
224:      D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
225:      light.Range        = 200.0f;
226:      pDevice->SetLight( 0, &light );
227:      pDevice->LightEnable( 0, TRUE );
228:        // レンダリング
229:        pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
230:                        D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
231:
232:        if( SUCCEEDED( pDevice->BeginScene() ) )
233:        {
234:                        for( DWORD i=0; i<dwNumMaterials; i++ )
235:        {
236:            pDevice->SetMaterial( &pMeshMaterials[i] );
237:            pDevice->SetTexture( 0, pMeshTextures[i] );
238:            pMesh->DrawSubset( i );
239:        }
240:                        pDevice->EndScene();
241:        }
242:        pDevice->Present( NULL, NULL, NULL, NULL );
243: }
244:
245: //
246: //VOID FreeDx()
247: // 作成した DirectX オブジェクトの開放
248: VOID FreeDx()
249: {
250:        SAFE_RELEASE( pMesh );
251:        SAFE_RELEASE( pDevice );
252:        SAFE_RELEASE( pD3d );
253: }
```

D3DXMATRIXA16 matWorld,matPosition,matRotation;
D3DXMatrixIdentity(&matWorld);
D3DXMatrixTranslation(&matPosition,fPosX,fPosY,fPosZ);
D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
pDevice->SetTransform(D3DTS_WORLD, &matWorld);
このワールドトランスフォーム処理は、chapter5-1 と全く同様です。
D3DXMatrixTranslation 関数に渡している引数 fPosX,fPosY,fPosZ はユーザーのキー入力により変化するようにウィンドウプロシージャー内で更新していますので、キー入力とともにジオメトリの位置が変化する、つまり移動するということになります。

ジオメトリの回転方法について解説します。ここでも平行移動と同様、注目するコードはワールドトランスフォーム部分だけです。

サンプルプロジェクト名：Chapter5-3

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5:
 6: LPDIRECT3D9 pD3d;
 7: LPDIRECT3DDEVICE9 pDevice;
 8: LPD3DXMESH pMesh = NULL;
 9: D3DMATERIAL9* pMeshMaterials = NULL;
10: LPDIRECT3DTEXTURE9* pMeshTextures  = NULL;
11: DWORD dwNumMaterials = 0;
12: FLOAT fHeading=0,fPitch=0,fBank=0;
13:
14: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
15: HRESULT InitD3d(HWND);
16: VOID Render();
17: VOID FreeDx();
18:
19: //
```

```
20: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
21: // アプリケーションのエントリー関数
22: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
23: {
24:         HWND hWnd=NULL;
25:         MSG msg;
26:         // ウィンドウの初期化
27:         static char szAppName[] = "Chapter5-3" ;
28:       WNDCLASSEX  wndclass ;
29:
30:       wndclass.cbSize            = sizeof (wndclass) ;
31:       wndclass.style             = CS_HREDRAW | CS_VREDRAW ;
32:       wndclass.lpfnWndProc     = WndProc ;
33:       wndclass.cbClsExtra       = 0 ;
34:       wndclass.cbWndExtra      = 0 ;
35:       wndclass.hInstance         = hInst ;
36:       wndclass.hIcon             = LoadIcon (NULL, IDI_APPLICATION) ;
37:       wndclass.hCursor           = LoadCursor (NULL, IDC_ARROW) ;
38:       wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
39:       wndclass.lpszMenuName = NULL ;
40:       wndclass.lpszClassName = szAppName ;
41:       wndclass.hIconSm          = LoadIcon (NULL, IDI_APPLICATION) ;
42:
43:       RegisterClassEx (&wndclass) ;
44:
45:       hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
46:                   0,0,800,600,NULL,NULL,hInst,NULL) ;
47:
48:      ShowWindow (hWnd,SW_SHOW) ;
49:      UpdateWindow (hWnd) ;
50:        // ダイレクト3D の初期化関数を呼ぶ
51:        if(FAILED(InitD3d(hWnd)))
52:        {
53:                    return 0;
54:        }
55:        // メッセージループ
56:      ZeroMemory( &msg, sizeof(msg) );
57:      while( msg.message!=WM_QUIT )
58:       {
59:                     if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
60:          {
61:                                 TranslateMessage( &msg );
62:              DispatchMessage( &msg );
63:          }
64:          else
65:                    {
66:                                 Render();
67:                    }
68:         }
69:         // メッセージループから抜けたらオブジェクトを全て開放する
70:         FreeDx();
71:         // OS に戻る （アプリケーションを終了する）
72:       return (INT)msg.wParam ;
73: }
74:
75: //
76: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
77: // ウィンドウプロシージャ関数
78: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
79: {
80:         switch(iMsg)
81:         {
82:                    case WM_DESTROY:
83:                               PostQuitMessage(0);
84:                    break;
85:                    case WM_KEYDOWN:
86:                    switch((CHAR)wParam)
87:                    {
88:                               case VK_ESCAPE:
89:                                          PostQuitMessage(0);
```

```
90:                                 break;
91:                                 case VK_LEFT:
92:                 fHeading-=0.1f;
93:                                 break;
94:                                 case VK_RIGHT:
95:                 fHeading+=0.1f;
96:                                 break;
97:                                 case VK_DOWN:
98:                 fPitch-=0.1f;
99:                                 break;
100:                                case VK_UP:
101:                fPitch+=0.1f;
102:                                break;
103:                                case 'Z':
104:                fBank-=0.1f;
105:                                break;
106:                                case 'X':
107:                fBank+=0.1f;
108:                                break;;
109:                    }
110:                    break;
111:        }
112:        return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
113: }
114:
115: //
116: //HRESULT InitD3d(HWND hWnd)
117: // ダイレクト 3D の初期化関数
118: HRESULT InitD3d(HWND hWnd)
119: {
120:        //「Direct3D」 オブジェクトの作成
121:        if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
122:        {
123:                    MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
124:          return E_FAIL;
125:        }
126:        //「DIRECT3D デバイス」 オブジェクトの作成
127:        D3DPRESENT_PARAMETERS d3dpp;
128:        ZeroMemory( &d3dpp, sizeof(d3dpp) );
129:        d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
130:        d3dpp.BackBufferCount=1;
131:        d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
132:        d3dpp.Windowed = TRUE;
133:      d3dpp.EnableAutoDepthStencil = TRUE;
134:      d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
135:
136:         if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
137:                                         D3DCREATE_MIXED_VERTEXPROCESSING,
138:                                         &d3dpp, &pDevice ) ) )
139:      {
140:                    MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
141:                    if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
142:                                         D3DCREATE_MIXED_VERTEXPROCESSING,
143:                                         &d3dpp, &pDevice ) ) )
144:                    {
145:                                MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
146:                                 return E_FAIL;
147:                    }
148:      }
149:        // X ファイルからメッシュをロードする
150:        LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
151:
152:        if( FAILED( D3DXLoadMeshFromX( "Tomato.x", D3DXMESH_SYSTEMMEM,
153:              pDevice, NULL, &pD3DXMtrlBuffer, NULL,
154:                                         &dwNumMaterials, &pMesh ) ) )
155:      {
156:              MessageBox(NULL, "X ファイルの読み込みに失敗しました ",NULL, MB_OK);
157:              return E_FAIL;
158:      }
159:        D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
```

```
160:    pMeshMaterials = new D3DMATERIAL9[dwNumMaterials];
161:    pMeshTextures  = new LPDIRECT3DTEXTURE9[dwNumMaterials];
162:
163:      for( DWORD i=0; i<dwNumMaterials; i++ )
164:      {
165:                  pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
166:        pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
167:        pMeshTextures[i] = NULL;
168:        if( d3dxMaterials[i].pTextureFilename != NULL &&
169:            lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
170:        {
171:            if( FAILED( D3DXCreateTextureFromFile( pDevice,
172:                                        d3dxMaterials[i].pTextureFilename,
173:                                        &pMeshTextures[i] ) ) )
174:            {
175:                MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
176:                            }
177:        }
178:    }
179:      pD3DXMtrlBuffer->Release();
180:      // Z バッファー処理を有効にする
181:    pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
182:      // ライトを有効にする
183:      pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
184:      // アンビエントライト（環境光）を設定する
185:       pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
186:       // スペキュラ（鏡面反射）を有効にする
187:       pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
188:      return S_OK;
189: }
190:
191: //
192: //VOID Render()
193: //X ファイルから読み込んだメッシュをレンダリングする関数
194: VOID Render()
195: {
196:      // ワールドトランスフォーム（絶対座標変換）
197:       D3DXMATRIXA16 matWorld,matRotation,matRotation2;
198:       D3DXMatrixIdentity(&matWorld);
199:       D3DXMatrixIdentity(&matRotation);
200:       D3DXMatrixRotationX(&matRotation2,fPitch);
201:       D3DXMatrixMultiply(&matRotation,&matRotation,&matRotation2);
202:       D3DXMatrixRotationY(&matRotation2,fHeading);
203:       D3DXMatrixMultiply(&matRotation,&matRotation,&matRotation2);
204:       D3DXMatrixRotationZ(&matRotation2,fBank);
205:       D3DXMatrixMultiply(&matRotation,&matRotation,&matRotation2);
206:       D3DXMatrixMultiply(&matWorld,&matWorld,&matRotation);
207:     pDevice->SetTransform( D3DTS_WORLD, &matWorld );
208:       // ビュートランスフォーム（視点座標変換）
209:
210:     D3DXVECTOR3 vecEyePt( 0.0f, 1.0f,-3.0f ); // カメラ（視点）位置
211:     D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
212:     D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
213:     D3DXMATRIXA16 matView;
214:     D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
215:     pDevice->SetTransform( D3DTS_VIEW, &matView );
216:       // プロジェクショントランスフォーム（射影変換）
217:     D3DXMATRIXA16 matProj;
218:     D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
219:     pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
220:       // ライトをあてる 白色ライト、鏡面反射有りに設定
221:       D3DXVECTOR3 vecDirection(1,1,1);
222:     D3DLIGHT9 light;
223:     ZeroMemory( &light, sizeof(D3DLIGHT9) );
224:     light.Type       = D3DLIGHT_DIRECTIONAL;
225:     light.Diffuse.r  = 1.0f;
226:     light.Diffuse.g  = 1.0f;
227:     light.Diffuse.b  = 1.0f;
228:       light.Specular.r=1.0f;
229:       light.Specular.g=1.0f;
```

```
230:        light.Specular.b=1.0f;
231:      D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
232:      light.Range        = 200.0f;
233:      pDevice->SetLight( 0, &light );
234:      pDevice->LightEnable( 0, TRUE );
235:        // レンダリング
236:        pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
237:                        D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
238:
239:        if( SUCCEEDED( pDevice->BeginScene() ) )
240:        {
241:                        for( DWORD i=0; i<dwNumMaterials; i++ )
242:        {
243:            pDevice->SetMaterial( &pMeshMaterials[i] );
244:            pDevice->SetTexture( 0, pMeshTextures[i] );
245:            pMesh->DrawSubset( i );
246:        }
247:                        pDevice->EndScene();
248:        }
249:        pDevice->Present( NULL, NULL, NULL, NULL );
250: }
251:
252: //
253: //VOID FreeDx()
254: // 作成した DirectX オブジェクトの開放
255: VOID FreeDx()
256: {
257:       SAFE_RELEASE( pMesh );
258:       SAFE_RELEASE( pDevice );
259:       SAFE_RELEASE( pD3d );
260: }
```

D3DXMATRIXA16 matWorld,matRotation,matHeading,matPitch,matBank;
最終的なワールドトランスフォーム行列 (matWorld)、ヘディング回転用行列 (matHeading)、ピッチ回転用行列 (matPitch)、バンク回転用行列 (matBank) を用意（宣言）します。ここで、3 種類の回転の呼び方について触れておきます。ヘディングとは Y 軸を基準にした回転です。ピッチは X 軸を基準にした回転、ロールは Z 軸を基準にした回転です。
DirectX のヘルプでは、それぞれヨー、ピッチ（これは同じ呼び方）、ロールと呼んでいます。ヨー、ピッチ、ロールの方は航空用語のようで格好良いですが、筆者は 3D ソフトでの呼び方に慣れているせいかヘディング、ピッチ、バンクのほうが、馴染みがあります。
ヘディング（Heading）、ヨー (Yaw)　Y 軸周りの回転
ピッチ（Pitch）　X 軸周りの回転
バンク（Bank）、ロール (Roll)　Z 軸周りの回転

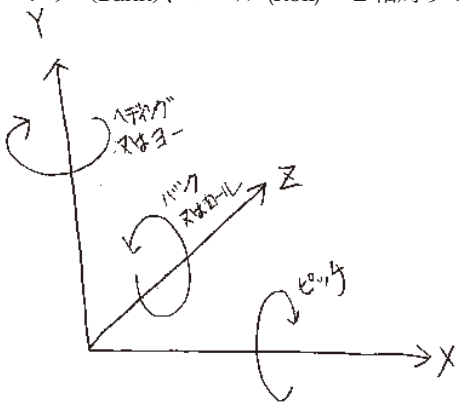


図 5-7
D3DXMatrixIdentity(&matWorld);
例により、まず最終的なワールドトランスフォーム行列である matWorld を単位行列として初期化します。
D3DXMatrixIdentity(&matRotation);
3 軸の回転行列をまとめる回転行列 matRotation を単位行列として初期化しておきます。単位行列で初期化しておかないと（メンバーがゼロだと）、この後で 3 軸の回転行列を掛け合わせていってもゼロになってしまいますので、ここで初期化しておきます。
D3DXMatrixRotationY(&matHeading,fHeading);
matHeading を Y 軸周りの回転行列として初期化します。

D3DXMatrixRotationX(&matPitch,fPitch);
matPitch を X 軸周りの回転行列として初期化します。
D3DXMatrixRotationZ(&matBank,fBank);
matBank を Z 軸周りの回転行列として初期化します。

見て分かる通り、D3DXMatrixRotation は座標軸周りの回転行列を作成します。最後の X、Y,Z が軸を表しています。
D3DXMatrixMultiply(&matRotation,&matRotation,&matHeading);
ヘディング回転を、掛け合わせます。
D3DXMatrixMultiply(&matRotation,&matRotation,&matPitch);
ピッチ回転を、掛け合わせます。この時点で matRotation はヘディングとピッチの回転行列になっています。
D3DXMatrixMultiply(&matRotation,&matRotation,&matBank);
バンク回転を、掛け合わせます。この時点で matRotation はヘディング、ピッチ、バンク回転全ての合成行列になっています。なお、同じ種類の行列であれば、この場合は回転行列同士なので順番は任意です。ピッチからでも、バンクからでも掛け合わせることが出来ます。しかし、例えば回転行列と移動行列という種類が異なる行列同士の合成は、合成の順番で結果が変わってしまいますので注意してください。
D3DXMatrixMultiply(&matWorld,&matWorld,&matRotation);
最終的な演算結果ワールドトランスフォーム行列を matWorld に格納します。
pDevice->SetTransform(D3DTS_WORLD, &matWorld);
レンダリングパイプラインにワールドトランスフォーム行列を登録します。
本サンプルコードでは、分かりやすさを優先しているため、行列の中間演算を“しっかり”（？）行っています。3 軸の回転行列について、その都度中間式用行列としての matRotation を用意しておいて、それに掛け合わせています。（合成を重ねています）。行列の乗算は、全部で 4 回行っていますが、これらは次のように短縮できます。

D3DXMatrixMultiply(&matWorld,&matWorld,&matHeading);
D3DXMatrixMultiply(&matWorld,&matWorld,&matPitch);
D3DXMatrixMultiply(&matWorld,&matWorld,&matBank);
最初から最終的なワールドトランスフォーム行列 matWorld に掛け合わせていけば乗算は 3 回で済み、また、回転の中間式格納用の matRotation を用意する必要がなくなります。
また、次のように回転行列を作成した後にすぐ乗算していけば、3 軸全ての回転行列格納用行列を用意する必要がなくなります。
D3DXMatrixRotationY(&matRotation,fHeading);
D3DXMatrixMultiply(&matWorld,&matWorld,&matRotation);
D3DXMatrixRotationX(&matRotation,fPitch);
D3DXMatrixMultiply(&matWorld,&matWorld,&matRotation);
D3DXMatrixRotationZ(&matRotation,fBank);
D3DXMatrixMultiply(&matWorld,&matWorld,&matRotation);

ジオメトリの拡大・縮小（スケーリング）方法について解説します。平行移動、回転と同様に、注目するコードはワールドトランスフォーム部分だけです。

サンプルプロジェクト名：Chapter5-4

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5:
 6: LPDIRECT3D9 pD3d;
 7: LPDIRECT3DDEVICE9 pDevice;
 8: LPD3DXMESH pMesh = NULL;
 9: D3DMATERIAL9* pMeshMaterials = NULL;
10: LPDIRECT3DTEXTURE9* pMeshTextures  = NULL;
11: DWORD dwNumMaterials = 0;
12: FLOAT fScale=1.0f;
13:
14: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
15: HRESULT InitD3d(HWND);
16: VOID Render();
17: VOID FreeDx();
18:
19: //
20: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
21: // アプリケーションのエントリー関数
22: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
23: {
24:         HWND hWnd=NULL;
```

```
25:       MSG msg;
26:       // ウィンドウの初期化
27:       static char szAppName[] = "Chapter5-4" ;
28:     WNDCLASSEX  wndclass ;
29:
30:     wndclass.cbSize          = sizeof (wndclass) ;
31:     wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
32:     wndclass.lpfnWndProc     = WndProc ;
33:     wndclass.cbClsExtra      = 0 ;
34:     wndclass.cbWndExtra      = 0 ;
35:     wndclass.hInstance       = hInst ;
36:     wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
37:     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
38:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
39:     wndclass.lpszMenuName = NULL ;
40:     wndclass.lpszClassName = szAppName ;
41:     wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
42:
43:     RegisterClassEx (&wndclass) ;
44:
45:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
46:                 0,0,800,600,NULL,NULL,hInst,NULL) ;
47:
48:    ShowWindow (hWnd,SW_SHOW) ;
49:    UpdateWindow (hWnd) ;
50:       // ダイレクト3D の初期化関数を呼ぶ
51:       if(FAILED(InitD3d(hWnd)))
52:       {
53:                   return 0;
54:       }
55:       // メッセージループ
56:    ZeroMemory( &msg, sizeof(msg) );
57:    while( msg.message!=WM_QUIT )
58:     {
59:                    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
60:         {
61:                                TranslateMessage( &msg );
62:             DispatchMessage( &msg );
63:         }
64:         else
65:                   {
66:                                Render();
67:                   }
68:       }
69:       // メッセージループから抜けたらオブジェクトを全て開放する
70:       FreeDx();
71:       // OS に戻る （アプリケーションを終了する）
72:     return (INT)msg.wParam ;
73: }
74:
75: //
76: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
77: // ウィンドウプロシージャ関数
78: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
79: {
80:       switch(iMsg)
81:       {
82:                   case WM_DESTROY:
83:                               PostQuitMessage(0);
84:                   break;
85:                   case WM_KEYDOWN:
86:                   switch((CHAR)wParam)
87:                   {
88:                               case VK_ESCAPE:
89:                                           PostQuitMessage(0);
90:                               break;
91:                               case VK_LEFT:
92:               fScale-=0.1f;
93:                               break;
94:                               case VK_RIGHT:
```

```
95:                 fScale+=0.1f;
96:                             break;
97:                     }
98:                     break;
99:         }
100:        return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
101: }
102:
103: //
104: //HRESULT InitD3d(HWND hWnd)
105: // ダイレクト 3D の初期化関数
106: HRESULT InitD3d(HWND hWnd)
107: {
108:        //「Direct3D」 オブジェクトの作成
109:        if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
110:        {
111:                    MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
112:          return E_FAIL;
113:        }
114:        //「DIRECT3D デバイス」 オブジェクトの作成
115:        D3DPRESENT_PARAMETERS d3dpp;
116:        ZeroMemory( &d3dpp, sizeof(d3dpp) );
117:        d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
118:        d3dpp.BackBufferCount=1;
119:        d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
120:        d3dpp.Windowed = TRUE;
121:      d3dpp.EnableAutoDepthStencil = TRUE;
122:      d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
123:
124:         if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
125:                                         D3DCREATE_MIXED_VERTEXPROCESSING,
126:                                         &d3dpp, &pDevice ) ) )
127:      {
128:                    MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
129:                    if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
130:                                         D3DCREATE_MIXED_VERTEXPROCESSING,
131:                                         &d3dpp, &pDevice ) ) )
132:                    {
133:                                 MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
134:                                  return E_FAIL;
135:                    }
136:      }
137:        // X ファイルからメッシュをロードする
138:        LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
139:
140:        if( FAILED( D3DXLoadMeshFromX( "Melon.x", D3DXMESH_SYSTEMMEM,
141:              pDevice, NULL, &pD3DXMtrlBuffer, NULL,
142:                                 &dwNumMaterials, &pMesh ) ) )
143:      {
144:              MessageBox(NULL, "X ファイルの読み込みに失敗しました ",NULL, MB_OK);
145:              return E_FAIL;
146:      }
147:        D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
148:      pMeshMaterials = new D3DMATERIAL9[dwNumMaterials];
149:      pMeshTextures  = new LPDIRECT3DTEXTURE9[dwNumMaterials];
150:
151:        for( DWORD i=0; i<dwNumMaterials; i++ )
152:        {
153:                    pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
154:          pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;
155:          pMeshTextures[i] = NULL;
156:          if( d3dxMaterials[i].pTextureFilename != NULL &&
157:              lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
158:          {
159:              if( FAILED( D3DXCreateTextureFromFile( pDevice,
160:                                          d3dxMaterials[i].pTextureFilename,
161:                                          &pMeshTextures[i] ) ) )
162:              {
163:                  MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
164:                                  }
```

```
165:         }
166:     }
167:       pD3DXMtrlBuffer->Release();
168:       // Z バッファー処理を有効にする
169:     pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
170:       // ライトを有効にする
171:       pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
172:       // アンビエントライト（環境光）を設定する
173:        pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
174:        // スペキュラ（鏡面反射）を有効にする
175:        pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
176:       return S_OK;
177: }
178:
179: //
180: //VOID Render()
181: //X ファイルから読み込んだメッシュをレンダリングする関数
182: VOID Render()
183: {
184:       // ワールドトランスフォーム（絶対座標変換）
185:        D3DXMATRIXA16 matWorld,matScale;
186:        D3DXMatrixIdentity(&matWorld);
187:        D3DXMatrixScaling(&matScale,fScale,fScale,fScale);
188:        D3DXMatrixMultiply(&matWorld,&matWorld,&matScale);
189:      pDevice->SetTransform( D3DTS_WORLD, &matWorld );
190:       // ビュートランスフォーム（視点座標変換）
191:
192:      D3DXVECTOR3 vecEyePt( 0.0f, 1.0f,-3.0f ); // カメラ（視点）位置
193:      D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
194:      D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
195:      D3DXMATRIXA16 matView;
196:      D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
197:      pDevice->SetTransform( D3DTS_VIEW, &matView );
198:       // プロジェクショントランスフォーム（射影変換）
199:      D3DXMATRIXA16 matProj;
200:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
201:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
202:       // ライトをあてる 白色、鏡面反射有りに設定
203:       D3DXVECTOR3 vecDirection(0,0,1);
204:      D3DLIGHT9 light;
205:      ZeroMemory( &light, sizeof(D3DLIGHT9) );
206:      light.Type       = D3DLIGHT_DIRECTIONAL;
207:      light.Diffuse.r  = 1.0f;
208:      light.Diffuse.g  = 1.0f;
209:      light.Diffuse.b  = 1.0f;
210:        light.Specular.r=1.0f;
211:        light.Specular.g=1.0f;
212:        light.Specular.b=1.0f;
213:      D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
214:      light.Range       = 200.0f;
215:      pDevice->SetLight( 0, &light );
216:      pDevice->LightEnable( 0, TRUE );
217:       // レンダリング
218:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
219:                   D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
220:
221:       if( SUCCEEDED( pDevice->BeginScene() ) )
222:       {
223:                   for( DWORD i=0; i<dwNumMaterials; i++ )
224:       {
225:           pDevice->SetMaterial( &pMeshMaterials[i] );
226:           pDevice->SetTexture( 0, pMeshTextures[i] );
227:           pMesh->DrawSubset( i );
228:       }
229:                   pDevice->EndScene();
230:       }
231:       pDevice->Present( NULL, NULL, NULL, NULL );
232: }
233:
234: //
```

```
235: //VOID FreeDx()
236: // 作成した DirectX オブジェクトの開放
237: VOID FreeDx()
238: {
239:       SAFE_RELEASE( pMesh );
240:       SAFE_RELEASE( pDevice );
241:       SAFE_RELEASE( pD3d );
242: }
```

D3DXMATRIXA16 matWorld,matScale;
最終的なワールドトランスフォーム行列 (matWorld)、スケーリング行列 matScale を用意（宣言）します。
D3DXMatrixIdentity(&matWorld);
matWorld を単位行列として初期化します。
D3DXMatrixScaling(&matScale,fScale,fScale,fScale);
D3DXMatrixScaling はスケーリング行列を作成する関数です。ここでは、matScale を、fScale 変数をパラメーターとしてスケーリング行列として初期化しています。第 1、
第 2、第 3 引数はそれぞれ X、Y、Z 軸にたいする拡大・縮小率です。ここでは、全てに同じパラメーターを指定しているのでジオメトリ形状の比率は変化せずに拡大縮小します。
D3DXMatrixMultiply(&matWorld,&matWorld,&matScale);
pDevice->SetTransform(D3DTS_WORLD, &matWorld);
スケーリング行列を最終的なワールドトランスフォーム行列に掛け合わせて、レンダリングパイプラインにそのワールド行列を登録しておきます。
なお、このコードは次のように短縮できます。

```
D3DXMATRIXA16 matWorld;
D3DXMatrixIdentity(&matWorld);
D3DXMatrixScaling(&matWorld,fScale,fScale,fScale);
pDevice->SetTransform( D3DTS_WORLD, &matWorld );
```

視点（カメラ）を移動・回転させる（ビュートランスフォーム）

サンプルプロジェクト名：Chapter5-5

```
 1:
 2: #include <windows.h>
 3: #include <d3dx9.h>
 4:
 5: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 6: #define THING_AMOUNT 3+1
 7:
 8:
 9: struct THING
10: {
11:       LPD3DXMESH pMesh;
12:       D3DMATERIAL9* pMeshMaterials;
13:       LPDIRECT3DTEXTURE9* pMeshTextures ;
14:       DWORD dwNumMaterials;
15:       D3DXVECTOR3 vecPosition;
16:       THING()
17:       {
18:                 ZeroMemory(this,sizeof(THING));
19:       }
20: };
21:
22: LPDIRECT3D9 pD3d;
23: LPDIRECT3DDEVICE9 pDevice;
24: FLOAT fCameraX=0,fCameraY=1.0f,fCameraZ=-3.0f,
25:                         fCameraHeading=0,fCameraPitch=0;
26:
27: THING Thing[THING_AMOUNT];
28:
29: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
30: HRESULT InitD3d(HWND);
```

```
31: HRESULT InitThing(THING *,LPSTR,D3DXVECTOR3*);
32: VOID Render();
33: VOID RenderThing(THING*);
34: VOID FreeDx();
35:
36: //
37: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
38: // アプリケーションのエントリー関数
39: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
40: {
41:       HWND hWnd=NULL;
42:       MSG msg;
43:       // ウィンドウの初期化
44:       static char szAppName[] = "Chapter5-5" ;
45:     WNDCLASSEX  wndclass ;
46:
47:     wndclass.cbSize          = sizeof (wndclass) ;
48:     wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
49:     wndclass.lpfnWndProc     = WndProc ;
50:     wndclass.cbClsExtra      = 0 ;
51:     wndclass.cbWndExtra      = 0 ;
52:     wndclass.hInstance       = hInst ;
53:     wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
54:     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
55:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
56:     wndclass.lpszMenuName = NULL ;
57:     wndclass.lpszClassName = szAppName ;
58:     wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
59:
60:     RegisterClassEx (&wndclass) ;
61:
62:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
63:                 0,0,800,600,NULL,NULL,hInst,NULL) ;
64:
65:    ShowWindow (hWnd,SW_SHOW) ;
66:    UpdateWindow (hWnd) ;
67:       // ダイレクト3D の初期化関数を呼ぶ
68:       if(FAILED(InitD3d(hWnd)))
69:       {
70:                 return 0;
71:       }
72:       // メッセージループ
73:    ZeroMemory( &msg, sizeof(msg) );
74:    while( msg.message!=WM_QUIT )
75:    {
76:                  if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
77:        {
78:                           TranslateMessage( &msg );
79:            DispatchMessage( &msg );
80:        }
81:        else
82:                 {
83:                           Render();
84:                 }
85:       }
86:       // メッセージループから抜けたらオブジェクトを全て開放する
87:       FreeDx();
88:       // OS に戻る （アプリケーションを終了する）
89:     return (INT)msg.wParam ;
90: }
91:
92: //
93: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
94: // ウィンドウプロシージャ関数
95: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
96: {
97:       switch(iMsg)
98:       {
99:                 case WM_DESTROY:
100:                       PostQuitMessage(0);
```

```
101:                    break;
102:                    case WM_KEYDOWN:
103:                    switch((CHAR)wParam)
104:                    {
105:                            case VK_ESCAPE:
106:                                    PostQuitMessage(0);
107:                            break;
108:                            // カメラの移動
109:                            case 'A':
110:                                    fCameraX-=0.1f;
111:                            break;
112:                            case 'D':
113:                                    fCameraX+=0.1f;
114:                            break;
115:                            case 'Q':
116:                                    fCameraY-=0.1f;
117:                            break;
118:                            case 'E':
119:                                    fCameraY+=0.1f;
120:                            break;
121:                            case 'W':
122:                                    fCameraZ-=0.1f;
123:                            break;
124:                            case 'C':
125:                                    fCameraZ+=0.1f;
126:                            break;
127:                            // カメラの回転
128:                            case VK_LEFT:
129:                                    fCameraHeading-=0.1f;
130:                                    break;
131:                            case VK_RIGHT:
132:                                    fCameraHeading+=0.1f;
133:                            break;
134:                            case VK_UP:
135:                                    fCameraPitch-=0.1f;
136:                            break;
137:                            case VK_DOWN:
138:                                    fCameraPitch+=0.1f;
139:                            break;
140:                    }
141:                    break;
142:        }
143:        return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
144: }
145:
146: //
147: //HRESULT InitD3d(HWND hWnd)
148: // ダイレクト 3D の初期化関数
149: HRESULT InitD3d(HWND hWnd)
150: {
151:        //「Direct3D」オブジェクトの作成
152:        if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
153:        {
154:                    MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
155:          return E_FAIL;
156:        }
157:        //「DIRECT3D デバイス」オブジェクトの作成
158:        D3DPRESENT_PARAMETERS d3dpp;
159:        ZeroMemory( &d3dpp, sizeof(d3dpp) );
160:        d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
161:        d3dpp.BackBufferCount=1;
162:        d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
163:        d3dpp.Windowed = TRUE;
164:     d3dpp.EnableAutoDepthStencil = TRUE;
165:     d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
166:
167:        if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
168:                                        D3DCREATE_MIXED_VERTEXPROCESSING,
169:                                        &d3dpp, &pDevice ) ) )
170:     {
```

```
171:                    MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
172:                    if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
173:                                    D3DCREATE_MIXED_VERTEXPROCESSING,
174:                                    &d3dpp, &pDevice ) ) )
175:                    {
176:                                MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
177:                                 return E_FAIL;
178:                    }
179:    }
180:
181:      // X ファイル毎にメッシュを作成する
182:      InitThing(&Thing[0],"Ground.x",&D3DXVECTOR3(0,-0.5,0));
183:      InitThing(&Thing[1],"Can.x",&D3DXVECTOR3(0,0,0));
184:      InitThing(&Thing[2],"Bottle.x",&D3DXVECTOR3(1,0,1));
185:
186:      // Z バッファー処理を有効にする
187:    pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
188:      // ライトを有効にする
189:      pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
190:      // アンビエントライト（環境光）を設定する
191:       pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
192:       // スペキュラ（鏡面反射）を有効にする
193:       pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
194:      return S_OK;
195: }
196:
197: //
198: //
199: //
200: HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
201: {
202:      // メッシュの初期位置
203:      memcpy(&pThing->vecPosition,pvecPosition,sizeof(D3DXVECTOR3));
204:      // X ファイルからメッシュをロードする
205:      LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
206:
207:      if( FAILED( D3DXLoadMeshFromX( szXFileName, D3DXMESH_SYSTEMMEM,
208:          pDevice, NULL, &pD3DXMtrlBuffer, NULL,
209:                          &pThing->dwNumMaterials, &pThing->pMesh ) ) )
210:    {
211:          MessageBox(NULL, "X ファイルの読み込みに失敗しました ",szXFileName, MB_OK);
212:          return E_FAIL;
213:    }
214:      D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
215:      pThing->pMeshMaterials = new D3DMATERIAL9[pThing->dwNumMaterials];
216:    pThing->pMeshTextures  = new LPDIRECT3DTEXTURE9[pThing->dwNumMaterials];
217:
218:      for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
219:      {
220:                    pThing->pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
221:       pThing->pMeshMaterials[i].Ambient = pThing->pMeshMaterials[i].Diffuse;
222:       pThing->pMeshTextures[i] = NULL;
223:       if( d3dxMaterials[i].pTextureFilename != NULL &&
224:          lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
225:       {
226:          if( FAILED( D3DXCreateTextureFromFile( pDevice,
227:                                        d3dxMaterials[i].pTextureFilename,
228:                                        &pThing->pMeshTextures[i] ) ) )
229:          {
230:              MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
231:                          }
232:       }
233:    }
234:      pD3DXMtrlBuffer->Release();
235:
236:      return S_OK;
237: }
238:
239: //
240: //VOID Render()
```

```
241: //X ファイルから読み込んだメッシュをレンダリングする関数
242: VOID Render()
243: {
244:     pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
245:                     D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
246:
247:     if( SUCCEEDED( pDevice->BeginScene() ) )
248:     {
249:                     for(DWORD i=0;i<THING_AMOUNT;i++)
250:                     {
251:                                     RenderThing(&Thing[i]);
252:                     }
253:                     pDevice->EndScene();
254:     }
255:     pDevice->Present( NULL, NULL, NULL, NULL );
256: }
257: //
258: //
259: //
260: VOID RenderThing(THING* pThing)
261: {
262:     // ワールドトランスフォーム（絶対座標変換）
263:     D3DXMATRIXA16 matWorld,matPosition;
264:     D3DXMatrixIdentity(&matWorld);
265:     D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,
266:                     pThing->vecPosition.z);
267:     D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
268:   pDevice->SetTransform( D3DTS_WORLD, &matWorld );
269:     // ビュートランスフォーム（視点座標変換）
270:     D3DXMATRIXA16 matView,matCameraPosition,matHeading,matPitch;
271:   D3DXVECTOR3 vecEyePt( fCameraX,fCameraY,fCameraZ ); // カメラ（視点）位置
272:   D3DXVECTOR3 vecLookatPt( fCameraX,fCameraY-1.0f,fCameraZ+3.0f );// 注視位置
273:   D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
274:     D3DXMatrixIdentity(&matView);
275:     D3DXMatrixRotationY(&matHeading,fCameraHeading);
276:     D3DXMatrixRotationX(&matPitch,fCameraPitch);
277:     D3DXMatrixLookAtLH( &matCameraPosition, &vecEyePt, &vecLookatPt, &vecUpVec );
278:     D3DXMatrixMultiply(&matView,&matView,&matHeading);
279:     D3DXMatrixMultiply(&matView,&matView,&matPitch);
280:     D3DXMatrixMultiply(&matView,&matView,&matCameraPosition);
281:   pDevice->SetTransform( D3DTS_VIEW, &matView );
282:     // プロジェクショントランスフォーム（射影変換）
283:   D3DXMATRIXA16 matProj;
284:   D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
285:   pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
286:     // ライトをあてる 白色で鏡面反射ありに設定
287:     D3DXVECTOR3 vecDirection(1,1,1);
288:   D3DLIGHT9 light;
289:   ZeroMemory( &light, sizeof(D3DLIGHT9) );
290:   light.Type       = D3DLIGHT_DIRECTIONAL;
291:   light.Diffuse.r  = 1.0f;
292:   light.Diffuse.g  = 1.0f;
293:   light.Diffuse.b  = 1.0f;
294:     light.Specular.r=1.0f;
295:     light.Specular.g=1.0f;
296:     light.Specular.b=1.0f;
297:   D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
298:   light.Range      = 200.0f;
299:   pDevice->SetLight( 0, &light );
300:   pDevice->LightEnable( 0, TRUE );
301:     // レンダリング
302:     for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
303:   {
304:                     pDevice->SetMaterial( &pThing->pMeshMaterials[i] );
305:       pDevice->SetTexture( 0,pThing->pMeshTextures[i] );
306:       pThing->pMesh->DrawSubset( i );
307:   }
308: }
309:
310: //
```

```
311: //VOID FreeDx()
312: // 作成した DirectX オブジェクトの開放
313: VOID FreeDx()
314: {
315:       for(DWORD i=0;i<THING_AMOUNT;i++)
316:       {
317:                  SAFE_RELEASE( Thing[i].pMesh );
318:       }
319:       SAFE_RELEASE( pDevice );
320:       SAFE_RELEASE( pD3d );
321: }
```

ワールドトランスフォームが、個々のジオメトリに対する変換であったのに対し、ビュートランスフォームは、全てのジオメトリに対する変換です。ワールドトランスフォームでジオメトリを絶対座標に配置した後に、もう一段階このビュートランスフォームで絶対座標を更に変換します。変換した座標はカメラ座標と呼ばれます。更に前もって言うと事項で解説するプロジェクショントランスフォームで微調整し、最終的な画面出力となります。
流れとしては次のようになります。
ワールドトランスフォーム
機能　個々のジオメトリのローカル座標を絶対座標に変換して配置
対象　個々のジオメトリのローカル座標。

ビュートランスフォーム
機能　絶対座標を更にカメラ座標に変換。
対象　全てのジオメトリの絶対座標

プロジェクショントランスフォーム
機能　カメラ座標を更に最終的なスクリーン（画面、モニター）座標に変換。
対象　全てのジオメトリのカメラ座標

ビュートランスフォームというプロセスがあるお陰で、例えば一人称シューティングゲームで主人公の視点変化を簡単に実現することができます。
どうゆうことなのかというと、例えば、主人公が右を向いたとします。このとき主人公から見れば（視界にある）全ジオメトリは向いた方向と逆方向（左）に移動（回転）するはずです（と言うよりも、「そうなるようにしなくてはならない」と言った方が良いかもしれませんが）。もし、ビュートランスフォームというプロセスが無いならば、主人公の視点変化を実現するためには、全てのジオメトリの絶対座標を一つ一つ計算して移動しなくてはならないことになります。これは非常に非効率ですし、なにより面倒です。ジオメトリ全体の配置をシーン等に見立てて、そのシーンを統一的に移動変換できれば便利です。ビュートランスフォームはまさにこのような視点変化を実現するための変換なのです。
更に、図で説明することにしましょう。今、主人公は４つのリンゴジオメトリを真っ直ぐ見ていることとします。この時の絶対座標を仮にテーブルに例えると図のようになります。リンゴジオメトリはテーブルという絶対座標上に配置されています。最初、主人公は真っすぐそれらのリンゴを見ています。次に主人公が右を向いたとしましょう。主人公から見たリンゴ達は相対的に左に移動（回転）したように見えなくてはなりません。その時、もし、ビュートランスフォームのような変換が無ければ、リンゴジオメトリ一個一個を左に回転しなくてはなりません。それよりも、テーブルを動かせば、その上にあるリンゴは全て一斉に動きます。このテーブルという絶対座標を動かすことがビュートランスフォームなのです。

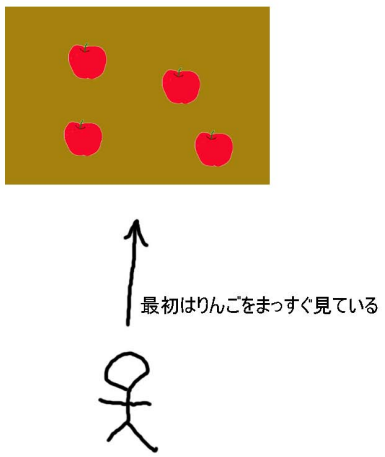


図 5-8

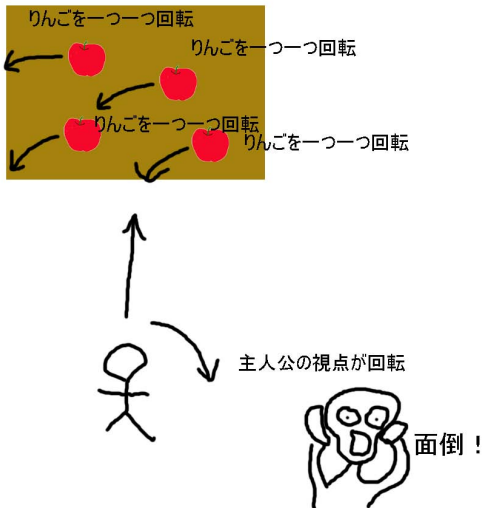


図 5-9

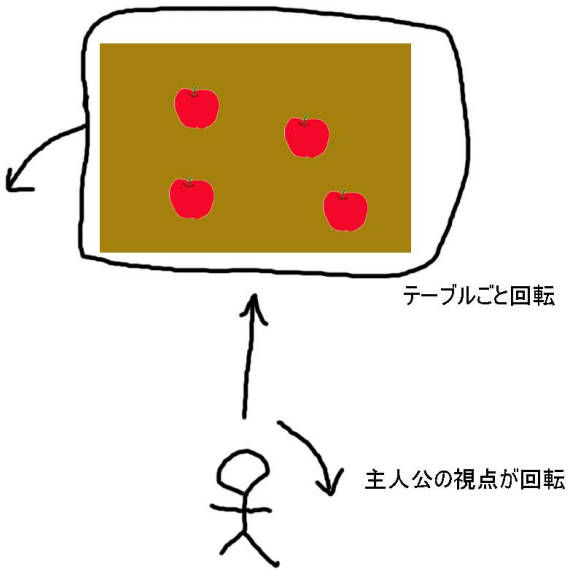


図 5-10

では、コードの解説をしましょう。ワールドトランスフォーム
読み下すコードは、関数 VOID　RenderThing（）内のビュートランスフォーム部分だけです、その他の部分は今までとたいして変わりはありません。

```
D3DXMATRIXA16 matView,matCameraPosition,matHeading,matPitch;
D3DXVECTOR3 vecEyePt( fCameraX,fCameraY,fCameraZ ); // カメラ（視点）位置
D3DXVECTOR3 vecLookatPt( fCameraX,fCameraY-1.0f,fCameraZ+3.0f );// 注視位置
D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
D3DXMatrixIdentity(&matView);
D3DXMatrixRotationY(&matHeading,fCameraHeading);
D3DXMatrixRotationX(&matPitch,fCameraPitch);
D3DXMatrixLookAtLH( &matCameraPosition, &vecEyePt, &vecLookatPt, &vecUpVec );
```

```
D3DXMatrixMultiply(&matView,&matView,&matHeading);
D3DXMatrixMultiply(&matView,&matView,&matPitch);
D3DXMatrixMultiply(&matView,&matView,&matCameraPosition);
pDevice->SetTransform( D3DTS_VIEW, &matView );
D3DXMATRIXA16 matView,matCameraPosition,matHeading,matPitch;
```

最終的なビュートランスフォーム行列を格納するために matView を、カメラの平行移動行列用に matCameraPosition を、カメラのヘディング回転用とピッチ回転用にそれぞれ、matHeading と matPitch を用意します。バンク回転は今回考慮しません。

```
D3DXVECTOR3 vecEyePt( fCameraX,fCameraY,fCameraZ ); // カメラ（視点）位置
D3DXVECTOR3 vecLookatPt( fCameraX,fCameraY-1.0f,fCameraZ+3.0f );// 注視位置
D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
```

この 3 行がビュートランスフォームで最も重要なキーとなるコードですので、意味を理解していただきたいと思います。この 3 行の説明に入る前に、カメラの位置や回転とビュートランスフォームの関係を説明する必要があるでしょう。そして、そもそも " カメラ " とは何であるのかということを解説しなくてはなりません。

コンピューター上での 3 次元ジオメトリのレンダリングにおいて、3D ジオメトリのディメンションは 3 次元なのにモニターは平面なので 3 次元ではなく 2 次元のデバイスです。したがって、3D ジオメトリは、レンダリング出力されて最終的に画面（モニター、ディスプレイ）上では 2 次元に変換されます。

さて、実際のカメラ、特に動画を撮るビデオカメラを想像してみてください。ビデオカメラで撮る対象は景色だったり人物だったりするわけですが、とにかく現実の物なのですから当然 3 次元です。ところがモニターは 2 次元平面です。これは、コンピューター 3D レンダリングの記述と全く同じです。さらに、カメラの方向や位置によって写る像の変化も 3D レンダリングと現実のカメラは同一です。ただ、実際のビデオカメラの場合、投影の必要はありません、なぜなら、現実世界の物体は作り出されたものではなく文字通り現実に存在しているものであり、（当たり前ですが、）ビデオカメラはそれらの座標やマテリアルなどを自分自身で持っているものではなく、また、その必要もないのは言うまでもないでしょう。カメラレンズに入ってくる光は計算されたものではない、というより計算する必要がないからであり、現実のカメラは物体からの反射光を単純に画素に対応させればいいわけです。これに対してコンピューター 3D の場合は、物体自体が作り出されたものであり、それらの頂点座座標、マテリアル等はすべて計算されたものです、現実世界のように画素に対応されればいいだけの光を自然に得られるわけでもありませんのでラスタライズするプロセスが必要になるのです。

このように、投影の有無はありますが、それは問題ではなく、コンピューター 3D レンダリングと現実のカメラの " 機能 " 面に注目していただきたいのです。それらは、ほぼ同一と言えます。事項で解説するプロジェクショントランスフォームを読んでいただければ、もっと現実のカメラと機能的に酷似していることがわかると思います。コンピューター 3D において " カメラ " という概念を引用しているのは、このような理由からだと思われます。あるいは、最初から、カメラという概念を基にコンピューター 3D が設計されてきたのかもしれませんが、筆者はそこまでは正直わかりません。

3D レンダリングにおいて " カメラ " と呼ぶ理由がわかったところで、この 3 行を理解していきましょう。

```
D3DXVECTOR3 vecEyePt( fCameraX,fCameraY,fCameraZ ); // カメラ（視点）位置
```

コメントの通り、カメラの位置ベクトルとして vecEyePt を初期化します。

```
D3DXVECTOR3 vecLookatPt( fCameraX,fCameraY-1.0f,fCameraZ+3.0f );// 注視位置
```

注視位置ベクトルとして vecLookatPt を初期化します。

" 注視 " 位置とは何なのか、どのような意味があり、どのように機能するかと言うと、次のとおりです。

カメラが写す像を計算するとき、計算に必要な要素はカメラの位置と方向です。カメラの位置は既に初期化して vecEyePt に入っています。あとは方向が必要なわけですが、その方向を出す目的で定義するのがこの " 注視 " 位置です。カメラの方向、言い換えればカメラがどっちを向いているのかを、カメラ位置ベクトルとカメラ注視位置ベクトルの引き算から求めます。

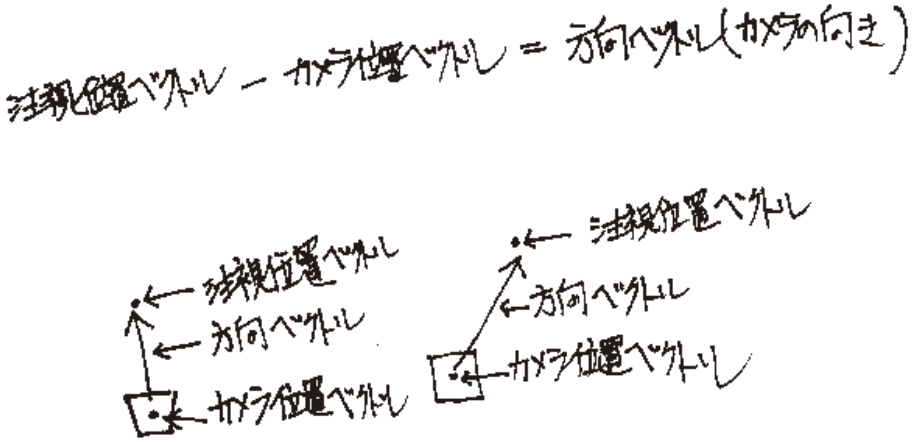


図 5-11

注視位置ではなく、最初からカメラ方向として定義すれば良いのではないかと思うかもしれません。素直な発想であり、確かにそうすれば、ベクトルの引き算を省略できて一見スマートなような感じもしまが、注視点を定義することでなにかと便利なことがあるのです。例えば、よく TV などで人物は動かずにカメラがその周りを旋回しているというシーンを見ると思います。あるジオメトリを中心にカメラを、そのジオメトリの周りを回転・移動させたいと思ったとします。カメラがどんなに動いてもピッタリそのジオメトリを " 注視 " しているように見せたいわけです。文字通り " 注視 " です。

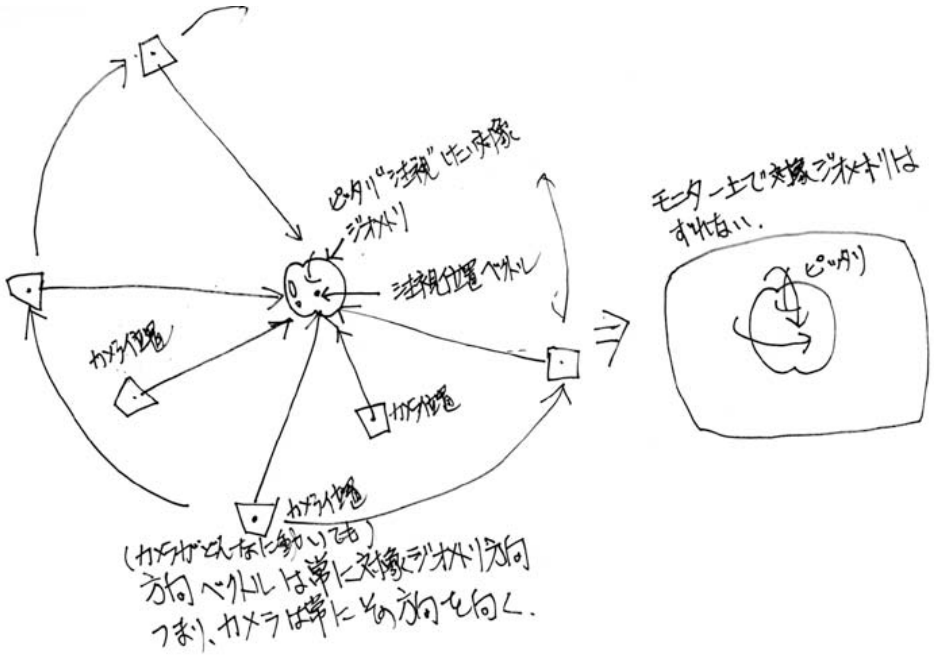


図 5-12

図のように、注視位置を対象ジオメトリの位置に設定するだけでカメラはピッタリそのジオメトリを捕捉することができます。これを、注視位置ではなく最初から方向ベクトルで処理しようとすると、結局、ベクトルの引き算をしなくてはならないばかりか、中間式用のベクトルを用意しなくてはなりません。Direct3D の設計者は最初から、このような演出をし易いように、カメラ位置と注視位置を敢えて分けた仕様にしたのではないでしょうか。（Direct3D は OpenGL に多大な影響を受けているので OpenGL の設計者と言ったほうが良いかもしません）

D3DXVECTOR3 vecUpVec(0.0f, 1.0f, 0.0f);// 上方位置

これは、単純にこのようにするものだと覚える程度で良いかと思います。このパラメーターを変更することはまずありません。意味としては、レンダリングパイプラインに「上方はこの方向ベクトルで考えなさい」と指示しているに過ぎません。パラメーターを見ても分かるとお方向ベクトルとしては Y 軸と重なる文字通り上方へのベクトルです。

D3DXMatrixIdentity(&matView);

最終的なビュートランスフォーム行列用の matView 行列を単位行列で初期化します。このあと位置と回転の行列をこれに掛け合わせていくので、掛けられる元の行列は単位行列で初期化しておきます。（メンバーが全てゼロの行列に何を掛け合わせていっても値は全てゼロになってしまうので）

D3DXMatrixRotationY(&matHeading,fCameraHeading);

fCameraHeading はカメラのヘディング回転状態を保持するためにグローバルに宣言している変数であり、値はユーザー入力により変動します。それを用いて matHeading をヘディング回転行列として初期化しています。

D3DXMatrixRotationX(&matPitch,fCameraPitch);

ヘディング回転と同様にピッチ回転処理をしています。

D3DXMatrixLookAtLH(&matCameraPosition, &vecEyePt, &vecLookatPt, &vecUpVec);

D3DXMatrixLookAtLH はビュー行列を作成する関数です。最終的なビュー行列は matView に格納すると述べました。ここで作成したビュー行列は位置情報のみを反映したビュー行列であり、いわば中間式です。

D3DXMatrixMultiply(&matView,&matView,&matHeading);

D3DXMatrixMultiply(&matView,&matView,&matPitch);

matView 行列にヘディング及びピッチ回転行列を掛けます。

D3DXMatrixMultiply(&matView,&matView,&matCameraPosition);

matView 行列に位置情報を格納しているビュー行列を掛けます。

この計算は回転行列を掛けた後にする必要があります。回転行列より先に位置行列を掛けると結果が異なってしまいます。

この時点で、matView 行列は完全なビュー行列となりましたので例によって、

pDevice->SetTransform(D3DTS_VIEW, &matView);

により、レンダリングパイプラインにビュー行列を登録します。

ズームイン・アウトする（プロジェクショントランスフォーム）

サンプルプロジェクト名：Chapter5-6

```
1:
2: #include <windows.h>
3: #include <d3dx9.h>
4:
5: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
```

```
 6: #define THING_AMOUNT 6+1
 7:
 8:
 9: struct THING
10: {
11:         LPD3DXMESH pMesh;
12:         D3DMATERIAL9* pMeshMaterials;
13:         LPDIRECT3DTEXTURE9* pMeshTextures ;
14:         DWORD dwNumMaterials;
15:         D3DXVECTOR3 vecPosition;
16:         THING()
17:         {
18:                         ZeroMemory(this,sizeof(THING));
19:         }
20: };
21:
22: LPDIRECT3D9 pD3d;
23: LPDIRECT3DDEVICE9 pDevice;
24:
25: THING Thing[THING_AMOUNT];
26: FLOAT fCameraX=0,fCameraY=1.0f,fCameraZ=-3.0f;
27: FLOAT fPerspective=4.0f;
28:
29: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
30: HRESULT InitD3d(HWND);
31: HRESULT InitThing(THING *,LPSTR,D3DXVECTOR3*);
32: VOID Render();
33: VOID RenderThing(THING*);
34: VOID FreeDx();
35:
36: //
37: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
38: // アプリケーションのエントリー関数
39: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
40: {
41:         HWND hWnd=NULL;
42:         MSG msg;
43:         // ウィンドウの初期化
44:         static char szAppName[] = "Chapter5-6" ;
45:       WNDCLASSEX  wndclass ;
46:
47:       wndclass.cbSize           = sizeof (wndclass) ;
48:       wndclass.style            = CS_HREDRAW | CS_VREDRAW ;
49:       wndclass.lpfnWndProc      = WndProc ;
50:       wndclass.cbClsExtra       = 0 ;
51:       wndclass.cbWndExtra       = 0 ;
52:       wndclass.hInstance        = hInst ;
53:       wndclass.hIcon            = LoadIcon (NULL, IDI_APPLICATION) ;
54:       wndclass.hCursor          = LoadCursor (NULL, IDC_ARROW) ;
55:       wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
56:       wndclass.lpszMenuName = NULL ;
57:       wndclass.lpszClassName = szAppName ;
58:       wndclass.hIconSm          = LoadIcon (NULL, IDI_APPLICATION) ;
59:
60:       RegisterClassEx (&wndclass) ;
61:
62:       hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
63:                         0,0,800,600,NULL,NULL,hInst,NULL) ;
64:
65:      ShowWindow (hWnd,SW_SHOW) ;
66:      UpdateWindow (hWnd) ;
67:         // ダイレクト3D の初期化関数を呼ぶ
68:         if(FAILED(InitD3d(hWnd)))
69:         {
70:                         return 0;
71:         }
72:         // メッセージループ
73:      ZeroMemory( &msg, sizeof(msg) );
74:      while( msg.message!=WM_QUIT )
75:       {
```

```
76:                     if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
77:         {
78:                             TranslateMessage( &msg );
79:             DispatchMessage( &msg );
80:         }
81:         else
82:                     {
83:                             Render();
84:                     }
85:         }
86:         // メッセージループから抜けたらオブジェクトを全て開放する
87:         FreeDx();
88:         // OS に戻る （アプリケーションを終了する）
89:     return (INT)msg.wParam ;
90: }
91:
92: //
93: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
94: // ウィンドウプロシージャ関数
95: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
96: {
97:       switch(iMsg)
98:       {
99:                     case WM_DESTROY:
100:                            PostQuitMessage(0);
101:                    break;
102:                    case WM_KEYDOWN:
103:                    switch((CHAR)wParam)
104:                    {
105:                            case VK_ESCAPE:
106:                                    PostQuitMessage(0);
107:                            break;
108:                            // カメラの移動
109:                            case 'A':
110:                                    fCameraX-=0.1f;
111:                            break;
112:                            case 'D':
113:                                    fCameraX+=0.1f;
114:                            break;
115:                            case 'Q':
116:                                    fCameraY-=0.1f;
117:                            break;
118:                            case 'E':
119:                                    fCameraY+=0.1f;
120:                            break;
121:                            case 'W':
122:                                    fCameraZ-=0.1f;
123:                            break;
124:                            case 'C':
125:                                    fCameraZ+=0.1f;
126:                            break;
127:                            case VK_UP:
128:                            // 視野角の変更 （ズームイン）
129:                            fPerspective+=0.1;
130:                            break;
131:                            case VK_DOWN:
132:                            // 視野角の変更 （ズームアウト）
133:                            fPerspective-=0.1;
134:                    }
135:                    break;
136:
137:
138:                    break;
139:      }
140:      return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
141: }
142:
143: //
144: //HRESULT InitD3d(HWND hWnd)
145: // ダイレクト 3D の初期化関数
```

```
146: HRESULT InitD3d(HWND hWnd)
147: {
148:     //「Direct3D」オブジェクトの作成
149:     if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
150:     {
151:                 MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
152:       return E_FAIL;
153:     }
154:     //「DIRECT3D デバイス」オブジェクトの作成
155:     D3DPRESENT_PARAMETERS d3dpp;
156:     ZeroMemory( &d3dpp, sizeof(d3dpp) );
157:     d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
158:     d3dpp.BackBufferCount=1;
159:     d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
160:     d3dpp.Windowed = TRUE;
161:   d3dpp.EnableAutoDepthStencil = TRUE;
162:   d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
163:
164:     if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
165:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
166:                                 &d3dpp, &pDevice ) ) )
167:   {
168:                 MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
169:                 if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
170:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
171:                                 &d3dpp, &pDevice ) ) )
172:                 {
173:                             MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
174:                              return E_FAIL;
175:                 }
176:   }
177:
178:     // X ファイル毎にメッシュを作成する
179:     InitThing(&Thing[0],"Ground.x",&D3DXVECTOR3(0,-1,0));
180:     InitThing(&Thing[1],"Can.x",&D3DXVECTOR3(0,0,0));
181:     InitThing(&Thing[2],"Apartment.x",&D3DXVECTOR3(0.5,0,50));
182:     InitThing(&Thing[3],"Corn.x",&D3DXVECTOR3(-0.5,0,2));
183:     InitThing(&Thing[4],"Tomato.x",&D3DXVECTOR3(1,0,3));
184:     InitThing(&Thing[5],"Melon.x",&D3DXVECTOR3(0,0,4));
185:     // Z バッファー処理を有効にする
186:   pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
187:     // ライトを有効にする
188:     pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
189:     // アンビエントライト（環境光）を設定する
190:      pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
191:      // スペキュラ（鏡面反射）を有効にする
192:      pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
193:     return S_OK;
194: }
195:
196: //
197: //
198: //
199: HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
200: {
201:     // メッシュの初期位置
202:     memcpy(&pThing->vecPosition,pvecPosition,sizeof(D3DXVECTOR3));
203:     // X ファイルからメッシュをロードする
204:     LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
205:
206:     if( FAILED( D3DXLoadMeshFromX( szXFileName, D3DXMESH_SYSTEMMEM,
207:         pDevice, NULL, &pD3DXMtrlBuffer, NULL,
208:                             &pThing->dwNumMaterials, &pThing->pMesh ) ) )
209:   {
210:         MessageBox(NULL, "X ファイルの読み込みに失敗しました ",szXFileName, MB_OK);
211:         return E_FAIL;
212:   }
213:     D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
214:     pThing->pMeshMaterials = new D3DMATERIAL9[pThing->dwNumMaterials];
215:   pThing->pMeshTextures  = new LPDIRECT3DTEXTURE9[pThing->dwNumMaterials];
```

```
216:
217:       for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
218:       {
219:                   pThing->pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
220:         pThing->pMeshMaterials[i].Ambient = pThing->pMeshMaterials[i].Diffuse;
221:         pThing->pMeshTextures[i] = NULL;
222:         if( d3dxMaterials[i].pTextureFilename != NULL &&
223:             lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
224:         {
225:             if( FAILED( D3DXCreateTextureFromFile( pDevice,
226:                                         d3dxMaterials[i].pTextureFilename,
227:                                         &pThing->pMeshTextures[i] ) ) )
228:             {
229:                 MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
230:                             }
231:         }
232:     }
233:       pD3DXMtrlBuffer->Release();
234:
235:       return S_OK;
236: }
237:
238: //
239: //VOID Render()
240: //X ファイルから読み込んだメッシュをレンダリングする関数
241: VOID Render()
242: {
243:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
244:                   D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
245:
246:        if( SUCCEEDED( pDevice->BeginScene() ) )
247:        {
248:                   for(DWORD i=0;i<THING_AMOUNT;i++)
249:                   {
250:                               RenderThing(&Thing[i]);
251:                   }
252:                   pDevice->EndScene();
253:        }
254:        pDevice->Present( NULL, NULL, NULL, NULL );
255: }
256: //
257: //
258: //
259: VOID RenderThing(THING* pThing)
260: {
261:       // ワールドトランスフォーム （絶対座標変換）
262:        D3DXMATRIXA16 matWorld,matPosition;
263:        D3DXMatrixIdentity(&matWorld);
264:        D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,
265:                   pThing->vecPosition.z);
266:        D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
267:      pDevice->SetTransform( D3DTS_WORLD, &matWorld );
268:        // ビュートランスフォーム （視点座標変換）
269:
270:      D3DXVECTOR3 vecEyePt( fCameraX,fCameraY-0.5f,fCameraZ ); // カメラ （視点） 位置
271:      D3DXVECTOR3 vecLookatPt( fCameraX,fCameraY-0.5,fCameraZ+3.0f );// 注視位置
272:      D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
273:      D3DXMATRIXA16 matView;
274:      D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
275:      pDevice->SetTransform( D3DTS_VIEW, &matView );
276:        // プロジェクショントランスフォーム （射影変換）
277:      D3DXMATRIXA16 matProj;
278:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/fPerspective, 1.0f, 1.0f, 100.0f );
279:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
280:        // ライトをあてる 白色で鏡面反射ありに設定
281:        D3DXVECTOR3 vecDirection(1,1,1);
282:      D3DLIGHT9 light;
283:      ZeroMemory( &light, sizeof(D3DLIGHT9) );
284:      light.Type       = D3DLIGHT_DIRECTIONAL;
285:      light.Diffuse.r  = 1.0f;
```

```
286:     light.Diffuse.g  = 1.0f;
287:     light.Diffuse.b  = 1.0f;
288:       light.Specular.r=1.0f;
289:       light.Specular.g=1.0f;
290:       light.Specular.b=1.0f;
291:     D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
292:     light.Range        = 200.0f;
293:     pDevice->SetLight( 0, &light );
294:     pDevice->LightEnable( 0, TRUE );
295:       // レンダリング
296:       for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
297:     {
298:                       pDevice->SetMaterial( &pThing->pMeshMaterials[i] );
299:           pDevice->SetTexture( 0,pThing->pMeshTextures[i] );
300:           pThing->pMesh->DrawSubset( i );
301:     }
302: }
303:
304: //
305: //VOID FreeDx()
306: // 作成した DirectX オブジェクトの開放
307: VOID FreeDx()
308: {
309:       for(DWORD i=0;i<THING_AMOUNT;i++)
310:       {
311:                       SAFE_RELEASE( Thing[i].pMesh );
312:       }
313:       SAFE_RELEASE( pDevice );
314:       SAFE_RELEASE( pD3d );
315: }
```

カメラ座標まで変換された段階で最後の変換を掛けます。3 次元の座標を 2 次元デバイスであるモニターでは表示できません。そこで、カメラ座標をスクリーン座標という 2 次元座標に変換します。少ない次元に変換することを投影または射影と呼びます。プロジェクションとは日本語で「投影」という意味です。難しいことはなく、我々人間の可視効果と同じことです。我々も絵を描くときに 3 次元の物体を 2 次元の紙に書くでしょう、それは投影以外のなにものでもありません。3 次元の情報を 2 次元に変換しているのです。
では、実際のコードを見ていきましょう。
と言っても、今回は 3 行しかありません。
プロジェクショントランスフォームは射影変換ですので、変換のパラメーターにより射影の形態を変化させることができます。今回のテーマであるズームイン、ズームアウトも射影変換のパラメーターを変化させることにより実現できます。

```
D3DXMATRIXA16 matProj;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/fPerspective, 1.0f, 1.0f, 100.0f );
pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

D3DXMATRIXA16 matProj;
最終的なプロジェクションとランフォーム行列格納用に matProj という行列を用意します。

```
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/fPerspective, 1.0f, 1.0f, 100.0f );
```

D3DXMatrixPerspectiveFovLH はプロジェクショントランスフォーム行列を作成する関数です。
第 1 引数は、演算結果である射影行列、第 2 引数は視野角、第 3 引数はスクリーンのアスペクト比、第 4 引数は可視できる最小距離、第 5 引数は可視できる最大距離です。
今回のテーマはカメラのズームなので、第 2 引数が可変値となっています。つまり定数ではなく変数になています。fPerspective はユーザー入力によりウィンドウプロシージャー内で値が変化しています。したがって、ユーザー入力によって視野角が変化します。視野角が変化するとズームイン、ズームアウトの効果が生まれます。

```
pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

レンダリングパイプラインに最終的なプロジェクションとランフォーム行列を登録します。

これで、3D レンダリングの 3 つの変換についての解説を一通り終えたわけですが、ここでこれら 3 つの変換を再度まとめてみましょう。先にビュー変換でリンゴとテーブルに例えました、ここではちょっと気分を変えてメロンを用いて例えてみます。なお、今度はカメラも用います。

ワールドトランスフォーム

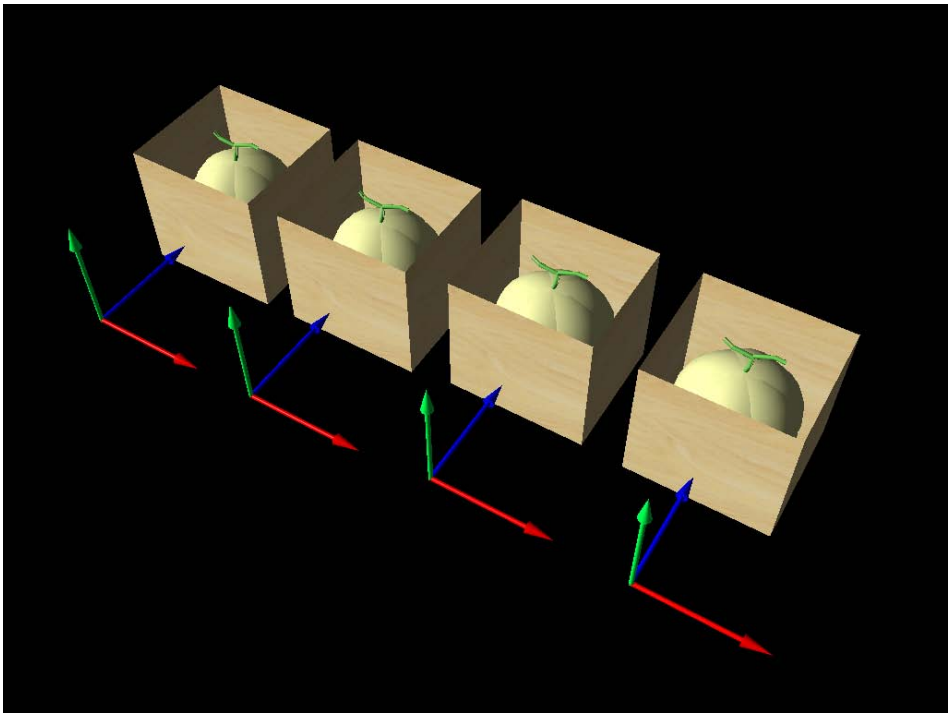


図 5-13

メロンジオメトリが 4 個あるとします。メロンジオメトリは、自分自身のローカル座標系においてその形状が定義されています。図では横に並んでいますが、ローカル座標系だけではお互いの位置関係を定義することは不可能ですので、これはあくまでもジオメトリが 4 つあるということを表しているにすぎません。ローカル座標だけで見れば、メロンは一つ一つ木箱に入った独立したジオメトリと言えばいいでしょうか。(木箱に入ってるのですから、このメロンは高級なメロンです（笑)）

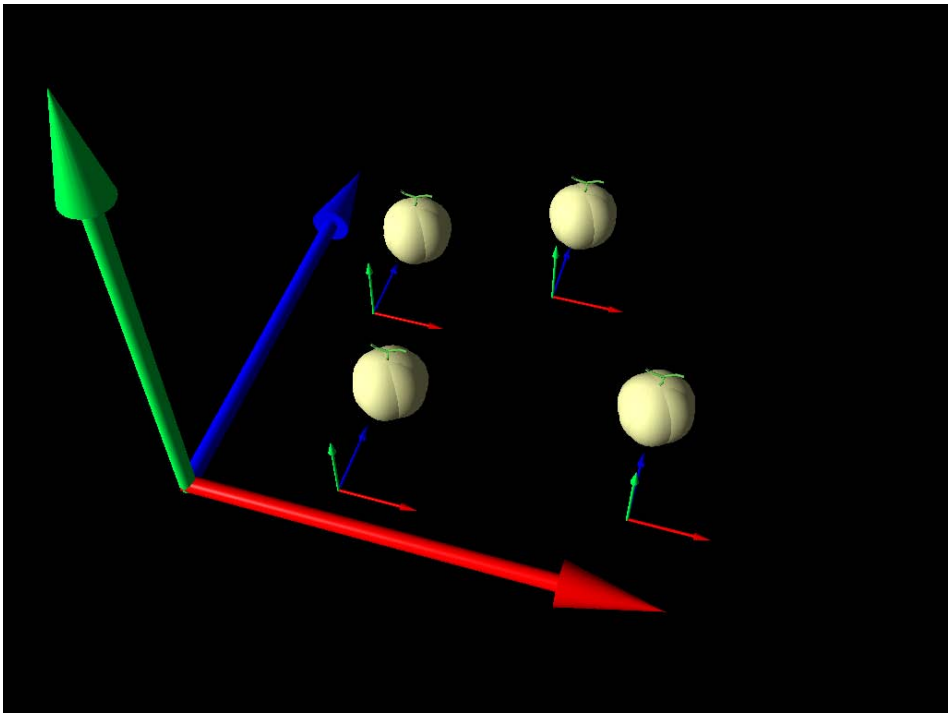


図 5-14
第 1 の変換であるワールドトランスフォームを掛けた状態です。メロンは木箱から出され、外の世界に配置されました。
この個々のジオメトリを絶対座標上に配置することで、初めてジオメトリ同士の相対的位置関係が識別できることになります。

ビュートランスフォーム

ここで、カメラを考慮に入れましょう。
最初、カメラはまっすぐ前方を向いている状態とします。次にカメラを右に回転させたとすると、カメラに写る像は、その逆の左に回転したように見えなくてはならないので、絶対座標ごと（テーブルごと）、ごっそり回転させるのが簡単です。カメラの動きに連動して動かされた絶対座標をカメラ座標と呼ぶことは前述しました。

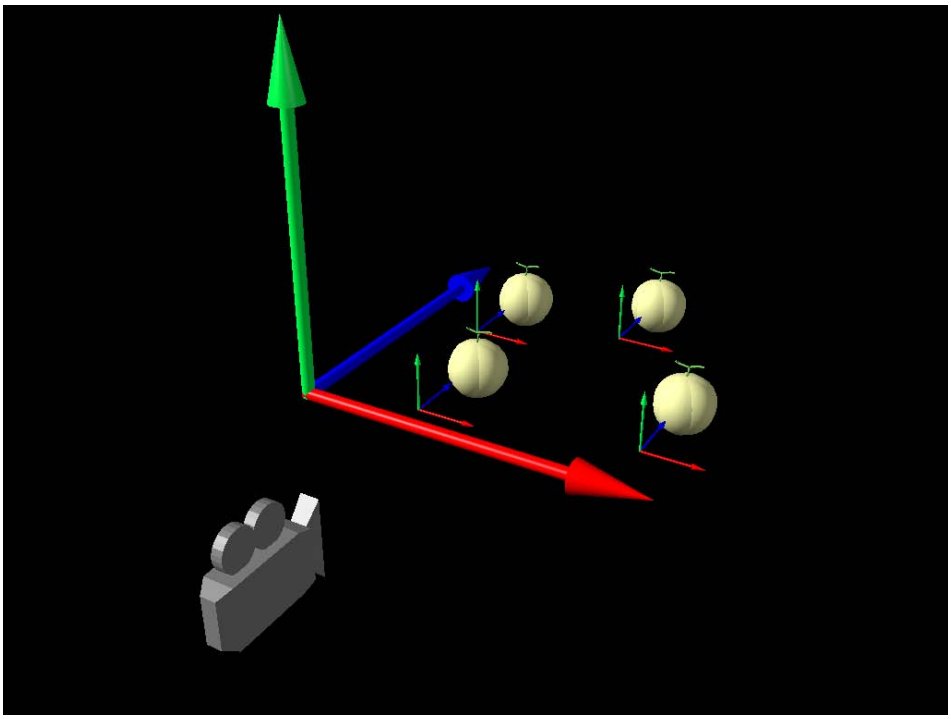


図 5-15

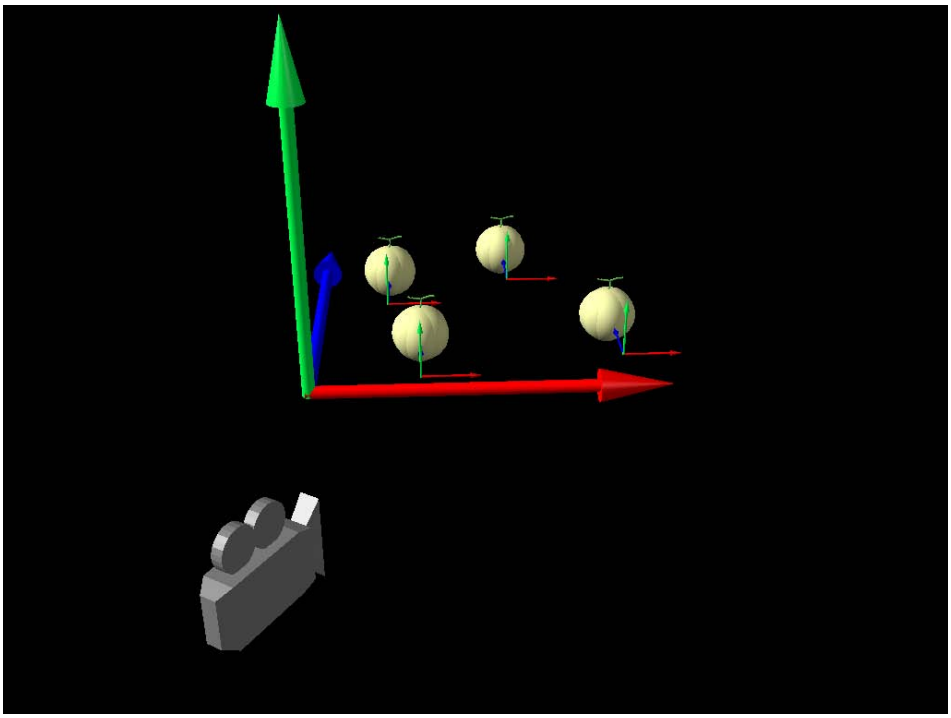


図 5-16

プロジェクショントランスフォーム

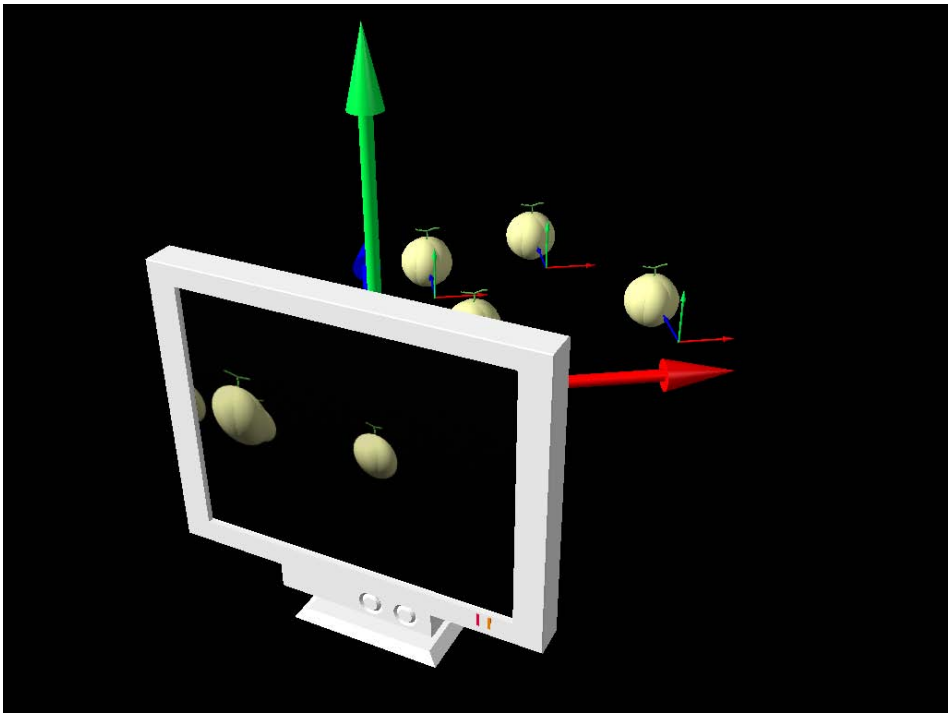


図 5-17

カメラ座標を更に、スクリーン座標に変換します。ここまでの変換が終わって初めてモニター上に表示できます。

このように座標の変化で見れば、メロンジオメトリの座標は、次のように変換を重ねながら最終的にモニターで表示できる 2 次元座標（スクリーン座標）となりモニターに表示されるというわけです。
ローカル座標
↓
ワールド座標
↓
カメラ座標
↓
スクリーン座標

レンダリングパイプラインとは？

今まで、何回かレンダリングパイプラインという言葉が登場しましたが、意図的に説明することを避けていました。最初からいきなり言葉だけで説明するよりコード中でどのような時に呼び出されるのかということを先に見てから説明するほうが良いと思ったからです。ここでは、複雑な図を使ってレンダリングパイプラインの全てを解説することはしません。大体の概要、いや概要というより雰囲気を知るだけでむしろ今は十分かと思っています。
レンダリングパイプラインはその言葉から連想されるようにレンダリングの重要な部分ということは分かると思います。平たく言えば、文字通りレンダリングパイプラインとは “ 入力されたデータ（頂点、マテリアル、テクスチャ）をレンダリング、さらにラスタイジングまで行い画面に出力するもの ” であり、実体はビデオカードの中にあるハードウェアです。
レンダリングパイプラインは一旦レンダリングを開始すると一心不乱に計算しはじめます。具体的な表現をするとレンダリング処理中はアクセス禁止状態になり計算に集中するわけです。もしレンダリングの計算途中でプログラム側から何らかのアクセスを可能にしてしまうと多大なタイムロスとなるのは明らかです。そのデバイスの機能に集中するのは PC、CPU とはハードウェア的に独立しているデバイスの特徴ですが、ビデオカード特に 3D レンダリングは、計算が膨大なこともあり、計算中は排他的になる必要があるのです。計算中はプログラム側から操作できないことから、今まで、コード中で何回が登場した IDirect3DDevice::SetRenderState() によってレンダリングの仕様（レンダリングステートをレンダリングのタイミングに先立って指示していたのです。例えて言うなら、
プログラムを発注者とした場合、
ジオメトリデータ等の原材料を
レンダリングステートで指定された仕様通りに、
ラスタライズされた最終的なスクリーンイメージを成果品として引き渡す
住宅メーカーのようなものといえます。

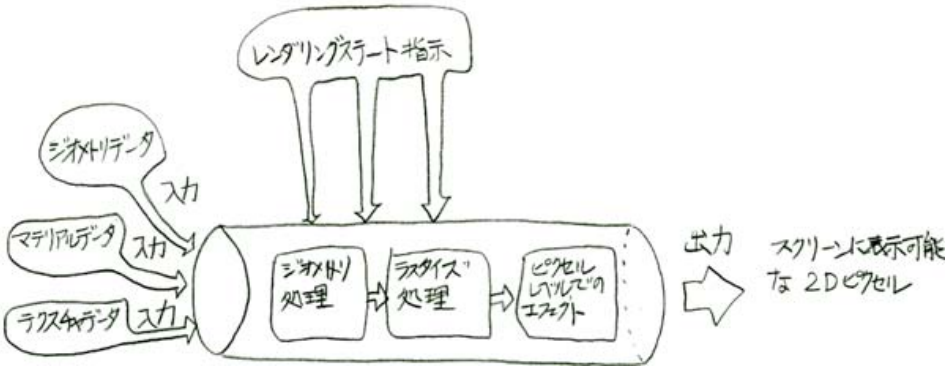


図 5-18

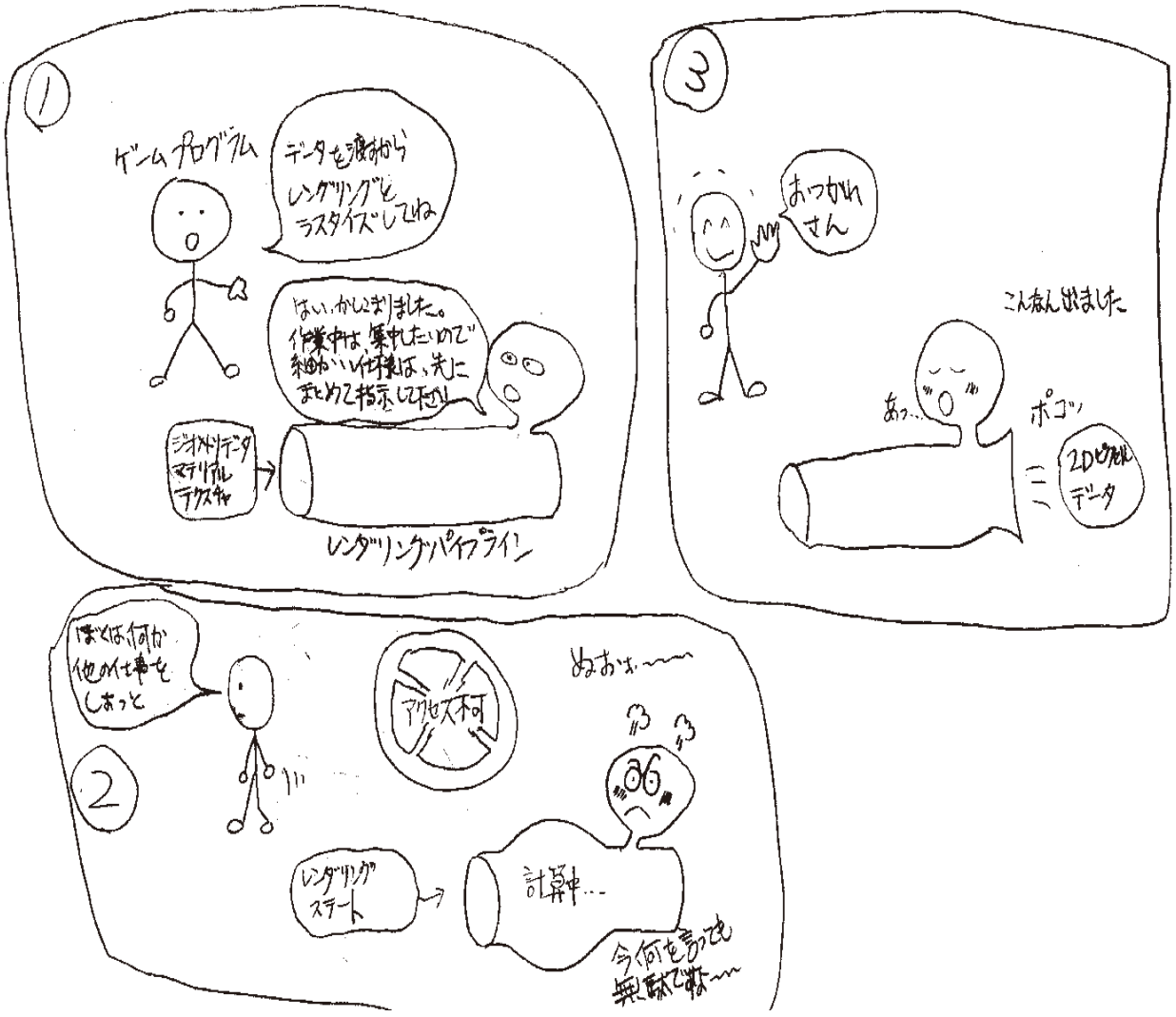


図 5-19

Section 2

セクション１で最低限の基礎知識を紹介しました。このセクション２では最低限の知識を有していることを前提として記述していきます。セクション１で解説した概念や用語について、再度解説することはしません。
セクション１の執筆を終えた段階で思ったことは、基礎知識の解説が予想以上に大変だったということです。おそらく原因は、いろいろな読者層を想定しながら、「この記述でいいのだろうか？、もしかすると、解説の解説を付ける必要があるのではないだろうか？」などと常に（極端な表現をすると１行ごとに）悩んでいたことが大きいと思います。
そして、ようやくある程度基礎的な解説をせずとも話しを進められるこのセクションに来て少しほっとしています。筆者にとってある意味このセクションの方が執筆は楽なような気がします。
セクション１と違い、このセクションはステップアップ形式ではなく、個別参照形式となっているのが特徴です。章を順番に読む必要はなく、知りたい事柄や興味のある事柄に対する章を順不同で参照できるように、各章は独立した内容になっています。但し、６章だけは、このセクションで用いられるＣ（C++）の構文の基本を解説しています。６章で解説していることは、ポインター関連と再帰関数についてです。これらはこのセクションのサンプルコードで広く用いられる構文と思われます。再帰関数については使用するかどうかは未定ですが、ポインターは必ずコードの中に出てきます。もし、読者がＣ言語のポインター及びポインターを実際にコード内で使用する方法をあまり理解していない場合は一読することをお勧めします。このセクション内のある章を読んでいる途中、ポインター等の使用方法によりコードを読み下せなくなった場合６章を読み、またその章に戻り、また別の章を読んでいる途中で再度６章を呼んで、また戻る、というスタイルで読んでいただければと思います。

引数をポインターにする理由

関数コールパフォーマンスの理由から

関数は何らかの型のデータを引数に取ることが殆どですが、関数に渡すデータを、そのまま渡すか、そのデータ型のポインターで渡すかの違いで、状況によっては大きくパフォーマンスに影響します。
今、次のように構造体を定義します。これはただ単に“非常に大きいデータ“ということを強調するための構造体であり、そのサイズは 2048x4=8 キロバイトにもなります。名前もそのままビッグデータです。

```
#define MAX_DATA 2048
struct BIGDATA
{
    DWORD dwData[MAX_DATA+1];
};
```

引数をデータ型そのまま（データのインスタンス）で受け取る関数は例えば次のようになります。

```
VOID InstanceArgument(BIGDATA Bigdata)
{
}
```

引数をデータのポインターで受け取る関数は次のようになります。

```
VOID PointerArgument(BIGDATA* pBigdata)
{
}
```

引数の型名の後にアスタリスクが付きます。
BYTE、WORD、DWORD、FLOAT、DOUBLE 型等のせいぜい 8 バイト以下のデータ型を除いて、ポインターを引数にしたほうが、データのインスタンスをそのまま渡すよりもパフォーマンスは上がります。
その理由は、次のとおりです。
ポインターを引数にした場合、引数が取るメモリサイズはデータ型の“アドレス”（4 バイト）とデータ型の型情報の合計でせいぜい数バイトオーダーであり、型のサイズに影響されず、常に一定のサイズしか必要としません。
一方、データ型をそのまま渡した場合（インスタンスを渡した場合）となると、引数が取るメモリサイズは一定ではありません。そのデータ型のサイズ分だけメモリーを必要とします。
したがって、渡すデータ型のサイズが大きくなればなるほどパフォーマンスの差が広がることになります。ポインター引数の場合、渡すデータ型のサイズに関わり無く常に一定で、しかもごく小さなメモリーしか必要としないというのが、非常に大きなメリットです。そして、引数をポインターにすることのデメリットについて、少なくとも筆者には見つけることが出来ません。関数コールの際にデータにアンパサンドを付けることが面倒というくらいなものでしょうか（これは半分冗談です（笑））。
なお、先に述べたように、データ型が BYTE、WORD…等の組み込みの型でサイズが小さいものであればポインターでもインスタンスでもパフォーマンスは変化しません。
構造体は多くの場合、単純なデータ型よりサイズが大きい場合が殆どでしょうから、引数に構造体をとる場合はそのポインターを引数にすべきでしょう。クラスに至っては、単純なデータ型よりサイズの小さいものを作ることは不可能ですので、必ずポインターで渡すべきものです。クラスのインスタンスをそのまま渡すようなコードは見たことはありませんが、念のため。
パフォーマンスの違いについて、サンプルコードとその結果を示します。このサンプルでは、純粋に関数コールだけの所要時間を計測できるように関数は両方とも中身はありません。

```
 1: #include <stdio.h>
 2: #include<conio.h>
 3: #include <time.h>
 4: #include <windows.h>
 5: //
 6: //
 7: // マクロ宣言
 8: #define MAX_DATA 2048
 9: //
10: //
11: // 構造体宣言
12: struct BIGDATA
13: {
14:         DWORD dwData[MAX_DATA+1];
15: };
16: //
17: //VOID PointerArgument(BIGDATA* pBigdata)
18: // ポインター引数による関数
19: VOID PointerArgument(BIGDATA* pBigdata)
20: {
21: }
22: //
```

```
23: //VOID InstanceArgument(BIGDATA Bigdata)
24: // インスタンス引数による関数
25: VOID InstanceArgument(BIGDATA Bigdata)
26: {
27: }
28:
29: //
30: //VOID main()
31: // エントリー関数
32: VOID main()
33: {
34:         DWORD i;
35:         BIGDATA Bigdata;
36:         time_t tTimer;
37:
38:         // ポインターを引数にした場合
39:         tTimer=clock();
40:         for(i=0;i<1000000;i++) // 関数を 100 万回コールする
41:         {
42:                         PointerArgument(&Bigdata);
43:         }
44:         printf("¥n ポインターを引数にした場合の所要時間 : %d ¥n",clock()-tTimer);
45:
46:         // インスタンスそのものを引数にした場合
47:         tTimer=clock();
48:         for(i=0;i<1000000;i++) // 関数を 100 万回コールする
49:         {
50:                         InstanceArgument(Bigdata);
51:         }
52:         printf("¥n インスタンスそのものを引数にした場合の所要時間 : %d ¥n",clock()-tTimer);
53:
54:         printf(" 何かキーを押してください ");
55:         while( !_kbhit() );
56: }
57:
```

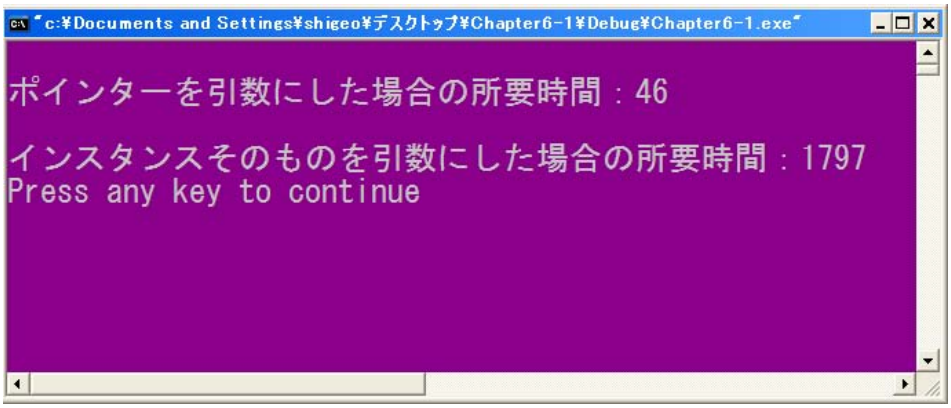


図 6-1

必然性の理由から

２つの引数の除算をして、その答えを返す単純な関数を考えてみましょう。
引数として除算する数と除算される数の２つだけをとる場合が関数 Divide1 で、引数に答えを格納する変数のポインターもとる場合は関数 Divide2 のようになります。

```
FLOAT Divide1(FLOAT fA,FLOAT fB)
{
  Return fA / fB;
}

VOID Divide2(FLOAT* piResult,FLOAT fA,FLOAT fB)
{
    *pResult=fA / fB;
}
```

p Result は呼び出し側の変数へのポインターです。
Divide1 関数の場合、計算結果を戻り値としているのに対し、Divide2 関数はポインターにより呼び出し側の変数にダイレク

トに結果を代入しています。
この２つの関数は、機能的に全く同じと言えます。但し Divide2 が VOID 型の関数である場合に限ります。どうゆうことかと言えば、例えば計算において何らかのエラーが出た場合にエラー情報を戻り値にしたいと考えた場合、Divide2 は問題無く修正することができます。

```
HRESULT Divide2(FLOAT* piResult,FLOAT fA,FLOAT fB)
{
  If(fB == 0) Return E_FAIL;
    *pResult=fA / fB;
}
```

もしも、fB がゼロであればエラーコードとともにその時点で関数から抜けるというものです。ゼロ除算は、普通プログラムがクラッシュしますので、それを回避しています。
一方、Divide1 関数は、戻り値は計算結果用に使ってしまっているので、エラーコードを呼び出し側に知らせる手段がありません。fB がゼロである場合に静かに return するくらいしか出来ません。

```
FLOAT Divide1(FLOAT fA,FLOAT fB)
{
        If(fB == 0) Return;
  Return fA / fB;
}
```

その場合呼び出し側は、エラーで戻ったのか正常終了して戻ったのかは全く分かりません。ある意味これは、深刻なバグと言え、このような “ 静かなエラー ” を生む関数を作っては絶対にいけません。開発者から見れば、いっそのことクラッシュしてくれたほうがまだ親切（？）というものです。（対処することができます）
なお、インスタンスそのものを引数にすることによるメリットもあります。それは各関数の間で値の独立性を比較的保てるということです。C 言語において基本的に関数内のローカルな値は他の関数から参照することはできません。ところが、ポインターを引数にして渡せば他の関数から参照することも変更することも “ 出来てしまいます ”。参照することは問題になることはありませんが、変更することは時としてバグの原因につながることがあります。先の Devide2 関数では正に呼び出し側の値（pResult）を変更しています。Divide2 関数程度の機能である場合、バグになるということは考えられませんが、もっと規模の大きなプログラムの場合、なんらかの不具合が発生した時にポインターを辿って関連する関数全てをチェックする必要が出てきます。
このようにインスタンスを渡す場合では、参照は出来ますが変更はできませんので値はリードオンリーです。一方、ポインターを渡す場合において値はフルアクセスの状態であることを意識するようにしてください。

“ ポインターのポインター ” の使いどころ

ポインターのポインターは、いろいろなコードでよく見かける構文であり珍しいものではありません。ところがプログラミング初心者の多くは、その意味するところはなんとなく分かってはいるものの、いざその使い方、ひいては、どのような時に使えばいいものなのかということが分かっていないことが多いようです。これは筆者のつたない講師経験から感じ取ったことです。ポインターのポインターは、文字通り何らかの値のアドレスのアドレスです。言葉の意味は分かっていても、その存在理由と使用局面がわからないため、コードの中でポインターのポインターを表す “**”2 重アスタリスクが出てきた時点で嫌悪感を感じてしまうようです。
使いどころ　その 1
空のポインターを何らかのデータへの適正なアドレスで初期化したい場合。

```
HRESULT InitPointer(INT** ppiVal)
{
  *ppiVal=(INT* )malloc(10);
  Return S_OK;
}
….

INT* piVal=NULL;
InitPointer(&piVal);
```

引数 ppiVal は、INT 型のポインター変数のアドレス、すなわち “ ポインターのポインター ” です。
このような用途であれば、初心者でも想像できるかもしれませんし、これはポインターのポインター特有の用途とは言えません。“ 使いどころ ” というほどの積極的な意味は無く、どちらかという必要に迫られてポインターのポインターにしたという感じでしょう。
使いどころ　その２
これから解説する内容が、真の意味での “ 使いどころ ” ではないかと筆者は思っています。コードはコンソールアプリケーション用です。
その前に、まずは、単なる “ ポインター ” の使いどころから見てみてください。

今、仮に INT 型のデータ 10 個をゼロで初期化するプログラムを考えてみます。データを格納する変数を配列として宣言すると次のようになります。

```
//　グローバル宣言
```

```
#define AMOUNT 10
INT iVal[AMOUNT+1];
// エントリー関数
VOID main()
{
    for(DWORD i=0;i<AMOUNT;i++)
    {
        iVal[i]=0;
    }
}
```
この場合、データの個数が最初から 10 個と分かっているので配列を使用することが出来ましたが、普通このように配列を用いることができる状況は限られています。最初からデータの個数が不定の場合は配列を宣言できないのは明らかです。それとも無理やり十分に大きい配列を宣言して（iVal[10000] とか !!）データ個数がそれ以上にならないことを祈るでしょうか？いえ、それは開発者としてはやってはいけないことです。（とはいえ筆者はたまにやることですが…みなさんはもっとまっとうなプログラミングを目指してください）
データの個数が不定の場合、ポインターを用いると次のようにすっきりコードが書けます。
```
//   グローバル宣言
INT* piVal=NULL;
DWORD dwAmount=0;
// エントリー関数
VOID main()
{
    printf("\n   データの個数を入力してください \n");
    scanf("%d",&dwAmount);
    piVal=(INT*)malloc(sizeof(INT)*dwAmount+1);
    for(DWORD i=0;i<dwAmount;i++)
    {
        piVal[i]=0;
    }
}
```
piVal はポインターなのに配列のように扱っていますが、これは C 言語においてポインターと配列が密接な関係である証です。
＊
これであれば、データの個数が 10 個でも 10,000 個でも（メモリーが許す限り）対応できます。
これが、“ ポインター ” の使いどころです。次では、“ ポインターのポインター ” もこれと同様の流れで考えてみましょう。

今度は INT 型の 20 個のデータが “10 セット ” あり、各データをゼロで初期化するプログラムを考えます。
先程と同じように、最初に配列形式のコードを示すと、次のようになります。
```
//   グローバル宣言
#define SET_AMOUNT 10
#define AMOUNT 20
INT iVal[SET_AMOUNT][AMOUNT+1];
// エントリー関数
VOID main()
{
    for(DWORD i=0;i<SET_AMOUNT;i++)
    {
      for(DWORD k=0;k<AMOUNT;k++)
      {
            iVal[i][k]=0;
      }
    }
}
```
先の配列によるコードが、ただ 2 次元配列になっただけです。
先の流れと同様に、この場合でも、データの個数が最初から 20 個× 10 セットと分かっているので配列を使用することが出来ましたが、最初からデータの個数が不定の場合は配列を使用できません。
データの個数が不定の場合、今度はいよいよ “ ポインターのポインター ” の出番です。
```
//   グローバル宣言
INT** ppiVal=NULL;
DWORD dwSetAmount=0;
DWORD dwAmount=0;
// エントリー関数
VOID main()
{
    printf("\n   データとセットの個数を入力してください \n");
    scanf("%d%d",&dwAmount,&dwSetAmount);
```

```
    ppiVal=(INT**)malloc(sizeof(INT*) * dwSetAmount);
    for(DWORD i=0;i<dwSetAmount;i++)
    {
       INT* piVal=(INT*)malloc(sizeof(INT) * dwAmount+1);
       ppiVal[i]=piVal;
     }

    for(DWORD i=0;i<dwSetAmount;i++)
    {
       for(DWORD k=0;k<dwAmount;k++)
       {
          ppiVal[i][k]=0;
        }
     }
}
```

データの個数が不定で、しかもデータのセット数さえも不定の場合は、当然配列表現は出来ません。なお、データのセット数のみ分かっている場合はポインター配列 piVal[セット数] という風に出来ますが、スマートに全てポインターで柔軟な構造にしたいものです。配列で表現できる場合でも、「私はどうしても配列で容量を固定するのは嫌だ」という人もいるでしょう。

データの個数を dwAmount に、データセットの個数を dwSetAmount にそれぞれ格納します。

```
ppiVal=(INT**)malloc(sizeof(INT*) * dwSetAmount);
```

一番最初に行う処理は、“ ポインターのポインター “ に格納する ” ポインター “ の数分だけメモリーを確保します。これで ppiVal にポインターを格納する準備ができました。

```
for(DWORD i=0;i<dwSetAmount;i++)
{
    INT* piVal=(INT*)malloc(sizeof(INT) * dwAmount+1);
    ppiVal[i]=piVal;
}
```

次に、セット数分のループ中で、毎回 データ数分のメモリーを確保して、そのポインターを ppiVal の該当するセット番号の領域にコピーします。

これで、ポインターのポインターとしての準備は完了です。あとは、ポインターポインターを、まるで 2 次元配列のように扱えます。

結論をまとめると

「未知の数のメモリーを確保する必要がある場合は、そのデータ型のポインターを使う」

「未知の数自体が階層になっている場合（[2][3] など）はポインターのポインターを使う」

さらに、

「未知の数が 3 次元以上の階層になっている場合（[2][3][3] など）は、ポインターのポインターのポインターを使う」

これから分かるように、未知の数の階層だけポインターも階層にすればいいというわけです。

さて、ここで筆者としては、読者が上手く理解したかどうか不安です、特にポインターのポインターの例を上手く消化してくれたかどうかは実に不安です。そこで、図により再度流れを説明することにします。

1 次元配列とポインターのメモリー配置

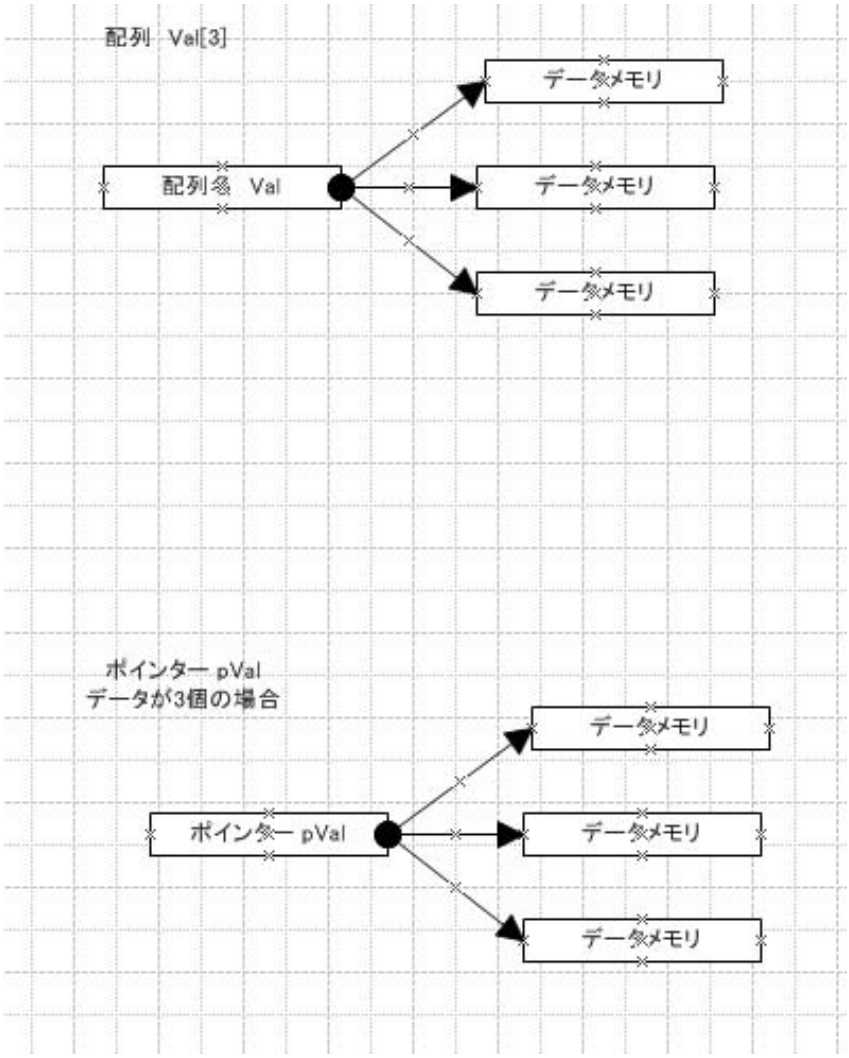


図 6-2

2 次元配列とポインターのポインターのメモリー配置

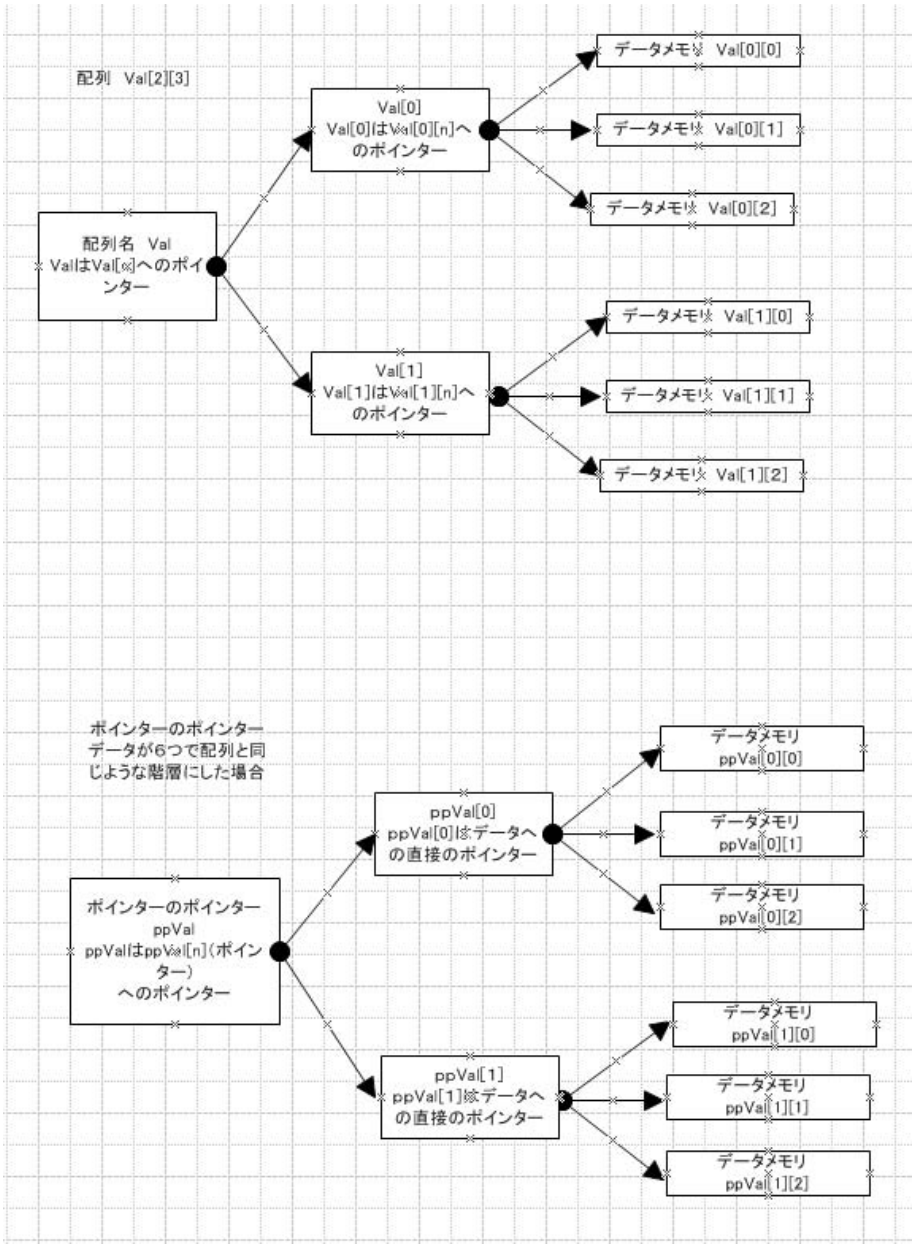


図 6-3
2 次元配列では不可能なメモリー配置

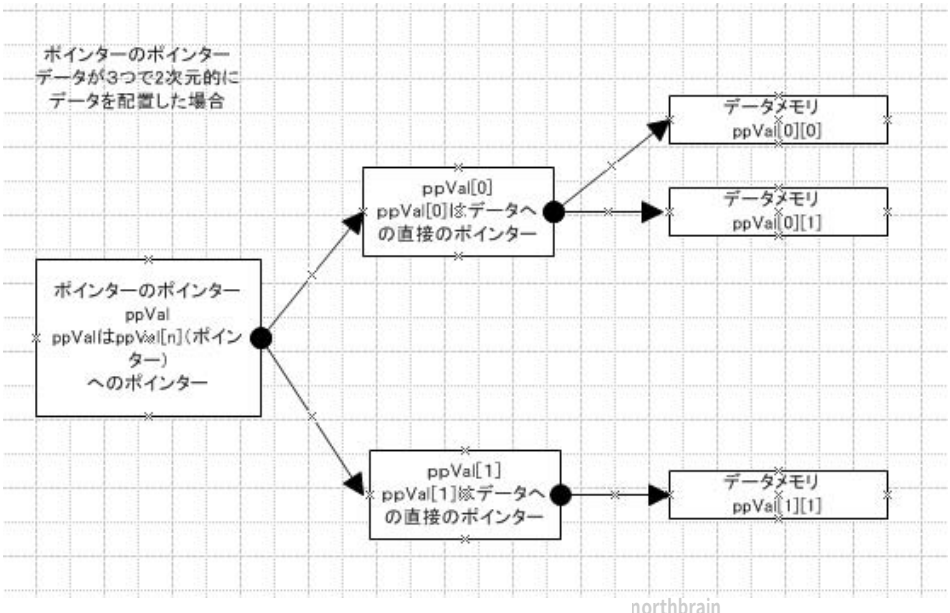


図 6-4
この図から分かるように、ポインターのポインターによるメモリー確保の場合、より柔軟で無駄の無いメモリー確保が可能です。なぜなら、配列の場合、例えば Val[2][3] と宣言すると Val[0] に 3 個分のデータメモリが確保され Val[1] にも全く同じく 3 個分のメモリーが確保されます。一方、ポインターのポインターの場合、（配列表現すると）pVal[0] には 2 個分のデータメモリを確保して、pVal[1] には 1 つ分のデータメモリしか確保しないということが出来ます。時としてデータ領域は、配列のように綺麗に（？）領域を確保する必要が無い場合もあります。そのような時、ポインターのポインターによりメモリーを節約することができるわけです。

以上の事柄から配列とポインター（のポインター）の利点と欠点をまとめてみました。

配列
　利点
　宣言した時点でメモリーが自動的に確保される。
　欠点
　そのサイズを動的に変更できないため、未知の要素数に対応できない。
　多次元配列の場合は、メモリーを無駄に確保してしまう可能性がある。

ポインター（及びそのポインター）
　利点
　サイズを動的に変更できるため、未知のデータ個数に対応できる。
　ポインターのポインターの場合、“ ルートポインターが指し示す各ポインター ” が指し示すデータの個数は（当然）同じである必要はありません。そのことにより、メモリーを無駄にする可能性が低い。
　欠点
　プログラマーの責任において、メモリーを確保しなくてはならない。

　この関係から分かるとおり、配列とポインターの利点と欠点は、トレードオフになっています。実務上の見地から言うと、規模の小さい、又は、テストプログラムのようなプログラムでは簡単に扱える配列を使用し、そして、データ量が非常に多くなる可能性のあるプログラムでその量を確定することができない場合にはポインターを使用するというのが一般的です。
メモリーの浪費を気にすることのないテストプログラムを除いて、ポインターの利点から得られる恩恵は、その欠点を補って余りあるものであることと言えます。

再帰による階層データの参照

再帰関数は本書ではアニメーションメッシュの階層走査に登場しますので、再帰関数の原理と、その使用例について解説したいと思います。関数を再帰的にすることにより、同様の機能を通常の関数で行う場合よりスマートに処理できる場合があります。
再帰関数の使用例を 3 DCG に関連のあるロジックに絡めて解説したほうが理解し易く、また、再帰関数の意義も同時に意識できると考え、本節ではそれぞれのメッシュが階層構造になっているメッシュアニメーションを再帰関数で読み込む方法を最後に紹介します。
その前に、少し準備が必要です。最も簡単な再帰関数を例に “ 再帰 ” とはどうゆうことなのかということを理解しましょう。
今、次のような処理をする関数を考えます。その関数は与えられた整数を 1 減算して表示し、数がゼロになるまで表示を繰り返すという仕様です。この処理を、再帰関数を用いてコードにすると次のようになります。

```
VOID FuncA(INT iVal)
{
        printf("val=%d \n",iVal);
        if(iVal>0)
        {
                FuncA(iVal-1);
        }
}

VOID main()
{
        FuncA(10);
}
```

エントリー関数 main() は、関数 FuncA() をコールしているだけです。引数として値 10 を渡しています。
FuncA() 内での処理を見てみましょう。FunkA 内で自分自身である FunkA を呼び出しています。このように、その関数内で自分自身をコールすることが再帰であり、その関数は再帰関数といえます。
FuncA 関数が（外部から）最初にコールされると、値がゼロになるまで FincA は自分自身をコールし続けています。この流れを図にすると次のとおりとなります。

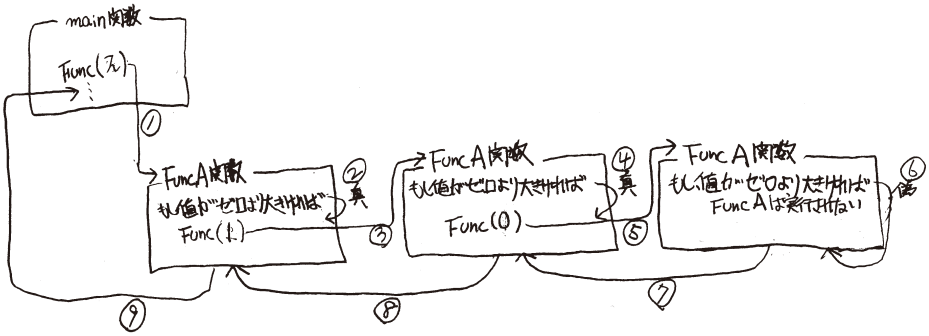


図 6-5

紙面の都合上、FuncA に渡す初期値は 2 にしています。図中 3 つの FuncA 関数はもちろんただ一つの FuncA 関数です。
ここで、いまひとつ再帰がイメージできないという読者のために、補足します。再帰構造は goto 文で表すことができます。そして、再帰構造から goto 文への " 変換 " は機械的に行うことが出来ます。機械的に goto 文へ変換できるので、実際コンパイラーが再帰構造をマシン後に変換する際は、この goto 文への機械的な変換を行っているのです。(goto 文があるわけではありません、ある命令から別の命令へのジャンプという意味です)
FuncA を再帰ではなく goto 文を用いて非再帰に書き直すと次のように変換できます。

```
#include <windows.h>

//
//VOID FuncA(INT iVal)
//goto 文による再帰の除去
VOID FuncA(INT iVal)
{
start:;
	if(iVal<0) goto end;
	printf("val=%d \n",iVal);
	iVal--;
	goto start;
end:;
}
//
//VOID main()
// エントリー関数
VOID main()
{
	FuncA(10);
}
```

なお、これは " 再帰の除去 (recursion removal)" と呼ばれ、コンパイラーがコードを機械語に翻訳するときに行っていることと原理は同じです。
さて、再帰がどのようなものか述べたので、これから、再帰の真価を実感できる処理を考えていきましょう。
ところで、先の FuncA 関数を見て、読者の中に「再帰 (ましてや禁断の goto 文) を使用しなくても出来る処理だ」と思った人がいるかもしれません。たしかに " その通りです "。FuncA は次のように for 文であっさり書くことが出来ます。

```
//
//VOID FuncA(INT iVal)
// 通常の関数
VOID FuncA(INT iVal)
{
	for(INT i=iVal;i>=0;i--)
	{
		printf("val=%d \n",i);
	}
}
//
//VOID main()
// エントリー関数
VOID main()
{
	FuncA(10);
}
```

しかし、再帰を適用できる処理を for 文（及び他のロジック）でも書くことができるのは FuncA のような非常に単純なコードの場合です。
14 章で解説するアニメーションメッシュの参照は再帰以外では、スマートに書くことができない処理です。

Direct3Dでは､単一メッシュ或いは複数メッシュアニメーションを独自のフォーマットでXファイルというファイルに収録し、ゲームの初期時にそのXファイルを読み込むという一つの流れがあります。メッシュアニメーションの場合、大きく２通りのアプローチがあり、一つはメッシュの頂点自体を個別に動かすことによりメッシュそのものを変形しアニメーションさせる手法と、メッシュを変形させずメッシュ間の相対的な姿勢を変化させアニメーションさせるものです。ときには、２つの手法が混在することもあります。
いずれにしても、複数のメッシュから成るアニメーションにおいて、個々のメッシュは、それぞれ互いに関連し合い、関連するメッシュの姿勢に影響を受ける場合が殆どです。
具体例を書くと、例えば、人間が歩くアニメーションの場合、頭は頭メッシュ、胴体は胴体メッシュ、腕は腕メッシュ…などというように体の各パートごとのパーツメッシュとして別々にメッシュを作成します。
“脚”が腿（脚の付け根から踵まで)、膝（膝から足首まで)、足首（踵からつま先)、と３つのパーツから成っている場合、歩行アニメーションをさせるには、まず腿をそれなりに傾けます、この時、踵と足首も動かさなければなりません。(膝や足首が動かないで腿だけ動いている様を想像してください）膝と足首が腿に連動して動くようにすれば楽です。あるメッシュを別のメッシュに連動して動かすようにするためにメッシュに親子関係を設定するのが一般的です。親子関係があれば親の姿勢変化が子に伝えられ、親が動くたびにいちいち子の姿勢をそれぞれ計算する必要がありませんし、なにより、独自に子の動きを計算していたのでは動きが不自然になる可能性があります。
この例で言うと、親子関係は次のように設定します。
（親） 腿
（腿の子） 膝
（膝の子） 足首
したがって足首は腿の“孫”という風に考えることもできます。
人間の体は、脚のほかにも手や胴など他のパートがありますので、それらにも全て親子関係を設定します。この“親子関係”は階層構造になります。その階層構造を参照するには再帰が有効であり、実際、Xファイルのアニメーション情報は再帰による読み込みを想定して格納されています。

Chapter7 エレガントなコードを目指す

綺麗なコードとは？

「綺麗なコード」はどのようなコードなのでしょうか？「綺麗な」という言葉の抽象性が高いため、その定義は様々だと思います。「字下げが統一されている」、「コメントが丁寧」、「変数や関数名が分かりやすい」、「関数やクラスの独立性が高い」、「メンテナンスが容易」など、色々定義できますが、最終的には、もっと抽象的に「読みやすいコード」、「バグの無いコード」と定義するのが無難なところでしょう。
最初から「読みやすいコードを目指す」という題にしておけば、このようなある種哲学的な文面にならなかったのですが、「エレガントな」という言葉からは、直接的には「綺麗な」という言葉が連想されるので「綺麗」という言葉を使用しました。
エレガントなコードが読みやすいコード、バグの無いコードであるならば、それを目指すのは当然でしょう。では、どうすればエレガントなコーディングが出来るのでしょうか？
コードをエレガントにするためには幾つかのルールを守ることが必要です。次節以降でそれらのルールを解説しますが、それらのルールや知識の他にもう一つ重要な要素があります。それは“経験“です、経験だけは書籍から得ることは不可能です。もし読者がプログラミング初心者であるならば、最初からエレガントなコードを書けなくてもいいのです、どんな天才でも知識だけで最初からエレガントなコードを書くことは不可能なのですから、ただ、読みやすいコードを書く努力をしたり、意識を持ちながらコーディングするという姿勢は忘れないほうがいいでしょう。時間をかけて、時には失敗しながらだんだん綺麗なコードを書けるようになってくものだと筆者は思います。

マクロの奨め、ハードコードの禁止

例えば、DrawText(INT iX,INT iY,TCHAR* szStr) という関数があったとします。この関数の機能は、スクリーン座標 (iX,iY) に szStr が指す文字列を描画するものとします。
そして、コード中で次のように使用したとしましょう。
DrawText(100,200,”hello”);
一見何も問題ないかのように見えます。ところが、この使用例は思ったより深刻な問題をはらんでいます。何が問題かというと、100,200 という“生の数字”を書いていることです。
これは、次のようにするべきです。
#define OOWIDTH 100
#define OOHEIGHT 200
DrawText(OOWIDTH,OOHEIGHT,”hello”);
#define プリプロセッサにより数字を分かりやすい言葉に置き換えます。OOWIDTH と OOHEIGHT はプログラマーが分かりやすい言葉であれば何で良く、このように定義された言葉はマクロと呼ばれ、#define によるこのような定義をマクロ定義といいます。
一方、“生の数字”をマジックナンバーなどと呼び、そのようなコードを書くことをハードコード (Hard Code) とも呼ばれていますが、どちらも、もちろん良い意味ではありません。
なぜ、生の数字であるマジックナンバーを使用することがいけないのか、解説しましょう。
理由は２つあります。
理由その 1
仮に DrawText 関数をコールしているステートメント（行）がプログラム全体を通して、50 行あったとします。そして、全ての描画座標を 100,200 としているものとします。座標を生の数字で書いていた場合、もし、その座標を変更する必要が出た時に 50 行全てを変更しなくてはなりません。ところが、座標をマクロ定義していれば、修正箇所はマクロ定義の部分 1 箇所だけで済みます。
この例えに限って、エディターの検索と置換等の機能を使えば同様の修正ができそうですが、一般的にどんなに上手く正規表現を使用して置換してもマクロほど綺麗に修正できませんし、なにより、新たなバグを発生させる原因にもなります。
理由その２
マジックナンバーは、その意味するところが分かり辛いということもあります。他人が書いたコードを見ている中で、マジックナンバーが出てきたら、その意味を理解するのは容易ではありません。いや、容易ではないと言うよりも、意味を読み取るのは基本的には不可能といっていいでしょう。たとえ、自分が書いたコードであっても、書いてから 1 ヶ月も経てば他人のコードのようなものです（失礼、これは記憶力の無い筆者の感覚かもしれません）。自分の書いたコードがどのような処理をしているのか、ほぼ忘れてしまうくらい時間が経過したときにコードを見ている中で、マジックナンバーの意味を思い出すことに時間を費やすのは嫌なものです。
筆者の経験から言ってマジックナンバーを使用することは百害あって一利無しと断言できます。マクロを使用してください。
なお、もし、DrawText をコールしているステートメントが 1 行だけしかない場合はどうでしょうか、1 行だけなら問題ないと思うことを積極的に否定はしません、たしかに、1 行だけであれば先の２つの問題も表面化しないでしょう。しかし、プログラム全体の統一性がその時点で低下していることは確かです。これは次節“鉄の掟を貫く“でも解説しますが、統一性は大切です。マジックナンバーがプログラム中に”在ると分かっている“のと”無いと分かっている“のとでは、精神的にも実務的にもけっこう違うものです。実際に大規模なプログラムで全くマジックナンバーの無いプログラムというものがあるのかどうか分かりませんが（少なくとも筆者はマジックナンバーが 1 個たりとも無い大規模プログラムを書いた記憶はありません）、たとえ 1 箇所だけだとしてもマクロ定義するくらいの”意識”は持つべきだと思います。

“鉄の掟”を貫く

「鉄の掟」とは、どうゆうことかというと、一貫性、統一性を維持するための決まりという意味です。単に「鉄則」と言ってしまうよりもインパクトがあり筆者はこの言葉をプログラミングにおける大前提にしています。

プログラミングはもちろん、設計の段階においても一貫性は非常に重要です。特にチームで共同作業する場合には、一貫性を維持しないと作業能率が極度に低下する場合があります。
一貫性を維持するためにのルールを本節では「掟」と呼ぶことにします。
掟は、広く一般的に知られているものと、自分で決めるものがあります。一般的に知られている掟は抽象的で範囲の広いものであり、自分で決めるものは、具体的で詳細なものになるのが普通でしょう。
一般的な掟の例
Goto 文は一切使用しない。
グローバル変数の使用は控える。
変数名や関数名は、分かりやすいものにする。
ルーチンの部品化を意識する。
など。
Goto 文を一切使わないというのは、異論があるかもしれませんが、筆者は “ 一切使用しない派 ” でありそれを支持します。グローバル変数は一切使用しないとまでは言いません、どうしてもグローバル変数を使用しなければならない場合があります。あくまで使用頻度を控えるということです。ルーチンの部品化については次節で解説します。
自分で決める掟の例
人によってはクラスのメンバー変数は全て private や protected にするというポリシーを持っている人もいるでしょう。徹底しすぎている感があるますが、その人にとっては重要な掟の場合もあります。
そのような普遍的な掟と同時に、一時的・局所的な掟を決めるべき時もあります、例えば、現在進行中のあるプロジェクトあるいはあるプログラムまたはルーチン固有の開発方針やデバッグ時の修正方針は、具体的になる場合があり、しかも一時的なものです。そのため一般的な掟として定義されていない場合があります。そのような場合は、自分で掟を決めることとなります。
掟が適正なものなのかどうなのかということより、もっと大切なのは、これら掟を “ 貫く ” ことです。貫くということは 100％実践するという意味です。なにせ “ 鉄の ” 掟なのですから。いかなる時も破ることはできません。時にはたとえ掟が間違っていても守らなければならない状況があります。それは、一貫性を保つためです。特にソフトウェア開発において一貫性を 100％維持する意識は大切で、100％の一貫性と 99％の一貫性は大違いです。
100％と 99％がなぜそんなにも違うのでしょうか、具体的な例えとして次のシナリオを考えてみてください。
あなたは、5 年もの長い年月を費やし、ついに自作（しかも大作）ゲームをほぼ完成させました。コードは全体で 10 万行にもなる力作です。コード内では前出の DrawText 関数を 1000 箇所で使用していたとします。DrawText の引数は、努めてマクロ（1000 箇所なので数種類のマクロを定義した）を使用したつもりだったのですが、たった一箇所だけマジックナンバーである数字を直接書いた箇所がありました。1000 分の 1 箇所です、そのパーセンテージはたった 0.1％であり、一貫性は 99.9％です。そして、完成目前という時にあなたは、スクリーン座標の横と縦を逆に考えていたことに気付きました（それまで気付かなかったのかという疑問は抱かないでください）。一瞬、青ざめましたが、すぐにマクロの利点を生かすことを考えほっとしました。マクロ定義部分の数字だけ書き換えればいいということを…さっそく座標用に定義した数種類のマクロ定義部分のみを変更し、プログラムを実行してみました。ところが、なぜか、横と経が修正されていない部分が残っています。そう、たった 1 箇所のマジックナンバーによる DrawText 部分です。DrawText は 1000 箇所で使用され、しかもコードは全体で 10 万行もあります。10 万行のコードから、そのマジックナンバーを見つけるのに浪費する時間は少なくないでしょう。
これは、一貫性が大切だということを説明するために、いささか極端かつ誇張したシナリオですが、一貫性が、ほんの少しでも崩れるとそれだけリスクが発生するということを感じてもらえればと思い想定したものです。
今度は、別の意味での一貫性もまた重要だということを別のシナリオで見てみましょう。
あなたは、あるゲーム開発チームの一員で、チームメンバーと共同でコーディングしているとします。チーム内のメンバーの一人は初心者で、その彼がとんでもない認識不足から考えられないミスをしでかしました。そのミスはこうです。
変数をインクリメント（1 加算する）するのに、本来は (変数)++ としなければならないところを（変数）+ としてしまっていたのです。これは、コンパイル時に警告が出てくれるので分かりやすい部類の間違いですが、その初心者メンバーはコーディング中一度もコンパイルせずに全てをコーディングしてから初めてコンパイルしたので間違いに気付きませんでした。そして、その間違い箇所はおおよそ 100 箇所あったとします。
あなたは、なんとか少ない労力で機械的にその間違いを修正しようと考えました、まず、思いついたのは、エディターの文字列置換機能を使うというものでしたが、変数は 1 種類ではなく、さらに＋記号は他の正常な部分でも頻繁に使用されているので、置換機能はむしろ傷口を広げる恐れがあります、どうやら一つ一つ手作業で修正する以外に無いという結論に達しました。軽い目まいを覚えつつチーム内で次のような修正手順を決めました。
「（変数）+ となっている箇所を （変数）++ に修正する。」
というもので、各チームメンバーは、それを掟とし修正作業にかかりました。修正作業が 8 割ほど進んだところで、あなたはある重大なことに気付きました、インクリメントしている箇所は実は全てデクリメント（1 減算する）するべきものであるということが分かったのです。さぁ、あなたはどうするべきでしょう？
“ それ以降の修正はすべてデクリメント、つまり (変数)— と修正する ”
のか、それとも、
” 今までどおり（変数）+ を (変数)++ に修正する “
のどちらでしょう。答えは、後者で今までどおりの修正規則を守るということです。インクリメントが間違った処理、つまりは修正規則自体が間違っているということを分かっていても、あえて間違った修正規則を貫くのです。なぜかというと、その初心者メンバーはコード中全体を通して ++ 演算子は使用していないことが分かっていて、間違い箇所が全て ++ に統一されていればエディターの文字列一括置換機能を使えるからです。
間違っていようが、“++“ に統一されていさえすれば一括置換で “++” を “—” にボタン一発で変換できるのです。
プログラミングに限らず、修正規則自体が間違っていることに気付いても、あえてその規則を最後まで貫き、一括処理し易いように統一するという状況はたまにあるのではないでしょうか？

ルーチンの部品化

概して、初心者は一つの実装ファイル (.c や .cpp ファイル) に殆どのルーチンを詰め込んだり、一つの関数に何千行もコー

ドを書き込む傾向があります。何でもかんでも、一箇所に放り込んでしまい、コードの可読性を悪くしてしまうというのをよく見かけます。他人がそれを見て理解するのは容易ではありませんし、本人すらもコード量が多くなるにしたがってなにがなんだか分からなくなってしまいます。
プログラミングというものがこの世に誕生した時から、このような状況を改善し、分かり易いコードを目指す方法論が議論されてきたはずです。公に提唱された最初の方法論は「構造化プログラミング」というもので、1960 年代にエズガー・ダイクストラ（Edsger Wybe Dijkstra）という人が提唱しました。
その後、構造化プログラミングの普遍的理念を継承しつつ、当時のプログラムをさらに保守性の高いものとするべくモジュールプログラミングが登場し、そのモジュール化プログラミングから更にクラスベースプログラミングという大きなパラダイムシフトが起こることとなります。モジュールからクラスというパラダイムが “ シフト ” したということに異論を唱える人がいるかもしれません、しかし、実務上、それらはたしかにシフトしています。シフトしているのですから、後発のパラダイムが優れているということであり、筆者もそう思います。構造化プログラミングの考え方はモジュールプログラミング、クラスベースプログラミングに内在する共通の枠組みです。したがって構造化プログラミングというパラダイムは当分生き続けることになるでしょう。
なお、クラスベースプログラミングをオブジェクト指向プログラミングと同一視する人がいますが、クラスベースプログラミングはオブジェクト指向プログラミングの一つの手段であり、コードをその書式的な側面から見たパラダイムです。コードの設計側面から見たオブジェクト指向とはそもそも次元が異なるものですし、当然同一でもありません。
よく、構造化プログラミングとオブジェクト指向プログラミングを同じ土俵で “ どちらが優れているのか ” という議論を見かけますが、それは、「CPU と PC は、どちらが優れているか」、あるいは、「エンジンと車は、どうちらが優れているのか」と言っているようなものです。書いていて赤面するほどナンセンスな文章で、議論の趣旨すら掴めません。“ あるもの ” と、そのあるものを ” 構成しているもの ” について、どちらが優れているか比較することは無意味であるし、そもそも不可能です。
構造化プログラミングとは、コンピューターがノイマン型である限り、すべてのプログラムパラダイムにとっての基盤となる基本理念的な概念であり、当然オブジェクト指向でコーディングする際にも基本となる概念です。おそらく、構造化プログラミング時代の “ プログラミング ” と比較しているのでしょう。そう考えても、もし、両者を書式や保守性といったコーディング局面から比べているのであれば、40 年前と現代のパラダイムを比較するのは少々不公平というものです。
そして、もし、両者を、その対象アプリケーションの設計アプローチから比較したとしても、処理を分割基準とする非オブジェクト指向と物（オブジェクト）を分割基準とするオブジェクト指向プログラミングを比較して、答えが出るとは思えません。議論で出ている答えは全て “ 言葉 ” であり、実際のコードで証明されているものは見たことがありません。現実に学術研究やビジネスアプリケーションの設計、コーディング、保守にはオブジェクト指向が有効であるということはなんとなく同意できますが、極限的高速性を要求するゲームとなると、オブジェクト指向が馴染むものなのか疑問とするところです。ということで、オブジェクト指向は本書では考えないでください。
しかし、クラス形式でコーディングすることは積極的に推奨します。というよりも、クラスベースでプログラミングしてください。なぜこのようなことを言うかというと、筆者は以前に、モジュールプログラミングとクラスベースプログラミングのどちらが優れているかということを真剣に考えたことがあります。それぞれの利点と欠点を列挙していく形で追及していくにつれ、次のことに気付いたのです、モジュールで出来ることはクラスでも出来る、クラスで出来ることはモジュールでも “ 概ね ” 出来る。つまり、クラスで出来てモジュールで出来ないことが存在するのです。両者の利点と欠点は、ほぼ同じものでしたが、クラスの方が “ 若干 ” 利便性が上でした。さすがは、後発のパラダイムだけのことはあります。したがって、積極的にモジュールプログラミングする理由は無く、クラス形式でコーディングしない理由はありません。
さて、構造化プログラミングから始まって、クラスベースプログラミングまで、長い年月を掛けて、コーディング及び設計手法は熟成されてきました。それらは、言うまでも無く「見易いコード」「保守性の高いコード」というものを共通の目的にしているわけですが、そのアプローチも “ ルーチンの部品化 ” という点では全く同じです。モジュールプログラミングにおいて、部品化は「モジュール化」「関数化」となり、クラスベースプログラミングでは「クラス化」というふうになります。
いまどき、初心者でも goto 文で処理を分岐させるということはしません、とりあえず、形としてはモジュール形式やクラス形式になっています、問題は、どこまでを部品として処理を分けるかというクオリティーの範疇です。筆者の感想としては、本節の最初で書いたように、えてして初心者はだらだらと長いルーチンにしてしまいがちです。もっと細かく部品化する必要があります。こう言うと、初心者はなんでもかんでも、細かく部品化してしまうことがよくあります、極端な話、一つの関数に一つのモジュール（実装ファイル）を用意してしまうなど。しかし、それは成長過程の一段階と考えられ、だらだら長いよりはましです。
さて、最後に結論を平たくまとめるとこうなります。
「コードはクラス形式で書く。部品化を意識して、意味が一区切りできるルーチンを部品化していく。部品化とはこの場合、クラス自体の分類と、クラス内のメソッド（関数）の分類のこと。なお、コードのアーキテクチャをオブジェクト指向にしたほうが良いのかどうかは（筆者には）分からない」

Chapter8　ゲームにはビジュアルが必要

ビジネスアプリケーションを開発するのとは違い、ゲーム開発の場合は、プログラムと同時にビジュアルも作成しなくてはなりません。今から 20 数年前、第 1 次パソコンブーム時代（マイコンブームとも呼ばれた時代）であれば、ゲームのビジュアルは全てドット絵と呼ばれるものでした。ドット絵がどのようなものかは、おそらくお分かりかと思いますが、念のため説明します。第 1 次パソコンブームよりも更に 4,5 年前のドット絵を例に挙げると分かり易いでしょう。インベーダーゲームや平安京エイリアンのビジュアルを頭に浮かべてください。キャラクターやその他オブジェクトの “ かくかく ” した形状、まさしく “ ドット（点）” から成る絵であり、誰でも方眼用紙と鉛筆さえあれば物の 30 分程度で書いてしまうような単純な絵です。これは、当時のコンピューターのメモリーやディスプレイの表示能力からくる制約という側面もあります。第 1 次パソコンブームの中期にはスキャナーから写真を読み込み作成したであろうと思われる画像がちらほらゲームで使用されたりもしましたが、主に背景やタイトルで使用されるもので、リアルタイムパートでは依然にドット絵が主流でした。筆者が高 3 から大学 2 年にかけてアルバイトをしていた札幌の某ゲームメーカーでも、ドット絵専門の方が毎日せっせとディスプレイにペン（ペンの先端からでる光を認識するデバイス、名称は忘れました）を当ててドット 1 点 1 点を打ち込んで絵を作成していたのを覚えています。現在のゲームにおけるビジュアルを当時の方法で作成するとなると気の遠くなるような作業量になりますが、当

時のゲームにおけるビジュアルはハード的にそれしかできなかったし、だからこそ、それで良かったのです。
その後、コンピューターの性能が向上するにつれ絵のディティールは向上しますが、３DCG やイラストソフトが普及するまでは基本的にドット絵であることには変わりはなく、色や絵が詳細になったというだけで、あくまでもボリュームの向上の領域です。業界には俗に“ドッター”と呼ばれるドット絵を描くアーティストが存在し、中にはとてもドット絵とは思えないほどのディティールを持った絵を世に生み出しましたアーティストも居ました。
そして、ようやく今から約 10 年前ごろから３DCG ソフトやイラストソフトで作られた絵がゲームで使用されるようになります。３DCG やイラストソフトで作られる絵とドット絵の違いは、その作成プロセスにあります。最終的な画像データはもちろん両者ともにピクセルデータなのでピクセルから成る絵をドット絵と定義してしまうと全てドット絵ということになってしまいます。
ドット絵の作成プロセスは、１ピクセル単位で描いていくもので、点画を書くように絵を描いていくものです。相当の根気と忍耐の勝負となります。
現在は、まず、大きく２種類のビジュアルが存在します。２D と３D です。ドット絵は２D の絵を描く手法ですが、現在は２D でもドット単位で描くなどということはぜずに、３D ソフトでレンダリングしたものを使用するのが効率及び仕上がりの面から一般的です。また、３D ゲームの場合は、もはや、絵ではなく３D ジオメトリそのものを作成し、ゲーム内ではそれを直接レンダリングするというビジュアルスタイルとなり、３D 全盛の今、こちらのビジュアルスタイルのほうが目にすることは多いでしょう。
本当のゲーム草創記であれば、先に述べたようにハード的な表現力が低いこともあり簡単なドット絵をビジュアルとし、ゲーム開発＝プログラミングと言えるものであり、ビジュアルは、いわば“おまけ”のようなものでした。見方を変えれば、簡単な絵で済んだので、ビジュアル作成は極めて簡単かつ短時間に行えるものでした。そして、コンピューターの表現力が向上するにつれ、ビジュアルのディティールを上げることが可能となる反面、ビジュアル作成に掛ける時間比率も増える結果となってきます。開発者としては、表現力が向上することは嬉しいことですが、反面、それなりに見栄えのするビジュアルを作成しなければならないことになります。
さて、読者のみなさんは、ビジュアルを作成することが難しいと思っているでしょうか？「昔ならともかく、現在の市販ゲームのようなビジュアルを自分で作成できるのだろうか」と思っている読者も少なくないと思います。いえいえ、それほど、難しいことはありません。たしかに、２D イラスト絵の場合は絵心とセンスが必要ですが、殆どのビジュアルは３DCG ソフトとペイント系ソフトを使えば、さほど難しいものではありません。
では、２D と３D のビジュアルに分けて、それぞれの作成プロセスを紹介します。

２D ビジュアル

ゲームで使用する絵はほとんど３DCG ソフトでレンダリングした絵を使用します。筆者が使用している３DCG ソフトは NewTek 社の LightWave です。LightWave は 4.0 から使い始め現在は 7.5 を使用しています。他に 3DStudioMAX,SoftImage, MAYA,Cinema4D などのソフトがありますが、ゲーム内で使用する静止画を作成するのであれば、出力結果は全く同じと言えます。あとは使い勝手の問題ですが、筆者は LightWave 以外使ったことがないので他のソフトに関しては何とも言えません。LightWave はファイルの管理が幾分独特であり、たまにファイルを上書きしてしまいますが、それは筆者の操作が悪い部分もあるのかもしれません。LightWave 等の３DCG ソフトでレンダリングした画像の多くは、ペイント系ソフトで加工して最終的な絵として仕上げます。筆者は Adobe 社の PhotoShop5.0 を使用しています。
ここでは、LightWave でレンダリングし、PhotoShop で仕上げる一例を紹介します。
図 8-1 は、添付サンプルゲーム CompanyWars2004（以降 CW2004 と表記）で使用している“お菓子”のモデルデータを LightWave に読み込んだ状態で、それをレンダリングしたものが図 8-1-2 です。

図 8-1

図 8-1-2

この時点での、解像度は 420x420 ピクセルです。ゲーム内では解像度 84x84 で使用しますので、解像度を編集し縮小します。多くのレンダリング画像は大きめにレンダリングしてから縮小して使用します。最初からゲーム内での解像度でレンダリングすると、きめが粗くなってしまうのでレンダリングは実際より大きめに行います。
解像度を最初から実寸（ゲーム内でのサイズ）でレンダリングしたものと、大きめにレンダリングしてから縮小したものの仕上がりを比較したものが図 8-2 です。

図 8-2
縮小する時には気をつけなければならないことがあります。縮小するとどうしても絵と背景に境界線を引いたような線が出てしまいます。これは縮小アルゴリズムによるものです。背景と絵の輪郭の境界がその中間色になってしまい、(背景が黒の場合)

黒に近い色ではあるが真の黒（RGB 値 00:00:00）ではない色が生成されてしまうためです。
図 8-3 は、CW2004 に出てくる対空ミサイルのレンダリング直後のイメージで、解像度は 600x600 ピクセルあります。ゲーム内では、それを縮小して使用しますが、縮小直後のイメージ図 8-3-2 を見てください。縮小直後でまだ修正を施してない状態のイメージには、絵に汚い輪郭のような線が出てしまっています。図 8-3-2 では、汚い輪郭を分り易いように白にしています。（実際は黒に近い色です）。それを、そのままゲームにしようすると図 8-3-3 のようになってしまいます。

図 8-3

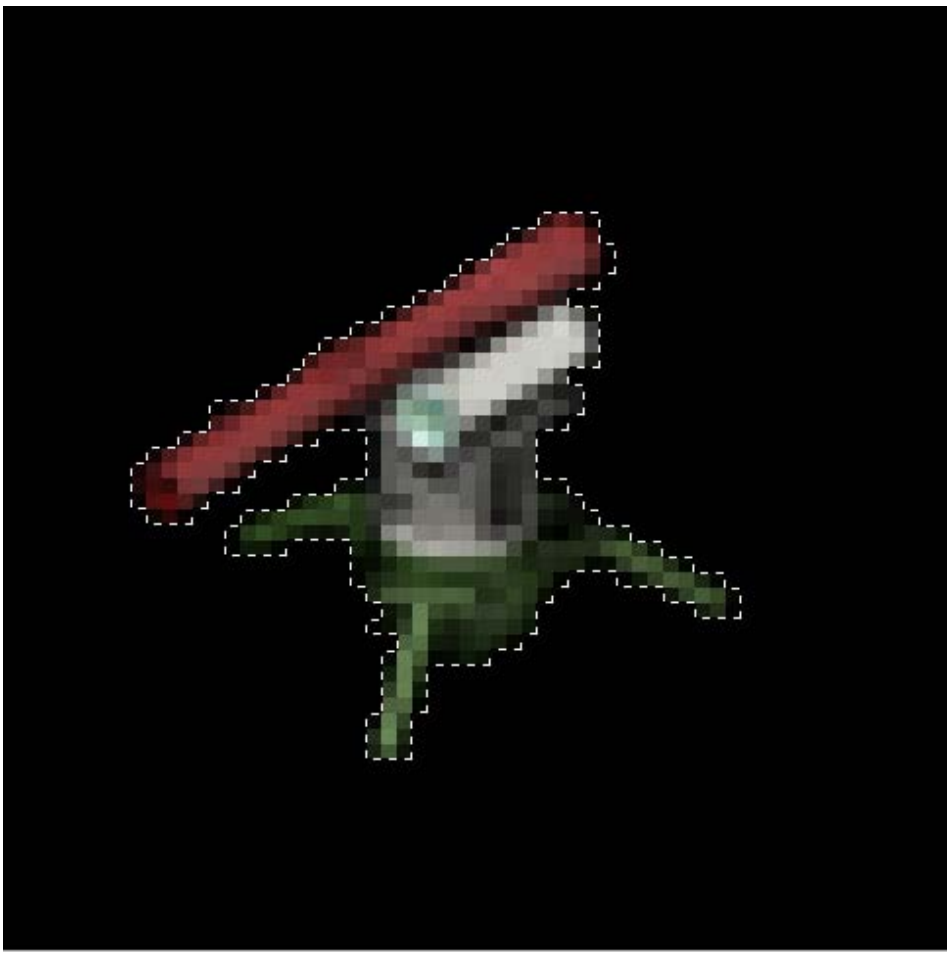


図 8-3-2

図 8-3-3

この離隔を縁取るように生成されてしまう中間色は実際にゲーム内で描画するとカラーキーと微妙に RGB 値が異なるためそのまま描画されてしまい（カラーキーが効かない）、黒っぽい（カラーキーに黒を指定した場合）離隔として残ってしまいます。まるで、絵が宙に浮いているように見え、絵を重ね塗りしていることがばればれになっていしまいます。この不要な線は

PhotoShop 等により手作業で消すか、あるいは、専用ツールを作る必要があります。付属ディスクにこのような線を消す目的で指定範囲の色を黒に変換する自作ツール「FadeToBlack.exe」を収録しましたので適宜使ってください。
不要な輪郭を、完全な黒、つまりカラーキー値にして、その部分を描画しないようにし、さらに、それに影などを付けると図 8-3-4 のように綺麗な仕上がりになり完璧です。

図 8-3-4

縮小すると、境界が中間色で補正されるので修正が必要ですが、時にはゲーム内での解像度そのものでレンダリングしてもさほど見栄えが変わらない場合があります。ゲーム内での解像度が小さい場合、例えば 20 ピクセル X20 ピクセルなどの解像度の場合は、大きめにレンダリングぜずに、20x20 でレンダリングした絵をそのまま使用してもかまわないときもあります。ただし、レンダリングはアンチエイリアスをかけないでレンダリングしてください。アンチエイリアスを効かせてレンダリングすると、これもまた境界の中間色が縁取りのようになってしまいます。アンチエイリアスが効いているとレンダリング出力自体が境界を中間色で補正されたものになってしまうので、そのままのサイズでレンダリングした意味がなくなってしまいます。アンチエイリアスは無効にしてレンダリングしてください。

3D ビジュアル

3D ベースゲームの場合、ピクセルデータである " 絵 " と、ジオメトリデータとしてのメッシュの作成 2 つがあります。背景、タイトル、文字、また、メッシュのテクスチャー用等にはピクセルデータ、3D 物体にはジオメトリデータを使用します。
2D と大きく違うところはデータ形式がピクセルデータではなくジオメトリデータというところです。ピクセルデータでアニメーションを実現する場合には、モーション毎の絵を連続表示する、いわゆる " ぱらぱらアニメ " なので、アニメーションの時間が長ければ長いほど多くの絵を必要とします。ジオメトリデータはピクセルデータのようにすでに完成されている静止画ではなく、ゲーム内で動的にレンダリングされるのでアニメーションさせる場合でもアニメーション時間に関わりなくジオメトリは 1 個で済みます。ジオメトリデータの作成は LightWave 等の 3DCG ソフトで作成します。
ここでは、LightWave でオブジェクト（3 DCG ソフトではジオメトリデータをオブジェクトと呼びます）を作成し、そのオブジェクトを Direct3D が読み込める X ファイルに変換するまでの一例を紹介します。
図 8-5 は、LightWave のモデラーでオブジェクトをモデリングしているところです。

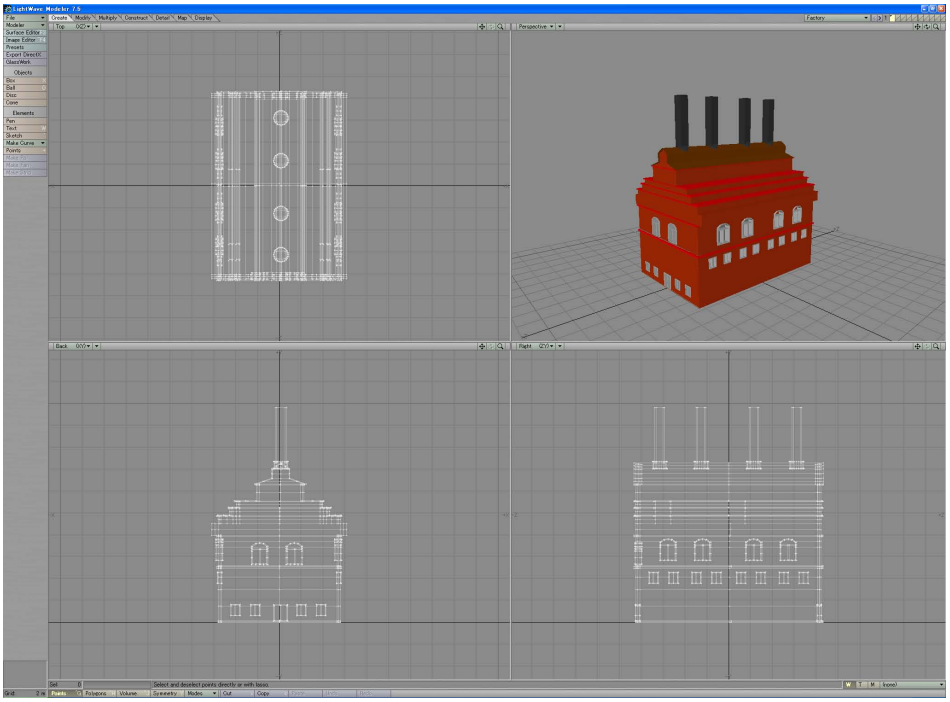


図 8-4

モデリングが終了したら、そのオブジェクトを Direct3D が読み込めるように X ファイルフォーマットに変換します。LightWave には DirectXExport という無料のプラグインがあるので、それを使用します。なお、筆者が作成したコンバーターも付属ディスクに収録しています、これは LightWave のオブジェクトファイル又はシーンファイルを読み込んで X ファイルを吐き出す（コンバートする）ものですので、適宜使用してください。このコンバーターは、もともと筆者が現在取り組んでいる 3D ベースゲームにおけるオリジナルフォーマットのジオメトリデータを作成・編集する目的で開発し始めて現在も開発中のものですが、現時点でも LightWave->DirectX コンバーターとしては利用できるので収録しました。コンバートできる 3DCG フォーマットは現在のところ LightWave だけです。

Chapter9 ゲームのセーブとロード

そのときプレイしているゲーム状態を保存（セーブ）しておいて、後日それを読み込み（ロード）、その続きからプレイできるという機能が多くのゲームに搭載されていて、便利な機能です。
ゲームのセーブとロード機能をコーディングすることは非常に簡単です。簡単すぎるので本章を設けることをためらいましたが、セーブロードの方法に対する需要が多いことと、簡単なことを知る機会がないということを防ぐべく、目次の見出しでも目立つように 1 つの章にしました。

セーブとロードの基本原理

セーブとロードは、構造的には全く同じ処理であり、ただデータの流れを逆にしただけのものと言えます。

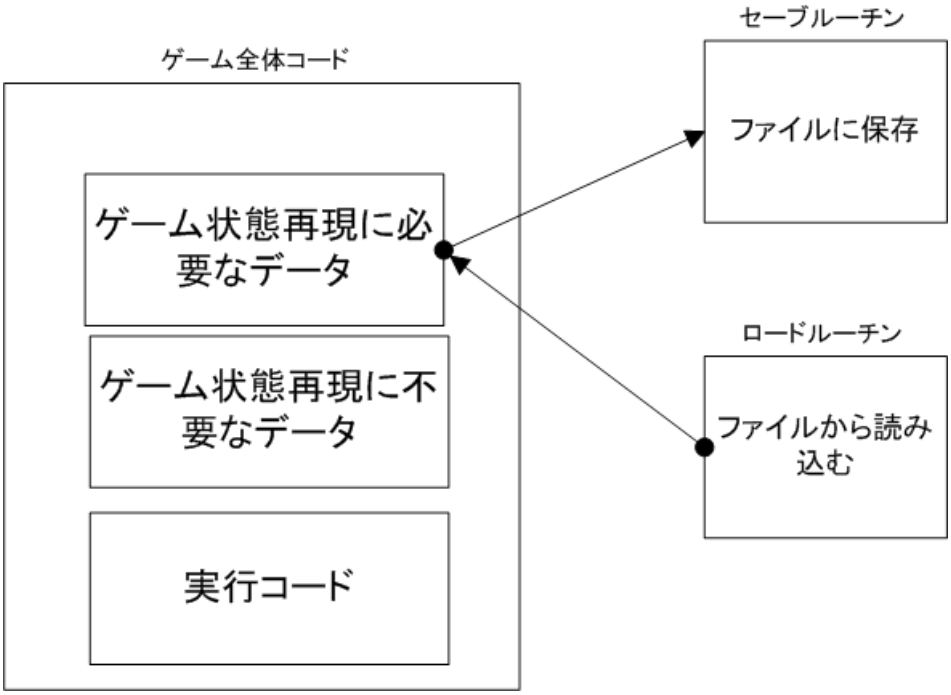


図 9-1

ゲームを途中から再開するのに必要なものは、セーブ時点での “ ゲーム状態に関連した変数・構造体など ” です。つまりゲーム再現に必要なデータです。これを、ファイルに保存すれば電源を切ったとしてもハードディスクに残るわけですから、あとでそれを読み込めばいいだけのことです。
ここで簡単なプログラムをゲームと見立てて実際のコードを示します。

```
 1: #include <stdio.h>
 2: #include <windows.h>
 3: #include <time.h>
 4: #include <conio.h>
 5:
 6: //
 7: //
 8: //
 9: #define MAX_ROUND 12
10: #define JAB '1'
11: #define FOOK '2'
12: #define STRAIGHT '3'
13: #define BLOCK '4'
14: #define GAMESAVE 's'
15: #define GAMELOAD 'l'
16: #define FORCE_JAB 10
17: #define FORCE_FOOK 20
18: #define FORCE_STRAIGHT 30
19: #define STAMINA 100
20:
21: struct GAME_DATA
22: {
23:         BYTE bRound;
24: };
25:
26: struct PLAYER_DATA
```

```
27: {
28:         BYTE bRound;
29:         INT iForce;
30:         INT iStamina;
31:         BYTE bAction;
32:         TCHAR szAction[MAX_PATH+1];
33: };
34:
35: GAME_DATA gd;
36: PLAYER_DATA pdMe,pdTyson;
```

```
 1: #include "Chapter9-1.h"
 2:
 3: //
 4: //VOID Save()
 5: // 現在のゲーム状態をファイルとして保存する
 6: VOID Save()
 7: {
 8:         FILE* fp;
 9:         fp=fopen("gamedata.dat","wb");
10:         fwrite(&gd,1,sizeof(gd),fp);
11:         fwrite(&pdMe,1,sizeof(pdMe),fp);
12:         fwrite(&pdTyson,1,sizeof(pdTyson),fp);
13:         fclose(fp);
14:         printf(" ゲームをセーブしました ");
15: }
16: //
17: //VOID Load()
18: // 保存してあるファイルからゲームデータを読み込む
19: VOID Load()
20: {
21:         FILE* fp;
22:         fp=fopen("gamedata.dat","rb");
23:         fread(&gd,1,sizeof(gd),fp);
24:         fread(&pdMe,1,sizeof(pdMe),fp);
25:         fread(&pdTyson,1,sizeof(pdTyson),fp);
26:         fclose(fp);
27:         printf(" ゲームをロードしました ");
28: }
29: //
30: //VOID Fight(PLAYER_DATA* ppdA,PLAYER_DATA* ppdB)
31: // 攻防計算。成功率はジャブ 50%、フック 30%、ストレート 20%
32: VOID Fight(PLAYER_DATA* ppdA,PLAYER_DATA* ppdB)
33: {
34:         INT iRnd=0;
35:         // 攻防計算
36:         ppdB->bAction != BLOCK ? iRnd=(rand()*100)/32768 : iRnd=100;
37:         switch(ppdA->bAction)
38:         {
39:                 case JAB:
40:                         if(iRnd<50)
41:                         {
42:                                 ppdB->iStamina-=FORCE_JAB;
43:                         }
44:                 break;
45:                 case FOOK:
46:                         if(iRnd<30)
47:                         {
48:                                 ppdB->iStamina-=FORCE_FOOK;
49:                         }
50:                 break;
51:                 case STRAIGHT:
52:                         if(iRnd<20)
53:                         {
54:                                 ppdB->iStamina-=FORCE_STRAIGHT;
55:                         }
56:                 break;
```

```
57:         }
58: }
59: //
60: //VOID main()
61: // エントリー関数　今回エントリー関数に殆どの処理を詰め込んだ
62: VOID main()
63: {
64:         // 構造体初期化
65:         ZeroMemory(&gd,sizeof(gd));
66:         ZeroMemory(&pdMe,sizeof(pdMe));
67:         ZeroMemory(&pdTyson,sizeof(pdTyson));
68:         pdMe.iStamina=pdTyson.iStamina=STAMINA;
69:         // 乱数初期化
70:         srand( (unsigned)time( NULL ) );
71:         // タイトル
72:         printf(" インライン　ボクシング　---M. タイソンに挑戦！ ---¥n");
73:         // ラウンド分だけループ
74:         while(gd.bRound<MAX_ROUND)
75:         {
76:                 gd.bRound++;
77:                 // ラウンド数と現在のスタミナ表示
78:                 printf("¥n********** %d　ラウンド **********¥n",gd.bRound);
79:                 printf(" あなたの体力：%d　タイソンの体力：%d¥n",pdMe.iStamina,pdTyson.iStamina);
80:                 // 自分のアクション
81:                 printf("¥n アクションを入力してください。");
82:                 printf("¥n( ジャブ：1　フック：2　ストレート：3 ブロック：4 セーブ：s　ロード：l)　");
83:                 pdMe.bAction=getchar();
84:                 while(getchar() != '¥n');
85:                 switch(pdMe.bAction)
86:                 {
87:                         case JAB:
88:                                 strcpy(pdMe.szAction," ジャブ ");
89:                         break;
90:                         case FOOK:
91:                                 strcpy(pdMe.szAction," フック ");
92:                         break;
93:                         case STRAIGHT:
94:                                 strcpy(pdMe.szAction," ストレート ");
95:                         break;
96:                         case BLOCK:
97:                                 strcpy(pdMe.szAction," ブロック ");
98:                         break;
99:                         case GAMESAVE:
100:                                Save();
101:                                gd.bRound--;
102:                                continue;
103:                        break;
104:                        case GAMELOAD:
105:                                Load();
106:                        break;
107:                }
108:                // タイソンのアクション ジャブ 30% フック 20% ストレート 20% ブロック 30%
109:                INT iRnd=(rand()*100)/32768;
110:                if(iRnd<30)
111:                {
112:                        pdTyson.bAction=JAB;
113:                        strcpy(pdTyson.szAction," ジャブ ");
114:                }
115:                else if(iRnd<50)
116:                {
117:                        pdTyson.bAction=FOOK;
118:                        strcpy(pdTyson.szAction," フック ");
119:                }
120:                else if(iRnd<70)
121:                {
122:                        pdTyson.bAction=STRAIGHT;
123:                        strcpy(pdTyson.szAction," ストレート ");
124:                }
125:                else
126:                {
```

```
127:                            pdTyson.bAction=BLOCK;
128:                            strcpy(pdTyson.szAction," ブロック ");
129:                    }
130:                    // 自分のスタミナ計算
131:                    Fight(&pdMe,&pdTyson);
132:                    // タイソンのスタミナ計算
133:                    Fight(&pdTyson,&pdMe);
134:                    // 結果表示
135:                    printf("¥n¥n あなたのアクション : %s  タイソンのアクション : %s ¥n",pdMe.szAction,pdTyson.szAction);
136:                    //" KO" 勝敗チェック
137:                    if((pdMe.iStamina<=0) && (pdTyson.iStamina<=0))
138:                    {
139:                            printf("========= 引き分けです。=========¥n");
140:                            break;
141:                    }
142:                    else if(pdTyson.iStamina<=0)
143:                    {
144:                            printf("========= あなたの KO 勝ちです。=========¥n");
145:                            break;
146:                    }
147:                    else if(pdMe.iStamina<=0)
148:                    {
149:                            printf("========= あなたの      KO 負けです。=========¥n");
150:                            break;
151:                    }
152:                    else if(gd.bRound == MAX_ROUND)
153:                    {
154:                            // " 判定" 勝敗
155:                            if(pdMe.iStamina>pdTyson.iStamina)
156:                            {
157:                                    printf("========= あなたの判定勝ちです。=========¥n");
158:                            }
159:                            else if(pdMe.iStamina<pdTyson.iStamina)
160:                            {
161:                                    printf("========= あなたの判定負けです。=========¥n");
162:                            }
163:                            else
164:                            {
165:                                    printf("========= 判定で引き分けました。=========¥n");
166:                            }
167:                    }
168:            }
169:            printf("¥n 終了。何かキーを押してください。");
170:            while(!kbhit());
171: }
```

このプログラムは、コンソール・テキストベースで「ボクシング」の試合を行うものです、構想 1 時間、製作 5 時間の超大作（笑）です。本章の目的はセーブロードなので、ゲーム部分をいかに短くかつゲームとして成り立つ最低ラインを意識しつつ考えた末、このようなテキストベースのボクシングになったわけでが、ゲームの内容はここでは重要ではなく、着眼すべき部分はセーブとロードルーチンだけです。

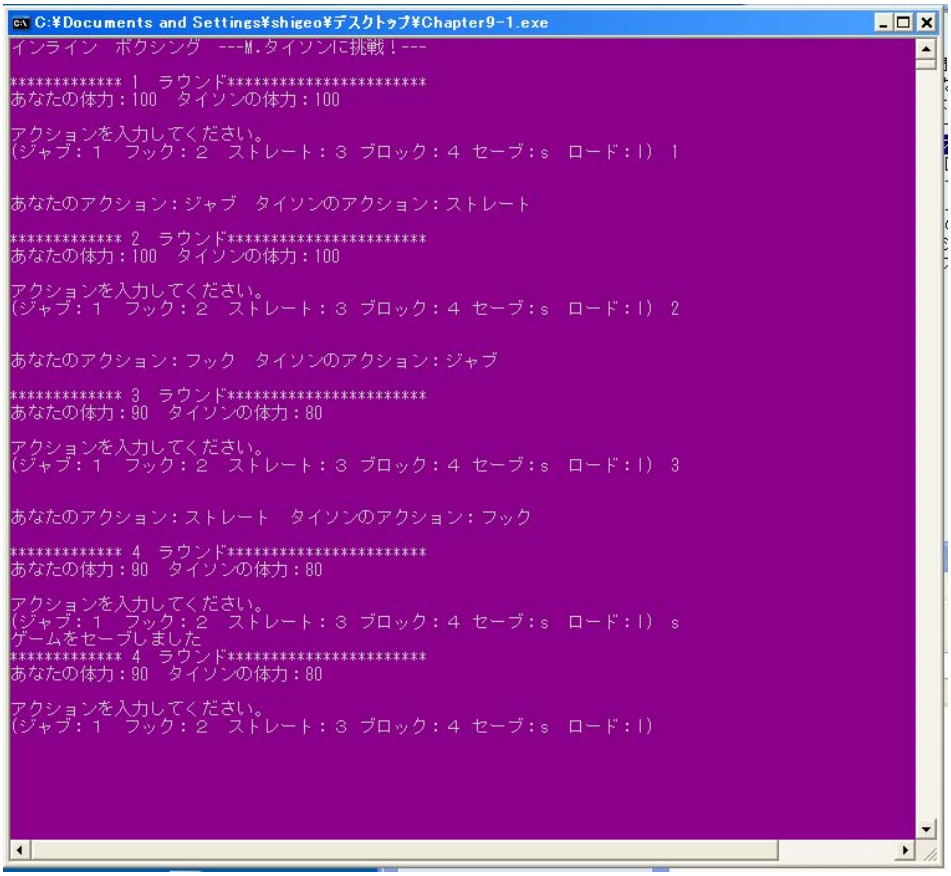


図 9-2

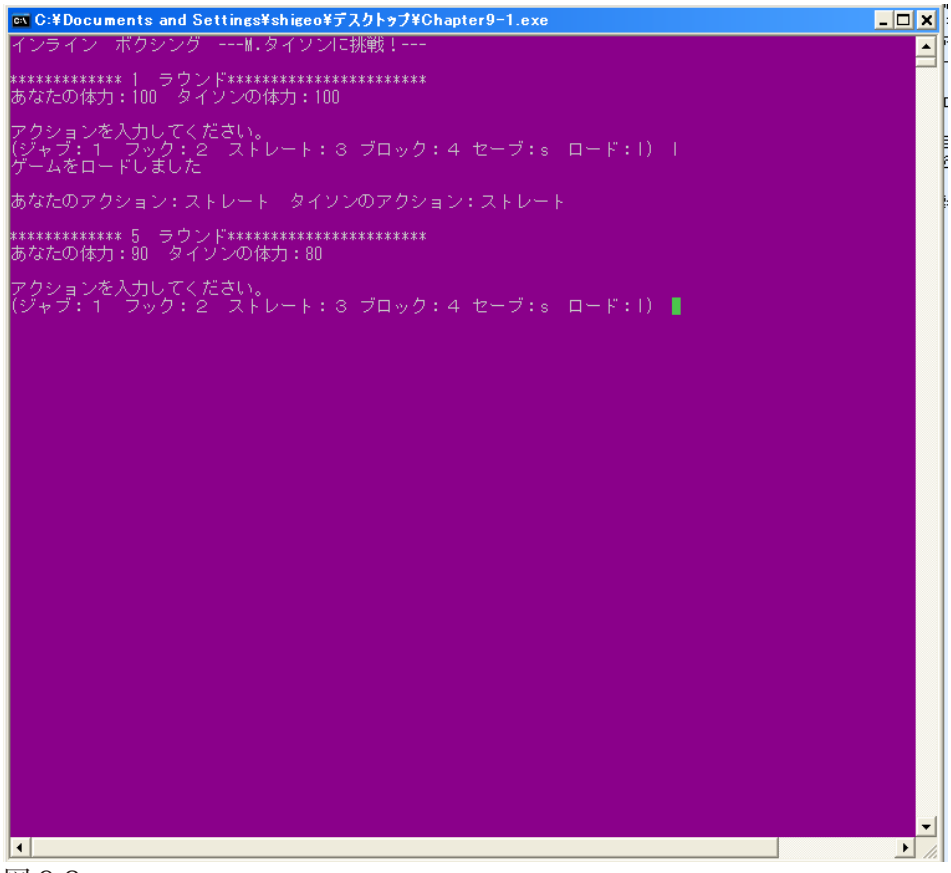


図 9-3

6 行目～ 28 行目がセーブとロードルーチンであり、この部分の処理が本章のテーマです。その他の部分は、そのプロジェクトによって当然変わるわけですが、セーブとロードルーチンにおけるその基本原理は変わりません。
このゲームでは、ヘッダーファイルでマクロ定義と構造体定義を定義しています。構造体は 2 種類で、1 つはプレイヤー共通

のゲームデータである GAME_DATA 型、もう 1 つはプレイヤー毎のデータ用として PLAYER_DATA 型です。GAME_DATA はラウンド数を格納し、PLAYER_DATA はプレイヤー情報を格納します。PLAYER_DATA 型のインスタンスは自分と " タイソン（コンピュータープレイヤー）" 用にそれぞれ 1 個ずつ計 2 個作成するのでゲーム全体での構造体インスタンスは 3 つになります。
まず、セーブルーチンで行っている処理から解説します。6 〜 14 行目の Save 関数を見てください。Save 関数で行っている処理は、次のとおりです。
ファイルを書き込み専用でオープンし、書き込みの準備をしてから、fwrite によりデータをファイルに書き込みます。
これだけです。簡単ですね。
次にロードルーチンを見てみましょう。ロードは Load() 関数で行っています。Load() 関数で行っている処理は、"Save 関数の逆 " です。
ファイルを読み込み専用でオープンして、fread によりデータをファイルから読み込みます。
セーブルーチンを作ってしまえば、ロードルーチンの作成は機械的に行えます。セーブルーチンのコードをコピー＆ペーストし、ファイルオープンを書き込みから読み込みに変更（fopen(…,"wb") から fopen(…,"rb")）し、fwrite を fRead に置換すればいいのです。コピー＆ペーストでらくらく作成できてしまいます。変な話、別々に書くよりコピー＆ペーストの方が、間違いがありません。なぜかというと、Save 関数でファイルに書き込む順番どおりに読み込むコードが出来るからです。書き込みと読み込みのデータの順番は、全く同じでなくてはなりません。順番が違うとクラッシュあるいは原因不明の不具合が発生することとなります。
もちろん、ロードルーチンを先に書いた場合も、コピー＆ペーストしてからデータの流れが逆になるように書き換えればセーブルーチンができます。ようするに、どちらか一方を作成すればもう一方は機械的に作成できます。

Chapter10「常にこちらを向く板」、ビルボード

ビルボードとは

ポリゴンやメッシュをレンダリングする時、通常はカメラの姿勢に対応してスクリーン座標上を動きますが、場合によってはカメラの姿勢に反応させない、反応して欲しくないときがあります。カメラの平行移動は通常通り適用してカメラの回転は適用しない、あるいはカメラの平行移動、回転両方とも適用したくない場合があります。カメラの姿勢に影響を受けないということは、言い換えると、視点がどうであろうと常に一定の方向を向くようになるということであり、カメラの平行移動にさえ影響させないようにした場合は、ジオメトリは常に一定の方向、一定の位置に留まることとなります。このように、スクリーン上で一定の方向や一定の位置あるいはその両方を実現することをビルボーディングと呼び、そのジオメトリをビルボードとも呼びます。ビルボード（Billboard）とは、同名のアメリカ音楽チャートのことではなく、本来は「路上（宣伝）広告（看板）」という意味です。3DCG におけるビルボードはこの本来の意味における " 看板 " を思わせることからこのような名称で広く呼ばれています。

ビルボードにする理由

どのような場合にこのような要求が出るのか例を列挙すると次のようになります。

カメラの回転だけに反応させたくない場合
雲、煙、炎　等を、板ポリゴンを利用してレンダリングしたい時。
カメラの平行移動と回転両方に反応させたくない場合
ポリゴンを完全な 2D イメージとしてレンダリングしたい時。

ではどうしてこれらの場合にカメラの姿勢の影響を受けないようにしたいのか、言い換えると、どうしてビルボードにしたいのか、その理由は次のとおりです。
雲などの、非常に不規則で粒子の細かい物体をジオメトリ的に表現するとなると、膨大な頂点、ポリゴンを用意しなければなりません。それは、レンダリングコストを考えると非現実的であり、不可能であるとさえ言えます。ちなみに、3DCG ソフト（LightWave 等）で雲などをジオメトリ的にレンダリングするとコーヒーを一杯入れられるだけの時間は軽くかかります。リアルタイムレンダリングである Direct3D で、非現実なのは明らかです（毎秒 0.001 フレーム 0.001fps のゲームを想像してみてください）。雲などの物体は、もっとも単純な板ポリゴン（3 頂点や 4 頂点）に雲などの絵をテクスチャーとして貼り付けて、" それらしく " 見せます。その際、通常のようにカメラの姿勢変化に反応させて板ポリゴンが移動・回転 " してしまうと " それが板ポリゴンであることが分かってしまい、演出は台無しです。そこで、板ポリゴンにカメラの姿勢の影響が及ばないようにすれば板ポリゴンを常にこちらに向かせるようにすることが可能となり、雲が雲らしく見えます。なぜなら、常にこちらを向いていれば見ている側からは、それが単なる板ポリゴンだろうが完全なジオメトリ（まず有り得ないが）だろうが同じように見えます。
このように、ビルボードは、レンダリングコストが非常に大きい特殊なジオメトリを高速にレンダリングするトリックです。

ビルボーディング方法

ビルボーディングの方法はいつくかありますが、最もシンプルな方法を紹介します。それは、ビルボーディングをかけたいジオメトリに対して、「ビュートランスフォームを打ち消す処理をする」というもので、具体的には、その時点においてビュートランスフォームで使用しているビュー行列の逆行列を、そのジオメトリ用のワールド行列に掛けるという極めて簡単なものです。ビュー行列の逆行列は、ビュートランスフォームを打ち消す（効果を無効にする）機能があります。それを、ビルボードにしたいジオメトリ用に計算したワールド行列に掛けるので、他の非ビルボードジオメトリに影響はありません。
では、実際のコードを見ながらビルボーディング技法を見ていきましょう。

```
 1:
 2: #include <windows.h>
 3: #include <d3dx9.h>
 4:
 5: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 6: #define THING_AMOUNT 3+1
 7:
 8:
 9: struct THING
10: {
11:         LPD3DXMESH pMesh;
12:         D3DMATERIAL9* pMeshMaterials;
13:         LPDIRECT3DTEXTURE9* pMeshTextures ;
14:         DWORD dwNumMaterials;
15:         D3DXVECTOR3 vecPosition;
16:         THING()
17:         {
18:                         ZeroMemory(this,sizeof(THING));
19:         }
20: };
21:
22: LPDIRECT3D9 pD3d;
23: LPDIRECT3DDEVICE9 pDevice;
24: FLOAT fCameraX=0,fCameraY=1.0f,fCameraZ=-3.0f,
25:                                 fCameraHeading=0,fCameraPitch=0;
26:
27: THING Thing[THING_AMOUNT];
28: THING Billboard;
29: BOOL boBillboading=FALSE;
30:
31: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
32: HRESULT InitD3d(HWND);
33: HRESULT InitThing(THING *,LPSTR,D3DXVECTOR3*);
34: VOID NormalTransform(THING* pThing);
35: VOID BillboardingTransform(THING* pThing);
36: VOID Render();
37: VOID RenderThing(THING*);
38: VOID FreeDx();
39:
40: //
41: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
42: // アプリケーションのエントリー関数
43: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
44: {
45:         HWND hWnd=NULL;
46:         MSG msg;
47:         // ウィンドウの初期化
48:         static char szAppName[] = "Chapter10" ;
49:       WNDCLASSEX  wndclass ;
50:
51:       wndclass.cbSize            = sizeof (wndclass) ;
52:       wndclass.style             = CS_HREDRAW | CS_VREDRAW ;
53:       wndclass.lpfnWndProc    = WndProc ;
54:       wndclass.cbClsExtra      = 0 ;
55:       wndclass.cbWndExtra     = 0 ;
56:       wndclass.hInstance         = hInst ;
57:       wndclass.hIcon             = LoadIcon (NULL, IDI_APPLICATION) ;
58:       wndclass.hCursor           = LoadCursor (NULL, IDC_ARROW) ;
59:       wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
60:       wndclass.lpszMenuName = NULL ;
61:       wndclass.lpszClassName = szAppName ;
62:       wndclass.hIconSm          = LoadIcon (NULL, IDI_APPLICATION) ;
63:
64:       RegisterClassEx (&wndclass) ;
65:
66:       hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
67:                   0,0,800,600,NULL,NULL,hInst,NULL) ;
68:
```

```
69:     ShowWindow (hWnd,SW_SHOW) ;
70:     UpdateWindow (hWnd) ;
71:       SetWindowText(hWnd," スペースキーでビルボーディングをトグルします。 ");
72:       // ダイレクト3D の初期化関数を呼ぶ
73:       if(FAILED(InitD3d(hWnd)))
74:       {
75:                 return 0;
76:       }
77:       // メッセージループ
78:     ZeroMemory( &msg, sizeof(msg) );
79:     while( msg.message!=WM_QUIT )
80:      {
81:                  if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
82:        {
83:                           TranslateMessage( &msg );
84:            DispatchMessage( &msg );
85:        }
86:        else
87:                 {
88:                           Render();
89:                 }
90:       }
91:       // メッセージループから抜けたらオブジェクトを全て開放する
92:       FreeDx();
93:       // OS に戻る （アプリケーションを終了する）
94:     return (INT)msg.wParam ;
95: }
96:
97: //
98: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
99: // ウィンドウプロシージャ関数
100: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
101: {
102:      switch(iMsg)
103:      {
104:                 case WM_DESTROY:
105:                           PostQuitMessage(0);
106:                 break;
107:                 case WM_KEYDOWN:
108:                 switch((CHAR)wParam)
109:                 {
110:                           case VK_ESCAPE:
111:                                     PostQuitMessage(0);
112:                           break;
113:                           //ビルボーディング  オン、オフ  トグル
114:                           case VK_SPACE:
115:                                     boBillboading== FALSE ? boBillboading=TRUE : boBillboading=FALSE;
116:                                     break;
117:                           // カメラの移動
118:                           case 'A':
119:                                     fCameraX-=0.1f;
120:                           break;
121:                           case 'D':
122:                                     fCameraX+=0.1f;
123:                           break;
124:                           case 'Q':
125:                                     fCameraY-=0.1f;
126:                           break;
127:                           case 'E':
128:                                     fCameraY+=0.1f;
129:                           break;
130:                           case 'W':
131:                                     fCameraZ-=0.1f;
132:                           break;
133:                           case 'C':
134:                                     fCameraZ+=0.1f;
135:                           break;
136:                           // カメラの回転
137:                           case VK_LEFT:
138:                                     fCameraHeading-=0.1f;
```

```
139:                                    break;
140:                            case VK_RIGHT:
141:                                    fCameraHeading+=0.1f;
142:                            break;
143:                            case VK_UP:
144:                                    fCameraPitch-=0.1f;
145:                            break;
146:                            case VK_DOWN:
147:                                    fCameraPitch+=0.1f;
148:                            break;
149:                    }
150:                    break;
151:    }
152:    return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
153: }
154:
155: //
156: //HRESULT InitD3d(HWND hWnd)
157: // ダイレクト 3D の初期化関数
158: HRESULT InitD3d(HWND hWnd)
159: {
160:    //「Direct3D」オブジェクトの作成
161:    if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
162:    {
163:            MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
164:     return E_FAIL;
165:    }
166:    //「DIRECT3D デバイス」オブジェクトの作成
167:    D3DPRESENT_PARAMETERS d3dpp;
168:    ZeroMemory( &d3dpp, sizeof(d3dpp) );
169:    d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
170:    d3dpp.BackBufferCount=1;
171:    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
172:    d3dpp.Windowed = TRUE;
173:   d3dpp.EnableAutoDepthStencil = TRUE;
174:   d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
175:
176:     if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
177:                                     D3DCREATE_MIXED_VERTEXPROCESSING,
178:                                     &d3dpp, &pDevice ) ) )
179:   {
180:            MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
181:            if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
182:                                     D3DCREATE_MIXED_VERTEXPROCESSING,
183:                                     &d3dpp, &pDevice ) ) )
184:            {
185:                            MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
186:                             return E_FAIL;
187:            }
188:   }
189:    // X ファイル毎にメッシュを作成する
190:    InitThing(&Thing[0],"Ground.x",&D3DXVECTOR3(0,-0.5,0));
191:    InitThing(&Thing[1],"Can.x",&D3DXVECTOR3(-1,0,-0.5));
192:    InitThing(&Thing[2],"Bottle.x",&D3DXVECTOR3(1,0,1));
193:    InitThing(&Billboard,"Billboard.x",&D3DXVECTOR3(0,0,3));
194:    // カリングはしない
195:    pDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
196:    // Z バッファー処理を有効にする
197:   pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
198:    // ライトを有効にする
199:    pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
200:    // アンビエントライト（環境光）を設定する
201:    pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
202:    // スペキュラ（鏡面反射）を有効にする
203:    pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
204:    // ライトをあてる 白色で鏡面反射ありに設定
205:    D3DXVECTOR3 vecDirection(1,1,1);
206:   D3DLIGHT9 light;
207:   ZeroMemory( &light, sizeof(D3DLIGHT9) );
208:   light.Type       = D3DLIGHT_DIRECTIONAL;
```

```
209:     light.Diffuse.r  = 1.0f;
210:     light.Diffuse.g  = 1.0f;
211:     light.Diffuse.b  = 1.0f;
212:       light.Specular.r=1.0f;
213:       light.Specular.g=1.0f;
214:       light.Specular.b=1.0f;
215:     D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
216:     light.Range        = 200.0f;
217:     pDevice->SetLight( 0, &light );
218:     pDevice->LightEnable( 0, TRUE );
219:       return S_OK;
220: }
221:
222: //
223: //
224: //
225: HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
226: {
227:       // メッシュの初期位置
228:       memcpy(&pThing->vecPosition,pvecPosition,sizeof(D3DXVECTOR3));
229:       // X ファイルからメッシュをロードする
230:       LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
231:
232:       if( FAILED( D3DXLoadMeshFromX( szXFileName, D3DXMESH_SYSTEMMEM,
233:             pDevice, NULL, &pD3DXMtrlBuffer, NULL,
234:                               &pThing->dwNumMaterials, &pThing->pMesh ) ) )
235:     {
236:             MessageBox(NULL, "X ファイルの読み込みに失敗しました ",szXFileName, MB_OK);
237:             return E_FAIL;
238:     }
239:       D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
240:       pThing->pMeshMaterials = new D3DMATERIAL9[pThing->dwNumMaterials];
241:     pThing->pMeshTextures  = new LPDIRECT3DTEXTURE9[pThing->dwNumMaterials];
242:
243:       for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
244:       {
245:                     pThing->pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
246:         pThing->pMeshMaterials[i].Ambient = pThing->pMeshMaterials[i].Diffuse;
247:         pThing->pMeshTextures[i] = NULL;
248:         if( d3dxMaterials[i].pTextureFilename != NULL &&
249:             lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
250:         {
251:             if( FAILED( D3DXCreateTextureFromFile( pDevice,
252:                                         d3dxMaterials[i].pTextureFilename,
253:                                         &pThing->pMeshTextures[i] ) ) )
254:             {
255:                 MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
256:                               }
257:         }
258:     }
259:       pD3DXMtrlBuffer->Release();
260:
261:       return S_OK;
262: }
263:
264: //
265: //VOID Render()
266: //X ファイルから読み込んだメッシュをレンダリングする関数
267: VOID Render()
268: {
269:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
270:                     D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
271:
272:       if( SUCCEEDED( pDevice->BeginScene() ) )
273:       {
274:                       // その他メッシュを通常レンダリング
275:                     for(DWORD i=0;i<THING_AMOUNT;i++)
276:                     {
277:                                   NormalTransform(&Thing[i]);
278:                                   RenderThing(&Thing[i]);
```

```
279:                    }
280:                    if(boBillboading)
281:                    {
282:                            // ボードメッシュをビルボードレンダリング
283:                            BillboardingTransform(&Billboard);
284:                            RenderThing(&Billboard);
285:                    }
286:                    else
287:                    {
288:                            //  ボードメッシュを通常レンダリング
289:                            NormalTransform(&Billboard);
290:                            RenderThing(&Billboard);
291:                    }
292:
293:
294:                    pDevice->EndScene();
295:            }
296:            pDevice->Present( NULL, NULL, NULL, NULL );
297: }
298: //
299: //
300: //
301: VOID NormalTransform(THING* pThing)
302: {
303:            // ワールドトランスフォーム（絶対座標変換）
304:            D3DXMATRIXA16 matWorld,matPosition;
305:            D3DXMatrixIdentity(&matWorld);
306:            D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,
307:                    pThing->vecPosition.z);
308:            D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
309:          pDevice->SetTransform( D3DTS_WORLD, &matWorld );
310:
311: }
312: //
313: //
314: //
315: VOID BillboardingTransform(THING* pThing)
316: {
317:            // ワールドトランスフォーム（絶対座標変換）
318:            D3DXMATRIXA16 matWorld,matCurrentView,matPosition;
319:            D3DXMatrixIdentity(&matWorld);
320:            D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,
321:                    pThing->vecPosition.z);
322:            D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
323:            // 現在のビュー行列を得て、、
324:            pDevice->GetTransform(D3DTS_VIEW,&matCurrentView);
325:            // それを逆行列にして、、
326:            D3DXMatrixInverse(&matCurrentView,NULL,&matCurrentView);
327:            // ワールド行列に掛け合わせると、ビュー変換を打ち消すことになる
328:            D3DXMatrixMultiply(&matWorld,&matWorld,&matCurrentView);
329:
330:          pDevice->SetTransform( D3DTS_WORLD, &matWorld );
331:
332: }
333: //
334: //
335: //
336: VOID RenderThing(THING* pThing)
337: {
338:            // ビュートランスフォーム（視点座標変換）
339:            D3DXMATRIXA16 matView,matCameraPosition,matHeading,matPitch;
340:          D3DXVECTOR3 vecEyePt( fCameraX,fCameraY,fCameraZ ); // カメラ（視点）位置
341:          D3DXVECTOR3 vecLookatPt( fCameraX,fCameraY-1.0f,fCameraZ+3.0f );// 注視位置
342:          D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
343:            D3DXMatrixIdentity(&matView);
344:            D3DXMatrixRotationY(&matHeading,fCameraHeading);
345:            D3DXMatrixRotationX(&matPitch,fCameraPitch);
346:            D3DXMatrixLookAtLH( &matCameraPosition, &vecEyePt, &vecLookatPt, &vecUpVec );
347:            D3DXMatrixMultiply(&matView,&matView,&matHeading);
348:            D3DXMatrixMultiply(&matView,&matView,&matPitch);
```

```
349:        D3DXMatrixMultiply(&matView,&matView,&matCameraPosition);
350:      pDevice->SetTransform( D3DTS_VIEW, &matView );
351:        // プロジェクショントランスフォーム（射影変換）
352:      D3DXMATRIXA16 matProj;
353:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
354:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
355:        // レンダリング
356:        for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
357:      {
358:                        pDevice->SetMaterial( &pThing->pMeshMaterials[i] );
359:          pDevice->SetTexture( 0,pThing->pMeshTextures[i] );
360:          pThing->pMesh->DrawSubset( i );
361:      }
362: }
363:
364: //
365: //VOID FreeDx()
366: // 作成した DirectX オブジェクトの開放
367: VOID FreeDx()
368: {
369:        for(DWORD i=0;i<THING_AMOUNT;i++)
370:        {
371:                        SAFE_RELEASE( Thing[i].pMesh );
372:        }
373:        SAFE_RELEASE( pDevice );
374:        SAFE_RELEASE( pD3d );
375: }
```

着目すべき部分は、BillboardingTransform() 関数だけですが、対比のために NormalTransform() 関数も眺めてみてください。NormalTranform() 関数は、Chapter5-x のコードと基本的に同じです。BillboardingTransform() 関数はそれに、ビュー行列を打ち消す処理を加えているだけです。

pDevice->GetTransform(D3DTS_VIEW,&matCurrentView);

まず、現在レンダリングパイプラインに登録されているビュー行列を取得します。

D3DXMatrixInverse(&matCurrentView,NULL,&matCurrentView);

現在のビュー行列を直接逆行列にします。

D3DXMatrixMultiply(&matWorld,&matWorld,&matCurrentView);

その逆行列をビルボードジオメトリのワールド行列に掛けます。

これで、このジオメトリはビュートランスフォームの影響を受けません。

厳密に言うと “ 影響を受けた結果が現在と同じになる ” のです。ここで言う “ 打ち消す ” 処理は、まず前もってビュートランスフォームの影響量の逆だけ動かしておいて、そのあと影響を受けると丁度相殺されるという仕組みです。

図にすると次のとおりです。

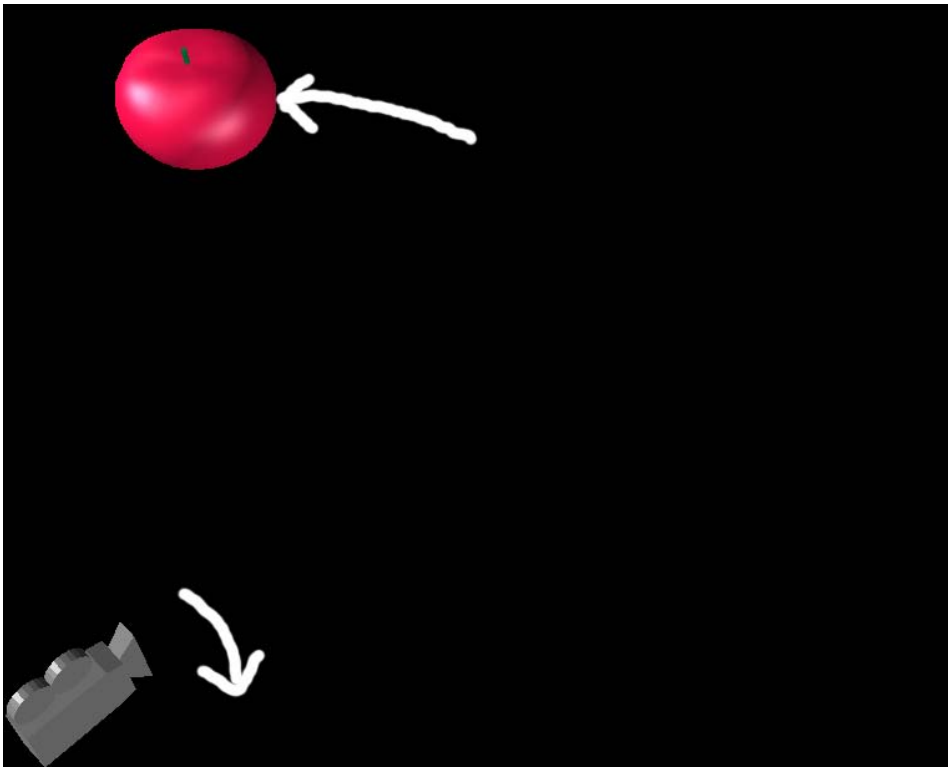


図 10-1
通常のジオメトリはこのように、カメラが向こうとする方向と逆方向に影響を受けますが、

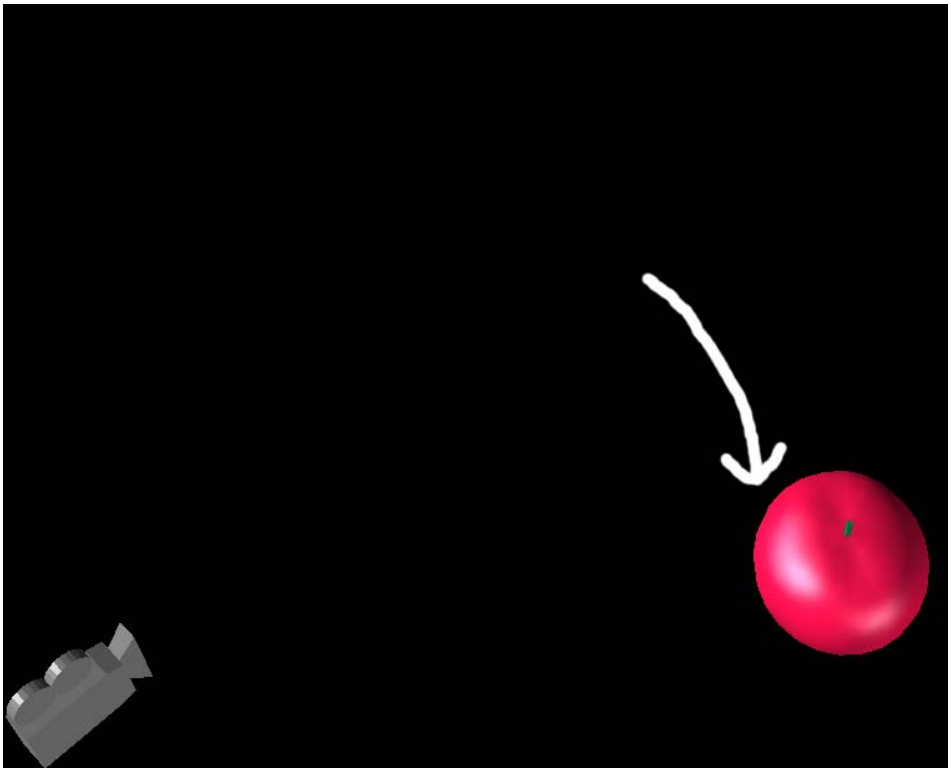


図 10-2

ビルボードの場合、まず最初に、カメラの影響と逆方向に前もって移動しておきます。
この処理をコードで表現したものが、BillboardingTransform() 関数の追加部分です。
matWorld 行列は、この図 10-2 における逆方向への増加分を保持しているのです。

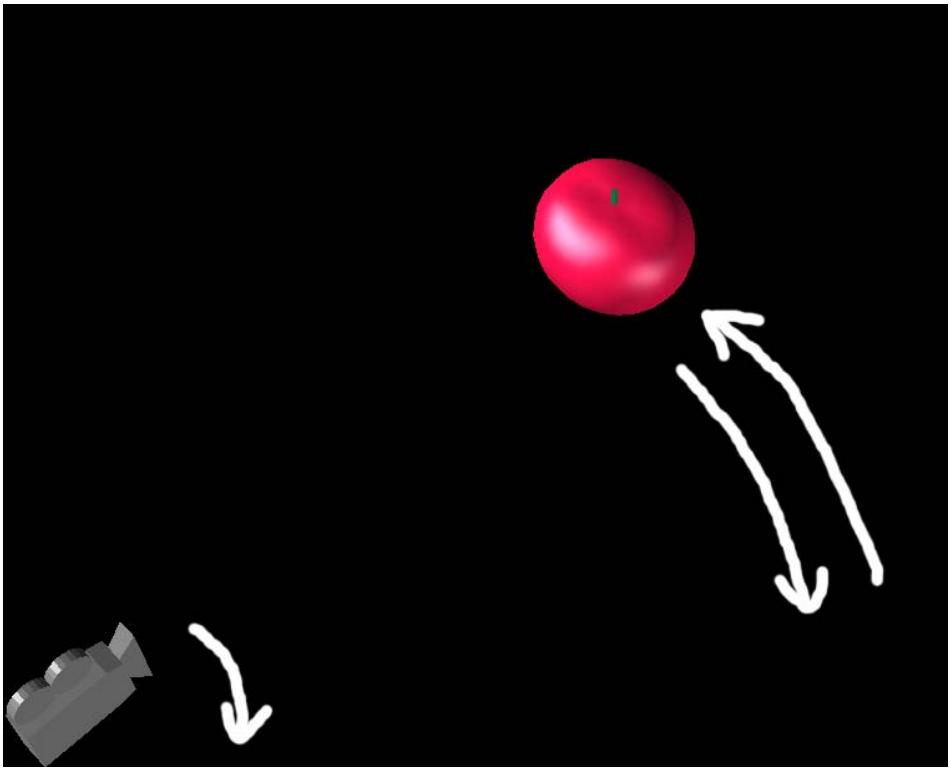


図 10-3

そのあと、カメラの回転の影響を受けても、ちょうど元に戻ることになり、結果的に動きません。

任意の座標を中心として円運動をさせる

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5: #define THING_AMOUNT 3+1
 6:
 7:
 8: struct THING
 9: {
10:         LPD3DXMESH pMesh;
11:         D3DMATERIAL9* pMeshMaterials;
12:         LPDIRECT3DTEXTURE9* pMeshTextures ;
13:         DWORD dwNumMaterials;
14:         D3DXVECTOR3 vecPosition;
15:         D3DXMATRIX matRotation;
16:         THING()
17:         {
18:                         ZeroMemory(this,sizeof(THING));
19:         }
20: };
21:
22: LPDIRECT3D9 pD3d;
23: LPDIRECT3DDEVICE9 pDevice;
24:
25: THING Thing[THING_AMOUNT];
26:
27: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
28: HRESULT InitD3d(HWND);
29: HRESULT InitThing(THING *,LPSTR,D3DXVECTOR3*);
30: VOID Render();
31: VOID RenderThing(THING*);
32: VOID RotaryMove(THING*,D3DXVECTOR3*,FLOAT);
33: VOID FreeDx();
34:
35: //
36: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
37: // アプリケーションのエントリー関数
38: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
39: {
40:         HWND hWnd=NULL;
41:         MSG msg;
42:         // ウィンドウの初期化
43:         static char szAppName[] = "Chapter11" ;
44:       WNDCLASSEX  wndclass ;
45:
46:       wndclass.cbSize            = sizeof (wndclass) ;
47:       wndclass.style             = CS_HREDRAW | CS_VREDRAW ;
48:       wndclass.lpfnWndProc     = WndProc ;
49:       wndclass.cbClsExtra       = 0 ;
50:       wndclass.cbWndExtra      = 0 ;
51:       wndclass.hInstance         = hInst ;
52:       wndclass.hIcon             = LoadIcon (NULL, IDI_APPLICATION) ;
53:       wndclass.hCursor           = LoadCursor (NULL, IDC_ARROW) ;
54:       wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
55:       wndclass.lpszMenuName = NULL ;
56:       wndclass.lpszClassName = szAppName ;
57:       wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
58:
59:       RegisterClassEx (&wndclass) ;
60:
61:       hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
```

```
62:                     0,0,800,600,NULL,NULL,hInst,NULL) ;
63:
64:     ShowWindow (hWnd,SW_SHOW) ;
65:     UpdateWindow (hWnd) ;
66:       // ダイレクト3D の初期化関数を呼ぶ
67:       if(FAILED(InitD3d(hWnd)))
68:       {
69:                     return 0;
70:       }
71:       // メッセージループ
72:     ZeroMemory( &msg, sizeof(msg) );
73:     while( msg.message!=WM_QUIT )
74:      {
75:                      if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
76:         {
77:                                     TranslateMessage( &msg );
78:             DispatchMessage( &msg );
79:         }
80:         else
81:                     {
82:                                     Render();
83:                     }
84:        }
85:        // メッセージループから抜けたらオブジェクトを全て開放する
86:        FreeDx();
87:        // OS に戻る （アプリケーションを終了する）
88:      return (INT)msg.wParam ;
89: }
90:
91: //
92: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
93: // ウィンドウプロシージャ関数
94: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
95: {
96:       switch(iMsg)
97:       {
98:                     case WM_DESTROY:
99:                                     PostQuitMessage(0);
100:                    break;
101:                    case WM_KEYDOWN:
102:                    switch((CHAR)wParam)
103:                    {
104:                                    case VK_ESCAPE:
105:                                                    PostQuitMessage(0);
106:                                    break;
107:                                    case VK_LEFT:
108:                                                    Thing[2].vecPosition.x-=0.1f;
109:                                    break;
110:                                    case VK_RIGHT:
111:                                                    Thing[2].vecPosition.x+=0.1f;
112:                                    break;
113:                                    case VK_UP:
114:                                                    Thing[2].vecPosition.z+=0.1f;
115:                                    break;
116:                                    case VK_DOWN:
117:                                                    Thing[2].vecPosition.z-=0.1f;
118:                                    break;
119:                    }
120:                    break;
121:      }
122:      return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
123: }
124:
125: //
126: //HRESULT InitD3d(HWND hWnd)
127: // ダイレクト 3D の初期化関数
128: HRESULT InitD3d(HWND hWnd)
129: {
130:       //「Direct3D」 オブジェクトの作成
131:       if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
```

```
132:     {
133:                 MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
134:       return E_FAIL;
135:     }
136:     //「DIRECT3D デバイス」オブジェクトの作成
137:     D3DPRESENT_PARAMETERS d3dpp;
138:     ZeroMemory( &d3dpp, sizeof(d3dpp) );
139:     d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
140:     d3dpp.BackBufferCount=1;
141:     d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
142:     d3dpp.Windowed = TRUE;
143:   d3dpp.EnableAutoDepthStencil = TRUE;
144:   d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
145:
146:     if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
147:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
148:                                 &d3dpp, &pDevice ) ) )
149:   {
150:                 MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
151:                 if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
152:                                 D3DCREATE_MIXED_VERTEXPROCESSING,
153:                                 &d3dpp, &pDevice ) ) )
154:                 {
155:                             MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
156:                              return E_FAIL;
157:                 }
158:   }
159:
160:     // X ファイル毎にメッシュを作成する
161:     InitThing(&Thing[0],"Arrow.x",&D3DXVECTOR3(0,0,0));
162:     InitThing(&Thing[1],"Human.x",&D3DXVECTOR3(0,1.2,0));
163:     InitThing(&Thing[2],"Center.x",&D3DXVECTOR3(0,0,0));
164:     // Z バッファー処理を有効にする
165:   pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
166:     // ライトを有効にする
167:     pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
168:     // アンビエントライト（環境光）を設定する
169:      pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
170:      // スペキュラ（鏡面反射）を有効にする
171:      pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
172:     return S_OK;
173: }
174:
175: //
176: //HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
177: //
178: HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
179: {
180:     // メッシュの初期位置
181:     memcpy(&pThing->vecPosition,pvecPosition,sizeof(D3DXVECTOR3));
182:     // メッシュの初期回転行列
183:     D3DXMatrixIdentity(&pThing->matRotation);
184:     // X ファイルからメッシュをロードする
185:     LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
186:
187:     if( FAILED( D3DXLoadMeshFromX( szXFileName, D3DXMESH_SYSTEMMEM,
188:         pDevice, NULL, &pD3DXMtrlBuffer, NULL,
189:                                 &pThing->dwNumMaterials, &pThing->pMesh ) ) )
190:   {
191:         MessageBox(NULL, "X ファイルの読み込みに失敗しました ",szXFileName, MB_OK);
192:         return E_FAIL;
193:   }
194:     D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
195:     pThing->pMeshMaterials = new D3DMATERIAL9[pThing->dwNumMaterials];
196:   pThing->pMeshTextures  = new LPDIRECT3DTEXTURE9[pThing->dwNumMaterials];
197:
198:     for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
199:     {
200:                 pThing->pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
201:       pThing->pMeshMaterials[i].Ambient = pThing->pMeshMaterials[i].Diffuse;
```

```
202:         pThing->pMeshTextures[i] = NULL;
203:         if( d3dxMaterials[i].pTextureFilename != NULL &&
204:             lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
205:         {
206:             if( FAILED( D3DXCreateTextureFromFile( pDevice,
207:                                     d3dxMaterials[i].pTextureFilename,
208:                                     &pThing->pMeshTextures[i] ) ) )
209:             {
210:                 MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
211:                             }
212:         }
213:     }
214:       pD3DXMtrlBuffer->Release();
215:
216:       return S_OK;
217: }
218:
219: //
220: //VOID Render()
221: //X ファイルから読み込んだメッシュをレンダリングする関数
222: VOID Render()
223: {
224:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
225:                       D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
226:
227:        if( SUCCEEDED( pDevice->BeginScene() ) )
228:        {
229:                       RotaryMove(&Thing[1],&Thing[2].vecPosition,0.5f);
230:                       for(DWORD i=0;i<THING_AMOUNT;i++)
231:                       {
232:                                   RenderThing(&Thing[i]);
233:                       }
234:                       pDevice->EndScene();
235:        }
236:        pDevice->Present( NULL, NULL, NULL, NULL );
237: }
238: //
239: //VOID RenderThing(THING* pThing)
240: //
241: VOID RenderThing(THING* pThing)
242: {
243:       // ワールドトランスフォーム （絶対座標変換）
244:        D3DXMATRIXA16 matWorld,matPosition;
245:        D3DXMatrixIdentity(&matWorld);
246:        D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,
247:                       pThing->vecPosition.z);
248:        D3DXMatrixMultiply(&matWorld,&matWorld,&pThing->matRotation);
249:        D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
250:      pDevice->SetTransform( D3DTS_WORLD, &matWorld );
251:        // ビュートランスフォーム （視点座標変換）
252:      D3DXVECTOR3 vecEyePt( 5.0f, 6.0f,-8.0f ); // カメラ （視点） 位置
253:      D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 1.0f );// 注視位置
254:      D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
255:      D3DXMATRIXA16 matView;
256:      D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
257:      pDevice->SetTransform( D3DTS_VIEW, &matView );
258:        // プロジェクショントランスフォーム （射影変換）
259:      D3DXMATRIXA16 matProj;
260:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
261:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
262:        // ライトをあてる 白色で鏡面反射ありに設定
263:        D3DXVECTOR3 vecDirection(1,1,1);
264:      D3DLIGHT9 light;
265:      ZeroMemory( &light, sizeof(D3DLIGHT9) );
266:      light.Type       = D3DLIGHT_DIRECTIONAL;
267:      light.Diffuse.r  = 1.0f;
268:      light.Diffuse.g  = 1.0f;
269:      light.Diffuse.b  = 1.0f;
270:        light.Specular.r=1.0f;
271:        light.Specular.g=1.0f;
```

```
272:        light.Specular.b=1.0f;
273:      D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
274:      light.Range        = 200.0f;
275:      pDevice->SetLight( 0, &light );
276:      pDevice->LightEnable( 0, TRUE );
277:        // レンダリング
278:        for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
279:      {
280:                        pDevice->SetMaterial( &pThing->pMeshMaterials[i] );
281:          pDevice->SetTexture( 0,pThing->pMeshTextures[i] );
282:          pThing->pMesh->DrawSubset( i );
283:      }
284: }
285: //
286: //VOID RotaryMove(THING* pThing,D3DXVECTOR3* pvecCenter,FLOAT fRadius)
287: // 円運動　円周の座標を計算する
288: VOID RotaryMove(THING* pThing,D3DXVECTOR3* pvecCenter,FLOAT fRadius)
289: {
290:       D3DXMATRIX matRotation;
291:       // まず最初に、原点に半径を足しただけの座標を用意する
292:       D3DXVECTOR3 vecTarget(-fRadius,0.0f,0.0f);
293:       // 次に、原点を中心とした回転（オイラー回転）の行列を作る
294:       D3DXMatrixRotationY(&matRotation,timeGetTime()/1000.0f);
295:       D3DXVec3TransformCoord(&vecTarget,&vecTarget,&matRotation);
296:       // 最後に本来の座標（回転対象の座標）を足す
297:       D3DXVec3Add(&vecTarget,&vecTarget,pvecCenter);
298:       pThing->vecPosition.x=vecTarget.x;
299:       pThing->vecPosition.z=vecTarget.z;
300:       pThing->matRotation=matRotation;
301: }
302: //
303: //VOID FreeDx()
304: // 作成した DirectX オブジェクトの開放
305: VOID FreeDx()
306: {
307:       for(DWORD i=0;i<THING_AMOUNT;i++)
308:       {
309:                        SAFE_RELEASE( Thing[i].pMesh );
310:       }
311:       SAFE_RELEASE( pDevice );
312:       SAFE_RELEASE( pD3d );
313: }
```

ここでは、対象ジオメトリを回転させる方法の一つを紹介します。回転と言っても回転の中心をジオメトリの外に置くので、円運動と言ったほうが正確でしょう。
本書では以降も
回転の中心が回転主体の内部にあれば " 回転 " で、
回転の中心が回転主体の外側にあれば " 円運動 " と定義することにします。

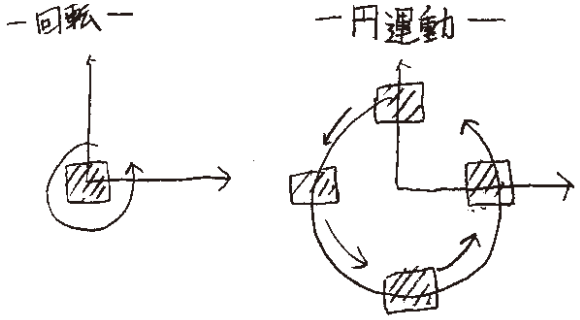


図 11-1
普通、ジオメトリのワールド座標系における姿勢は、ジオメトリごとのワールド変換により決定されますが、ワールド変換行列を作成する際に、まず回転行列を掛けてから平行移動行列を掛ける、言い換えれば、回転行列に平行移動行列を掛けたものをワールド変換行列としないと、意図した結果になりません。（図 11-2 参照）

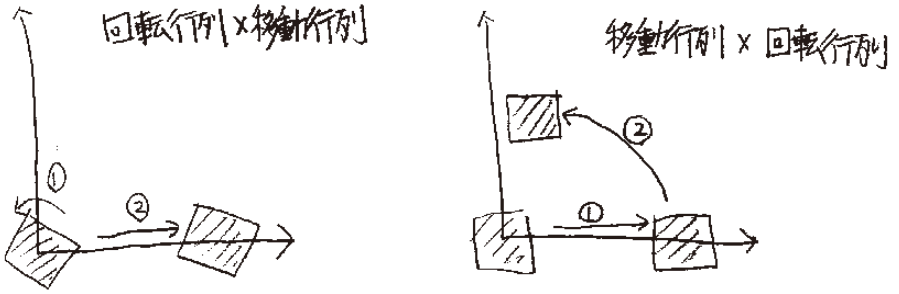


図 11-2

単純な円運動の場合、図中での間違った変換を使用しても実現できます。つまり、平行移動と回転の行列を掛ける順番を反転すれば、円運動を実現することができます。しかし、その方法が適用できるのはジオメトリがワールド系の原点にあるときだけです。よって本節では、その方法は取らず、もっと汎用性があり円運動を実現する手順が見えるようなコードにしました。
まず、任意の座標にあるジオメトリを円運動させるにあたり、普通に回転行列を掛けた場合の問題点を見ておきましょう。（図 11-3 参照）

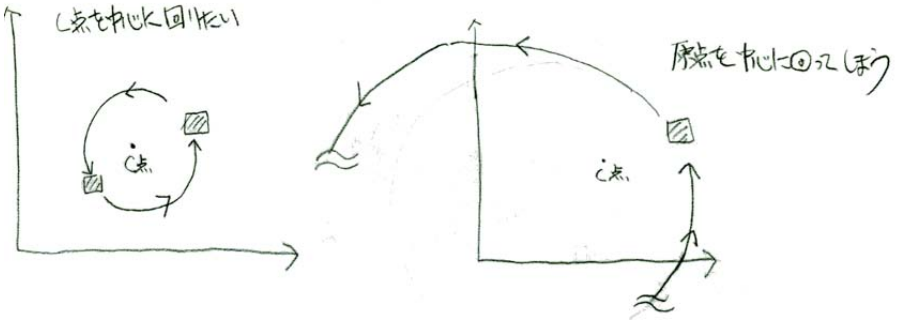


図 11-3
C 点を中心に円運動したいのに、普通に回転行列を掛けると原点を中心に回ってしまいます。
こうならないように正しく円運動させる考え方はこうです。
まず、原点（ワールド座標系の原点）を中心とし、意図する半径分の回転をします。
そして、それに対象ジオメトリの座標を足せば、ジオメトリが C 点の周りを円運動する。
という簡単な原理です。

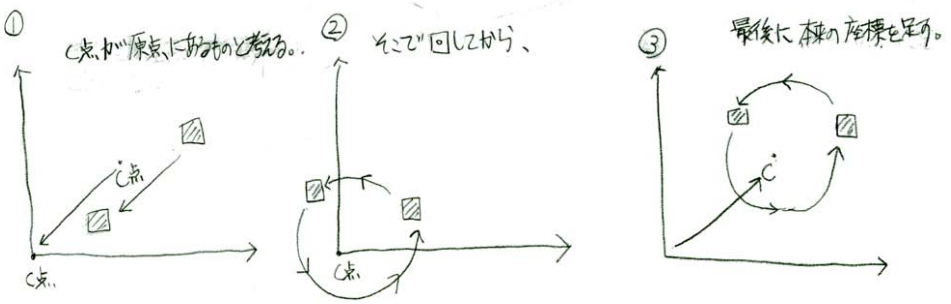


図 11-4

RotaryMove 関数では、この処理を行っているのです。pvecCenter が C 点に、そして vecTarget が回転対象にあたります。

D3DXVECTOR3 vecTarget(-fRadius,0.0f,0.0f);
回転対象の座標を最初から原点を中心に考えます。
D3DXVec3TransformCoord(&vecTarget,&vecTarget,&matRotation);
次に、回転行列を掛けます。
D3DXVec3Add(&vecTarget,&vecTarget,pvecCenter);
最後に、本来の座標を足せば、結果的に本来の座標での円運動になります。

なお、次の考えを補足しておきます。ジオメトリ全体が回転するということは、ジオメトリを構成する頂点が円運動するということです。（回転中心と重なる頂点を除く）

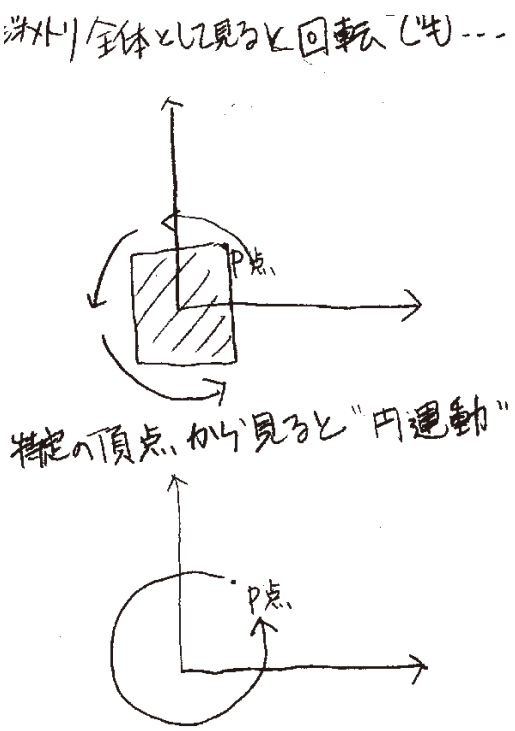


図 11-5

現在の姿勢からの平行移動

本節ではジオメトリを絶対座標系で 4 方向（前、後ろ、左、右）に平行移動することを考えます。簡単のため上下方向は考えず、動きとしては XZ 平面の 2 次元的なものです。
ユーザーの入力をもとに単純にジオメトリの座標を増減させると図 11-6 のような不自然さが表面化します。

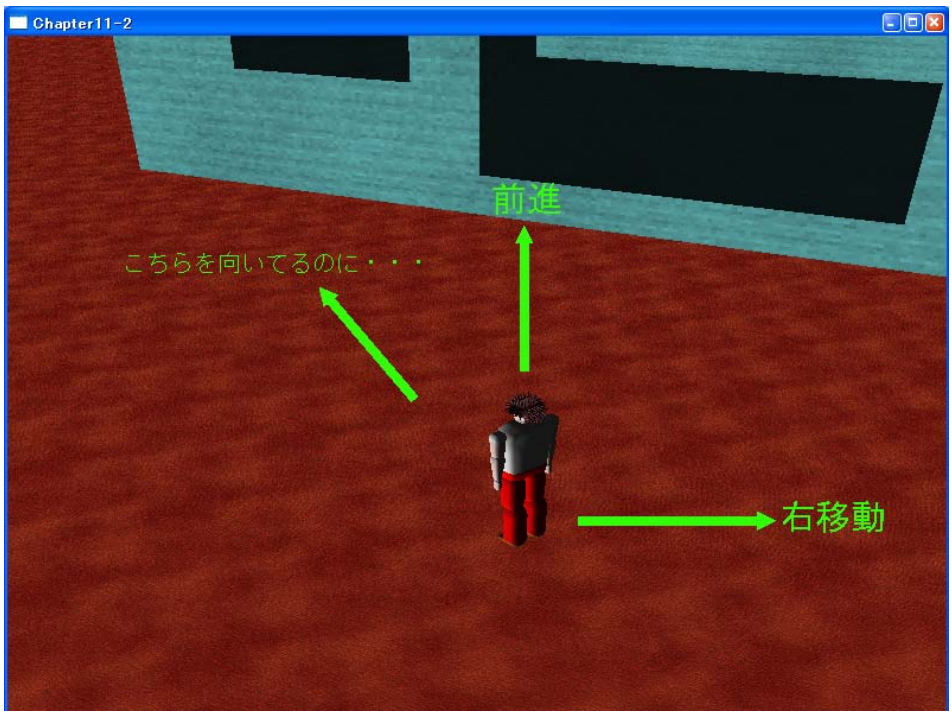


図 11-6
対象が斜め前方に向いているのに、絶対的な前後左右、言い換えると、X 軸、Z 軸方向に動かしてしまうと、不自然です。図 11-6 のような鳥瞰視点であれば、無理やりそれが仕様なのだと言い張ることも可能ですが、ジオメトリからの視点の場合（カメラがジオメトリとシンクロしている場合）は、不自然を通り越して操作不可能といってもいいでしょう。視点がジオメトリからであるいい例は一人称視点シューティングゲームです。1 人称シューティングで、このような動きは有り得ません。移動は図 11-7 のようになるべきです。

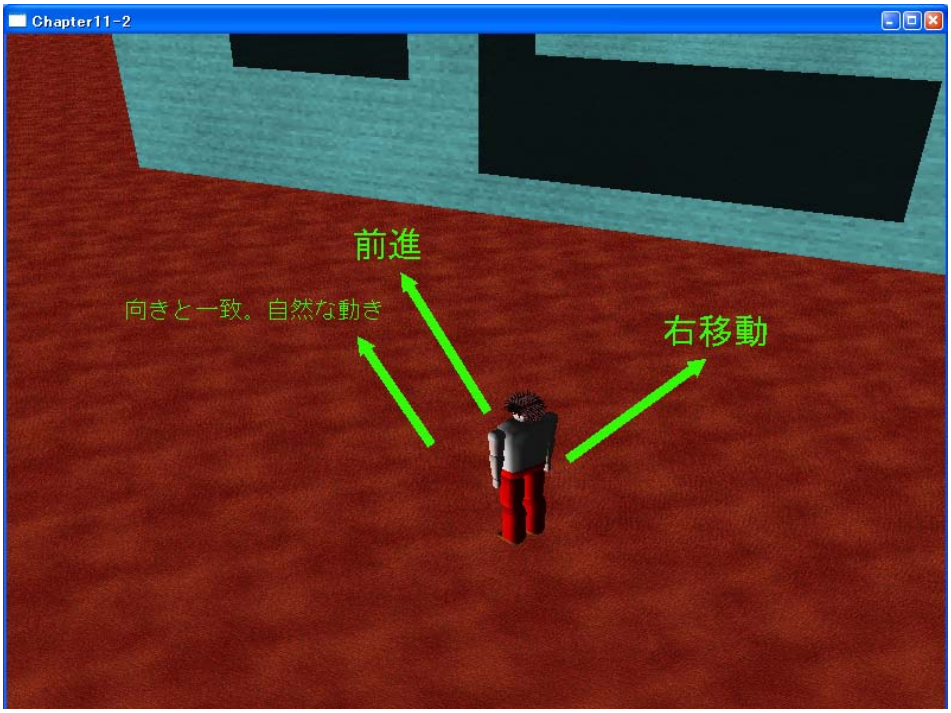


図 11-7

ある時点での“ジオメトリの向きにおける前後左右”は、絶対座標系の“絶対的な前後左右”とは異なることが殆どでしょう。その時点での姿勢、図の場合はその時点での向きに対する前後左右は絶対系の軸とはずれています。移動は図中の矢印方向に移動するのが自然です。このような現在の向きに対しての移動は 1 人称シューティングでは必須です、この場合の横の動きは、「ステップ移動」、または「かに歩き」などと呼ばれるものです。

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5: #define THING_AMOUNT 4+1
 6:
 7: enum DIRECTION
 8: {
 9:         STOP,
10:         FORWARD,
11:         BACKWARD,
12:         LEFT,
13:         RIGHT
14: };
15:
16: struct THING
17: {
18:         LPD3DXMESH pMesh;
19:         D3DMATERIAL9* pMeshMaterials;
20:         LPDIRECT3DTEXTURE9* pMeshTextures ;
21:         DWORD dwNumMaterials;
22:         D3DXVECTOR3 vecPosition;
23:         D3DXMATRIX matRotation;
24:         D3DXMATRIX matWorld;
25:         FLOAT fHeading;
26:         DIRECTION Dir;
27:         THING()
28:         {
29:                         ZeroMemory(this,sizeof(THING));
30:         }
31: };
32:
33: LPDIRECT3D9 pD3d;
34: LPDIRECT3DDEVICE9 pDevice;
```

```
35:
36: THING Thing[THING_AMOUNT];
37:
38: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
39: HRESULT InitD3d(HWND);
40: HRESULT InitThing(THING *,LPSTR,D3DXVECTOR3*);
41: VOID Render();
42: VOID RenderThing(THING*);
43: VOID StepMove(THING*);
44: VOID FreeDx();
45:
46: //
47: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
48: // アプリケーションのエントリー関数
49: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
50: {
51:       HWND hWnd=NULL;
52:       MSG msg;
53:       // ウィンドウの初期化
54:       static char szAppName[] = "Chapter11-2" ;
55:     WNDCLASSEX  wndclass ;
56:
57:     wndclass.cbSize          = sizeof (wndclass) ;
58:     wndclass.style           = CS_HREDRAW | CS_VREDRAW ;
59:     wndclass.lpfnWndProc     = WndProc ;
60:     wndclass.cbClsExtra      = 0 ;
61:     wndclass.cbWndExtra      = 0 ;
62:     wndclass.hInstance       = hInst ;
63:     wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
64:     wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
65:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
66:     wndclass.lpszMenuName = NULL ;
67:     wndclass.lpszClassName = szAppName ;
68:     wndclass.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;
69:
70:     RegisterClassEx (&wndclass) ;
71:
72:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
73:                 0,0,800,600,NULL,NULL,hInst,NULL) ;
74:
75:    ShowWindow (hWnd,SW_SHOW) ;
76:    UpdateWindow (hWnd) ;
77:       SetWindowText(hWnd," テンキーで回転、矢印キーで前後左右に移動します。");
78:       // ダイレクト3D の初期化関数を呼ぶ
79:       if(FAILED(InitD3d(hWnd)))
80:       {
81:                   return 0;
82:       }
83:       // メッセージループ
84:    ZeroMemory( &msg, sizeof(msg) );
85:    while( msg.message!=WM_QUIT )
86:     {
87:                   if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
88:         {
89:                               TranslateMessage( &msg );
90:             DispatchMessage( &msg );
91:         }
92:         else
93:                   {
94:                               Render();
95:                   }
96:       }
97:       // メッセージループから抜けたらオブジェクトを全て開放する
98:       FreeDx();
99:       // OS に戻る （アプリケーションを終了する）
100:     return (INT)msg.wParam ;
101: }
102:
103: //
104: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
```

```
105: // ウィンドウプロシージャ関数
106: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
107: {
108:       switch(iMsg)
109:       {
110:                   case WM_DESTROY:
111:                               PostQuitMessage(0);
112:                   break;
113:                   case WM_KEYDOWN:
114:                   switch((CHAR)wParam)
115:                   {
116:                               case VK_ESCAPE:
117:                                           PostQuitMessage(0);
118:                               break;
119:                               case VK_NUMPAD4:
120:                                           Thing[3].fHeading-=0.1f;
121:                                           break;
122:                               case VK_NUMPAD6:
123:                                           Thing[3].fHeading+=0.1f;
124:                                           break;
125:                               case VK_LEFT:
126:                                           Thing[3].Dir=LEFT;
127:                               break;
128:                               case VK_RIGHT:
129:                                           Thing[3].Dir=RIGHT;
130:                               break;
131:                               case VK_UP:
132:                                           Thing[3].Dir=FORWARD;
133:                               break;
134:                               case VK_DOWN:
135:                                           Thing[3].Dir=BACKWARD;
136:                               break;
137:                   }
138:                   break;
139:       }
140:       return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
141: }
142:
143: //
144: //HRESULT InitD3d(HWND hWnd)
145: // ダイレクト 3D の初期化関数
146: HRESULT InitD3d(HWND hWnd)
147: {
148:       //「Direct3D」 オブジェクトの作成
149:       if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
150:       {
151:                   MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
152:         return E_FAIL;
153:       }
154:       //「DIRECT3D デバイス」 オブジェクトの作成
155:       D3DPRESENT_PARAMETERS d3dpp;
156:       ZeroMemory( &d3dpp, sizeof(d3dpp) );
157:       d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
158:       d3dpp.BackBufferCount=1;
159:       d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
160:       d3dpp.Windowed = TRUE;
161:     d3dpp.EnableAutoDepthStencil = TRUE;
162:     d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
163:
164:        if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
165:                                    D3DCREATE_MIXED_VERTEXPROCESSING,
166:                                    &d3dpp, &pDevice ) ) )
167:     {
168:                   MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
169:                   if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
170:                                    D3DCREATE_MIXED_VERTEXPROCESSING,
171:                                    &d3dpp, &pDevice ) ) )
172:                   {
173:                               MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
174:                                return E_FAIL;
```

```
175:                     }
176:     }
177:
178:       // X ファイル毎にメッシュを作成する
179:       InitThing(&Thing[0],"Sky.x",&D3DXVECTOR3(0,0,0));
180:       InitThing(&Thing[1],"Ground.x",&D3DXVECTOR3(0,0,0));
181:       InitThing(&Thing[2],"Building.x",&D3DXVECTOR3(0,0,15));
182:       InitThing(&Thing[3],"Human.x",&D3DXVECTOR3(0,1.2,0));
183:       // Z バッファー処理を有効にする
184:     pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
185:       // ライトを有効にする
186:       pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
187:       // アンビエントライト（環境光）を設定する
188:        pDevice->SetRenderState( D3DRS_AMBIENT, 0x00111111 );
189:        // スペキュラ（鏡面反射）を有効にする
190:        pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
191:       return S_OK;
192: }
193:
194: //
195: //HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
196: //
197: HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
198: {
199:       // メッシュの初期位置
200:       memcpy(&pThing->vecPosition,pvecPosition,sizeof(D3DXVECTOR3));
201:       // X ファイルからメッシュをロードする
202:       LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
203:
204:       if( FAILED( D3DXLoadMeshFromX( szXFileName, D3DXMESH_SYSTEMMEM,
205:           pDevice, NULL, &pD3DXMtrlBuffer, NULL,
206:                               &pThing->dwNumMaterials, &pThing->pMesh ) ) )
207:     {
208:           MessageBox(NULL, "X ファイルの読み込みに失敗しました ",szXFileName, MB_OK);
209:           return E_FAIL;
210:     }
211:       D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
212:       pThing->pMeshMaterials = new D3DMATERIAL9[pThing->dwNumMaterials];
213:     pThing->pMeshTextures  = new LPDIRECT3DTEXTURE9[pThing->dwNumMaterials];
214:
215:       for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
216:       {
217:                   pThing->pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
218:         pThing->pMeshMaterials[i].Ambient = pThing->pMeshMaterials[i].Diffuse;
219:         pThing->pMeshTextures[i] = NULL;
220:         if( d3dxMaterials[i].pTextureFilename != NULL &&
221:           lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
222:         {
223:           if( FAILED( D3DXCreateTextureFromFile( pDevice,
224:                                       d3dxMaterials[i].pTextureFilename,
225:                                       &pThing->pMeshTextures[i] ) ) )
226:           {
227:               MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
228:                           }
229:         }
230:     }
231:       pD3DXMtrlBuffer->Release();
232:
233:       return S_OK;
234: }
235:
236: //
237: //VOID Render()
238: //X ファイルから読み込んだメッシュをレンダリングする関数
239: VOID Render()
240: {
241:       pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
242:                   D3DCOLOR_XRGB(100,100,100), 1.0f, 0 );
243:
244:        if( SUCCEEDED( pDevice->BeginScene() ) )
```

```
245:     {
246:             for(DWORD i=0;i<THING_AMOUNT;i++)
247:             {
248:                     StepMove(&Thing[i]);
249:                     RenderThing(&Thing[i]);
250:             }
251:             pDevice->EndScene();
252:     }
253:     pDevice->Present( NULL, NULL, NULL, NULL );
254: }
255: //
256: //VOID RenderThing(THING* pThing)
257: //
258: VOID RenderThing(THING* pThing)
259: {
260:      // ワールドトランスフォーム（絶対座標変換）
261:     pDevice->SetTransform( D3DTS_WORLD, &pThing->matWorld );
262:      // ビュートランスフォーム（視点座標変換）
263:     D3DXVECTOR3 vecEyePt( 5.0f, 6.0f,-8.0f ); // カメラ（視点）位置
264:     D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 1.0f );// 注視位置
265:     D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
266:     D3DXMATRIXA16 matView;
267:     D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
268:     pDevice->SetTransform( D3DTS_VIEW, &matView );
269:      // プロジェクショントランスフォーム（射影変換）
270:     D3DXMATRIXA16 matProj;
271:     D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
272:     pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
273:      // ライトをあてる 白色で鏡面反射ありに設定
274:      D3DXVECTOR3 vecDirection(1,1,1);
275:     D3DLIGHT9 light;
276:     ZeroMemory( &light, sizeof(D3DLIGHT9) );
277:     light.Type       = D3DLIGHT_DIRECTIONAL;
278:     light.Diffuse.r  = 1.0f;
279:     light.Diffuse.g  = 1.0f;
280:     light.Diffuse.b  = 1.0f;
281:      light.Specular.r=1.0f;
282:      light.Specular.g=1.0f;
283:      light.Specular.b=1.0f;
284:     D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
285:     light.Range       = 200.0f;
286:     pDevice->SetLight( 0, &light );
287:     pDevice->LightEnable( 0, TRUE );
288:      // レンダリング
289:      for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
290:     {
291:              pDevice->SetMaterial( &pThing->pMeshMaterials[i] );
292:         pDevice->SetTexture( 0,pThing->pMeshTextures[i] );
293:         pThing->pMesh->DrawSubset( i );
294:     }
295: }
296: //
297: //VOID StepMove(THING* pThing)
298: // 回転、ステップ移動を含むワールドトランスフォーム行列を計算
299: VOID StepMove(THING* pThing)
300: {
301:      D3DXMATRIXA16 matPosition;
302:      D3DXMatrixIdentity(&pThing->matWorld);
303:
304:      // 回転行列を計算
305:      D3DXMatrixRotationY(&pThing->matRotation,pThing->fHeading);
306:      // ステップ移動量を計算
307:
308:      // まずは軸ベクトルを用意する 今回は 2 軸のみ（Y 軸は考慮しない）
309:      D3DXVECTOR3 vecAxisX(1.0f,0.0f,0.0f);// 変換前の X 軸ベクトル
310:      D3DXVECTOR3 vecAxisZ(0.0f,0.0f,1.0f);// 変換前 Z 軸ベクトル
311:      //X、Z 軸ベクトルそのものをを現在の回転状態により変換する
312:      D3DXVec3TransformCoord(&vecAxisX,&vecAxisX,&pThing->matRotation);
313:      D3DXVec3TransformCoord(&vecAxisZ,&vecAxisZ,&pThing->matRotation);
314:
```

```
315:        switch(pThing->Dir)
316:        {
317:                    case LEFT:
318:                            pThing->vecPosition-=vecAxisX*0.1f;
319:                    break;
320:                    case RIGHT:
321:                            pThing->vecPosition+=vecAxisX*0.1f;;
322:                    break;
323:                    case FORWARD:
324:                            pThing->vecPosition+=vecAxisZ*0.1f;
325:                    break;
326:                    case BACKWARD:
327:                            pThing->vecPosition-=vecAxisZ*0.1f;
328:                    break;
329:        }
330:        pThing->Dir=STOP;
331:        D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,pThing->vecPosition.z);
332:        D3DXMatrixMultiply(&pThing->matWorld,&pThing->matRotation,&matPosition);
333: }
334: //
335: //VOID FreeDx()
336: // 作成した DirectX オブジェクトの開放
337: VOID FreeDx()
338: {
339:        for(DWORD i=0;i<THING_AMOUNT;i++)
340:        {
341:                    SAFE_RELEASE( Thing[i].pMesh );
342:        }
343:        SAFE_RELEASE( pDevice );
344:        SAFE_RELEASE( pD3d );
345: }
```

どのように現在姿勢に対する前後左右移動を実現しているのか、その考え方を説明します。
ジオメトリは固有の座標軸を持っていると考えます。本節の場合、上下移動はしないので Y 軸は考慮しません。ジオメトリは X 軸と Z 軸を独自に持っているとします。
ジオメトリの回転とシンクロさせて軸も回転させます。
あとは、その軸の方向に座標を増減させればいいわけです。

では、実際のコードを見ていきましょう。
着目すべき部分は、StepMove 関数です。
まず、ユーザーのキー入力は関数の外で行っていますが、キー入力があった場合、DIRECTION という列挙定数に FORWARD（前方）,BACKWARD（後方）,LEFT（左方）,RIGHT（右方）のいずれかが格納されてきます。

```
D3DXVECTOR3 vecAxisX(1.0f,0.0f,0.0f);// 変換前の X 軸ベクトル
 D3DXVECTOR3 vecAxisZ(0.0f,0.0f,1.0f);
```

vecAxisX と vecAxisZ というベクトルを用意します。これは、ジオメトリにシンクロさせる軸ベクトルです。最初は、それぞれ絶対的な軸ベクトル成分で初期化します。

```
D3DXVec3TransformCoord(&vecAxisX,&vecAxisX,&pThing->matRotation);
D3DXVec3TransformCoord(&vecAxisZ,&vecAxisZ,&pThing->matRotation);
```

その軸ベクトルにジオメトリの現在の回転を掛けます。

```
switch(pThing->Dir)
 {
            case LEFT:
                        pThing->vecPosition-=vecAxisX*0.1f;
            break;
            case RIGHT:
                        pThing->vecPosition+=vecAxisX*0.1f;;
            break;
            case FORWARD:
                        pThing->vecPosition+=vecAxisZ*0.1f;
            break;
            case BACKWARD:
                        pThing->vecPosition-=vecAxisZ*0.1f;
            break;
 }
```

あとは、DIRECTION で方向を場合分けして、X 座標、Z 座標の値を増減させればいいだけです。
これが、本節の趣旨の全てです。あとの部分は本サンプル固有の微調整的な部分です。

Chapter12　位置ベクトル（座標）による衝突判定

まず、再確認していただきたい事柄があります。Direct3D は３D 物体を容易に表現してくれるという素晴らしいものです。しかし、Direct3D が我々に提供してくれるのは、３D 物体の表示（レンダリング）までであるということを再認識してください。ゲームを開発する場合、レンダリングした後に、それらの物体の相互作用についてもコーディングする必要があり場合が殆どです。相互作用とは、例えば物体同士がぶつかって反発したり、空間にある物体が重力で落ちて、さらに床にぶつかり弾むなどの力学的作用（ダイナミクス）のことです。Direct3D は、ダイナミクスまでは提供せず、その実装を開発者に委ねています。このことはゲームごとにダイナミクスは最適化されるべきという理由から当然かもしれません。
本節のテーマである衝突判定は、別々のジオメトリ同士がぶつかっているとか重なっているという状態を調べることであり、よく「当たり判定」とも言われるものですが、衝突と言ったほうがより３次元を表現している感があるので本書では以降、衝突判定という言葉を使います。衝突判定はダイナミクスの一部です。ダイナミクスは衝突判定のみならず物理法則までも含むより広範囲な実装を指します。
衝突判定をしない場合、レンダリングされた物体は、お互いがぶつかろうが重なろうが、あるいは宙に浮いていようが、おかまいなしに自由に振舞ってしまいます。ただ単にレンダリングするだけのゲーム（そんなゲームがあるのだろうか？）なら関係ないですが、通常は、そのように自由に羽ばたいてもらっては困るはずです。現実の自然法則を忠実に実装する必要はないですが、すくなくとも物体同士がぶつかって見える場合は、何らかの処理をしたいと思うことでしょう。シューティングゲームの場合は、弾と敵についての衝突判定をしないことにはゲームにならないわけですから。
本章および次章で、この衝突判定を異なる２つのアプローチで解説します。
まず本章では、位置ベクトルによる衝突判定を解説します。
位置ベクトルとは、座標のことであり座標を判断要素とした衝突判定です。このアプローチは、変位ベクトル方式と比べると比較的処理が高速であるということが特徴であり、一方でやや精密度に欠けるという特徴もあります。判定の概要は、まず座標や半径により境界ボリュームを作成します。それぞれの物体は、それぞれ固有の境界ボリューム（ボリュームとは３次元的に閉じた領域、空間のこと）を持ち、その境界ボリュームの座標の大小関係で衝突しているかどうかを判断するというものです。境界ボリュームの形状は、球と立方体の２種類があり、それぞれ境界球、境界ボックスと呼ばれます。境界は英語でバウンディングなので、それぞれバウンディング･スフィア（スフィアは球のこと）、バウンディングボックスとも呼ばれます。
本章では、衝突判定の結果を確認するために画面に文字列をレンダリングするルーチンを加えています。極力コードは、単純でその章のテーマのみのコードにするというのが本書のコンセプトでもあるのですが、衝突のような“常にその状態が続く”ような事象の判定を確認するにはどうしても文字列による情報をレンダリングする必要があるのです。情報を確認するもっとも簡単な方法は Win32API のメッセージボックスを使用することですが、常に衝突状態である場合にメッセージボックスで確認するのは非常に困難です。メッセージボックスは常に出現し、プログラムを終了することすらままならない状態になってしまうからです。
文字列のレンダリングは Direct3D の ID3DXFont インターフェイスを使用すれば簡単に行え、また、本章に直接関連するものではないので解説はぜず、コード参照のみにします。

境界球（バウンディング・スフィア）による判定

境界球は、境界ボックスよりも、そして衝突判定方式全体で見ても最も簡単で、少ないコード量で実現できる方法です。球は楕円ではなく真円なので中心と（一定の）半径のみで表現でき、したがって判定に必要なものは、境界球の中心と半径の２つだけです。
判定原理はこうです。物体 A と物体 B があるとき、A の中心と B の中心の距離を L とし、A の半径と B の半径の合計を R とすると次のことが分かります。
L が R より大きい場合は、境界球は離れている。
L と R が等しい場合は、境界球が接触している。
L が R より小さい場合は、境界球は重なっている。

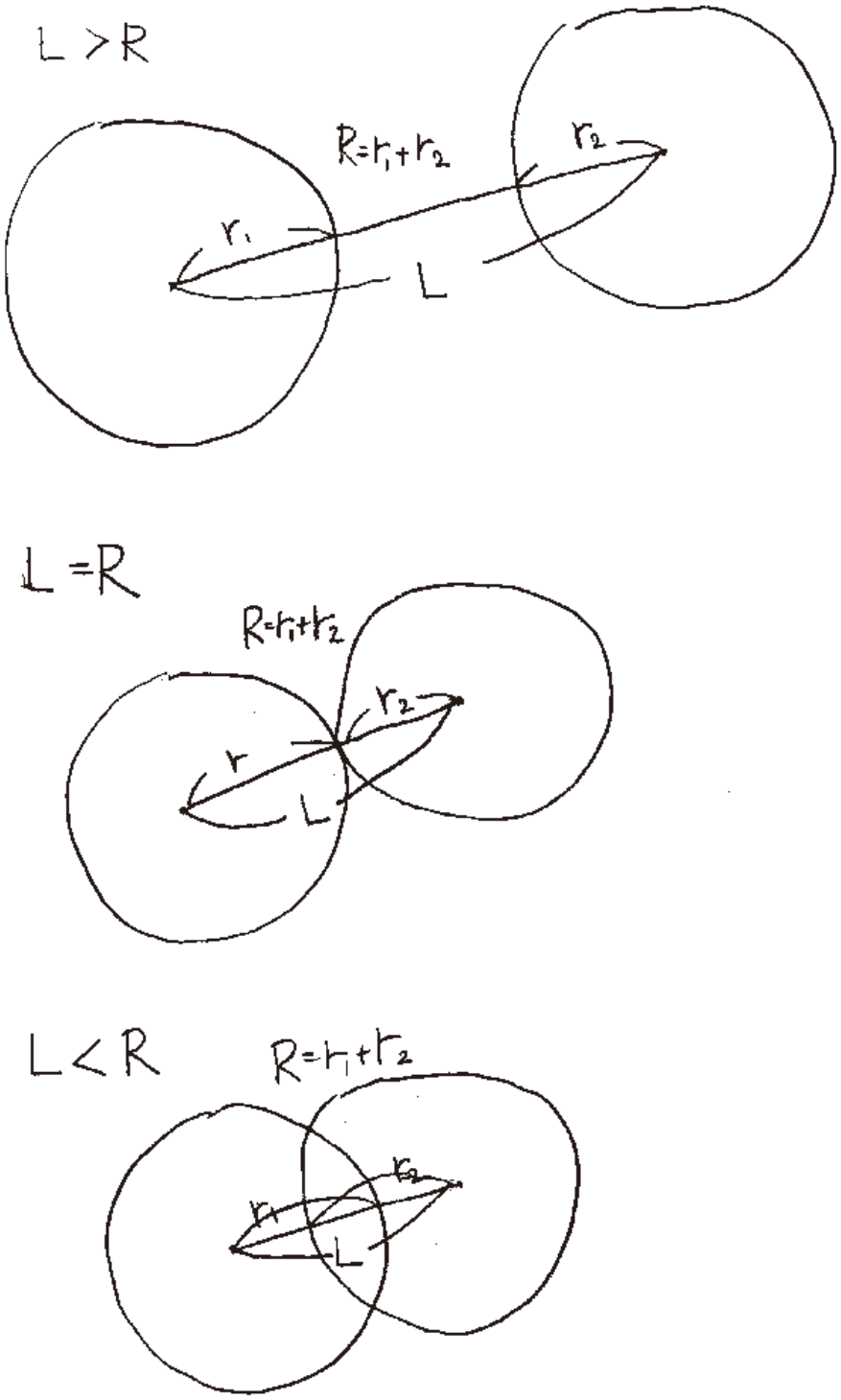


図 12-1

では、実際のサンプルコードを見ていきましょう。

そろそろ、一つのファイルに全てのコードを収めるのに無理が出てきましたので、１つのヘッダーファイルと２つの実装ファイル、計３つのファイルに分割しています。

着目すべき部分は全て Impact.cpp ファイル内にあります。衝突（判定）に関わる処理ということで Impact としました。Impact.cpp 内には２つの関数、InitSphere 関数と Impact 関数があります。衝突判定ルーチンは Impact 関数であり、InitSphere 関数はスフィアをメッシュとしてレンダリングするための準備がメインで本質的に関係のないコードではあります。ただ、開発中にデバッグのため本来レンダリングの必要ないスフィアを目で確認できるようにする目的でメッシュとしてレンダリングするこはよくあることで、特に学習用のサンプルではメッシュとしてレンダリングしなければ判定過程が解り辛いものとなってしまいます。そうゆう理由でスフィアをレンダリングする関数を作成したのですが、それが Impact.cpp 内の大半を占めている格好となっているので惑わされないでください。衝突判定を行っているのは Impact 関数です。

```
 1: #include <windows.h>
 2: #include <d3dx9.h>
 3:
 4: #define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
 5: #define THING_AMOUNT 2
 6:
 7: struct SPHERE
 8: {
 9:         D3DXVECTOR3 vecCenter;
10:         FLOAT fRadius;
11: };
12:
13: struct THING
14: {
15:         LPD3DXMESH pMesh;
16:         LPD3DXMESH pSphereMesh;
17:         D3DMATERIAL9* pMeshMaterials;
18:         D3DMATERIAL9* pSphereMeshMaterials;
19:         LPDIRECT3DTEXTURE9* pMeshTextures ;
20:         DWORD dwNumMaterials;
21:         D3DXVECTOR3 vecPosition;
22:         D3DXMATRIX matRotation;
23:         D3DXMATRIX matWorld;
24:         SPHERE Sphere;
25:
26:         THING()
27:         {
28:                         ZeroMemory(this,sizeof(THING));
29:         }
30: };
31:
32:
```

```
 1: #include "Chapter12-1.h"
 2:
 3: LPDIRECT3D9 pD3d;
 4: LPDIRECT3DDEVICE9 pDevice;
 5: LPD3DXFONT pFont;
 6: BOOL boRenderSphere=TRUE;
 7:
 8: THING Thing[THING_AMOUNT+1];
 9: LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
10: HRESULT InitD3d(HWND);
11: HRESULT InitThing(THING *,LPSTR,D3DXVECTOR3*);
12: VOID Render();
13: VOID RenderThing(THING*);
14: HRESULT InitSphere(LPDIRECT3DDEVICE9 ,THING* );
15: BOOL Impact(THING* ,THING* );
16: VOID RenderString(LPSTR ,INT,INT);
17: VOID FreeDx();
18:
19: //
20: //INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
21: // アプリケーションのエントリー関数
22: INT WINAPI WinMain( HINSTANCE hInst,HINSTANCE hPrevInst,LPSTR szStr,INT iCmdShow)
23: {
24:         HWND hWnd=NULL;
25:         MSG msg;
26:         // ウィンドウの初期化
27:         static char szAppName[] = "Chapter12-1" ;
28:       WNDCLASSEX  wndclass ;
29:
30:       wndclass.cbSize           = sizeof (wndclass) ;
31:       wndclass.style            = CS_HREDRAW | CS_VREDRAW ;
32:       wndclass.lpfnWndProc      = WndProc ;
33:       wndclass.cbClsExtra       = 0 ;
34:       wndclass.cbWndExtra       = 0 ;
```

```
35:     wndclass.hInstance        = hInst ;
36:     wndclass.hIcon            = LoadIcon (NULL, IDI_APPLICATION) ;
37:     wndclass.hCursor          = LoadCursor (NULL, IDC_ARROW) ;
38:     wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH) ;
39:     wndclass.lpszMenuName = NULL ;
40:     wndclass.lpszClassName = szAppName ;
41:     wndclass.hIconSm          = LoadIcon (NULL, IDI_APPLICATION) ;
42:
43:     RegisterClassEx (&wndclass) ;
44:
45:     hWnd = CreateWindow (szAppName,szAppName,WS_OVERLAPPEDWINDOW,
46:                 0,0,800,600,NULL,NULL,hInst,NULL) ;
47:
48:    ShowWindow (hWnd,SW_SHOW) ;
49:    UpdateWindow (hWnd) ;
50:       // ダイレクト3D の初期化関数を呼ぶ
51:       if(FAILED(InitD3d(hWnd)))
52:       {
53:                   return 0;
54:       }
55:       // メッセージループ
56:    ZeroMemory( &msg, sizeof(msg) );
57:    while( msg.message!=WM_QUIT )
58:    {
59:                    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
60:         {
61:                                TranslateMessage( &msg );
62:             DispatchMessage( &msg );
63:         }
64:         else
65:                   {
66:                                Render();
67:                   }
68:       }
69:       // メッセージループから抜けたらオブジェクトを全て開放する
70:       FreeDx();
71:       // OS に戻る （アプリケーションを終了する）
72:    return (INT)msg.wParam ;
73: }
74:
75: //
76: //LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
77: // ウィンドウプロシージャ関数
78: LRESULT CALLBACK WndProc(HWND hWnd,UINT iMsg,WPARAM wParam,LPARAM lParam)
79: {
80:       switch(iMsg)
81:       {
82:                   case WM_DESTROY:
83:                               PostQuitMessage(0);
84:                   return 0;
85:                   case WM_KEYDOWN:
86:                   switch((CHAR)wParam)
87:                   {
88:                               case VK_ESCAPE:
89:                                           PostQuitMessage(0);
90:                               return 0;
91:                               case VK_SPACE:
92:                                           boRenderSphere==TRUE ? boRenderSphere=FALSE : boRenderSphere=TRUE;
93:                                           break;
94:                               case VK_LEFT:
95:                                           Thing[1].vecPosition.x-=0.2f;
96:                               return 0;
97:                               case VK_RIGHT:
98:                                           Thing[1].vecPosition.x+=0.2f;
99:                               return 0;
100:                              case VK_UP:
101:                                          Thing[1].vecPosition.z+=0.2f;
102:                              return 0;
103:                              case VK_DOWN:
104:                                          Thing[1].vecPosition.z-=0.2f;
```

```
105:                                return 0;
106:                    }
107:                    break;
108:        }
109:        return DefWindowProc (hWnd, iMsg, wParam, lParam) ;
110: }
111:
112: //
113: //HRESULT InitD3d(HWND hWnd)
114: // ダイレクト 3D の初期化関数
115: HRESULT InitD3d(HWND hWnd)
116: {
117:        //「Direct3D」 オブジェクトの作成
118:        if( NULL == ( pD3d = Direct3DCreate9( D3D_SDK_VERSION ) ) )
119:        {
120:                    MessageBox(0,"Direct3D の作成に失敗しました ","",MB_OK);
121:         return E_FAIL;
122:        }
123:        //「DIRECT3D デバイス」 オブジェクトの作成
124:        D3DPRESENT_PARAMETERS d3dpp;
125:        ZeroMemory( &d3dpp, sizeof(d3dpp) );
126:        d3dpp.BackBufferFormat =D3DFMT_UNKNOWN;
127:        d3dpp.BackBufferCount=1;
128:        d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
129:        d3dpp.Windowed = TRUE;
130:      d3dpp.EnableAutoDepthStencil = TRUE;
131:      d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
132:
133:        if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
134:                                    D3DCREATE_MIXED_VERTEXPROCESSING,
135:                                    &d3dpp, &pDevice ) ) )
136:      {
137:                    MessageBox(0,"HAL モードで DIRECT3D デバイスを作成できません ¥nREF モードで再試行します ",NULL,MB_OK);
138:                    if( FAILED( pD3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
139:                                    D3DCREATE_MIXED_VERTEXPROCESSING,
140:                                    &d3dpp, &pDevice ) ) )
141:                    {
142:                                MessageBox(0,"DIRECT3D デバイスの作成に失敗しました ",NULL,MB_OK);
143:                                 return E_FAIL;
144:                    }
145:      }
146:
147:        // X ファイル毎にメッシュを作成する
148:        InitThing(&Thing[0],"OneMeshTank.x",&D3DXVECTOR3(6,0,6));
149:        InitThing(&Thing[1],"OneMeshLauncher.x",&D3DXVECTOR3(-3,0,-3));
150:        // メッシュごとのバウンディングスフィア （境界球） の作成
151:        InitSphere(pDevice,&Thing[0]);
152:        InitSphere(pDevice,&Thing[1]);
153:        // Z バッファー処理を有効にする
154:      pDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
155:        // カリングはしない
156:        pDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
157:        // ライトを有効にする
158:        pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
159:        // アンビエントライト （環境光） を設定する
160:        pDevice->SetRenderState( D3DRS_AMBIENT, 0x00555555 );
161:        // スペキュラ （鏡面反射） を有効にする
162:        pDevice->SetRenderState(D3DRS_SPECULARENABLE,TRUE);
163:        // スフィアを透明にレンダリングしたいのでアルファブレンディングを設定する
164:        pDevice->SetRenderState ( D3DRS_ALPHABLENDENABLE, TRUE );
165:        pDevice->SetRenderState ( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
166:        pDevice->SetRenderState ( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
167:        pDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
168:        // 文字列レンダリングの初期化
169:        HFONT hFont;
170:        if(NULL==(hFont=CreateFont(28,0,0,0,FW_REGULAR,FALSE,FALSE,FALSE,SHIFTJIS_CHARSET,
171:                    OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,PROOF_QUALITY,FIXED_PITCH | FF_MODERN,"tahoma"))) return E_
FAIL;
172:
173:        if(FAILED(D3DXCreateFont(pDevice,hFont,&pFont))) return E_FAIL;
```

```
174:
175:     return S_OK;
176: }
177:
178: //
179: //HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
180: //
181: HRESULT InitThing(THING *pThing,LPSTR szXFileName,D3DXVECTOR3* pvecPosition)
182: {
183:     // メッシュの初期位置
184:     memcpy(&pThing->vecPosition,pvecPosition,sizeof(D3DXVECTOR3));
185:     // X ファイルからメッシュをロードする
186:     LPD3DXBUFFER pD3DXMtrlBuffer = NULL;
187:
188:     if( FAILED( D3DXLoadMeshFromX( szXFileName, D3DXMESH_SYSTEMMEM,
189:         pDevice, NULL, &pD3DXMtrlBuffer, NULL,
190:                         &pThing->dwNumMaterials, &pThing->pMesh ) ) )
191:   {
192:         MessageBox(NULL, "X ファイルの読み込みに失敗しました ",szXFileName, MB_OK);
193:         return E_FAIL;
194:   }
195:     D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
196:     pThing->pMeshMaterials = new D3DMATERIAL9[pThing->dwNumMaterials];
197:   pThing->pMeshTextures  = new LPDIRECT3DTEXTURE9[pThing->dwNumMaterials];
198:
199:     for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
200:     {
201:                 pThing->pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
202:       pThing->pMeshMaterials[i].Ambient = pThing->pMeshMaterials[i].Diffuse;
203:       pThing->pMeshTextures[i] = NULL;
204:       if( d3dxMaterials[i].pTextureFilename != NULL &&
205:         lstrlen(d3dxMaterials[i].pTextureFilename) > 0 )
206:       {
207:         if( FAILED( D3DXCreateTextureFromFile( pDevice,
208:                                     d3dxMaterials[i].pTextureFilename,
209:                                     &pThing->pMeshTextures[i] ) ) )
210:         {
211:             MessageBox(NULL, " テクスチャの読み込みに失敗しました ", NULL, MB_OK);
212:                         }
213:       }
214:   }
215:     pD3DXMtrlBuffer->Release();
216:
217:     return S_OK;
218: }
219:
220: //
221: //VOID Render()
222: //X ファイルから読み込んだメッシュをレンダリングする関数
223: VOID Render()
224: {
225:     pDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
226:                 D3DCOLOR_XRGB(0,0,0), 1.0f, 0 );
227:
228:      if( SUCCEEDED( pDevice->BeginScene() ) )
229:      {
230:                 for(DWORD i=0;i<THING_AMOUNT;i++)
231:                 {
232:                         RenderThing(&Thing[i]);
233:                 }
234:                 if( Impact(&Thing[0],&Thing[1]) )
235:                 {
236:                         RenderString(" 衝突しています ¥n 境界ボリュームが重なっているということです ",10,10);
237:                 }
238:                 pDevice->EndScene();
239:      }
240:      pDevice->Present( NULL, NULL, NULL, NULL );
241: }
242: //
243: //VOID RenderThing(THING* pThing)
```

```
244: //
245: VOID RenderThing(THING* pThing)
246: {
247:       // ワールドトランスフォーム（絶対座標変換）
248:       D3DXMATRIXA16 matWorld,matPosition;
249:        D3DXMatrixIdentity(&matWorld);
250:        D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,
251:                    pThing->vecPosition.z);
252:        D3DXMatrixMultiply(&matWorld,&matWorld,&matPosition);
253:      pDevice->SetTransform( D3DTS_WORLD, &matWorld );
254:
255:       // ビュートランスフォーム（視点座標変換）
256:      D3DXVECTOR3 vecEyePt( 22.0f, 12.0f,-18.0f );  // カメラ（視点） 位置
257:      D3DXVECTOR3 vecLookatPt( 0.0f, 0.0f, 0.0f );// 注視位置
258:      D3DXVECTOR3 vecUpVec( 0.0f, 1.0f, 0.0f );// 上方位置
259:      D3DXMATRIXA16 matView;
260:      D3DXMatrixLookAtLH( &matView, &vecEyePt, &vecLookatPt, &vecUpVec );
261:      pDevice->SetTransform( D3DTS_VIEW, &matView );
262:       // プロジェクショントランスフォーム（射影変換）
263:      D3DXMATRIXA16 matProj;
264:      D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
265:      pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
266:       // ライトをあてる 白色で鏡面反射ありに設定
267:       D3DXVECTOR3 vecDirection(1,1,1);
268:      D3DLIGHT9 light;
269:      ZeroMemory( &light, sizeof(D3DLIGHT9) );
270:      light.Type       = D3DLIGHT_DIRECTIONAL;
271:      light.Diffuse.r  = 1.0f;
272:      light.Diffuse.g  = 1.0f;
273:      light.Diffuse.b  = 1.0f;
274:        light.Specular.r=1.0f;
275:        light.Specular.g=1.0f;
276:        light.Specular.b=1.0f;
277:      D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDirection );
278:      light.Range       = 200.0f;
279:      pDevice->SetLight( 0, &light );
280:      pDevice->LightEnable( 0, TRUE );
281:       // レンダリング
282:
283:       // 戦車のレンダリング
284:       for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
285:      {
286:                   pDevice->SetMaterial( &pThing->pMeshMaterials[i] );
287:          pDevice->SetTexture( 0,pThing->pMeshTextures[i] );
288:          pThing->pMesh->DrawSubset( i );
289:      }
290:       //  バウンディングスフィアのレンダリング
291:       if(boRenderSphere)
292:       {
293:                   pDevice->SetMaterial( pThing->pSphereMeshMaterials);
294:           pThing->pSphereMesh->DrawSubset( 0 );
295:       }
296: }
297: //
298: //VOID RenderString(LPSTR szStr,INT iX,INT iY)
299: // 情報表示用ルーチン
300: VOID RenderString(LPSTR szStr,INT iX,INT iY)
301: {
302:       RECT rect={iX,iY,0,0};
303:       // 文字列のサイズを計算
304:       pFont->DrawText(szStr,-1,&rect,DT_CALCRECT,NULL);
305:       // そのサイズでレンダリング
306:       pFont->DrawText(szStr,-1,&rect,DT_LEFT | DT_BOTTOM,0xff00ff00);
307:
308: }
309: //
310: //VOID FreeDx()
311: // 作成した DirectX オブジェクトの開放
312: VOID FreeDx()
313: {
```

```
314:      for(DWORD i=0;i<THING_AMOUNT;i++)
315:      {
316:                SAFE_RELEASE( Thing[i].pMesh );
317:      }
318:      SAFE_RELEASE( pDevice );
319:      SAFE_RELEASE( pD3d );
320: }
```

```
 1: #include "Chapter12-1.h"
 2:
 3: //
 4: //HRESULT InitSphere(LPDIRECT3DDEVICE9 pDevice,THING* pThing)
 5: // スフィアの計算およびスフィアを視認可能にするためにスフィアメッシュを作成する
 6: HRESULT InitSphere(LPDIRECT3DDEVICE9 pDevice,THING* pThing)
 7: {
 8:      HRESULT hr=NULL;
 9:      LPDIRECT3DVERTEXBUFFER9 pVB = NULL;
10:    VOID* pVertices = NULL;
11:      D3DXVECTOR3 vecCenter;
12:      FLOAT fRadius;
13:
14:      // メッシュの頂点バッファーをロックする
15:      if(FAILED(pThing->pMesh->GetVertexBuffer( &pVB )))
16:      {
17:                return E_FAIL;
18:      }
19:    if(FAILED(pVB->Lock( 0, 0, &pVertices, 0 )))
20:    {
21:       SAFE_RELEASE( pVB );
22:       return E_FAIL;
23:    }
24:      // メッシュの外接円の中心と半径を計算する
25:      hr=D3DXComputeBoundingSphere( (D3DXVECTOR3*)pVertices, pThing->pMesh->GetNumVertices(),
26:                                    D3DXGetFVFVertexSize(pThing->pMesh->GetFVF()), &vecCenter,
27:                                    &fRadius );
28:      pVB->Unlock();
29:    SAFE_RELEASE( pVB );
30:
31:      if(FAILED( hr ))
32:      {
33:       return hr;
34:      }
35:      pThing->Sphere.vecCenter=vecCenter;
36:      pThing->Sphere.fRadius=fRadius;
37:      // 得られた中心と半径を基にメッシュとしてのスフィアを作成する
38:      hr=D3DXCreateSphere(pDevice,fRadius,24,24,&pThing->pSphereMesh,NULL);
39:      if(FAILED( hr ))
40:      {
41:                return hr;
42:      }
43:      // スフィアメッシュのマテリアル　白色、半透明、光沢強
44:      pThing->pSphereMeshMaterials = new D3DMATERIAL9;
45:      pThing->pSphereMeshMaterials->Diffuse.r=1.0f;
46:      pThing->pSphereMeshMaterials->Diffuse.g=1.0f;
47:      pThing->pSphereMeshMaterials->Diffuse.b=1.0f;
48:      pThing->pSphereMeshMaterials->Diffuse.a=0.5f;
49:      pThing->pSphereMeshMaterials->Ambient=pThing->pSphereMeshMaterials->Diffuse;
50:      pThing->pSphereMeshMaterials->Specular.r=1.0f;
51:      pThing->pSphereMeshMaterials->Specular.g=1.0f;
52:      pThing->pSphereMeshMaterials->Specular.b=1.0f;
53:      pThing->pSphereMeshMaterials->Emissive.r=0.1f;
54:      pThing->pSphereMeshMaterials->Emissive.g=0.1f;
55:      pThing->pSphereMeshMaterials->Emissive.b=0.1f;
56:      pThing->pSphereMeshMaterials->Power=120.0f;
57:
58:      return S_OK;
59: }
60:
```

```
61: //
62: //BOOL Impact(THING* pThingA,THING* pThingB)
63: // 衝突判定
64: BOOL Impact(THING* pThingA,THING* pThingB)
65: {
66:         // 2つの物体の中心間の距離を求める
67:         D3DXVECTOR3 vecLength=pThingB->vecPosition-pThingA->vecPosition;
68:         FLOAT fLength=D3DXVec3Length(&vecLength);
69:         // その距離が、2 物体の半径を足したものより小さいということは、
70:         // 境界球同士が重なっている（衝突している）ということ
71:         if(fLength < pThingA->Sphere.fRadius+pThingB->Sphere.fRadius)
72:         {
73:                 return TRUE;
74:         }
75:
76:         return FALSE;
77: }
```

Impact 関数についての説明は既に、図によっても説明しています。前述の原理をそのままコードにしただけのものです。物体の中心間の距離と物体それぞれの半径を加算したものとを比較しているのです。重なっている場合のみ TRUE を返すようにしています。
スペースキーにより、境界ボリュームのレンダリングを有効・無効にできますので適宜切り替えてください。

境界ボックス（バウンディング・ボックス）による判定

境界ボックスは。境界球と並び高速な判定方法ですが、境界球よりは若干コード量が増えます。球の場合は中心と半径さえ分れば表現できるのですが、6 面体の場合は、頂点が 8 個あり、その 1 つ 1 つについて調べなければならないからです。しかし、それも大小関係を調べる処理なので処理としては高速です。次章のレイによる判定は乗算処理が必要なため、大小関係を調べることに比べどうしても重たくなってしましいます。
境界球のプログラムと同様に、ここでもファイルを 3 つに分け、注目すべきファイルは Impact.cpp 実装ファイルのみで、関数も同様に 2 つの関数のうち判定を行っているのは Impact 関数です。境界球プログラムの球の部分をボックスにしただけのプログラムと見て差し支えありません。
ボックスボリュームによる判定の原理は、次のとおりです。
物体 A のボックスボリュームの頂点それぞれが、物体 B のボックスボリュームに接しているかあるいは中に含まれる場合は衝突しているとするものです。調べる頂点を物体 B のものにしても同じことなので順序はどちらでも関係ありません。

8 つの頂点のいずれも相手のボリュームに接したり含まれたりしない場合は、物体同士は離れている。
8 つの頂点のうち 1 個以上が相手のボリュームに接している、あるいは含まれている場合は衝突している。

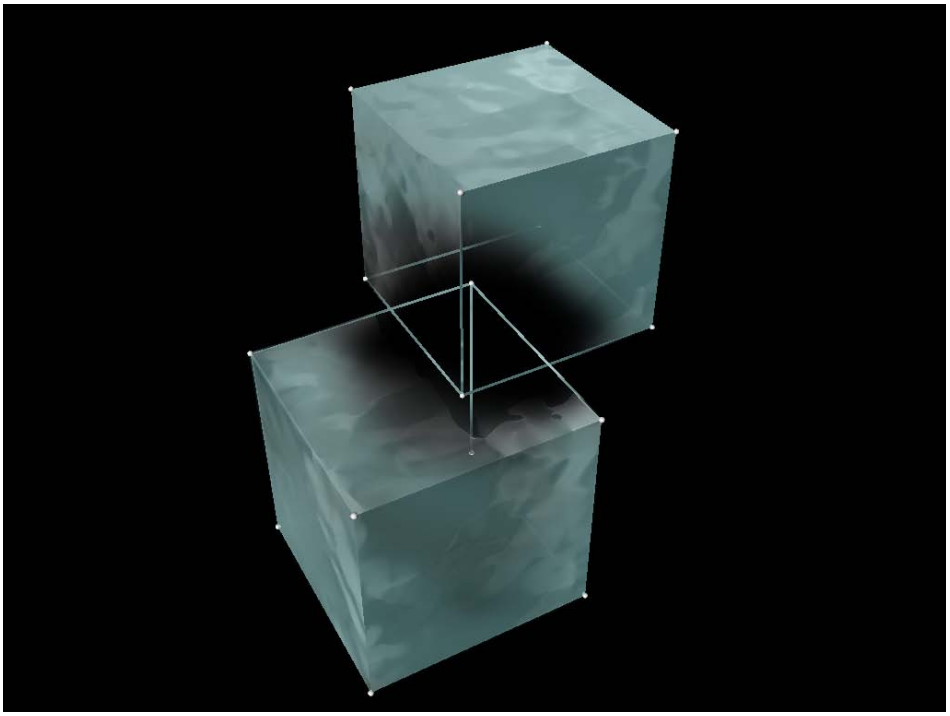


図 12-2

実際のコードを見ていきましょう。

Impact.cpp 内の Impact 関数だけに着目してください。
ここでも、境界球の場合と同様に、原理をそのまま単純なコードで実現しているだけであり、コードの理解は難しいものではないと考えます。8 つの頂点についてバウンディングボックスの最大と最小の頂点から他の 6 つの頂点を得ているところがミソでしょうか。あとは、単純に 8 つの頂点をループにかけて、頂点それぞれが相手のボリュームの内か外であるかを調べているにすぎません。
スペースキーにより、境界ボリュームのレンダリングを有効・無効にできますので適宜切り替えてください。

Chapter13 変位ベクトル（Ray：レイ）を併用した衝突判定

レイの意味

Ray（レイ）の単語としての意味は「光線」です。これは 3 DCG 分野でのレイ・トレーシング（光源追跡）技法におけるレイに由来しているものです。衝突判定における判定手段である変位ベクトルもレイという概念で捉えるという慣習があり、衝突判定においても、広くレイと呼ばれているものです。光線のベクトル的性質のみが重要であるので、“ 光線 ” でも “ 線分 ” でも、あるいは単にベクトルと言っても良く、判定の際にそれ以上の意味はありません。
前節の座標も本節のレイもコード上では D3DXVECTOR3 型、つまりベクトルで表現されるので外見は同じものです。異なるのは、その捉え方です。座標は、位置ベクトルと捉えるのでその X,Y,Z 成分の “ 位置関係、大小関係 ” を判断基準にするのに対し、レイは変位ベクトルとして捉えるのでその “ 向き ” を判断基準にするものです。同じベクトルでも見方あるいは捉え方によって位置ベクトルにも変位ベクトルにもなります。これは、例えば、ある数字が見方によって何かの順番、何かの価格、何かの重さ…というように色々な意味を取ることと同じです。

レイによる衝突判定の原理

原理の概要

主体ジオメトリ内の座標から任意の方向に向かう変位ベクトル（レイ）を考えます。レイが対象ジオメトリと交差するかどうかを調べ、交差している場合は、レイの始点（主体ジオメトリ内の座標）と交点の距離により衝突判断するというものです。

原理の詳細

実際にレイと交差判定するものは、対象ジオメトリを形成しているポリゴンです。対象ジオメトリが複数のポリゴンから成る場合、基本的に全てのポリゴンとの交差判定をすることになります。
レイとポリゴンとの交差判定は、直線と平面の交点を求めるという数学的な処理を行うことです。直線と平面の交点がレイによる衝突判定の本質なので、これを理解しないことには全ての理解は不可能であり、逆にこれを理解出来れば、その他の部分はたいしたことありません。しかし、それらを解説するとかなりのボリュームになるのでここでは詳細の解説は避け、SECTION3 第 18 章「ダイナミクスの基礎」において解説します。
Direct3D には、レイとメッシュの交差を判定してくれる D3DXIntersect という便利なヘルパー関数が存在し、それを使用する限りは、数学的知識は不要です。サンプルでも D3DXIntersect 関数により判定しています。18 章での解説は、D3DXIntersect 関数を使用せず、自前で交差判定ルーチンを作成する際の参考になることを目的としています。

では、サンプルコードを見ていきましょう。

今回も衝突判定に関わるのは Impact.cpp だけであり、着目すべきはそこに含まれる 2 つの関数、RenderRay 関数と Impact 関数だけです。
RenderRay 関数は、確認用にレイをレンダリングするもので、本質的に衝突の判定には関わりがありませんが、RenderRay 関数で行っている線分（ライン）のレンダリング処理は、レイのレンダリングに限らず、ほかの用途でも使用されると思われるので、ここで少し触れておきたいと思います。
pDevice->SetVertexShader(NULL);
直接、この関数に関係ないですが、関数が呼ばれる以前になんかのシェーダーが設定しれていると、その設定がそのまま適用されてしまうので、“ 念のため ” 設定をリセットします。
pDevice->SetFVF(D3DFVF_XYZ);
パイプラインのジオメトリレンダリング仕様を最も単純な “ 座標のみ ” に登録します。
IDirect3DDevice::DrawPrimitive メソッドは、頂点バッファーに格納されている頂点をレンダリングするメソッドであることはセクション 1 で述べた通りです。本サンプルでは DrawPrimitive の後に UP が付いています。UP は、Up「上」という意味ではなく、UsermemoryPointer（ユーザーメモリーポインター）の頭文字 U と P を意味しています。ユーザーメモリーポインターとは「頂点バッファーでなく、独自に用意した頂点のアドレス」という意味です。本サンプルの場合は、RenderRay 関数内でローカルに用意した D3DXVECTOR3 型の vecPnt[2] が用意した頂点 2 つであるので、そのアドレス vecPnt がユーザーメモリーポインターにあたります。
頂点バッファーをレンダリングするには、まず頂点タイプを定義し、頂点バッファーを確保して、頂点をそこに格納…云々という準備をする必要があるわけですが、本サンプルのように、最も単純な単なる線分用の 2 点（頂点）があればいいような場合にわざわざ、それらの事前準備をしなくてもいいように考えられたのがユーザーメモリーポインターによるレンダリング DrawPrimitiveUP です。
サンプルでは、レイは一本のみで、レイを出す Thing（この場合は Launcher メッシュ）から Z 軸プラス方向に 100 単位伸びるものを作成します。まず原点を始点として Z 軸に平行な長さが 100 単位の線分を定義するための 2 つの頂点を用意して、レンダリングパイプラインに Launcher メッシュのワールドトランスフォームをかけています。別の方法としては、最初から Launcher メッシュの位置ベクトルを始点とし、始点の Z 成分に 100 単位足したものを終点として頂点を用意すればワールドトランスフォームをかける必要はありません。どちらの方法でも構いません。なお、100 単位というのは、レイが到達するように十分な長さを確保するという意味で適当に設定した値です。
Impact 関数を見てみましょう。これが、衝突判定を行っている部分になります。
D3DXVECTOR3 vecStart,vecEnd,vecDirection;
レイの方向用に vecDirection を用意します。vecStart と vecEnd は、vecDirection を計算するためのものです。それぞれ始点

と終点を意味するもので、この 2 つのベクトルが定まれば vecDirection は vecEnd から vecStart を引くだけで求まります。
vecStart=vecEnd=pThingA->vecPosition;
とりあえず、始点も終点も物体 A（この場合は Launcher メッシュ）の座標で初期化します。
BOOL boHit=FALSE;
レイと物体 B（Tank メッシュ）が交差する場合の BOOL 値格納用です。
vecEnd.z+=1.0f;
終点の Z 成分を 1 単位増加させます。ここで増加させる値は正の値であればなんでも良く、0.1 単位でもいいですし、1000 単位でもとにかくプラスの値を増加させます。Z 成分を増加させる目的は方向を持たせるためであり、どんな値でもいいのです。レイは変位ベクトルですから方向が定まればそれで目的は達成されます。
D3DXMATRIX matWorld;
D3DXMatrixTranslation(&matWorld,pThingB->vecPosition.x,pThingB->vecPosition.y,pThingB->vecPosition.z);
D3DXMatrixInverse(&matWorld,NULL,&matWorld);
D3DXVec3TransformCoord(&vecStart,&vecStart,&matWorld);
D3DXVec3TransformCoord(&vecEnd,&vecEnd,&matWorld);
この部分は、レイを正しく対象に当てるコツとも言える部分です。この部分が無いと、対象メッシュ（Tank メッシュ）が動いた場合、レイが正しく当たらず、したがって正しい交差判定が出来ません。本サンプルにおいて Tank メッシュは静止しているので問題ないのですが、一般的にはこの部分の処理が必要になります。
この部分の処理は
“ 対象メッシュのワールドトランスフォーム行列の逆行列をレイに掛ける ” 処理です。
こうすれば、対象メッシュが動いていても、正しくレイが当たります。
vecStart と vecEnd に、逆行列を掛けています、これはレイに逆行列を掛けていることを意味します。
vecDirection=vecEnd-vecStart;
最終的に求めた始点と終点からレイを求めます。
D3DXIntersect(pThingB->pMesh,&vecStart,&vecDirection,&boHit,NULL,NULL,NULL, pfDistance,NULL,NULL);
D3DXIntersect 関数は前述した通り、レイとメッシュの交差を調べる関数です。メッシュポインタ pThingB->pMesh とレイの始点 vecStart、レイ vecDirection、交差の有無 boHit、始点から交点への距離 pfDistance を引数に渡しています。交差する場合は、boHit に TRUE が入り pfDistance に距離が入って帰ってきます。

以上が、レイによる衝突判定の手順です。
スペースキーにより、レイのレンダリングを有効・無効にできますので適宜切り替えてください。

Chapter14　アニメーションメッシュの扱い

他章で扱うジオメトリは、全て単一のメッシュでした。ジオメトリをアニメーションさせる場合は、複数のメッシュを一つのジオメトリとして扱います。単一のメッシュでもメッシュ自体を変形させてアニメーションを実現させる方法※がありますが、本書では扱いません。

メッシュの階層化と相対姿勢

戦車を例にすると、メインボディ、キャタピラー、砲台…などをパーツとして分けて考えます。各パーツがメッシュということです。

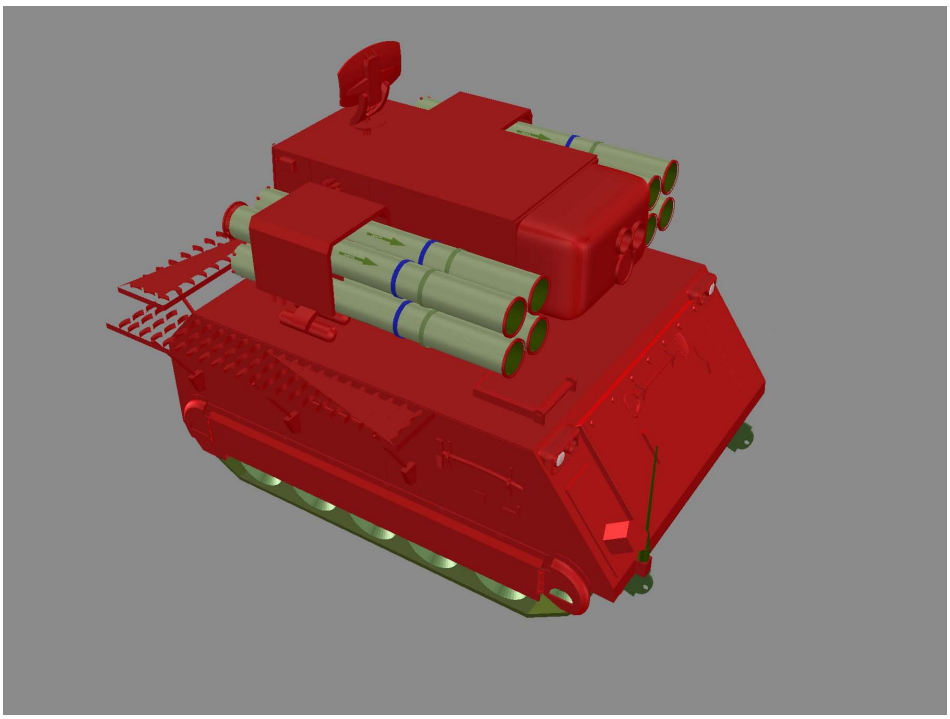


図 14-1

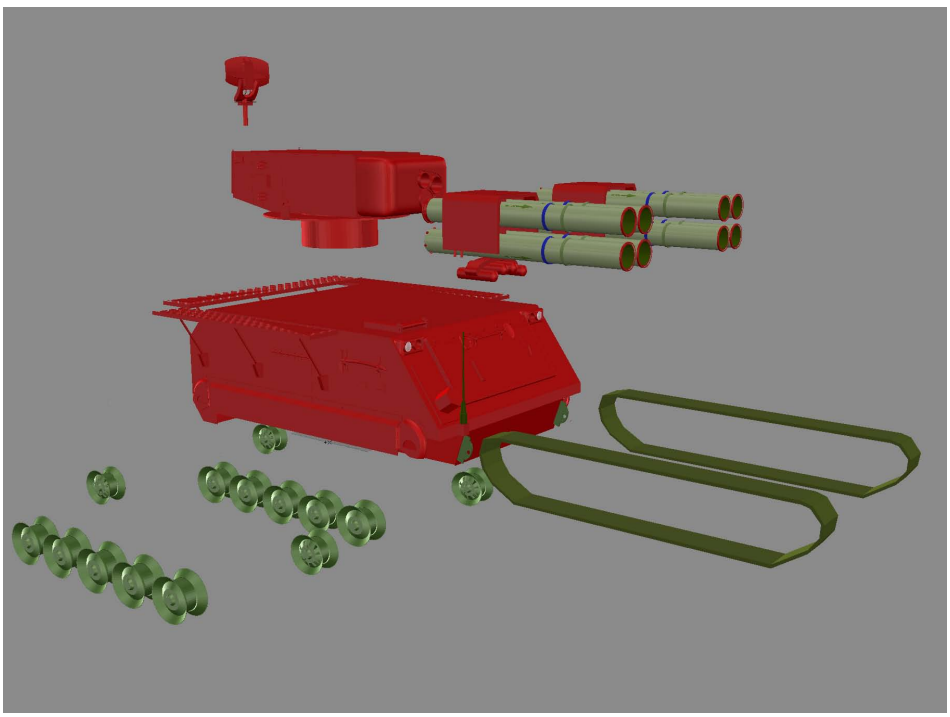


図 14-2

アニメーションメッシュの場合、各パーツメッシュは、階層（Hierarchy: ヒエラルキー）関係にあります。

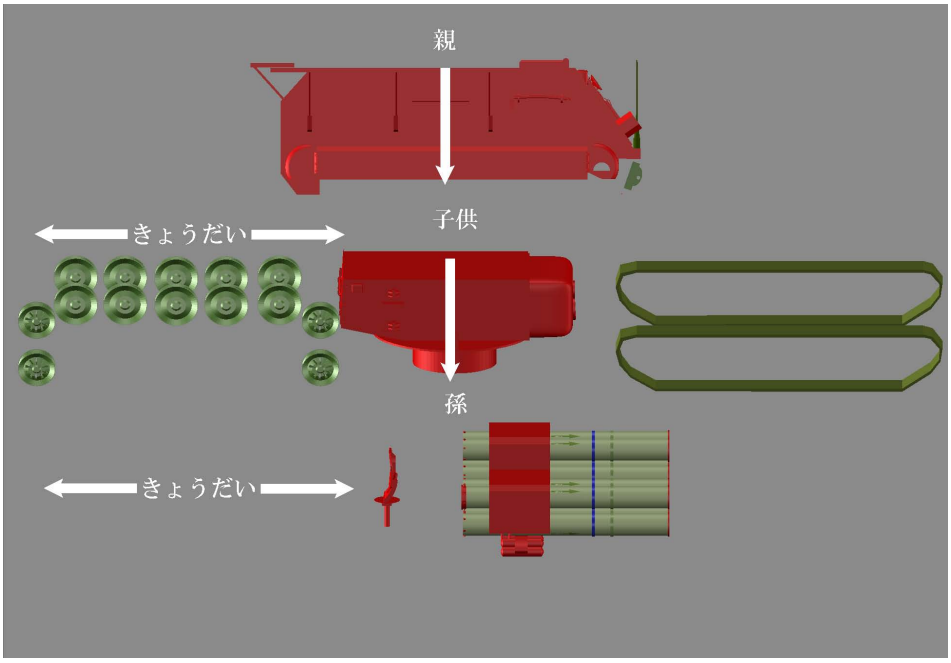


図 14-3

階層とは、親子関係及び兄弟関係を設定することです。親メッシュの動きや姿勢は子メッシュに影響を与え、親メッシュの動きに連動して子メッシュも動きます。兄弟同士は影響し合いません。
階層にする理由は、メッシュの姿勢を相対的にすることであり、相対姿勢にすることにより効率的にアニメーションが実現できるからです。
例えば、戦車全体が回転する場合、階層関係ではなく各メッシュが独立していると、全てのパーツについて回転を掛けなくてはなりません。また、戦車全体の回転と同時に砲台を回転させたい場合、階層関係にないと砲台の回転は全体の回転を含めて計算しなくてはなりません。
階層関係にある場合、砲台をメインボディの子に設定しておけば、砲台は砲台自身の回転をかけてやるだけでメインボディの回転と合成された回転をしてくれます。
実際に Chapter14-1 サンプルで、戦車（ミサイルランチャー）アニメーションメッシュにより、親メッシュに子メッシュが連動する様を確認してみてください。

階層の展開

アニメーションメッシュは、各メッシュが階層関係にあるため、読み込みとレンダリング時には、階層を展開してやる必要があります。階層を展開してやれば、あとは単一メッシュと同じように各メッシュを処理してやればいいのです。階層関係にあるため一手間かける必要がありますが、あとは通常のメッシュとして扱えますので、さほど難しいことではありません。
階層の展開は、関数を再帰させることで簡単かつスマートにコーディング出来ます。
次のコードは、Chapter14-1 から、メッシュをレンダリングするコードを抜粋したものです。

```
VOID DrawFrame(LPD3DXFRAME pFrame)
{
    LPD3DXMESHCONTAINER pMeshContainer;
    pMeshContainer = pFrame->pMeshContainer;
    while (pMeshContainer != NULL)
    {
        RenderMeshContainer(pMeshContainer, pFrame);
        pMeshContainer = pMeshContainer->pNextMeshContainer;
    }
    if (pFrame->pFrameSibling != NULL)
    {
        DrawFrame(pFrame->pFrameSibling);
    }
    if (pFrame->pFrameFirstChild != NULL)
    {
        DrawFrame(pFrame->pFrameFirstChild);
```

```
    }
}
```

DrawFrame が再帰関数であることはすぐに分ると思います。コード中に出てくる pFrame->pFrameSibling と pFrame->pFrameFirstChild が階層関係を表しています。Sibling ※は “ きょうだい “ という意味です。Child は子を表しているのは分るでしょう。きょうだい親子関係を手掛かりに、関数を再帰させると、階層を辿ることになり結果的に階層を展開していることになります。レンダリングに限らず、読み込み、さらには衝突判定の際にも階層を展開する必要がありますが、階層展開部分は全く同じです。その階層展開部分以外の部分でそれぞれ異なるコード（処理）を書いているにすぎません。
例えば、読み込み、レンダリング、そして衝突判定という 3 つの処理について、それぞれの処理固有の処理は単一メッシュの場合と同じであり、その下の部分に再帰判断部分であり、階層を走査している部分です。

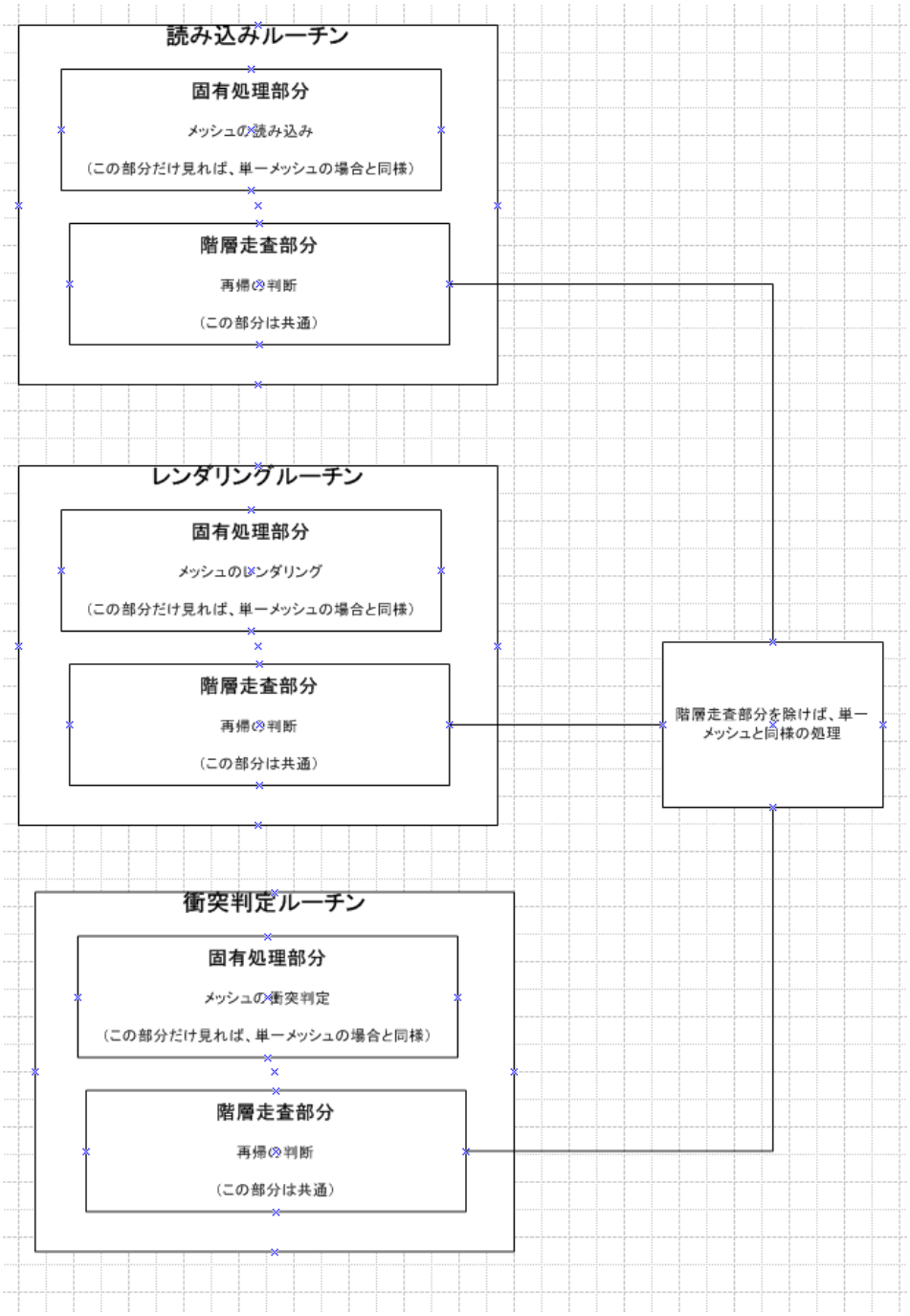
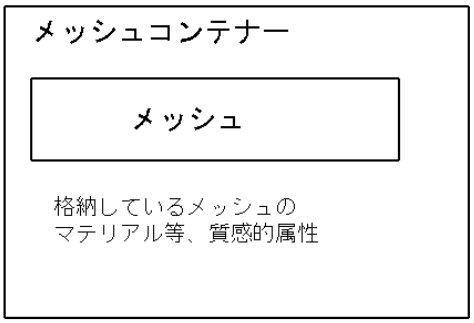


図 14-4
階層を展開していく中で各パーツメッシュに対しての処理は単一メッシュ処理と同様です。単一メッシュが理解できているならば、アニメーションメッシュを理解するということは、階層とその展開を理解することに尽きると言ってもいいでしょう。ただ、Direct3D は、アニメーションメッシュの読み込みに関して、開発者のアニメーション仕様を柔軟に組み込めるようにしているため、コードは若干複雑になりコード量も多くなってしまいます。これについては、すぐ後に解説します。

メッシュコンテナーとフレーム

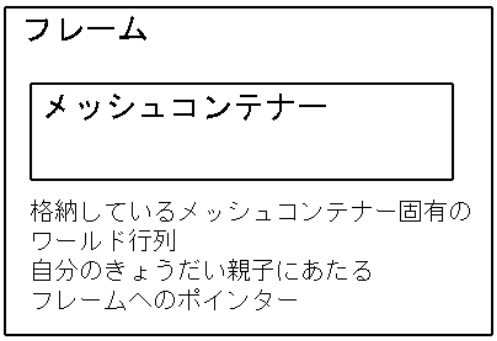
コードを読み下すためには、メッシュコンテナーとフレームという２つの概念について理解する必要があります。難しいことはありません２つとも単なる入れ物（構造体）にすぎず、開発者がよくやるカプセル化を最初から Direct3D で定義しているだけのことです。
メッシュコンテナーとフレームは、パーツメッシュとその属性をまとめて格納する入れ物という点では同じですが、メッシュコンテナーはフレームに含まれます。もし読者が、IFF ファイルフォーマットを知っているのであれば、フレームをチャンク、そしてメッシュコンテナーをサブチャンクと考えれば、当たらずとも遠からずといったところです。
大雑把に言うと、メッシュを一番小さい入れ物とすると、メッシュコンテナーが中くらいの入れ物、フレームが一番大きい入れ物です。

メッシュコンテナーとは、メッシュとその属性を格納する入れ物です。
各パーツメッシュがその属性（マテリアルなど）を自分自身で持っていると処理がし易いのですが、メッシュはジオメトリデータしか格納できないので、コンテナーという入れ物を新たに作ってメッシュとその属性を格納するわけです。したがって、メッシュがメッシュコンテナーに格納された後はメッシュコンテナーが使用されます。




```
typedef struct _D3DXMESHCONTAINER
{
  LPTSTR Name;
  D3DXMESHDATA MeshData;
  LPD3DXMATERIAL pMaterials;
  LPD3DXEFFECTINSTANCE pEffects;
  DWORD NumMaterials;
  DWORD *pAdjacency;
  LPD3DXSKININFO pSkinInfo;
  struct _D3DXMESHCONTAINER *pNextMeshContainer;
} D3DXMESHCONTAINER, *LPD3DXMESHCONTAINER;
```

図 14-5
フレームとは、メッシュコンテナーとその属性を格納する入れ物です。
メッシュコンテナーがメッシュの質感的属性を格納するのに対し、フレームはメッシュの姿勢属性（ワールド行列なそ）をメッシュコンテナーと併せて格納します。
フレーム自身も親子関係により入れ子状になることもあり、アニメーションメッシュであれば通常はフレームが入れ子状になります。単一メッシュプログラムでは、メッシュがジオメトリの基本単位であるのに対し、アニメーションメッシュプログラムではフレームがジオメトリの基本単位となるわけです。




```
typedef struct _D3DXFRAME
{
  LPTSTR Name;
  D3DXMATRIX TransformationMatrix;
  LPD3DXMESHCONTAINER pMeshContainer;
  struct _D3DXFRAME *pFrameSibling;
  struct _D3DXFRAME *pFrameFirstChild;
} D3DXFRAME, *LPD3DXFRAME;
```

図 14-6
メッシュ、メッシュコンテナー、フレームの包含関係図です。

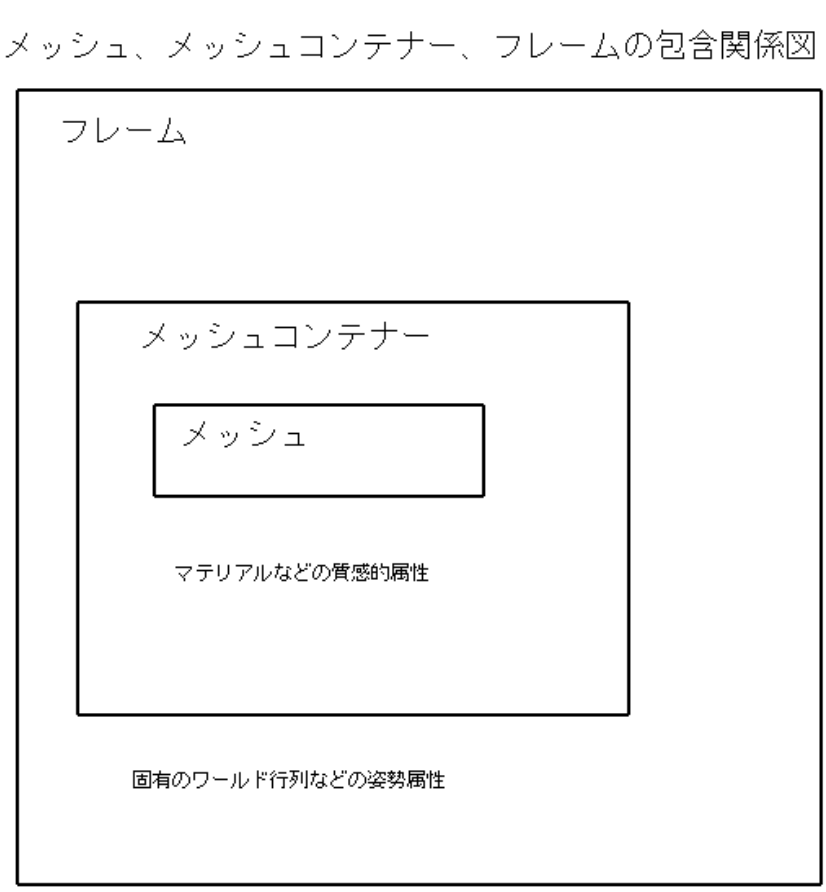


図 14-7

X ファイルからアニメーションを読み込む手順

前述のとおり、Direct3D において、アニメーションの読み込みは、少々複雑です。単一メッシュのときのように D3DXLoadMeshFromX 関数一発で読み込めるようにはなっていません。そのため、読み込みルーチンの作成と解説はどうしても長くなってしまいます。しかし、ここで心に留めておいていただきたいのは、読み込みルーチンが複雑なのは、あくまでも Direct3D の仕様の問題であり、本書で伝えたいアニメーションの本質ではありません。この部分で惑わされると階層や再帰によるアニメーションの本質を見失う危険性があります。もし本節を読み下すのが困難であると感じたなら、すぐに読み飛ばし、次節レンダリング手順に進んでください。読み込み手順を知る必要がある場合に読み直してもいいのですから。

手順の概要

手順は複雑なため、先に手順の骨格をざっと述べてから、詳細を説明します。
X ファイルからアニメーション階層を読むには D3DXLoadMeshHierarchyFromX という関数を使用しますが、この関数を使用するには事前準備が必要です。まず、事前に階層を展開する ID3DXAllocateHierarchy というクラス（インターフェイス）インスタンスを生成してそのポインターを渡さなければなりません。ところが、この ID3DXAllocateHierarchy クラスの主要なメソッド（関数）は仮想関数であるため、使用するには開発者自らが実装しなければなりません。なぜ、そのような仕組みなのかということは謎ですが、おそらく開発者の仕様を柔軟に取り入れることが出来るようにしたということでしょう。柔軟性の代償として、メソッドの実装という手間がかかるわけです。ID3DXAllocateHierarchy を使えるようにコーディングした後にはじめて D3DXLoadMeshHierarchyFromX が使えます。
では、ID3DXAllocateHierarchy クラスの、どのメソッドを、どのように実装すればいいのか説明します。なお、サンプルでは、ID3DXAllocateHierarchy を派生させ新たに MY_HIERARCHY というクラスを定義していますが、実装するメソッドと実装コードはもちろん同一です。Hierarchy.cpp 実装ファイルがその実装を全て行っています。

手順の詳細

実装が必要なのは、次の 4 つのメソッドです。
ID3DXAllocateHierarchy::CreateFrame メソッド
ID3DXAllocateHierarchy::CreateMeshContainer メソッド
ID3DXAllocateHierarchy::DestroyFrame メソッド
ID3DXAllocateHierarchy::DestroyMeshContainer メソッド

メソッド名からも分るように、各メソッドの目的・機能は次の通りです。
CreateFrame

フレームのメモリーを確保して、初期化する。
CreateMeshContainer
メッシュコンテナーのメモリーを確保して、初期化する
DestroyFrame
フレームをメモリーから開放する。
DestroyMeshContainer
メッシュコンテナーをメモリーから開放する。

CreateFrame の実装
CreateFrame は、アニメーションを X ファイルからロードする際に D3DXLoadMeshHierarchyFromX から呼ばれるコールバック関数ですので引数は決まっています。引数は 2 つで、X ファイル内のフレーム名と初期化されてないフレームポインターのアドレスです。
処理内容は、実装者によって自由ですが、基本的にすべき最低限の処理はほとんど変わらないでしょう、サンプルは基本的な処理をしています。

まず、フレーム構造体のメモリーを確保・初期化して、
フレーム名を保管し、
引数のフレームポインターが、確保したメモリーを指すようにしてやる。

というのが、基本的な処理で、これは誰が書いてもほぼ同一になるはずです。
Direct3D には D3DXFRAME というフレーム構造体が定義されていますが、本サンプルでは、各パーツメッシュに固有のワールド行列 CombinedTransformationMatrix を追加する必要があるため、この D3DXFRAME の派生型 MYFRAME を定義しています。このような行列は最初から D3DXFRAME に定義されていても良さそうなものなのですが、無いので派生させています。

CreateMeshContainer の実装
CreateMeshContainer も、アニメーションを X ファイルからロードする際に D3DXLoadMeshHierarchyFromX から呼ばれるコールバック関数ですので引数は決まっています。引数は 8 個もありますが、サンプルはそのうち基本的なもの 5 個を使用しています。その 5 個とは、
Name　メッシュコンテナー名
pMeshData　メッシュデータ
pMaterials　マテリアルポインター
NumMaterials　マテリアルの数
ppMeshContainer　初期化されていないメッシュコンテナーポインターのアドレス
の 5 つです。

これも仮想関数なわけですから処理内容は、最低限の処理を行えばあとは実装者の自由です。サンプルは基本的な最低限の処理をしています。

まず、メッシュコンテナーのメモリーを確保と初期化をして、
メッシュコンテナーの名前を保管し、
マテリアルのメモリー確保と初期化し、
引数として与えれれたポインターのが関数内で初期化したコンテナーメモリを指すようにしてやります。
フレームでもそうでしたが、Direct3D に最初から定義されている D3DXMESHCONTAINER 構造体も、在って欲しい変数が無かったりします。テクスチャー用のポインターは定義されているのですが、単なるポインターなのでテクスチャー 1 枚にしか対応できません。複数枚のテクスチャーに対応するにはポインターのポインターが無ければならないのです。ですので、ここでも派生型である MYMESHCONTAINER 型を定義して、ポインターのポインターを追加しています。

CreateFrame 関数と CreateMeshContainer 関数の処理が、処理内容がほぼ同一なのが分るでしょうか、2 つとも、メモリー確保とその初期化がメインの処理であり、最低限実装しなければならない処理です。

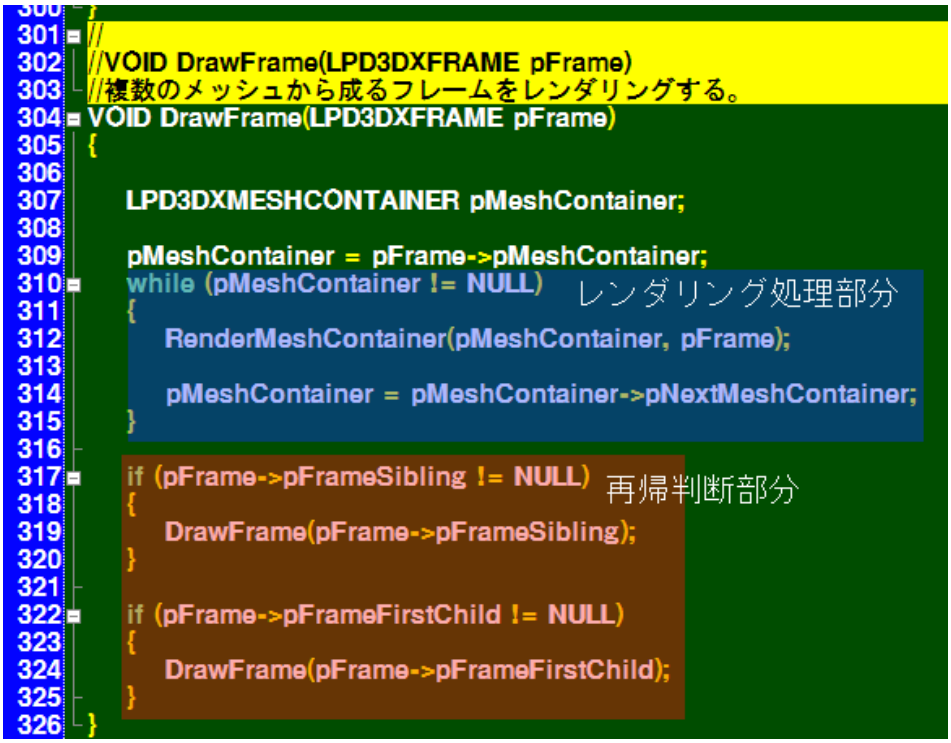
DestroyFrame と DestroyMeshContainer は、単なる開放関数なので、解説は割愛します。

これらの実装をしたら、あとは D3DXLoadMeshHierarchyFromX で X ファイルを読み込めば、アニメーション関連の初期化も全て Direct3D がやってくれます。
なお、今までのサンプルと違い、階層の読み込み時に各メッシュのマテリアルやテクスチャーを初期化しているので、InitThing 関数は逆にシンプルになっています。

アニメーションレンダリング手順

アニメーションのレンダリング関数は再帰関数になっています。その時点での階層ノードのメッシュをレンダリングしていき、最後の階層ノードに達するまで、言い換えると全てのパーツメッシュをレンダリングするまで自分自身をコールし続けます。再帰判断部分（階層走査部分）を除外して考えると、単一メッシュのレンダリングであることがわかります。この階層判断部分のコードは、きょうだい親子関係を辿るコードとも言えます。この部分はサンプル内の再帰関数すべてに共通のコードです。フレーム内のメッシュ毎にワールド変換行列を更新する関数である UpdateFrameMatrices 関数を見ていただければ分るように、再帰判断部分のコードは同じです。

サンプル実行中はスペースキーで、アニメーションの再生・一時停止が出来ますので適宜確認してください。




```
301 //
302 //VOID DrawFrame(LPD3DXFRAME pFrame)
303 //複数のメッシュから成るフレームをレンダリングする。
304 VOID DrawFrame(LPD3DXFRAME pFrame)
305 {
306
307     LPD3DXMESHCONTAINER pMeshContainer;
308
309     pMeshContainer = pFrame->pMeshContainer;
310     while (pMeshContainer != NULL)
311     {
312         RenderMeshContainer(pMeshContainer, pFrame);
313
314         pMeshContainer = pMeshContainer->pNextMeshContainer;
315     }
316
317     if (pFrame->pFrameSibling != NULL)
318     {
319         DrawFrame(pFrame->pFrameSibling);
320     }
321
322     if (pFrame->pFrameFirstChild != NULL)
323     {
324         DrawFrame(pFrame->pFrameFirstChild);
325     }
326 }
```

図 14-8

アニメーションメッシュの衝突判定

Direct3D はアニメーションの読み込みのコードが長くなってしまうので、読み込み関係のほうが多くの紙面を割いてしまいましたが、本来、本章が伝えたいことは、レンダリングと、本節の衝突判定であり、それらがアニメーション階層を展開・走査している以外は、単一メッシュとほとんど変わらないということなのです。
再帰判断部分は、本サンプル他の再帰関数と同じです。つまり、メッシュコンテナーのきょうだい親子情報を手掛かりに再帰して、階層を辿っていくというものです。
今、仮に、再帰判断部分が無いものとしてコードを見てください。そうして見ると、Chapter13 の衝突判定コードほとんどそのままであることがわかると思います。
再帰させる理由は、全てのパーツメッシュに対して D3DXIntersect を実行することです。
さて、これまでに読み込み、レンダリング、衝突判定の各段階で関数を再帰させて階層を走査するコードが出てきました。なにか、別の処理でもアニメーション階層を走査する必要があるときは、その処理固有のコードの下に再帰判断コードを加えれば、階層を辿り、アニメーションを構成するメッシュごとに、その処理をしてくれるようになります。これが、アニメーションメッシュを扱うコツであり、階層構造を扱うコツです。本章では、このことを一番に伝えたかったのです。

レイによる衝突判定の優位性

さて、ここでレイによる判定が、境界ボリューム方式よりも優れている点を、サンプルで解説します。
レイを戦車のレーダー付近になるように移動したのが図 14-9 です。この瞬間では、レイはレーダーに当たっていません、アニメーションが進むにつれレーダーが回転し、その結果レイに当たった瞬間が図 14-10 です。このように、レイによる判定は非常に正確で、きめ細かい判定が可能です。一方、境界ボリューム方式では、レーダーが回転していようが、常にボリュームで判断してしまうので、レイ方式よりも大雑把な判定になってしまいます。

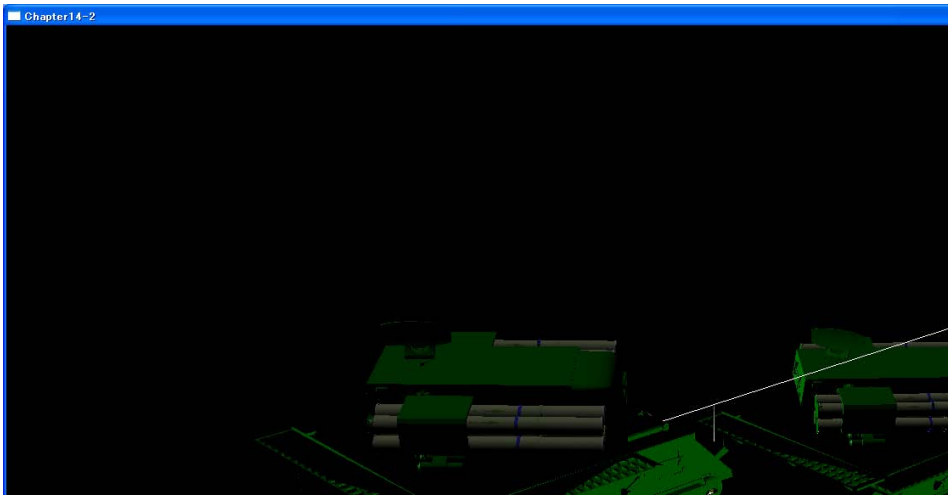


図 14-9

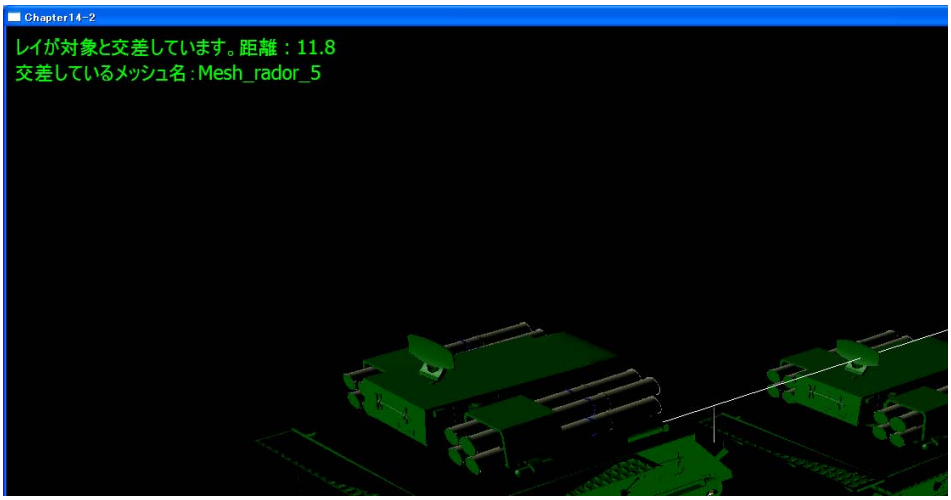


図 14-10
サンプル実行中はスペースキーで、レイのレンダリングを有効・無効にできますので適宜確認してください。

Chapter15　固定文字列高速描画クラス

文字列の描画は、いろいろな状況で考えられます。ロールプレイングゲーム等の台詞、ゲーム中に表示する操作説明、進行状況の表示、また開発中に変数の値などを表示したいときにも文字列を描画します。
文字列の描画においては、当然そのテキストデータが必要ですが、テキストデータの形態によって大きく２つに分けることができます。1 つは、データが動的なもの、2 つ目はデータが静的なもの、静的とは固定されたテキストデータということです。
動的なテキストデータは、例えば、デバッグ時に変数の値を表示するときの変数名や値、あるいはチャットでの入力文字の描画がそうです。これらは、動的に変化し、固定的ではないため事前にデータを用意することはできません。
静的なテキストデータは、例えば、台詞やメッセージ等で、そのテキストデータが変化しないもの、言い換えれば、固定のデータとしてあらかじめ用意することができるものです。
Direct3D には、ID3DXFONT というインターフェイスがあり、これには DrawText というテキストを簡単に描画するメソッドが用意されています。ID3DXFONT は、GDI を利用してテキストを描画するもので、固定テキストのみならず動的なテキストデータも描画できるため、特に一時的なデバッグ情報の描画などに重宝します。

ID3DXFONT インターフェイスの問題点

ID3DXFONT は、GDI を経由してフォントのアウトラインを生成し、それを描画するのですが、このアウトライン生成が低速であるということと、さらに GDI を介すること自体が大きなタイムロスとなるため、ID3DXFONT による文字列描画は非常に重たい処理になってしまいます。はじめてそのパフォーマンスを見たときは、「ここまで、重くなくても…」と思ってしまうほど劇遅ですが、何万種類もあるフォントの動的検索を毎回伴うのですから重たいのも当然であり、むしろ、その処理内容にしては高速なほうなのかもしれません。いずれにしても、リアルタイムゲームの完成バージョンでは使用を控えざるを得ませんが、文字列の描画を控えるのは無理がありますので、そこにジレンマが発生します。文字列描画についてなんらかの代替手段を考える必要があります。

“ もじもじ君 ” の誕生

そこで、筆者は文字列を高速に描画するルーチンをクラス形式で作りました。クラス名は “ もじもじ君 ” です。クラスを擬人化していますが、筆者のクセなので意味はありません。
もじもじ君を作成したそもそもの動機は、筆者が非常勤講師をしているゲームプログラミングコースでの話になります。コースの学生同士２，３人でチームを組み、ゲームを共同制作するという単元で、あるチームが興味深いゲームを企画しました。そのゲームは、マルチシナリオのアドベンチャーシステムで２D パートだけでなく、３D パートもあるという企画です。そのコースでは他の学生も含め、それまでも文字列のレンダリングには ID3DXFONT を使用していたのですが、その実行速度の遅さからリアルタイムパートでの使用を避けている向きがありました。しかし、そのチームの企画では３D パートでも文字列をレンダリングしなければならなかったので、どうしても高速な描画ルーチンが不可欠だったのです。そこで、このもじもじ君を作ろうと思ったわけです。実際、ゲーム内ではもじもじ君が使用され、全体のパフォーマンスを落とすことなく文字列を描画することに成功しました。これは余談ですが、チームメンバーの努力とゲームシステムの斬新性により、そのゲームは晴れて学校主催のコンテストで総合最優秀賞を獲得し、もじもじ君が多少でも貢献したことを嬉しく思った思い出があります。

インサイド “ もじもじ君 “

まず、もじもじ君はクラスとして設計したので、本書の他のサンプルとは書式が異なります。読み下すにはクラスの書式に慣れていることが望ましいのですが、コード量が多いため、コードを読み下すかどうかは、読者に委ねます。また、筆者としてもコードの細かい解説は避けます。なぜなら解説も長くなりますが、それ以上に解説を読むのが大変だと思うからです。ここでは、その仕組みと利用方法を解説します。

処理の概要

仕組みを簡単に述べると、
テキストデータを別ファイルで用意しておく。
ゲームの初期化タイミングに、ファイルからデータを読み込み、文字毎のフォントイメージを２D スプライトとしてとして作成する。
描画する際には、そのスプライトを単純に描画する。
というものです。
動的にフォントを作成するのではなく、初期化時にあらかじめフォントをスプライトとして用意しておけば、あとは単純な２D 描画なので高速に描画できるというわけです。３D ジオメトリに例えるなら、ジオメトリをリアルタイムレンダリングするのではなく、あらかじめレンダリングした “ 静止画 “ を単純にブリットするということです。ファイルとして最初から用意するので動的に変化するテキストは扱えません。したがって、デバッグ時の変数名や値、及び、チャットでの入力文字の描画等は出来ません。あくまで、事前にファイルに記録しておくことが出来る固定文字列の描画がその機能です。
なお、テキストデータ内の文字列は文字単位でスプライトサーフェイスに隙間無く記録され、全データ内の同じ文字は一つのスプライトとして記録します。したがって、例えば “ あああ…” と “ あ ” が 1 万個のテキストデータと、“ あ ” が 1 個のテキストデータは、必要なサーフェイスメモリは全く同じです。
なお、固定文字列の描画であれば、文字列を静止画として用意してそれを描画している読者もいると思いますが、文字列が例えば本一冊分もある場合や、開発中にテキストを頻繁に変更する場合にはお手上げです。もじもじ君はテキストファイルにテキストデータを用意してやれば、その静止画を作成してくれます。そうゆう部分では動的であり、柔軟性があります。

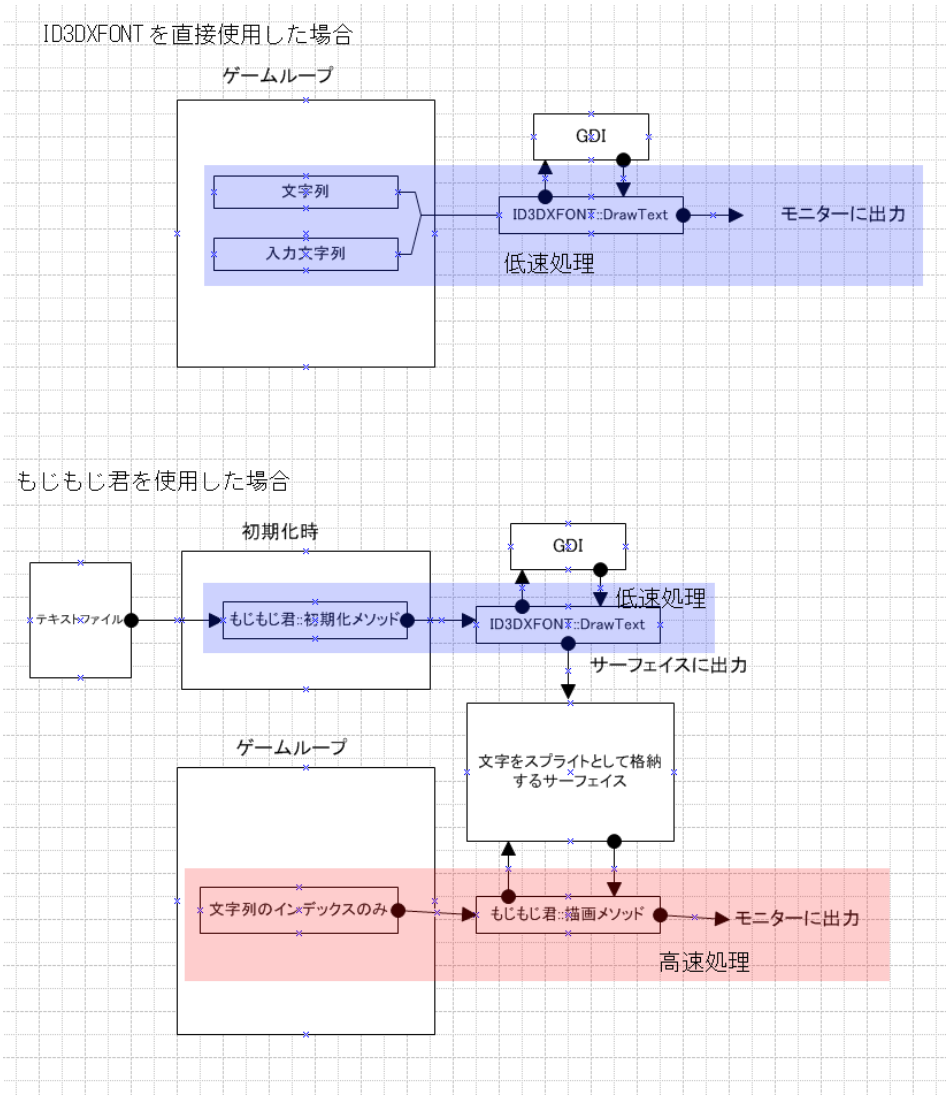


図 15-1

開発者としての使用方法

サンプルは、何も無いウィンドウに固定テキストデータを描画するだけの単純なもので、マウスの左クリックで“表示スピード”を一時的に速くなるようにしています。表示スピードは描画スピードのことではありません。ここでの表示スピードとは文字が現われる速さという意味です。文字列を 1 行表示する際に“左から、だんだん文字が現われるように”描画するというのがもじもじ君の仕様です。これは、アドベンチャーゲームの台詞の表示を意識した仕様です。行を一瞬で表示することは出来ますが（むしろ、そのほうが簡単ですが）、それでは、味気ないものとなってしまいます。もし、読者がもじもじ君を自分のプログラムで使用する際に、そのような演出効果が必要ない場合にはコードに手を加えてください。文字の表示仕様の変更であれば、簡単に出来るはずです。
描画スピード自体は先にも述べたとおりかなり高速です。例えば、文字を画面全体にびっしり描画する処理をもじもじ君で行った場合、フレームレートはほとんど変化しません。同じことを ID3DXFONT で行ったならば、おそらくプレイ出来ないほどフレームレートが落ちるでしょう。
さて、読者のプログラムにもじもじ君を取り入れる手順は簡単です。
まず、ヘッダーファイル CMOJIMOJIKUN.h と実装ファイル CMOJIMOJIKUN.cpp をプロジェクトに追加します。そして、もじもじ君を使用する実装ファイル内でヘッダーファイルをインクルードします。これで、ファイル的な準備は完了です。
次は、もじもじ君の初期化コードを追加します。もじもじ君はコンストラクタで初期化せずに初期化メソッドにより初期化を行いますので、インスタンスを生成した後にその初期化メソッド CMOJIMOJIKUN::Init をコールします。その書き方は、サンプルから抜粋すると次のとおりです。

```
// もじもじ君インスタンスの初期化
        MOJIKUN_INIT mi;
        mi.hWnd=m_hWnd;
        mi.pD3d=m_pcDx9->pDx9;
        mi.pDevice=m_pcDx9->pDevice;
        mi.boWideCharacter=TRUE;
        mi.wPosX=0;
        mi.wPosY=20;
```

```
	mi.wAltPosX=0;
	mi.wAltPosY=120;
	mi.boFullScreen=FALSE;
	mi.wWindowWidth=WINDOW_WIDTH;
	mi.wWindowHeight=WINDOW_HEIGHT;
	mi.bMaxFontWidth=14;
	mi.bMaxFontHeight=20;
	mi.dwFontColor=0xff00ff00;
	mi.szTextPath="test.txt";
	if(FAILED(cMojimojiKun.Init(&mi)))
	{
		MessageBox( 0," もじもじ君の初期化に失敗 ","",MB_OK);
		return E_FAIL;
	}
```

MOJIKUN_INIT 構造体の各メンバーに値を入れて、構造体のポインターを Init 関数に渡せば、初期化終了です。
各メンバーの意味は、上から順番に、
ウィンドウインスタンスハンドル
ダイレクト３D インスタンスのポインター
ダイレクト３D デバイスインスタンスのポインター
マルチ文字（日本語）か ANSI 文字（英数字）か
描画開始スクリーン座標（X 座標）
描画開始スクリーン座標（Y 座標）
選択モード時の描画開始スクリーン座標（X 座標）
選択モード時の描画開始スクリーン座標（Y 座標）
フルスクリーンかウィンドウモードか
ウィンドウの横幅
ウィンドウの縦幅
フォントの横幅
フォントの縦幅
フォントの色と透明度　ARGB 値
テキストファイルのパス

初期化が終われば、テキストファイルからテキストデータを読み込みます。
読み込んだデータは 1 行単位で、もじもじ君に登録します。登録は CMOJIMOJIKUN ::WriteTextStringToSurface メソッドにより行います。第 1 引数は、文字列ポインター、第 2 引数は登録フラグです。登録フラグは、文字データを一まとまりの段落としたい場合は WORDS_BLOCK を通常は NULL としてください。読み込みはクラスの外で行うものなので、読み込みルーチンは、適宜作成してください。または、サンプルの CMYAPPLICATION::LoadText() をそのまま使用するか、参考にしてください。サンプルは、もじもじ君を使用することだけが目的なシンプルなものなので参考になると思います。
以上で、全ての準備が完了しました。
運用形態は、必要に応じて CMOJIMOJIKUN::ChangeDrawSpeed により表示スピードをセットしながら CMOJIMOJIKUN::DrawBlock メソッドによって描画をしていく形となります。

Section 3

セクション１とセクション２において解説したことは、他の既存の書籍やインターネットからでも得ることができるものであり、目新しいものではありません。ただ、従来、一冊の書籍から得ることの出来ない情報を、本書ではゲーム開発の学習過程に沿って、必要な情報を１冊にまとめたという自負はあります。またそれは本書１冊で、初心者がゼロからのゲーム開発を開始できるようにと意識した結果でもあります。
そして、このセクション３は、既存の書籍では解説されていない高度な内容を扱います。
このセクションは、次の３つの技術について解説します。

1. 障害物回避システム
2. 通信対戦時におけるゲーム状態同期システム
3. ３D ダイナミクスシステムの基礎

この３つは､もはやセクション２のような小技的なものではなく高度なアルゴリズム及びその組み合わせとしての “ システム ” であり、ソースコードの規模は、かなり大きなものとなります。ここまでの規模になると、ソースコードを見ながら読み下すというアプローチは、不可能ではないものの非常に困難であり効率的とは言えません。また、一般的にシステムが高度になればなるほど、その目的に特化したロジックが絡み合い、汎用性が低くなります。したがって、複雑なロジックをソースコードで理解しようとすることは、仕組み・原理全体としての理解を阻害する危険性もあります。
本セクションは、ソースコードは見ません。システムの考え方や仕組みの解説をしていきます。
さて、原理や考え方を解説するには、その原理が実際に実現可能でなければならず、実際にコーディングされたプログラムが存在するという裏付けがなくてはなりません。本セクションで解説することは、すべて実際にコーディング及びビルドされ、その出力結果を確認しているものです。そして、これらのシステムは、市販レベルのゲームを作るのに必要なクオリティーを持ち、実際市販されたゲームに使用したエンジンです。その開発には多大の労力と時間を費やしたものであり、筆者は、これら３つの開発のためだけに丸１年程度はかけています。
障害物回避システムはミドルウェアとして利用できる KamPass.dll というダイナミックリンクライブラリを添付ディスクに収録しています。読者が自身のゲームプログラムにそれを組み込めば、開発に時間がかかり、また開発が困難とも言える障害物の回避思考ルーチンを自作する必要はありません。例えば、キャラクターが自らの思考でダンジョン内を滑らかに移動するのを苦労せずとも実現できます。
通信対戦同期システムと３D ダイナミクスは、コードは添付しません。同期システムはサンプルゲーム CompanyWars2004 での技術を解説するのですが、ライブラリとして分離するのが非常に困難であったのがその理由です。
３D ダイナミクスに関しては、現在進行形で筆者が開発中のゲームで使用しているものであり、これもライブラリ化するのには多大な時間を要するので、ここでは３D ダイナミクスを構築する際の必要な知識の解説となります。

Chapter16　障害物回避システム

障害物回避システムがどのような状況で必要になるかというと、例えば、本書サンプルゲーム CW2004 のようなリアルタイムストラテジーゲームにおけるユニットの移動がそうです。また、3D シューティングにおいても敵の移動思考には必要です。いずれにしても、特にコンピューターによるユニットの移動に必要となります。ユニットに目標地点を与えたら、ユニットは、その地点まで障害物をすり抜けることなく“回避“しながら自分の意思・思考で自動的に進み、目標地点にたどり着かなければなりません。既存のゲーム、特にコンソールマシン（家庭用ゲーム機）用のゲームでは、なかなかそのような思考を持たせた動きは見当たりません。PC ゲームであれば、おもにアメリカ産のストラテジーゲームで回避思考によるユニットの動きが見られますが、エレガントな思考のものは少ないようです。ユニットが障害物に引っかかったり、迷路状になった場所を行ったり来たりしてしまうことがままあります。また、3D シューティングで、相棒キャラ（主人公に追従して行動する見方キャラ）が進路を阻んで身動きが取れなくなるなんていうこともあります。障害物を回避するということは、迷路を解くことでもあります。システムは迷路を解けなくてはなりません。なぜなら、障害物が迷路状になることが十分考えられるからです。ユニットがたくさんある場所だったり、ユーザーがマップエディターで作成した結果、マップが迷路状になる場合だってあります。迷路を解けないシステムならば、障害物を完全に回避できず、途中で動きが固まることになります。障害物回避は、それだけ難しい処理なのです。
障害物が何もない場所を移動するのであれば、コードは 1 行で済みます。
(x,y) 自分の座標　(tx,ty)　目標地点の座標
x+=(x<tx)-(x>tx);
y+=(y<ty)-(y>ty);
などとすればいいだけのことです。
ところが、障害物があるとなると話は全然違ってきます。障害物を回避しながら目標にたどり着くような何らかの思考をさせる必要があり、コードは数千数万行になります。
障害物には色々なものが考えられます。2D ゲームでは、岩や木、建物、敵のユニットなど、3D シューティングでも建物や建物内部の壁、敵プレイヤー、主人公の相棒キャラが居る場合、その相棒も障害物です。とにかく、動かすキャラやユニット以外は基本的に障害物ですので、それを回避させないと、すり抜けてしまい不自然です。逆に言えば、自分以外の物を「障害物とする」必要があります。
現在地点から目標地点までの道すじをパスと言います。パスの意味としては道すじ、進路という意味であり、英語で言えば Path あるいは Pass になります。Path の方が直接的な意味を表していますが、ファイルパス FilePath と混同するためソースコード上は Pass の方を使っています。障害物回避システムはパス探索システムあるいはパス決定システムとも言えます。
パスを求めるアルゴリズムは、幾つかあるのでしょうが、高速な処理を要求されるゲームの場合、少なくとも筆者は本システムしか思いつきませんでした、パスは瞬時に求まる必要があるため、使えるシステムやアルゴリズムは決まってきます。パスを求める（計算する）のに十分な時間があるなら、乗算や方程式を多用した膨大な計算を CPU の力任せに行わせることも出来ます。一般的に、同じ目的を実現するのにすぐに思いつくのは、この力任せの計算のほうで、皮肉なことにそのほうがコードは理想的にシンプルに書けます。しかし、我々は、学術計算をプログラムするのではなくゲームをプログラムするのですから、大抵の場合、時間の掛かるアルゴリズムやシステムはすぐにボツになります。ゲームアルゴリズムが高速志向なのは昔も今も変わらないのです。高速に目的を実現するには、それなりの工夫やトリックが必要で、シンプルな正攻法は使えないことが殆どです。えてしてコードはトリック実現のために長く複雑に成らざるを得ず、開発時間もそれに比例して長いものとなってしまいます。
さて、本章では、まずシステムの概要を大まかに示した後に、概要の項目一つずつ詳しく解説していきます。本来、本セクションを解説するためには、本 1 冊分の紙面が欲しいところですが、そうもいかないので、なるべく分り易く説明しようと思います。

システムの基本原理

2 次元フィールドを、適当な解像度でブロック分けして、そのブロック情報をバッファーメモリーに格納します。バッファーメモリーは 2 次元配列です。なお、このブロック 1 つを「セル」と呼ぶことにします。セルのサイズは任意の横単位 x 任意の縦単位から成ります。サンプル Chapter16(2D) は 32x32 ピクセル、Chapter16-2(3D) は 32x32 単位としています。単位は 32 ピクセル x32 ピクセルだったり 32 メートル x32 メートルだったりとゲームの仕様によって異なります。セルは移動の最小単位より大きい単位でサンプリングするもので、それによって計算速度を高めるのが目的のです。

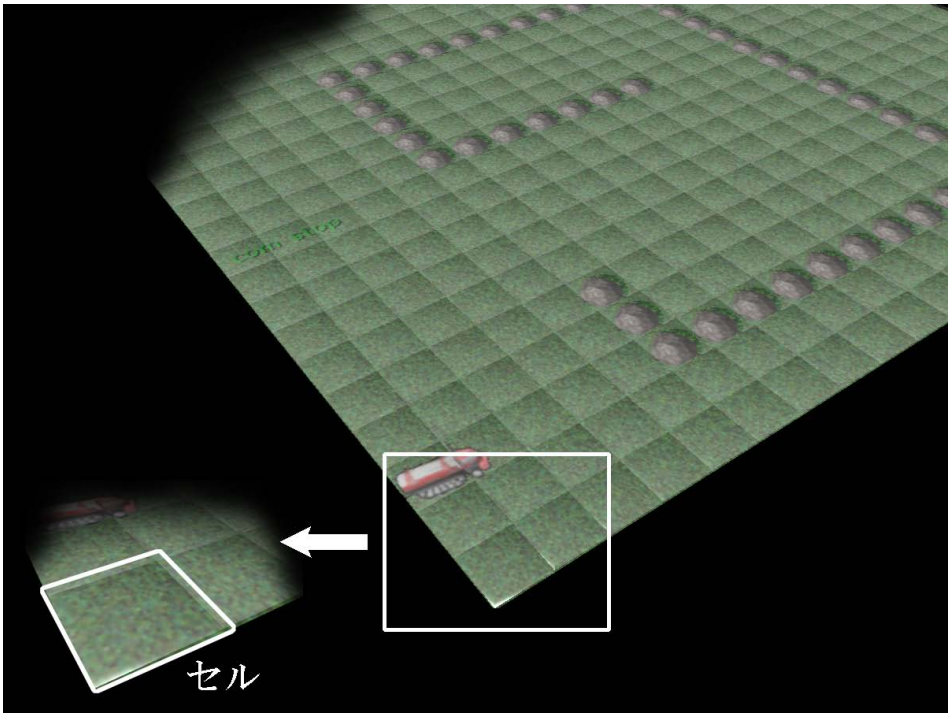


図 16-1
障害物回避の判断は、全てセルを基に判断するので、セルは常に現在のフィールド状態を表している必要があります。
セルを基準にする細かい理由は多くありますが、大きな目的は処理を高速に行うということただ 1 つです。先ほども述べましたが、処理に十分な時間をかけてもいいのなら、極端な話、全ての物体ごとに他の物体との位置関係を調べるループをかけてじっくり（？）計算すれば最高の精度で、しかもコード的にもシンプルに書けます。それが非現実なのは明らかであり、もしそうしてもいいのなら、なにもこのようなシステムを数ヶ月もかけて構築することもないのです。

セルを基準にして、パスを決定する概要は次のとおりです。

目標地点へ繋がる仮パスの作成

物体を移動する時に一番最初に行われる行程です。基本的に目標地点（目標セル）方向に向かって順順にセル調査を進め、障害物にぶつかった場合は障害物の周囲を沿うようにセルを進めていき、最終的に目標地点までセル調査を進めます。開始セル及び目標セルが障害物に完全に覆われている場合（出口がまったく無い閉じた空間）ではない限り必ず目標地点にセル調査が到達します。閉じた空間によりパス調査が到達不可能な場合のみ不可能フラグを立てて移動処理を中止します。
この行程で、状況によって複数のパスが候補に挙がることがあります。複数の候補が挙った場合は、最短のパスをひとまず決定します。

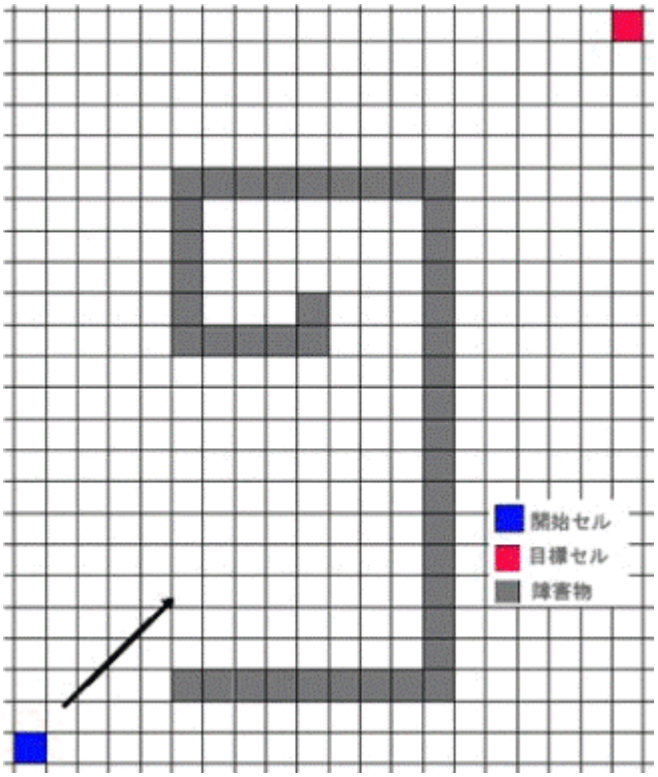


図 16-2

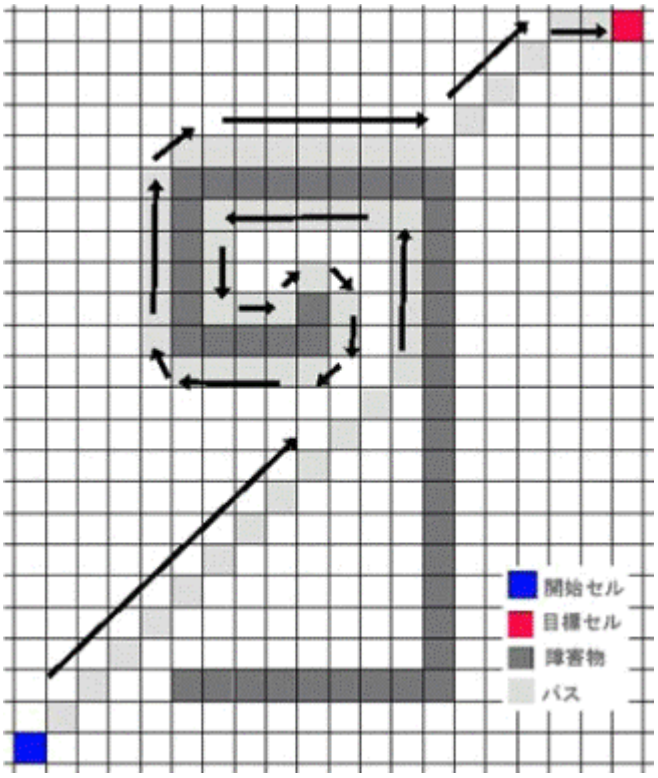


図 16-3

パスを最適化

障害物の形によってはパスが必要以上に長くなることがあります（図 16-4 参照）。そこで、パスを図 16-5 のように最適化します。

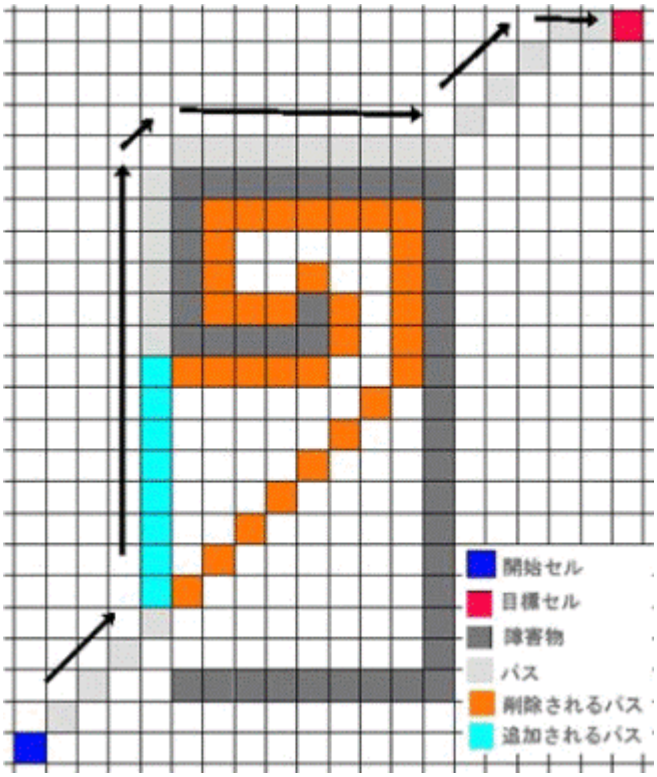


図 16-4

パス上の移動

物体は最適化後のパス上を単純に移動すればいいわけです。移動に関してはなにも計算することはありません。

新しいパスの再決定

物体がパスによる移動中にパス上に障害物が現れる場合があります（新しい建物の建設や他の移動物体の通過・停止など）。なお、他の移動物体は自分から見れば障害物です（移動要請を出して移動させることが出来るので純粋な障害物とは異なりますが）、このような場合、その状況が変わらないと判断した場合は新しいパスを再決定するため再度パスを計算し直します。

システムの詳細

原理の概要で述べた各プロセスのなかで最も難しくコード的にもボリュームがあるのは、最初の処理である “ 仮パスの作成 ” です。” パスの最適化 “ は仮パスが作成されていれば、それほど難しいことでありません。

仮パス作成プロセスの詳細

詳細をどうやって分り易くしかも簡潔に解説するかということを 24 時間悩んだ末、面白いアプローチを思いつきました。先ほどの図 16-4 でパスがセルの上を這っている様を思い出してください。ここでパスをミミズのような虫に例えます。名づけてパスワーム（PathWorm）です。（パスワームは、つい先ほど風呂の中で思いつきました）、パスワームは緑色の半透明で、目的地まで障害物を避けるように自分の体を伸ばすという非常に興味深い習性を持ち、もちろん昆虫学会未発表の新種です。（これは冗談です、気にしないでください…）
パスワームが障害物を回避する方法もこれまた非常にユニークです。まず、自分から目的地への直線上でどんどん体を細胞分裂しながら伸ばします。そして、伸びている最中、障害物にぶつかると、伸びるのを一時中断し、その場で子ワームを複数匹生みます。子ワームは、それぞれ別の方向に障害物がないか調査するため体を伸ばします。子ワームは自分が進んだ方向に障害物がなければ親に報告します。親ワームは障害物の無い方に向かって伸びを再開します。また、子ワームが障害物にぶつかった場合は、親と同じようにその場で止まって子供（一番最初のワームの孫）を複数匹生みます。孫ワームもまったく同様にそれぞれの方向の障害物の有無を調査し、調査結果を報告します。そのようにしてワームはぐにゃぐにゃと体をよじらせながら目的地に達します。

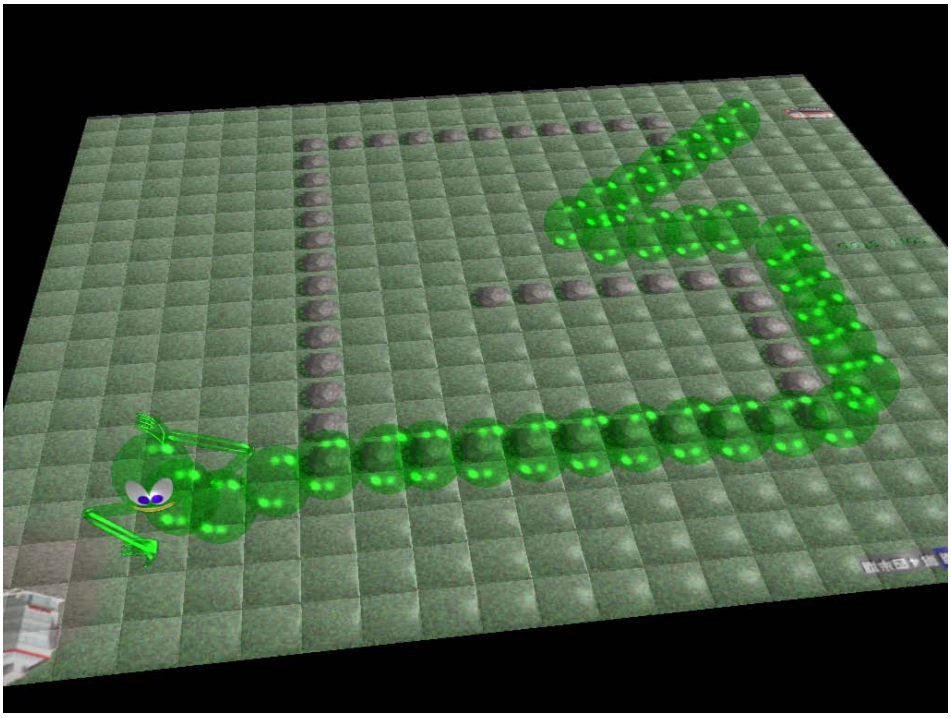


図 16-5
本書は別に奇妙な虫の生態を解説するものではありません。ゲーム開発解説書です。（笑）
しかし、このワームの習性は、本システムの仮パス作成プロセスを見事に表現しています。というのも、目的地に到達した時点でのワームの形状が仮パスだからです。
この時点でのパスは、まだ“仮”パス（最適化前のパス）の状態なので（図 16-4 参照）、パスをもうすこし加工して、無駄な部分を排除しなくてはなりません。それがパスの最適化です。

仮パス作成は、移動開始前に事前にワームに移動シミュレーションを行わせることと言えます。ワームは、複数の仮パスを作成してその全ての仮パスを報告します。そのうちでもっとも近いパスを採用するという仕組みです。
なお、事前にシミュレーションを行うと言うと時間がかかるように感じるかもしれませんが、ワームによるシミュレーションはセル単位で行われるのでナノ秒オーダーで終了します。これには、乗算や複雑な計算式を使用していないという理由もあります。ここで言う乗算とはアスタリスクによる乗算です。厳密に言うとビットシフトにより乗算を実現しているので、そうゆう意味では、浮動少数は一切使用していないと言ったほうがいいでしょうか。さらに、仮パスの作成タイミングは、基本的に対象を動かす指示があったときに 1 度だけ実行すればいいものなので全体パフォーマンスに影響は与えません。パスが一旦作成されれば、なにも考えることなくそのパスの上を移動すればいいのですから。

補足

実際のコードは、もっと複雑ですので、これ以外のポイントを列挙します。
ワームは、8 方向に伸びます。8 方向に優先順位を付けます。当然目標への方向が優先度 1 番です。
それぞれの方向へ伸びる際に、一度通った場所をマーキングして同じセルの上を再度延びることのないようにします。

パス最適化の詳細

仮パスを最終的なパスとして使用してもいいのですが、やはり、もっとスマートに移動したいものです。仮パスの無駄な部分をカットして、場合によっては新たなパスを追加する処理が必要です。このプロセスでのポイントは、どのように“無駄な部分”を認識するかということになります。
仮パスの形状を人間が見たときに無駄な部分は一目瞭然ですが、プログラムにはどうやって無駄な部分を“無駄”と判断されればいいのでしょう？
答えは、まったく簡単です。（と言っても答えを見つけるまで苦労しましたが…）
全てのセルに対して次のアルゴリズムを適用していきます。

最適判定

ワーム君の体は、偶然にも（…というか意図的に）セルと全く同じサイズの細胞から成るものとします。したがってワームの体の下にあるセルの数とワームの細胞の数は一致することになります。

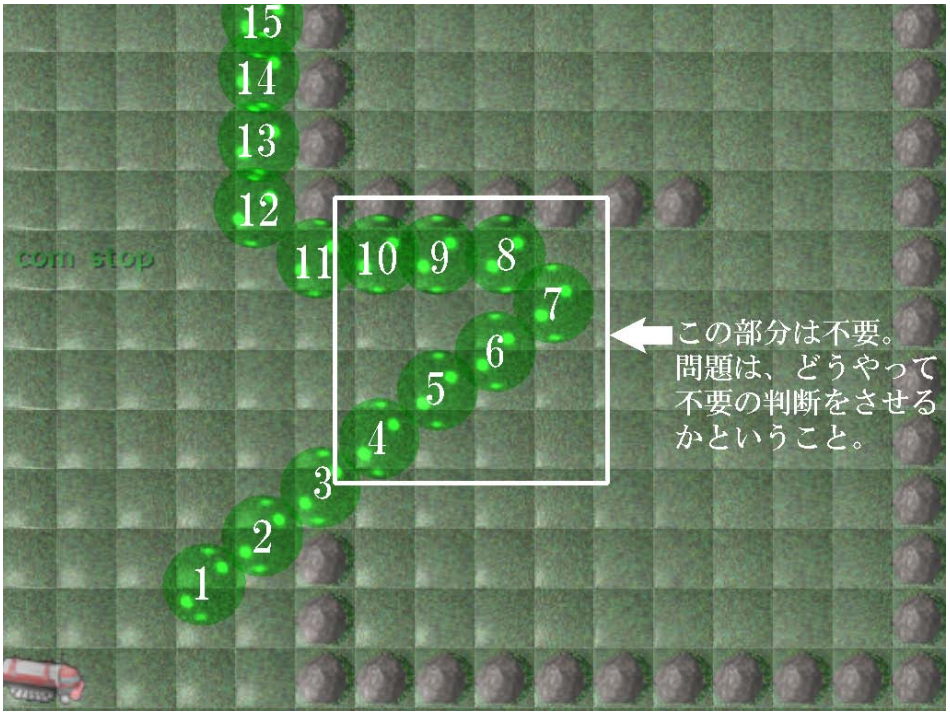


図 16-6

今、ワームの体の一部が図 16-8 のような形状だとし、全細胞調査のループの中で細胞 A まで調査が進んできたものとします。ワームは当該細胞の周り 8 方向に向かって、近くに他の細胞があるかどうか調べます。

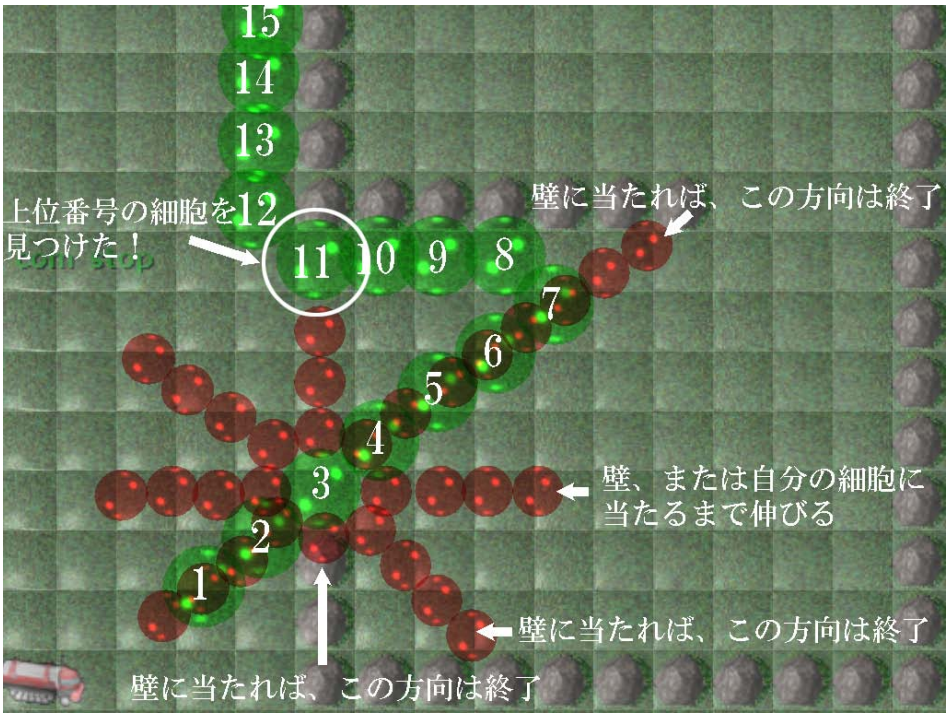


図 16-7

ワームは細胞 A の近くに細胞 B があることに気付きました。そして、ここからがミソですが、ワームは最初の分裂から分裂した順番で細胞に番号を付けています。細胞 B は細胞 A より番号が大きい、つまり、より新しい細胞です。より新しい細胞が近くにあり、かつ、細胞 A と細胞 B の間の u 番号の細胞が遠くにある場合、それら中間の細胞は “ 無駄 ” な細胞と特定できます。ワームは細胞 A と細胞 B を繋ぎ、中間にあった細胞は本体から切り離します。かくして、無駄な細胞は本体から離れ、死に絶えることとなります。
これが、最適化の原理です。ワーム君おみごと！
なお、本体から切り離され死に絶えた細胞が図 16-5 での “ 無駄なパス ” であり、細胞 A から細胞 B を結んだ新たな細胞が図

16-5 の “ 追加されるパス “ です。

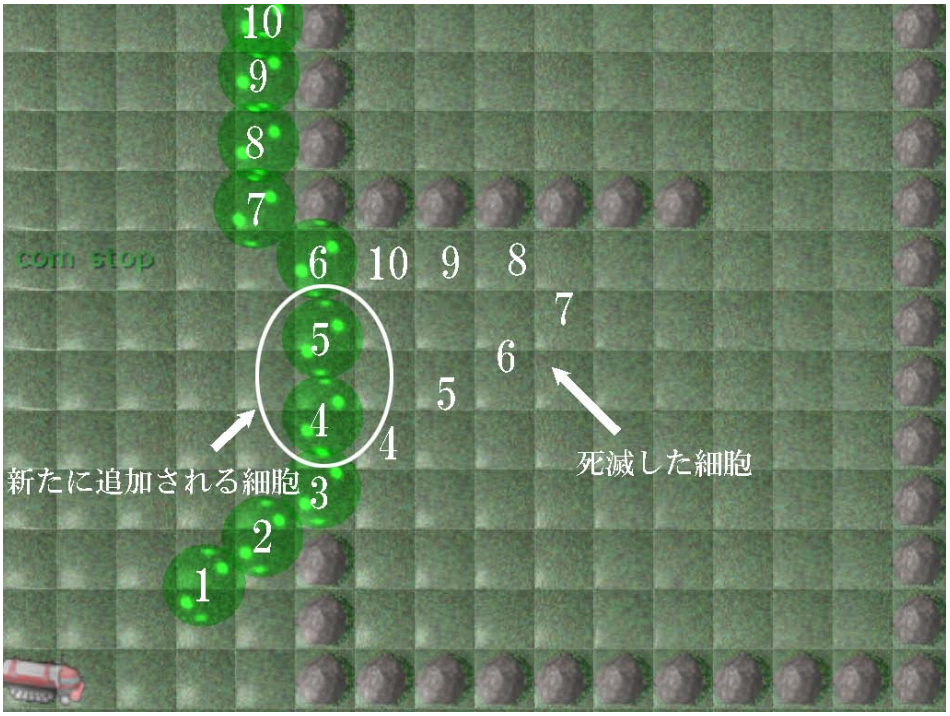


図 16-8

ライブラリの利用方法

ワーム君にはそろそろ自分の家（？）に帰ってもらい、ここからは、システムの導入及び運用の手順を解説します。
本システムは、クラスで作成したものをダイナミックリンクライブラリにしたものです。クラス名は CPASSDECISION です。PASS（進路）の DECISION（決定）という意味です。
ライブラリは 2 種類あり、DirectX7 の DirectDraw に特化したものと、DirectX のバージョンに依存しないものを作成しました。
DirectX7(DirectDraw) バージョンについて
このシステムを開発したのは、今から 5 年前であり、当時の DirectX バージョンは 7 でした。また、ストラテジーゲームに使用するエンジンとして開発したものなので DirectDraw を使用した描画メソッドを含んでいます。
ライブラリファイル名は、KamPassDX7-2D.lib と KamPassDX7-2D.dll です。
DirectX のバージョンに依存しないバージョンについて
このシステムは、殆どの部分が純粋な計算ルーチンから成り、DirectX のバージョンにも依存しませんし、DirectX を使用しないアプリケーションでも使用できます。描画メソッドは利便性を上げるために、おまけ程度に付け加えたものです。ですので、DirectX7 用のバージョンでも、描画系のメソッドを使用しなければ、DirectX9 ベースのアプリケーションでも当然使用できますが、混乱を避けるために、汎用バージョンを新たに作成しました。汎用バージョンと言っても、もともとのシステムから描画系メソッドを削除しただけです。
ライブラリファイル名は、KamPass.lib と KamPass.dll です。

読者が自身のプログラムに導入する手順は次のとおりです。（汎用バージョンを導入するものとします）
CPassDecision.h ヘッダーファイルをプロジェクトに追加します。
KamPass.lib と KamPass.dll を任意のディレクトリに置きます。普通はプロジェクトフォルダに入れます。（KamPass の Kam は Kamada の Kam です。）

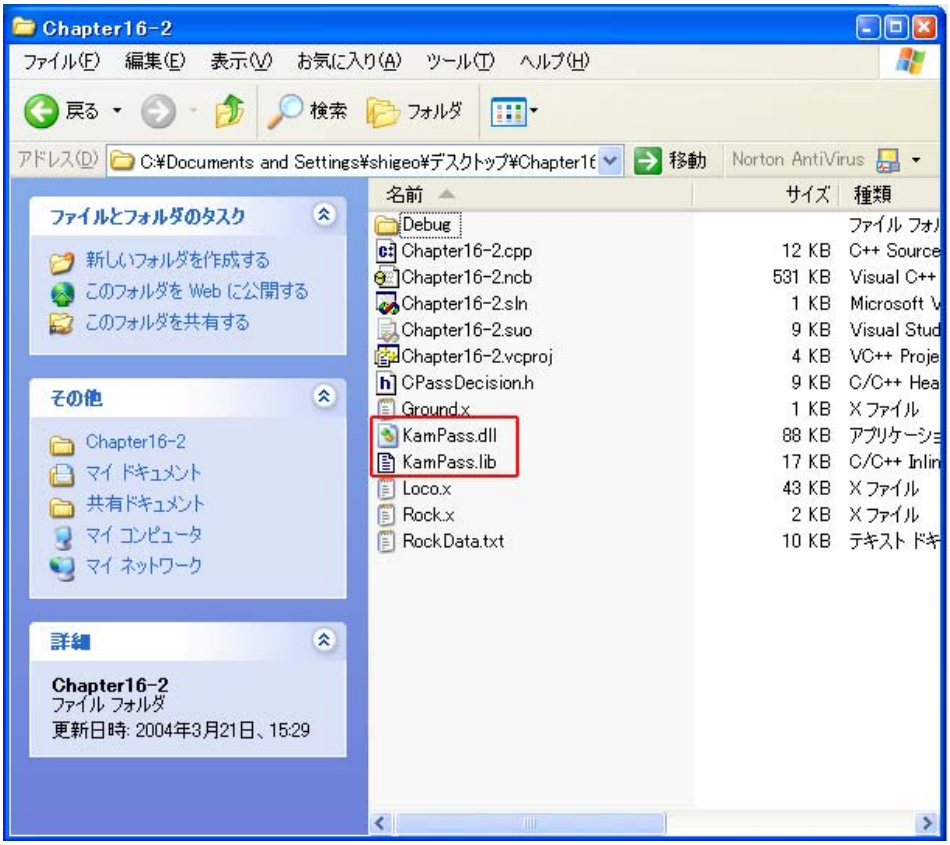


図 16-9

DirectX の lib ファイルと同じようにプロジェクトに KamPass.lib を登録します。KamPass.lib は DirectX のライブラリファイルと同じで、単なるインポートライブラリであり、ライブラリの実体である KamPass.dll を読みこむためのものです。

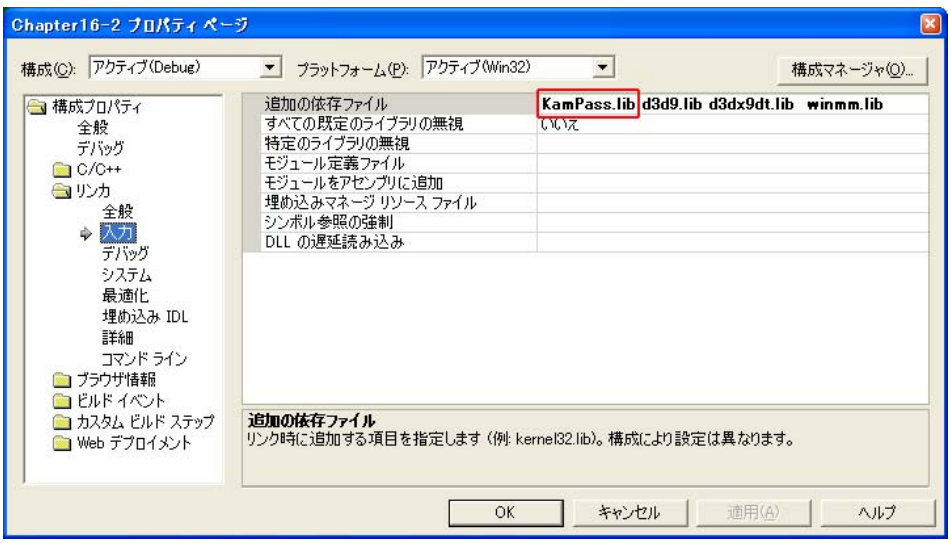


図 16-10

以上でファイル的な準備は完了です。
コード上での準備の手順は次のとおりです。
システムを使用するモジュールまたはクラスのヘッダーファイル等で CPassDecision.h をインクルードします。

各インスタンスの作成と初期化

CPASSDECISION の初期化
CPASSDECISION クラスは、コンストラクタで初期化はせずに、初期化メソッドで行います。初期化メソッドをコールする前に、まず、初期化用の構造体 CPASSDECISION_INIT のインスタンスを作成し、各メンバーを初期化します。次に、その構造体のポインターを初期化メソッド CPASSDECISION::Init() に渡してコールすれば初期化終了です。
初期化用構造体 CPASSDECISION_INIT の各メンバーの意味は次のとおりです。

（Chapter16-2 サンプルから抜粋）
```
// CPASSDECISION の初期化
        pcPass = new CPASSDECISION;
        CPASSDECISION_INIT pi;

        pi.pFuncRandom=NULL;
        pi.wCellWidth=CELL_SIZE;
        pi.wCellHeight=CELL_SIZE;
        pi.wFieldWidth=FIELD_WIDTH;
        pi.wFieldHeight=FIELD_HEIGHT;
        pi.wWindowWidth=WINDOW_WIDTH;
        pi.wWindowHeight=WINDOW_HEIGHT;
        pcPass->Init(&pi);
```

pFuncRandom
使用しません。NULL に設定します。
wCellWidth
セルの横幅
wCellHeight
セルの縦幅
wFieldWidth
フィールドの横幅
wFieldHeigh
フィールドの縦幅
wWindowWidth
ウィンドウの横幅
wWindowHeight;
ウィドウの縦幅

セルのサイズについて
サンプルでは、セルのサイズは 32x32 です。Chapter16 サンプルは DirectDraw ベースなので、単位はピクセルです。また、Chapter16-2 サンプルは３D ベースなので単位はメッシュの長さ 1 を 1 単位とした場合の 32 単位です。このように、単位は、利用するアプリケーションごとに変化します。変化するというより、捉え方を変えるといったほうがいいでしょう。要はシステムを使用しているアプリケーション内での単位を基準に考えるということです。また、たまたま両サンプルとも 32 単位としていますが、24 単位でも 20 単位でも、値は任意です。
セルのサイズは、フィールドのサイズと関連して決定する必要があります。この２つの組み合わせで、システムの解像度が決まるからです。解像度はフィールドサイズ / セルサイズで決まり、解像度が低いと、処理は高速になるが動きが粗くなり、一方、解像度が高いと、処理は低速になるが動きはより滑らかなものとなります。

移動体（派生 LOCOMOTOR 構造体）の作成
本システムは、動かす物（スプライトキャラやジオメトリ）を「移動体」という概念で捉えます。基本的な移動構造体型は、LOCOMOTOR 型としてすでに定義しています。利用する開発者は、自身が追加したい独自の変数等があるでしょうから、その場合は LOCOMOTOR 構造体から派生させて新たな独自の移動構造体を定義します。
LOCOMOTOR 構造体は、次のように移動に必要な基本要素をメンバーに持ちます。
（CPassDecision.h ヘッダーファイルから抜粋）
```
struct LOCOMOTOR
{
        INT wait_counter;
        BYTE wait;
        BYTE host_player_number;
        BYTE kind;
        WORD number;
        BYTE vector;
        BYTE speed;
        CHAR turn;
        BYTE wall;
        WORD pass_counter;
        BYTE pass;
        WORD cell1;
        WORD cell2;
        WORD cell3;
        BYTE meet_pattern;
        SHORT old_target_pointer_x;
        SHORT old_target_pointer_y;
```

```
        BYTE stop;
        CHAR escape;
        SHORT posx;
        SHORT posy;
        SHORT old_posx;
        SHORT old_posy;
        SHORT target_x;
        SHORT target_y;
        WORD mem[MEM_SIZE+1];
        BYTE jam;
        BYTE toomany;
};
```

各メンバーの意味を一応説明しますが、殆どのメンバー変数はシステムが内部で使用するもので、通常利用者は意識しないで構いません。

利用者が、使用または参照するメンバー変数
BYTE host_player_number;
プレイヤーが複数いるゲーム内での、移動体が属するプレイヤー番号。番号は 1 プレイヤーに対して一意であれば任意の値です。プレイヤー番号を付ける目的は、同じ種類の移動体でも、同じプレイヤー番号の移動体は互いに進路を譲り合いますが、プレイヤーが敵同士の場合には、敵の移動体を避けず、むしろブロックするようになります。システムはプレイヤー番号により、その判断をします。
BYTE kind;
移動体の種類を識別する一意の BYTE 値。値は 1 種類に対して一意であれば任意の値です。
例えば、戦車という種類の移動体を 1 に、人間という移動体の種類を 2 に設定するなど。
WORD number;
同じ種類の中での、その移動体の番号。例えば、戦車という移動体を 200 個作成した場合、何番目のインスタンスかということ。
BYTE speed;
移動体の移動スピード。値は 1 ～ 8 の範囲です。
CHAR turn;
移動体の方向は、24 方向です。時計の 12 時方向を 1 として、時計周りに 23 まであります。その時の進行方向を表しています。それぞれのゲームアプリケーションで、移動体の描画、レンダリングに、この方向を利用できます。
BYTE stop;
移動体を強制的に止めたい場合は、stop=1 にします。通常は、参照のみにしてください。
SHORT posx;
移動体の X 座標です。システム上、移動体は 2 次元座標ですが、 3D ゲームで利用する場合は、アプリケーション側で座標軸を増やすことにより対応してください。
SHORT posy;
移動体の Y 座標です。
SHORT target_x;
現在の目標地点 X 座標。
SHORT target_y;
現在の目標地点 Y 座標。

次のメンバー変数はシステム内部で使用します。通常利用者は使用しません。
INT wait_counter;
何らかの理由で、移動体が一時待機状態にある場合のカウント。
BYTE wait;
何らかの理由で、移動体が一時待機状態にある場合のフラグ。
BYTE wall;
障害物にぶつかっているときにこのフラグが立ちます。
BYTE vector;
移動体の方向。
WORD pass_counter;
現在通過中のパス番号。
BYTE pass;
パスが計算されているかいないかのフラグ。
WORD cell1;
WORD cell2;
WORD cell3;
セル情報。
BYTE meet_pattern;
他の移動体との衝突パターン。
SHORT old_target_pointer_x;
直前の目標地点 X 座標。

SHORT old_target_pointer_y;
直前の目標地点 Y 座標。
CHAR escape;
障害物を回避している最中は、回避モードになり、このフラグが立ちます。
SHORT old_posx;
直前の X 座標。
SHORT old_posy;
直前の Y 座標。
WORD mem[MEM_SIZE+1];
一時的なセル情報を記録するバッファー
BYTE jam;
何らかの理由により、身動きとれない場合のフラグ。
BYTE toomany;
複雑な迷路状のパスを計算するとき、演算時間が長くかかった場合にこのフラグが立ちます。

毎回（ゲームループ 1 回転）処理

目標地点の設定
CPASSDECISION::SetTarget メソッド

移動
CPASSDECISION::Move メソッド

移動体の表示
表示は、基本的にそれぞれのゲームアプリケーションで独自に表示します。ただ、Chapter16 サンプルでは、CPASSDECISION::DrawLocomotor 等のシステム描画メソッドにより描画し、スクロールまでさせています。システムで描画させるには、DirectX7 の初期化が前提になりますので、Chapter16 サンプルでは DirectX7 クラスを併用しています。読者が、もし DirectX7 で、しかも 2D タイプのゲームに本システムを採用するのであれば、Chapter16 サンプルは、その手順を与えるものとなるでしょう。
なお、表示した後には、CPASSDECISION::WriteMap メソッドを実行することを忘れないでください。

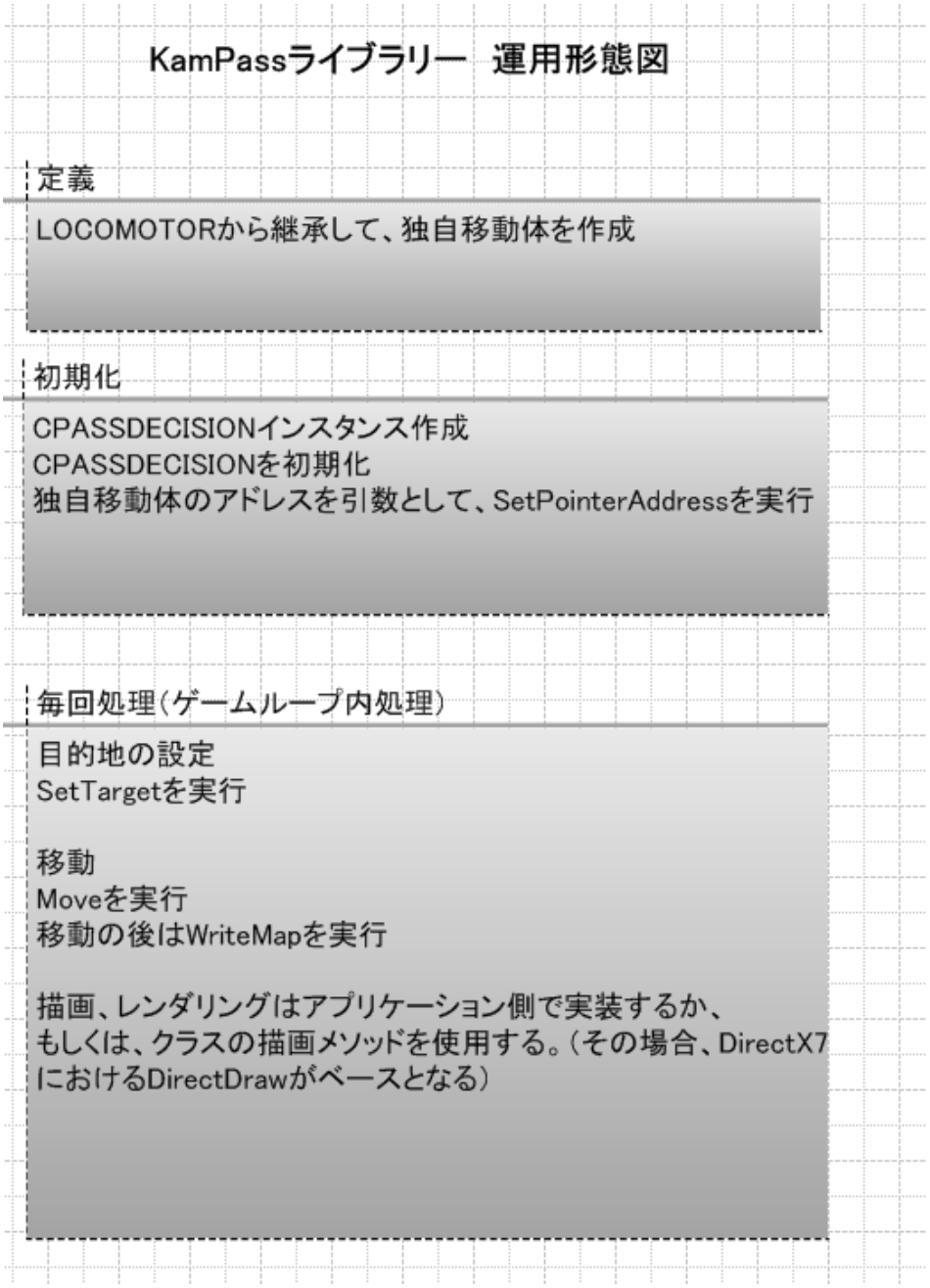


図 16-11

通信対戦ゲームでは、言うまでも無く他の PC と通信するためのルーチンが必要です。DirectX で通信ルーチンを作成するのであれば DirectPlay を使用しますが、ここでの解説は DirectPlay に特化したものではなく、他のライブラリにも当てはまる汎用性があります。
通信対戦を可能にするにあたって、やらなければならない処理は大きく２つに分けて考えることができます。１つは送受信、２つ目は送受信後の後処理です。
送受信とはゲームに関連する何らかのデータを物理的に離れた PC とやりとりすることです。これは言うまでもないでしょう。送受信するデータは、以降メッセージと呼びます。
では、送受信の後処理とはなんなのでしょう？ゲームメッセージは送受信されて各 PC を行き来しますが、メッセージはネットワークを経由すると、消失したり一部が破損したりします。そのメッセージを修復するなんかの処理が必要なのです。この“後処理”が“同期をとる”ということです。同期とはすなわち“同じゲーム状態を各リモート PC で再現すること“です。
ここで、考えてみてください、なぜ送受信するのでしょうか？言い換えると何のために送受信するのでしょうか？もちろん、送受信の目的はゲーム状態を全 PC 間でお互い同一に保つためのメッセージを相手に渡したり受け取ったりするためです、これは、言い換えると同期をとるために送受信することになります。送受信は同期の手段であり同期は送受信の目的です。
多くの書籍では、この手段の部分である“送受信の手順”のみが解説され、時には、「（送受信の手順が分ったので）通信対戦プログラムの準備ができました！あとは同期の問題に集中すればいいのです…」などとあっさり締めくくられて解説が終了しています。たしかに、準備ができたということはそうなのでしょうし、入門書であるなら、そこまででいいかもしれません。しかし、同期こそが大問題であり解説されるべき問題です。なぜなら、送受信が出来るようになった後、同期という壁はすぐにやってくるからです。本書はもう一歩踏みむことにします。少々乱暴な言い方になりますが、送受信の手順などは、同期処理に比べれば“どうとでもなること”と言わざるを得ず、ヘルプファイルを見れば簡単にコーディングできることです。（入門書において「簡単に…出来ること」と言ってしまうのは身も蓋も無いですね、すいません）
DirectPlay や WinSock ※は手段である送受信までしか提供しません。Direct3D がレンダリングまでを提供しダイナミクスまでを提供するものではないのと似ています。ただ、DirectPlay の場合は、ほんとうに最低限必要な“手段の部分”しか提供しません。比喩的な表現をすると DirectPlay は、初めて絵画を描こうとする人に筆（手段）だけを提供しているようなものです。Direct3D の場合は、筆と絵具と下書きくらいは提供しています。それでもやはり依然として絵を仕上げるのは開発者です。不特定多数の人が描こうとする絵を事前に提供することは誰にもできませんので、これは当たりまえのことです。つまり、DirectPlay や他のライブラリを使用しても、同期ルーチンを作るのは開発者です。
送受信処理だけならば DirectPlay じゃなく、もっと低レベルな WinSock を使用しても同様の処理ができます。むしろ、WinSock でコーディングしたほうが良い場合すらあります。
通信対戦プログラミングにおいて重要なのが、２つ目の処理、すなわち、ゲーム同期の部分です。送受信だけであれば、ものの１日でコーディングできます。
同期の解説は DirectX 環境、もっと言えば OS 環境にも依存しません。さらに、送受信については、特に DirectPlay を利用しなくても他のライブラリで実現できます。したがって、DirectPlay について言及することなく解説することが可能であり、そうべきものと考えます。DirectPlay にこだわらないのは、これらの理由からです。
しかしながら、DirectPlay は送受信の前段である、プロトコルの初期化を全て行ってくれるという初心者には便利な機能を持っていることも事実なので、最初のうちは便利なツールとして、DiretPlay を使うのが近道であるのは確かです。
一応、DirectPlay について認識しなければならないことだけ少し述べておきます。それは、DirectPlay はある意味で特殊なコンポーネントだということです。他の全てのコンポーネントとは決定的に異なる点が１つあるからです。それは、“メソッドの信頼性を意識する必要がある”ということす。他のコンポーネントの場合、あるメソッドの信頼性は PC が壊れてないかぎり 100％です。そもそも信頼性などと意識することすらありません。ところが、DirectPlay の場合、メソッドは正しい答えを返さないことがあり、ゲームのタイプによっては、正しい答えを出すほうが少ないというのが現状です。例えば、１台の PC 内におけるローカル処理の場合、1 ＋ 1 は CPU やバスが壊れていない限り　＝２と（当たり前ですが）返してくれます。一方、リモート環境では、１＋１は、＝３とか＝？だったり、時には答えが返ってこない、あるいは式が届かないことは日常茶飯事です。ネットワークの信頼性がそのまま処理内容に影響するので仕方ないことなのでしょう。LAN の場合はまだ良いですが、ことインターネット環境となるとパフォーマンスは劇的に不安定になります。インターネット対応のコーディング時には、その不安定さに涙がちょちょぎれる（笑）思いだったのを覚えています。言うなれば、“時々間違った答えを出す電卓を使って簿記検定を受けている”ようなものです。
DirectPlay が悪いと言っているのではありません、これらの原因は、全てネットワークの性能に依存しているということなのですから。もしも、100％完全にメッセージを送受信できるネットワークならば DirectPlay は常に正常に動作するはずです。これは WinSock でも同じです。原因は“ネットワークの性能”にあるのです。
なお、ここで言う同期とは DirectPlay の同期（非同期）送受信メソッド※とは関係ありません。先にも述べたように同期とは“同じゲーム状態を各リモート PC で再現すること“です。同期（非同期）通信は、単に送受信をメソッド呼び出しスレッドと同じか別スレッドで行うかどうかという意味しかありません。

WinSock
Windows Sockets の略称。“ソケット”という UNIX 互換 OS 上で発展したネットワーク用インターフェイス概念を Windows 上で実現したインターフェイス。
DirectPlay に比べ、より低レベルなインターフェイスである。
DirevtPlay は WinSock をベースにもつ。

非同期送受信メソッド
非同期メソッドの場合は、コールした後、直ちに次のステートメントに移ることができる。
これに対して、同期メソッドは、送受信が完了するまで待たなければならない。
送受信の成否をアプリケーション側で確認できるような場合、非同期メソッドは有効である。

通信対戦プログラミングの二重苦

“同期をとる“とは、異なる PC 間で同じゲーム状態を保つことですが、同期を困難にする問題が２つあります。そのうち一つは、ネットワーク特有の問題です。この２つの原因は、相互に関連し合うので、同期ルーチンは、この２つの壁を同時に克服するように設計することとなります。

問題その１　ネットワーク自体に内在する問題

通信対戦プログラミングにおいては、ネットワークを経由することによるメッセージの欠損という大きな壁が存在します。メッセージの欠損とは欠落（全くメッセージが届かないこと）と破損（到達したメッセージの中身が壊れていること）の両方の意味が含まれます。この壁は非常に大きく乗り越えることが困難であり、特にメッセージの一部が壊れる破損は深刻な問題で

す。

メッセージ欠損がもたらす怪奇現象？

今、A 君と B 君が 2 台の PC で 3D 格闘ゲームをしているとします。B 君のライフがゼロに近くなったとき、A 君は、とどめの一撃を放ちました。ところが、とどめの一撃を B 君の PC に伝えるメッセージがネットワークを通過する過程で欠落（消失）したとしたらどうなるでしょう？ A 君の PC 上では、B 君は倒れたため、A 君は次のステージ等に進みます。ところが、B 君の PC 上では依然として B 君は生きてきますし、そのため A 君はそのステージに存在しています。B 君は誰と戦っているのでしょう？？
また、相手が倒れたときでも次のステージに進まない仕様だった場合、A 君の PC 上では B 君は一人で何かと戦っていることになり、その光景は、さながら透明人間と戦っているように見えることでしょう。

一部破損は有り得る

まず、メッセージの欠落と破損の違いを再確認してください。欠落は、メッセージがネットワークの途中で完全に消失して相手に届かないことです。破損は、メッセージが届くことは届くものの、メッセージの中身が全部壊れている、あるいは一部分だけ壊れていることを言います。TCP/IP ※ではメッセージを分割して（分割されたメッセージはパケットと呼ばれます）送受信しますが、パケットは TCP 部分の情報をもとにパケットの内容の整合性をチェックしているので一部破損というのは有り得ないという建前があります。DirectPlay もヘルプファイルにデータの欠落は有り得るが、破損は有り得ない、つまり届くか届かないかの 2 つに 1 つであり、届いたデータであれば、そのデータの内容は保障されるとしています。しかし、それは " 嘘 " です。一部破損は起こります。場合によっては頻繁に起こります。DirectPlay には、確実にメッセージを送信する保証つき送信というメソッドがありますが、それを使用しても破損は発生します。欠落も当然のように（？）起こります。なぜ、破損が起きるのか、その原因は謎です。そうゆうものなのか、DirectPlay が原因なのか、筆者も DirectPlay を使用する 1 ユーザーでしかないので、分りかねますし、調べるつもりもありません。ただ、欠落のみならず破損も起こるのだということを認識しておいてください。

メッセージ欠損の特徴

メッセージが欠損する確立は、ある条件に対して比例的であるとう法則性があります。その条件とは " 送受信の間隔（タイミング）" と送受信メッセージ自体のサイズです。送受信間隔とは、ある瞬間に送受信を行った時に、次の送受信を行うまでの時間のことです。メッセージサイズは文字通りです。
他にも、通信経路の物理的距離や通信レスポンス（ピング :PING）、通信帯域（バンドワイズ :Band Width）などがあると思われますが、それらは物理的ハードの問題であり、調整不可能なので、プログラミングの範囲外の問題です。プログラミング上では、送受信間隔とメッセージサイズのみが欠損のファクターであるので、この 2 つを工夫することが、欠損を予防する唯一の手段と言えます。特にサイズよりも、送受信間隔のほうがより重要です。間隔が長ければサイズが大きいメッセージでも欠損は発生しませんが、間隔が短いと 1 バイトのメッセージでも欠損は発生します。

LAN とインターネット

同じ状況であれば、インターネットより LAN のほうが、欠損率が低いということは、ある程度想像できるのではないでしょうか。付け加えると、LAN の場合、欠損率はかなり低いです。インターネットと LAN では、データ欠損に大きな開きがあるということを覚えておいてください。
ただし、LAN でも、データ欠損の特徴自体はインターネットと同じです。つまり、送受信間隔が短いと LAN であろうと欠損は必ず発生するということです。

問題その 2　メッセージ処理タイミングの問題

（純粋な意味での同期）

同期自体が複雑なのに、さらに追い討ちのように、同期に使用するメッセージがそもそも不安定であるという問題がありますが、まずは、2 つの問題を切り離し、純粋な同期のみを考えてみます。したがって、ここでの解説は、送受信が完璧に行われ、お互いの PC が正しいメッセージをやりとりできるとしたときにはじめて成り立つ議論です。
タイミング調整の必要性を感じてもらうために、有名な例を挙げてみます。
「ドアのジレンマ」
昔から一般的に言われている「ドアのジレンマ」と呼ばれている現象を考えてみましょう。
A 君と B 君が 2 台の PC でロールプレイングゲーム（でもなんでもいいのですが）の通信対戦を行っているとします。ゲームの中で A 君は、あるドアの前に立っていて、B 君はドアの向こう側に立っているとしましょう。A 君がドアを開ける操作をした時は、その情報メッセージは B 君の PC に届き、B 君のゲーム内でドアが開きます。しかし、その時同時に B 君がドアを閉める操作をした場合、結果はどのようにすべきでしょうか？すぐに思いつくのは、両者の操作タイミングが早いほうの処理を先に実行するということでしょう。 1 台の PC 上で 2 人が対戦しているのなら、この判断は可能です。ところが、異なる PC 上で通信状態にある場合は、そうはいきません。なぜなら、メッセージ送受信のタイミングは、操作タイミングの判断ができるほど速くはないからです。言い換えると微妙なタイミングを検出できるど解像度は高くないのです。 1 台の PC の場合は、ゲームのフレームレートと同じ分解能で操作タイミングを検知できますが、 2 台の PC の場合、送受信のタイミングは、フレームレートより遥かに低いのです。その結果、両者の " 開ける " と " 閉める " が全く同一のタイミングに発生する状況は十分に有り得ます。同一タイミングとなってしまっては判断のしようがありません。
なお、「ドアは閉じたままでいい」とか「力の強い方にドアを開ける」という結論は一見可能に見えますが、それは人間の先

入観がもたらすもので、この条件では不可能な結論です。（ちろろんコーディングによりそのような追加属性を持たせることは当然できますが、この議論を何ら解決するものではありません）
ドアに限らず、シューティングゲームでも、同一の状況が存在します。例えば、レーザーを相手に発射するタイミングとシールドで防御するタイミングとかです、相手がシールドを張る前にレーザーを発射しているのに、同時に処理されてしまいレーザーが当たらないなど。同様に、槍を突いた操作と盾で防御する操作の場合もそうです、これはジレンマ（矛盾）の由来話風ですね。

このように、ネットワークがメッセージ欠損をすることなく完全にメッセージを送受信してくれるとしても、ゲーム状態を同期させることは簡単なことではなりません。そのうえメッセージ欠損が起こるのですから、同期ルーチンの開発は、データ欠損を考慮しつつタイミングにも対処できるようにコーディングする必要があり、さらに難しいものとなります。同期ルーチンの開発は、アマチュア、プロを問わず、開発には数ヶ月かかるはずです。筆者も、前章の障害物回避と並んで、この同期には2,3 ヶ月の時間を取られました。同期を取るには、この２つの壁を乗り越えなくてはならず、かなり難しい処理と言えるでしょう。

ソリューションの概要

しかし、心配は無用です。ほぼ完全に問題を解決するソリューション（解説策）を考案しました。考案したことなので、誰かに教わったことでもなければ、書籍から得た知識でもありません。先に述べたように同期にはさまざまな困難が伴い、それらはお互いに影響し合うため、それぞれを分離して解決策を解説するのは困難ですので、それらを同時に解説する一つのソリューションとして解説していきます。
ただし、考案したテクニックのなかで入力メッセージ交換は一般的な理論として知られているようなので、考案したと言うのはおこがましいかもしれません。もしかしたら、他のテクニックも一般的かもしれませんが、それは未確認です。同期に泣かされた人は、最終的に同じことを考えるということかもしれません。
ほぼ完璧と言うのは、ネットワークに 100％は有り得ないという経験則から念のためそう表現したものです。ネットワークが物理的に遮断している場合はもちろん、何らかの原因によりパケットデータが連続して届かない、あるいは連続して壊れたメッセージが届きそれがしばらく回復しない場合は、送受信の目的から見れば遮断されていることと全く同じこととなり、その場合はどんなに優れている同期ルーチンをもってしても同期は不可能です。ネットワークと同期の問題は、ゲームのみならず、システム系の開発者をも悩ませる最大の難関なのです。
このソリューションを組み合わせれば完全といえる同期がとれることは筆者の自作ゲームで実証済みです。ここでの考え方やテクニックは、後で詳しく図解します。
最初に断っておきますが、メッセージ欠損に対するソリューションと言っても、“ メッセージを欠損させない ” というソリューションではなく、メッセージ欠損を起こし辛くするという予防的なソリューション、及び、メッセージ欠損が起きたときの対処という事後的なソリューションです。メッセージを欠損させないようにすることは、通信キャリアーやネットワークデバイスメーカーの仕事であり、我々のするべきことではなく、我々ができるようなことではありません。メッセージ欠損そのものを無くすことは現状では出来ません。我々ゲーム開発者は、メッセージ欠損をなるべく起こさせないように努力し、欠損が起きたときには事後的に対処するしかないのです。

送受信間隔を長くとる（単位フレームという概念）

送受信間隔は出来る限り長くとります。最初のうちよくやりがちなのは、毎フレーム送受信をしてしまうことです。これはやってはいけません。なぜなのか、それは、フレーム間隔がネットワークの処理スピードを超えているからです。
ネットワークのレスポンスとは相手に送信したデータが相手に届いてさらに自分に帰ってくるまでの時間のことであり、いわゆる PING（ピング）と呼ばれています。LAN の場合、PING は 10ms 前後であり、インターネットの場合は、距離によってまちまちですが、国内の場合は数 10ms から 300ms あたりでしょう。一方、PC 内のフレームレートを 60fps としても、フレームとフレームの間の時間は 16ms（1000 / 60 秒）です。LAN 環境であれば、PING がフレーム間隔より速いので問題ないように見えますが、16ms という時間は LAN にとってもあまりにも短すぎる時間であり、16ms でデータを断続的に送受信したりすると、その大半は破損します。インターネットに至ってはインターネットが処理できる間隔をもともと超えてしまっているのですから、毎フレーム送受信は物理的に不可能であるのは明らかでしょう。インターネットにおいて毎フレーム送受信したデータは破損しないほうが奇跡的と言えます。
毎フレーム送受信を行いたい気持ちは分ります、ゲームループの 1 回転の中でのメッセージをその都度やりとりするというのが素直な発想ですし、もし可能であればそのようにするのが同期をとるうえでもコーディングの面でも一番やりやすいでしょう。しかし、ネットワークの現在の性能により、残念ながらそれは出来ません。
では、どうすればいいのでしょうか？
答えは、数フレームをまとめて一つの単位フレームとして考えるということになります。数フレームとは 10 フレームでも 20 フレームでもいいのですが、とにかくメッセージ欠損を予防しうるだけの出来る限り “ 時間を稼げる ” フレーム数ということになります。1 回のフレーム間隔が 16ms だとすると 10 フレーム分では 160ms になり、LAN 環境であれば、欠損が発生しづらい時間と言えます。20 フレームでは 320ms になるので、そこまでいくと、LAN 環境では、ほぼ欠損は発生しないと言って良いでしょう。
この数フレームを今後 “ 単位フレーム ” と呼びます。
メッセージをまとめるので、何らかの形で各フレームのメッセージを保存しなくてはなりません。また、メッセージには、何番目のフレームのメッセージであるか等のヘッダー情報を付けておくことも大切になってきます。このことは後ですぐ解説します。

入力情報のみ送受信する（入力メッセージ交換）

もし、ネットワークがローカルマシン並みのスループットを有し、かつ、メッセージの欠損が発生しないものであるなら、ゲーム内のあるとあらゆる変数、構造体の値をやり取りするようなコードを書けば、これほど楽で確実なことはありません。あるいは、ゲーム同期に必要なデータを全て送受信するというのもコーディングが簡単ですし、強力な同期が得られます。しかし、ご存知のようにネットワークのパフォーマンスはそこまで高くはありませんので、その方法は現実的ではありません。なぜ現実的ではないのか？もちろん、それはデータ量が莫大になりますし、メッセージを頻繁にやり取りすることになりからです。なお、メッセージサイズの小さいゲームであればこれは当てはまらないことになります。自分と敵の座標のみをやり取りすれば同期が得られるような場合は、全てのオブジェクトのデータをメッセージとして送受信するべきでしょう。（ただし、その場合でも間隔は長くとらなければなりません）
また、メッセージ送受信間隔がゲームの性質上そもそも長いゲームもあるます、例えばカードゲームや将棋、囲碁、あるいはモノポリーなどのボードゲーム、要するにターン制のゲームがそうです。ターン制のゲームでは基本的に、差し手の交代がそのまま送受信間隔と考えられます。差し手の交代間隔はたいてい分オーダーか、短くても数秒オーダーでしょう。（ミリ秒オーダーで差し手が交代するなんて状況はありませんよね？）、たとえ差し手が 1 秒で交代したとしても、同期にとって 1 秒は非常に長く十分な時間です。ターン制のゲームでは、メッセージ欠損にたいする対処としての同期を考えなくてもいいかもしれません。
しかし、ゲームは、そのような同期のとり易いものばかりではなく、同期が困難であるゲームが多く存在します。ここでの議論は、全てのゲームに対処できるような議論でなければなりせん。例えば、本書サンプルゲーム CW2004 ではリアルタイムにフレームが進行し、なおかつ全オブジェクトの総数は最大で 20000 個を超えます。その全ての状態を各 PC 間でやり取りするとなるとメッセージサイズは数メガバイトにもなります。これはちょうど、ゲームのセーブファイルの内容をまるごとやり取りすることと同じなので、送受信というよりは、もはやダウンロードの域です。これは火を見るより明らかに不可能です。
では、現在のネットワーク性能で、どうやってゲーム内大量オブジェクトの同一性を実現するのでしょうか？
これには、うまい方法があります。各プレイヤーの操作のみをやり取りすればいいのです。プレイヤーのキーボードやマウスの押下情報を入力メッセージと呼びます。上手くコーディングすれば入力メッセージのみにより各 PC 上で同一のゲーム状態を実現できます。実際 CW2004 が送受信しているのは入力メッセージ（と多少のフレームメッセージ）だけです。もし、読者が 2 つのモニターでゲーム画面を確認することができるなら、確認してみてください。ゲーム内の全オブジェクト（宝石、木、建物、人間、戦車、土の上のクレーターなどなど）は完全に一致しているのが確認できると思います。
ゲームと言っても、PC からすれば、言うまでもなく計算をしていることです。同じ式であれば PC が異なっても同じ答えを出します。全オブジェクトの状態を、その計算結果と考えると、同じになるのも当然というわけです。また、何らかの操作をするということは、その式を変更することであり、入力があったときに、式が変わったことだけ（入力メッセージだけ）を送受信すれば、あとは各 PC がローカルに式を計算し、当然、式の答えは全ての PC で同じものになります。
式の計算はローカル処理なので、必ず同じ答えになりますが、この “ 同じ式 ” に保つために送受信がからむので、そこに不安定さが出てきます。もう気付いたかもしれませんが、そこに同期の必要性が出てきます。
なお、ゲーム内のオブジェクトが乱数を利用している場合は同じ式でも違う結果がでるのではないかという疑問を持った読者もいるでしょう。乱数は各 PC で異なるのが普通であり、そのまま使ったのではもちろんすぐに同期が崩れます。乱数はそのまま使用せずに、乱数テーブルを作成し、あらかじめ全 PC に送受信します。ローカル PC 内でも乱数の本質は乱数テーブルを利用した擬似乱数にすぎません。異なる PC で異なる乱数を発生するのは、異なる乱数テーブルを使用する結果です。ですので、乱数のもとである乱数テーブルを全く同じものにしてやれば、全 PC が同じ乱数を生成するというわけです。乱数という言葉に惑わされないでください。実体は単なる数値テーブルです。このようにすれば乱数という名の単なる数値テーブルなのです。

フレーム志向の同期

フレームを基準に全 PC のタイミングをシンクロさせます。と言っても、全 PC を 1 フレーム単位でシンクロさせたのでは、頻繁に待機状態になる PC が出てしまいますので、数フレームをまとめた “ 単位フレーム ” でシンクロさせます。
シンクロさせるために、各 PC 上で単位フレーム内の最終フレームに到達した時点で、“ 単位フレーム終了メッセージ ” を他の PC 宛に送信します。各 PC は、自分以外の PC からのフレーム終了メッセージが全て揃うまで待機します。（フレーム進行を停止します）
そして、操作メッセージには、そのデータが発生したフレームの番号と、さらに、同じフレーム内での発生順番もヘッダー情報として付けます。（同一フレーム内で複数のデータが発生することは良くあります）このようにメッセージに発生した順番を付けておくことは大切です。
送信は、何らかの操作があった場合に、その時点ですぐに非同期送信します。（この場合の非同期とは、ここでの同期とは別の意味です。）非同期送信する理由は、すぐに送信メソッドが呼び出しスレッドに制御を返すようにするためです。同期送信を行うと、送信メソッドは送信が完了するまで制御を返してくれないので、深刻なタイムロスとなるからです。
受信は、常に行います。つまり、いつでも、アンテナを張っている状態にしておきます。自分に届くメッセージは、フレーム番号がまちまちなメッセージが届きます。時には、自分の単位フレームとは異なるメッセージも届きます。
自分も含めた（これがミソです）全員分のありとあらゆる受信メッセージを保存しておき、単位フレームの最初や最後のフレーム、とにかく 1 フレームで、発生順番どおりに順次処理していきます。自分も含めるということは、その数フレーム内で発生した自分のメッセージをローカルに処理していないということを意味します。これは、同期にとって重要なテクニックです。自分以外の人のメッセージを一括処理の際に処理しているのに、その前に自分のメッセージだけローカルに処理してしまっては、同期が壊れるからです。つまり、同期の際には、自分自身さえもリモートプレイヤー（他のプレイヤー全員）として扱い、特別扱いしないということがカギで、それにより、同期しやすくなります。具体的に言うと、自分の操作を送信し、それを自分で受信して、他のプレイヤーメッセージと同じように処理するということです。したがって、ネットワークが遮断されれば、自分自身の操作さえも反映されなくなります。

入力と反映をずらす

複数のフレームをまとめて処理すると、ある時点での入力が反映されるのは、フレーム一括処理の時です。つまり、入力と、

その反映結果にズレが生じるのです。
このようなズレは結果的に発生するものですが、それとは別に、さらに意図的にズレを発生させます。数フレームを 1 つの単位フレームとすると、実際的には 2 単位フレーム以上ずらします。その結果、場合によっては、入力と、ゲームの状態変化のズレが体感できるほど表面化しますが、フレーム進行をスムースに進めるためには、必要なことです。これは、一言で言えば、ネットワークのタイムラグをそのズレで吸収するということです。言わば、車のハンドルの遊びのようなもので、これが無いと相対的に速い PC は断続的に待機状態になりゲームになりません。詳細説明の部分で理由を解説します。

ソリューションの詳細

通常、各 PC の性能はまちまちなのが普通です。それは、フレーム進行のスピードもまちまちであることを意味します。全 PC の中で相対的に速い PC は、当然フレームも速く進みます。速い PC から送信されたメッセージは、遅い PC 側では、まだ到達していない将来のフレームメッセージになる可能性があり、逆に、遅い PC から送信されたメッセージは、速い PC 側では、過去の単位フレームメッセージである可能性があります。PC の性能が全く同じであっても、ネットワークスピードの違いがありますので、可能性というより、通常は常にそうゆう状態になります。このようなメッセージを遅延メッセージと呼ぶことにします。

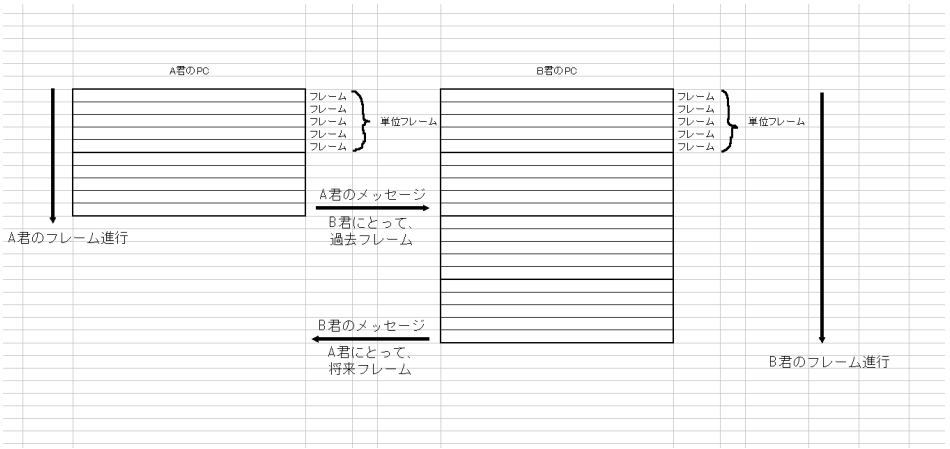


図 17-1

フレーム終了メッセージ

遅延メッセージの問題には、フレーム終了メッセージにより対処します。全 PC の単位フレーム進行を同期させるために、各 PC は自分の単位フレームの最後にフレーム終了メッセージを送信します。各 PC は、自分も含めた全 PC のフレーム終了メッセージを受信してから、次の単位フレームに進みます。このようにすれば、単位フレームをベースとした同期が得られます。

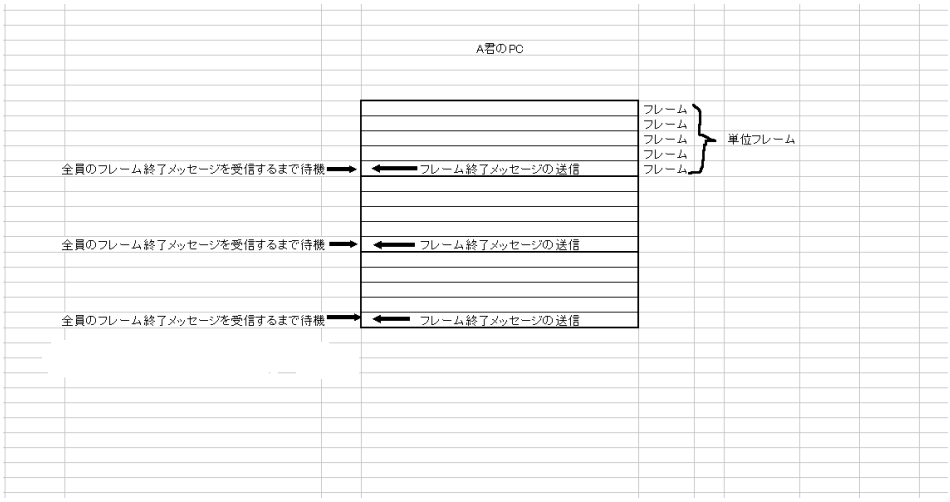


図 17-2

…と、言いたいところですが、それほど簡単にはゆきません。
フレーム終了メッセージにより、遅延メッセージの問題は解決するのですが、新たな問題が発生します。単位フレームの最後で全員の終了メッセージをチェックしてから進むため、速い PC は、単位フレームの最後で必ずウェイティング状態に陥ることになります。

ウェイティングへの対応

ウェイティングには、次の対応をします。
１．単位フレームを意図的にズラす（“ 遊び “ を持たせる）。（予防的対処）
２． メッセージを保存する。（積極的対処）

単位フレーム進行に “ 遊び ” を持たせる

メッセージ処理タイミングに処理するのは、現在の単位フレームではなく、その前の単位フレームのメッセージを処理します。これは先にも述べましたが、いわゆる “ ハンドルの遊び ” を作ることになります。“ 遊び ” を持たせることにより、送受信によるタイムラグを吸収できます。相手のメッセージは図の遊びの範囲内に届けばいいので、その分余裕が生まれます。できれば、さらにその前の単位フレームのものを処理するようにすれば、より円滑な同期が得られます。ただ、単位フレームをさかのぼればさかのぼるほど、コードは煩雑さを増します。ちなみに、CW2004 は、処理タイミングより 2 個前の単位フレームのメッセージを処理しています。
なお、“ 遊び “ を持たせることにより、先に解決したはずの遅延メッセージの問題を自ら再発させてしまう結果になりますが、フレーム終了メッセージによって、遅延メッセージの範囲は “ 遊び “ の範囲内であることが把握できるので、これには対処できます。

メッセージの保存

単位フレーム進行に “ 遊び ” を持たせることによって、現在の自分の単位フレームより前のメッセージや、自分から見て将来のフレームのメッセージが混在して届くことが考えられます。
これに対処するために、受信したメッセージは、そのまま処理せずに 1 クッションおきます。何らのバッファーを用意して、受信したメッセージを、まずそのバッファーに保存します。
メッセージの処理タイミングに、バッファーの中から同じ単位フレーム番号のメッセージだけを取り出し、処理すればいいわけです。

単位フレーム内での発生順番

フレーム終了メッセージは、単位フレームの最後に必ず 1 回だけ送信するものですが、操作メッセージ（入力メッセージ）は、何らかの操作があった時に送信されます。したがって、同じ単位フレーム内で、複数の入力メッセージが発生する可能性があります。そのため、入力メッセージには単位フレーム番号に加えて、フレーム番号もヘッダー情報として付けます。
バッファーに保存されている入力メッセージを処理する際には、まず、単位フレームが同じものを取り出し、さらに、フレーム番号順に処理することになります。同じ単位フレーム内でも、フレーム番号を無視して（発生順番を無視して）処理すると同期が崩れます。

以上の手順をまとめたものが図 17-3 です。

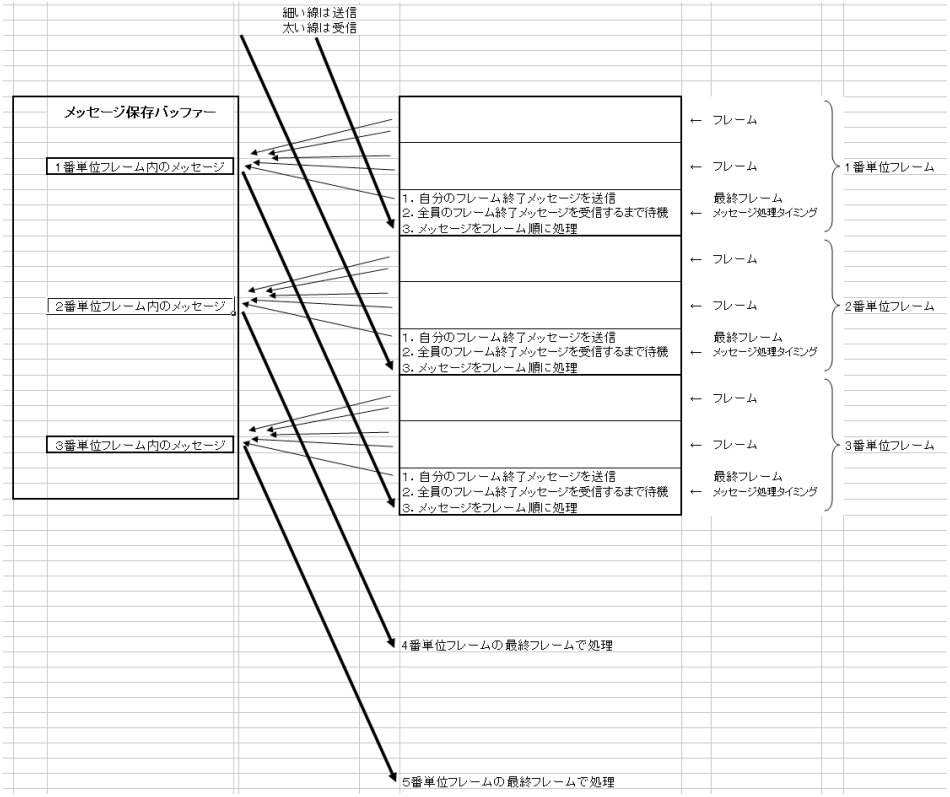


図 17-3

Chapter18 3D ダイナミクスの基礎

第 13 章において、レイとメッシュの交差判定に D3DXIntersect 関数を使用しました。
D3DIntersect 関数は、レイとメッシュの交差を判定しますが、ここでの解説は、レイと平面、つまりポリゴンとの判定手順です。
メッシュは、時として何千何万個のポリゴンから成る場合があり、そのようなメッシュに対して D3DIntersect 関数を実行するとかなりの時間がかかってしまうことになります。それを回避するためには、何らかの代替手段をとる必要がありますが、それを可能にするためには、ポリゴン単位での衝突判定方法は知っておくべきでしょう。

線分と平面の交点

直線と平面の交点を求めるプロセスは概ね次のようになります。
1. 平面方程式を求める。（この場合の平面は無限遠平面）
2. 線分と無限遠平面の交点を求める。
3. その交点が、意図する有限平面内にあるか内外判定をする。
内外判定で、交点が有限面内にある場合に衝突とする。

無限遠平面の方程式

コード上で平面を表現するにはどうしたらいいのでしょうか？
答えは " 平面方程式 " により表現するということです。例えば Direct3D で定義されている D3DXPLANE という平面構造体は、aX+bY+cZ+d=0 という一般平面方程式の係数から成っています。

```
typedef struct D3DXPLANE
{
    FLOAT a;
    FLOAT b;
    FLOAT c;
    FLOAT d;
} D3DXPLANE;
```

平面方程式が表す平面は、無限の面積をもつ無限遠変面です。いわゆる一般的な閉じた平面を表現するには、平面の内と外を区別する何らかの範囲が必要です。平面方程式には、そのような範囲を意味する係数はありません。また、D3DXPLANE 構造体を見ても分るように、面の範囲を表すようなメンバー変数はありません、面の傾きを表す 3 つの FLOAT 値（a,b,c）と原点からの距離を表す FLOAT 値 (d) が 1 つあるだけです。無限というと特別な意味を感じるかも知れませんが、一般的に方程式で現われるジオメトリは無限遠です。これは、直線でも同じことです、お馴染みの直線方程式 y=ax+b が表すのは、無限遠直線 (傾きと原点からの距離) であって、両端点がある線分ではありません。線分を表現するには、何らかの範囲情報が必要です。これが、交点の面内外判定をしなければならない理由です。

平面方程式を求める

コード上では、最初から平面方程式になっている場合もあるでしょうし、動的に平面方程式を求める必要もあろうかと思います。

任意の 3 点から平面を求める

D3DX ヘルパー関数の中に、そのものずばりの D3DXPlaneFromPoints 関数がありますが、ここでは自前で同様の関数を作っていきましょう。
与えられた 3 点をもとに、3 角形の 2 辺を導きます。その 2 辺の外積を計算すれば、平面の法線ベクトルを得ます。法線ベクトルの成分は平面方程式の各係数に対応しているので、この時点で平面方程式 ax+by+cz+d=0 の係数 a,b,c が一気に得られる形になります。
あと残るはスカラー値 d ですが、d は、平面内の任意の点を P とすると、次のように求めることが出来ます。
d=-(a * P の X 成分　+ b * P の Y 成分 + c * P の Z 成分)

```
HRESULT CalcPlane(D3DXPLANE* pPlane,D3DXVECTOR3* pvecA,
                                    D3DXVECTOR3* pvecB,D3DXVECTOR3* pvecC)
{
        // 辺ベクトル
        D3DXVECTOR3 vecAB,vecBC;
        vecAB=*pvecB-*pvecA;
        vecBC=*pvecC-*pvecB;
        // 平面法線ベクトル
        D3DXVECTOR3 vecNormal;
        D3DXVec3Cross(&vecNormal,&vecAB,&vecBC);
```

```
        // 法線は平面の傾きでもあるので、そのまま代入
        pPlane->a=vecNormal.x;
        pPlane->b=vecNormal.y;
        pPlane->c=vecNormal.z;
        // d を計算
        pPlane->d=-(pPlane->a*pvecA->x + pPlane->b*pvecA->y
                + pPlane->c*pvecA->z);

        return S_OK;
}
```

理論を、そのまま素直に分り易くコードにしたので、実行速度は改善の余地があります。高速化は読者にお任せします。

“d” の意味は？

平面方程式 ax+by+cz+d=0 の a,b,c がそれぞれ x,y,z の係数になることから、それらが面の傾きを表すことは、なんとなく分ると思います。では、d は何を表しているのでしょうか？
もし、”d” が無ければ（ゼロであれば）、その平面が原点を通ることを意味します。
“d” は、その平面の“原点からの距離”です。別の言い方をすれば、“その平面と平行で、かつ、原点を通る平面“からのシフト量（距離）とも言えます。

線分と無限遠平面の交点

線分とは、2 つ端点を持ち、その長さが有限な直線のことです。
線分の端点を P0(x0,y0,z0) と P1(x1,y1,z1) とすると、線分上の任意の点 P(x,y,z) は、媒介変数（パラメーター）t と端点同士の方向ベクトル V(vx,vy,vz) を用いて次のように書けます。

```
X=x0+t*vx
Y=y0+t*vy
Z=z0+t*vz;
```

但し 0<=t<=1

t がゼロの時は一方の端点と重なり、t が 1 のときは、他方の端点と重なります。t がゼロと 1 の間では、端点と端点の間の点を表すのですから、すなわち線分を表しているというわけです。パラメーターを用いたこのような表現をパラメーター形式とかパラメトリック方程式と言います。線分をパラメーター表示することにより、シンプルなコードで交点を求めることができます。

パラメーター表示した線分上の点を平面方程式に代入すると次のようになります。

```
A(x0+t*vx)+b(x0+t*vx)+c(x0+t*vx)+d=0
```

これを、t について解くと

```
T= -( a*x1+b*y1+c*z1+d ) / ( a*vx+b*vy+c*vz )
```

ベクトル V(vx,vy,vz) は、端点から求めたものなので、次のように、完全に端点と平面方程式の係数のみで表すこともできます。

```
T=-( a*x1+b*y1+c*z1+d ) / ( a*(x0-x1)+b*(y0-y1)+c*(z0-z1) )
```

あるいは、

```
T=-( a*x1+b*y1+c*z1+d ) / (( a*x0+b*y0+c*z0) - ( a*x1+b*y1+c*z1))
```

ここまで式が導出できれば、あとは簡単です。t を計算して、t がゼロから 1 の範囲外の場合は交差していない、 t がゼロ以上 1 以下の場合は交差しているということになります。その理由はおわかりでしょう、t がセロから 1 の範囲外ということは、線分の範囲外ということなのですから、交差しているわけがありません。
次のコードは、交点を求める関数ですが、この式をそのまま使用していることが分ると思います。

```
// 交差する場合は TURE を、交差しない場合は FALSE を返す
BOOL Intersect(D3DXPLANE* pPlane,D3DXVECTOR3* vecA,D3DXVECTOR3* vecB)
{
 FLOAT fT=0;
 fT=-( (pPlane->a*vecB->x)+(pPlane->b*vecB->y)
 +(pPlane->c*vecB->z)+pPlane->d)    /
```

```
  ( ((pPlane->a*vecA->x)+(pPlane->b*vecA->y)+(pPlane->c*vecA->z))
  - ((pPlane->a*vecB->x)+(pPlane->b*vecB->y)+(pPlane->c*vecB->z)) );

  if( (fT<0.0f) || (fT>1.0f))
  {
     （交差している時の処理）
    Return TRUE;
  }
  Return FALSE;
}
```

交点が面の範囲内か調べる

無限に広がり、範囲を持たない平面との交点なので、交点は意図した有限平面の面内には無い場合もあります。そこで、交点が面の内部にあるのか、面の外部にあるかの判定が必要になります。面の外部にあるときは、線分は、意図した平面（無限遠平面ではなく、頂点をもった平面）とは交差していることになりせん。

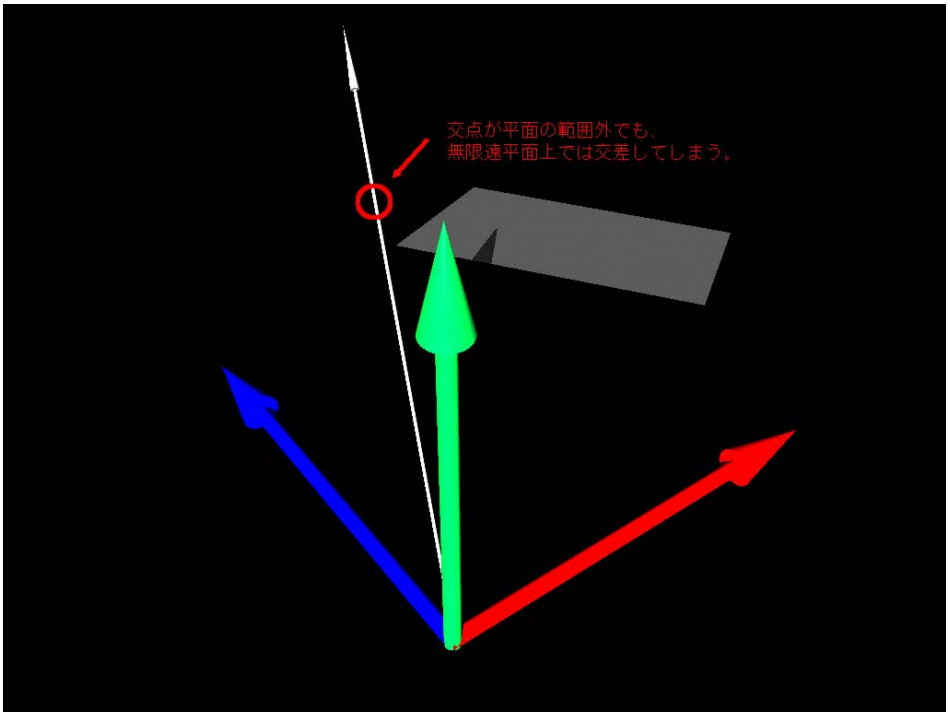


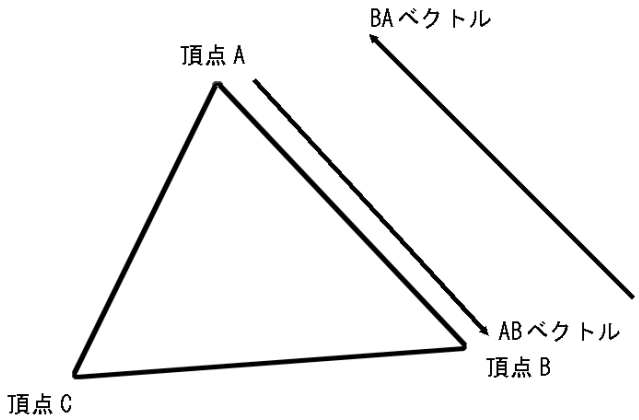
図 18-1

このサンプルコードを理解するうえで、注意しなくてはならないことがあります。ここでは、頂点と辺の組み合わせや、ベクトルの向き、及び外積をかける順番が大切ですが、背景理論を理解しないと、非常に複雑な組み合わせを丸暗記しなくてはなりません。これらの組み合わせのどれか一つ間違っても正しい結果は得られません。ここでは、丸暗記ではなく、なぜそのようななるのか、その仕組みを理解してください。仕組みが理解できれば、組み合わせは自由に変えることが出来ます。

まず、最初に次の事項を確認しておきたいと思います。

辺の表記

３つの頂点からなる平面（3 角形）の頂点それぞれ A,B,C とします。辺 AB と表記した場合、図 18-2 のように、A から B へ向かうベクトルです。B から A のベクトルではありません。（その場合、辺 BA と表記します）



頂点をベクトルとすると
ABベクトル＝Bベクトル－Aベクトル
BAベクトル＝Aベクトル－Bベクトル

図 18-2

外積について

外積はある２つのベクトル双方に垂直なベクトルを作るので、平面の法線や平面方程式を求める時によく使用しますが。図18-4のベクトルAとベクトルBに垂直なベクトルはN1とN2の２つあります。しかし､外積が返すベクトルは１本です､さて､どちらでしょう？

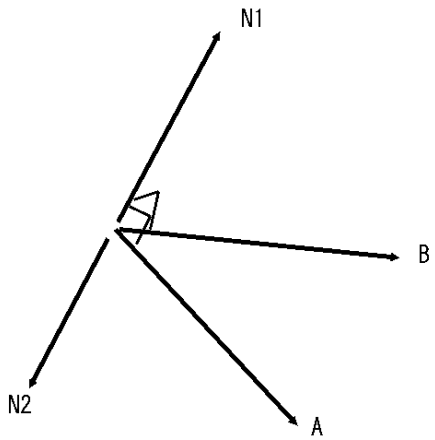
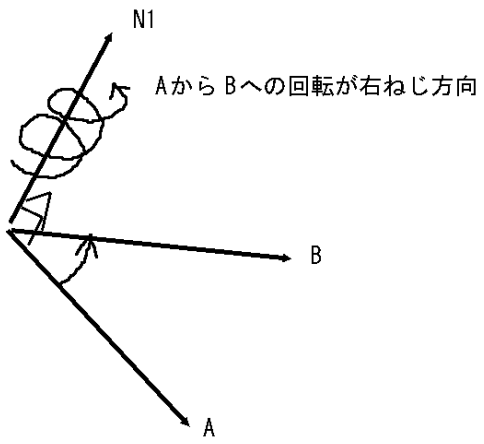


図 18-3

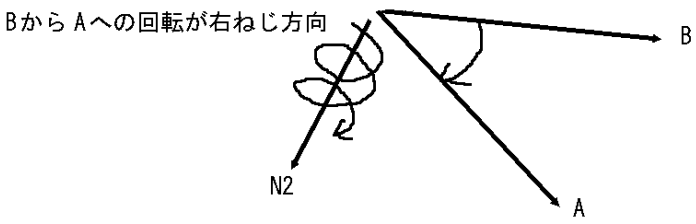
答えは、この文章だけでは、判断できません。
幾何的に言うと、
どちらのベクトルを右ねじの始点とするか、

また、計算式で言えば、
どちらのベクトルが先に書かれているか、
によって N1 か N2 の一方が外積結果になります。
A から B への回転が、右ねじになるような方向へ向かうベクトルが外積結果ベクトルです。



A X B = N1（Xは外積を意味する）

図 18-4



B X A = N2（X は外積を意味する）

図 18-5

図 18-4 では N1 がそうです。右ねじとは、ねじを締める回転方向です。あるいは水道の蛇口を閉める回転方向をイメージしてみてください。これを、式で書くと A に B を掛ける形になります。
A x B = N1
こう考えても良いでしょう、式で左に書いたベクトル (A) から、右に書いたベクトル (B) への回転が右ねじになる方のベクトルが外積結果としてのベクトルである。
B と A を逆にすると、外積の結果も逆（N2）になります。

B　x A　=　N2 = -N1
つまり、外積で、掛ける順番を反転させると結果ベクトルの符号も反転するということです。
なにか、ややこしくなってきましたが…重要なことなので我慢してください。

ベクトルの外積と内積を利用した判定

面は３角形として処理します。理由は３角形が最もシンプルな平面だからです。判定する平面が、それ以上の多角形の場合は３角形に分割してから、全ての３角形それぞれに対してこの判断をします。
判断の流れを簡単に述べると次のとおりです。

1. 辺をベクトルと考えます。辺ベクトルと、その辺ベクトルの始点となる頂点から交点へ向かうベクトル、２つのベクトルの外積を計算します。外積の結果はベクトルになりますので、３本の結果ベクトルが求まります。
2. ３つの結果ベクトルそれぞれと平面の法線ベクトルとの内積を計算します。内積の結果はスカラー値になりますので、３つの値が求まります。
3. その３つの値の符号で、面内判定ができます。３つの値の符号が全てプラスなら、交点は面内部にあり、一つでもマイナスなら交点は面の外にあることになります。

では、なぜ外積と内積で面の内外が判断できるのか、その仕組みを解説します。
この内外判定の原理において、内積はそれほど重要ではなく、内積を使用しなければ使用しないでも支障はありません。その場合、若干コードが増えるだけです。しかし、外積の性質はカギとなるものです。外積を利用しないで面積等で内外判定しようとすると、多くの場合分けが必要な複雑なコードを書かなければなりません。
まず交点が面の内部にある場合からか考えてみましょう。
頂点 A から交点へ向かうベクトルを V とします。ベクトル V と辺 AB ベクトル（BA ベクトルではありません）の外積を計算します。
式は　“辺ベクトル X　ベクトル V”　です。掛ける順番には注意してください。結果のベクトルは D1 とします。

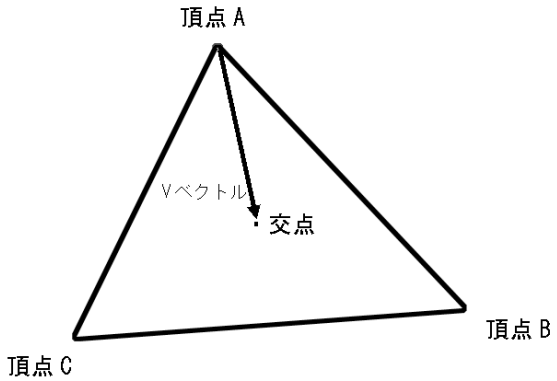


図 18-6

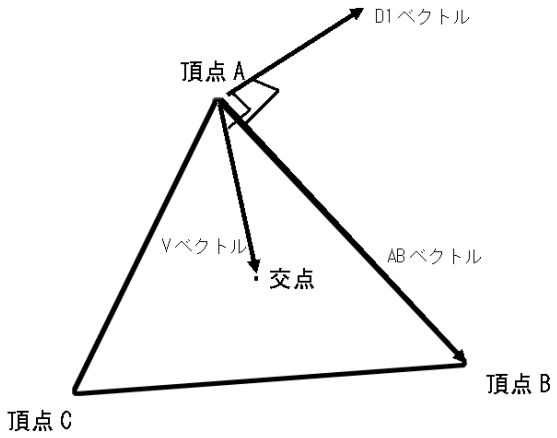


図 18-7

同様な手順を他の 2 頂点についても行います。
結果ベクトル D1,D2,D3 が出ました。結果ベクトルの方向に注目してください。

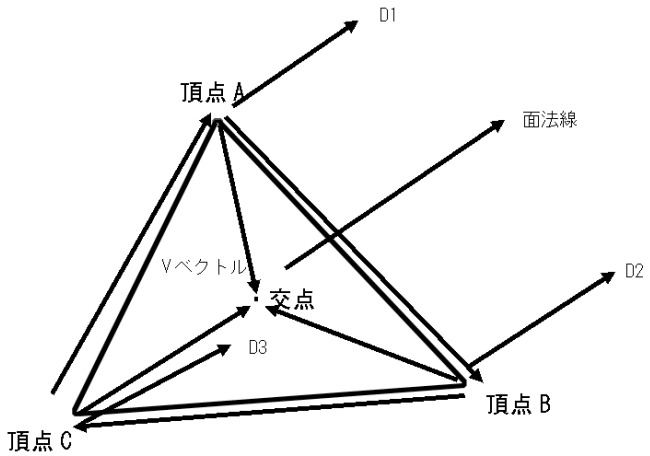


図 18-8

3 本とも、面の法線ベクトルと同じ方向ですね、式で辺ベクトルを左側に書いているので、交点が内側にある場合は、必ず、“ 辺ベクトルからベクトル V への回転を右ねじとするベクトルが外積の結果ベクトル “ になるので、そうなります。これは、外積の性質から当然です。
では、今度は交点が面の外にある場合を考えてみましょう。
同じように辺 AB　x ベクトル V　を計算します。この場合、辺 AB からベクトル V への回転は、内側の時と逆になります。したがって、外積の結果ベクトルは、法線とは逆の方向に向かいます。読者は、もう気付いたのではないでしょうか？この時点

で違いが出たのです。“違い”を出すことができれば、判断することは簡単です。
あとは、3本の結果ベクトルそれぞれと法線の方向が同じか違うかを比べればいいだけです。方向の違いは、内積を使えばあっさり計算できます。それぞれ法線との内積をとり、一つでもマイナスになった場合、それは法線と逆方向を意味し、交点が面の外にあるという判断をすることができます。

では、実際のコードを見ていきましょう。

```
BOOL InOrOut(D3DXVECTOR3* pvecI,D3DXVECTOR3* pvecA,D3DXVECTOR3* pvecB,
                                D3DXVECTOR3* pvecC)
{
        // 辺ベクトル用
        D3DXVECTOR3 vecAB,vecBC,vecCA;
        vecAB=*pvecB-*pvecA;
        vecBC=*pvecC-*pvecB;
        vecCA=*pvecA-*pvecC;
        // 辺ベクトルと「頂点から交点へ向かうベクトル」との、それぞれの外積用
        D3DXVECTOR3 vecCrossAB,vecCrossBC,vecCrossCA;
        //「外積結果のベクトル」と平面法線ベクトルとの、それぞれの内積用
        FLOAT fAB,fBC,fCA;
        // 法線用
        D3DXVECTOR3 vecNormal;
        // まず、3頂点から平面方程式を求める。これは、同時に平面の法線を求めることでもある
        D3DXPLANE pln;
        D3DXPlaneFromPoints(&pln,pvecA,pvecB,pvecC);
        vecNormal.x=pln.a;// 法線の x 成分は平面方程式の x 係数
        vecNormal.y=pln.b;// 法線の y 成分は平面方程式の y 係数
        vecNormal.z=pln.c;// 法線の z 成分は平面方程式の z 係数
        D3DXVec3Normalize(&vecNormal,&vecNormal);

        // 各頂点から交点 I に向かうベクトルを vecV とする
        D3DXVECTOR3 vecV;
        // 辺 AB ベクトル（頂点 B ベクトル - 頂点 A ベクトル）と、頂点 A から交点 I へ向かうベクトル、の外積を求める
        vecV=*pvecI-*pvecA;
        D3DXVec3Cross(&vecCrossAB,&vecAB,&vecV);
        // 辺 BC ベクトル（頂点 C ベクトル - 頂点 B ベクトル）と、頂点 B から交点 I へ向かうベクトル、の外積を求める
        vecV=*pvecI-*pvecB;
        D3DXVec3Cross(&vecCrossBC,&vecBC,&vecV);
        // 辺 CA ベクトル（頂点 A ベクトル - 頂点 C ベクトル）と、頂点 C から交点 I へ向かうベクトル、の外積を求める
        vecV=*pvecI-*pvecC;
        D3DXVec3Cross(&vecCrossCA,&vecCA,&vecV);
        // それぞれの、外積ベクトルとの内積を計算する
        fAB=D3DXVec3Dot(&vecNormal,&vecCrossAB);
        fBC=D3DXVec3Dot(&vecNormal,&vecCrossBC);
        fCA=D3DXVec3Dot(&vecNormal,&vecCrossCA);

        // 3つの内積結果のうち、一つでもマイナス符号のものがあれば、交点は外にある。
        if(fAB>=0 && fBC>=0 && fCA>=0)
        {
                // 交点は、面の内にある
                return TRUE;
        }
        // 交点は面の外にある
        return FALSE;
}
```

理論を順番どおりに、かつ忠実にコードにしたもので、理論と違う箇所は一つもありません。さらに、コメントを細かく書いているので、解説することが何もなくなってしまいました。しいて解説するとしたら、次の箇所くらいだと思います。

```
D3DXPLANE pln;
        D3DXPlaneFromPoints(&pln,pvecA,pvecB,pvecC);
        vecNormal.x=pln.a;// 法線の x 成分は平面方程式の x 係数
        vecNormal.y=pln.b;// 法線の y 成分は平面方程式の y 係数
        vecNormal.z=pln.c;// 法線の z 成分は平面方程式の z 係数
        D3DXVec3Normalize(&vecNormal,&vecNormal);
```

Pln は平面方程式です。D3DXPlaneFromPoints 関数により、3頂点から平面方程式を導出しています。（ここで、先ほどの自

前の関数で平面方程式を使用しても同様の結果がでます。）平面方程式の各係数は法線ベクトルの成分になるので、そのまま代入します。ここでは、法線が求まれば良いので、平面方程式の d 値は不要です。

さて、これで晴れて衝突判定ができることになります。D3DCIntersect 関数は、メッシュ（ポリゴンの集合）にたいての判定ですが、ここでの解説はポリゴン単位で判定しますので、柔軟な判定が可能となります。

衝突後の挙動

物体が壁に衝突した時の処理で最も簡単なのは、そこで動きを止めることでしょう。少なくとも壁のすり抜けは防げるわけですからそれでも悪くはないですが、市販の３D ゲームのように、壁に沿って滑るようにすれば、より操作し易く、ユーザーに親切といえるでしょう。また、急勾配の坂にいるときなどは、滑らせないと不自然です。
どのようにして、滑らせるのか、言い換えると、障害物に沿って移動させるのか、その原理は比較的簡単です。
滑らせる処理の前には、衝突判定が既にされているわけですから、衝突判定の際に使用したレイと障害物の平面方程式はすぐ利用できるはずです。その２つを利用して滑らせます。なお、D3DXintersect 関数などでメッシュに対して衝突判定をした場合は、メッシュからレイが当たったポリゴンを調べて、そのポリゴンの平面方程式を出すことになります。（平面方程式の出し方はわかりますよね？先に解説した方法です）

ここで､平面方程式から､滑るべき方向への方向ベクトルを求めます。滑るべき方向は平面内のベクトルのどれか一つです。(当たり前ですね）ただ、平面方程式は平面内のありとあらゆるベクトルを表しますから、候補に挙がるベクトルは無限個あります。どのベクトルを使用しても壁に沿うことはできますが、衝突した方向に対して不自然な方向に滑ってしまいます。

平面上に無限にあるベクトルのなかで、一本のベクトルを見つけるは次の方法をとります。
まず、レイの始点から平面に垂線を下ろします、垂線と平面の交点から、レイと平面の交点へ向かうベクトルが滑る方向ベクトルです。
なお、垂線もまたレイとして平面と交差判定し、その交点を得ます。また、垂線ベクトルは法線ベクトルの逆向きベクトルなので、これもまた最初から、求まっているわけです。

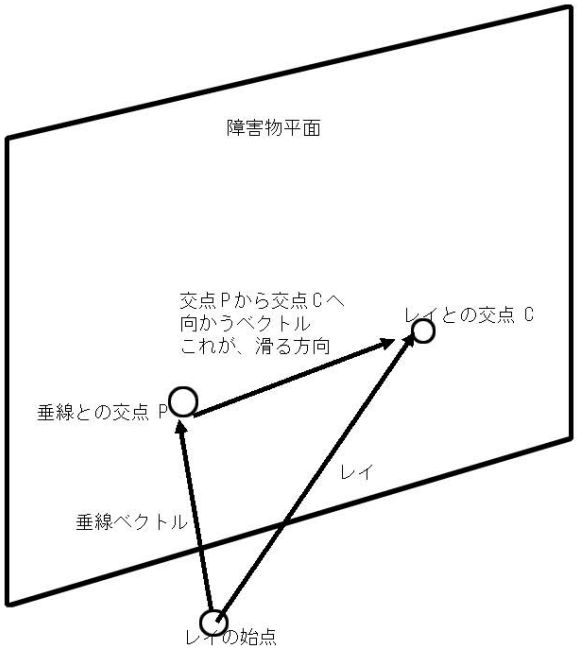


図 18-9

1 人称３D フィールドサンプル解説

このサンプルは、本章で解説したことを実際にコーディングしたものです。衝突判定、衝突後の処理は、全てここでの解説をそのままコーディングしています。

1 人称３D フィールドサンプル解説

このサンプルにおける、（視点の）回転と移動、衝突判定、衝突後の処理は、全て本書で解説した原理をそのままコーディングしています。

操作説明

操作は、キーボードとマウスについて、それぞれ次のように対応させています。

前進　W キー
後退　スペースキー
左平行移動　A キー
右平行移動　D キー
ジャンプ　マウス右ボタン
視点回転　マウス左ボタンを押しながらマウスを動かす（ドラック）
衝突判定用ポリゴン表示トグル　F1 キー

前進だけでも、視点を回転させることにより、意図する方向に進むことができます。

用いたロジック

このサンプルでの主要な各種処理は、全て本書で解説したロジックによりコーディングしました。次に、視点関係、衝突判定、衝突後の挙動の３つについて補足します。

視点の回転と平行移動

視点の移動は、第 11 章「現在の姿勢からの平行移動」の原理に基づいてコーディングしています。前進する方向は、あくまでも現在向いている方向であり、絶対的な Z 方向ではありません。例えば、視点が右を向いている状態で前進する場合は絶対座標系で見ると、視点は X 軸マイナス方向に移動することになります。主人公の回転にシンクロして、主人公固有の XYZ 軸そのものを回転させ、その回転させた Z 方向に進めば自然な前進になるというわけです。これは、現実の移動と同じ自然な移動です。そのため、前進ボタンと視点の回転を組み合わせることにより、横移動ボタンを使用しなくても意図する方向に移動できます。

衝突判定

衝突判定における大きな２つの手法について、第 12 章では境界ボリュームを解説し、第 13 章ではレイを解説しました。このサンプルでは、この２つの手法を併用しています。
まず、境界ボリュームで大まかな判定を行い、そこで衝突している場合に限り、更にレイにより精度の高い判定を行っています。２つの手法を併用している理由は２つあります。
1 つ目の理由は、計算コストの削減にあります。第 12・13 章で解説したとおり、レイによる判定は境界ボリュームに比べ計算コストが大きい処理、平たく言えば重たい処理です。常にレイを張り巡らせてしまうと、それだけ処理速度が低下してしまうため、常に計算することは避けたいところです。そこで、常に判定するのは境界ボリュームにより行い、境界ボリュームで衝突している場合に、はじめてレイにより厳密な判定を行います。このようにすれば、処理速度はかなり稼げます。
2 つ目の理由は、境界ボリュームだけでは、衝突後に視点が障害物を滑るようにするためにの処理に必要なレイを得ることができないためです。レイにより衝突判定を終了していれば、その後の滑らかな視点移動処理が計算し易くなります。
なお、このサンプルでは、メッシュに対して境界ボリューム判定及びレイ判定は行わっていません。メッシュそれぞれに対して、別個に “ 衝突判定用ポリゴン ” を作成し、そのポリゴンの境界ボリュームにより初期判定を行い、また、そのポリゴンにレイを当てています。
これは、メッシュにそのまま判定を行うよりも高速に判定を行うためです。当然、判定用ポリゴンは非常にシンプルなジオメトリであり、4 頂点ポリゴンを使用しています。筆者は、この判定用ポリゴンを “ 壁面ポリゴン ” あるいは単に “ 壁面 ” と呼び、その作成には自作ツールを使用しています。ツールといっても単独アプリケーションではなく、ライトウェーブ用 (6.0 以上) のプラグインとして開発しました。一応、そのプラグインも添付ディスクに収録しています。プラグインの名前は shigeo.p で、モデラー用です。モデラーのプラグインリストには ”shigeo” と表示されます。
プラグインの仕様は、シンプルです。(内部コードもシンプルですが…) まず、ライトウェーブのモデラー上で、単純な板ポリゴンを作成します。ポリゴンの頂点数は任意ですが、言うまでもなく、少ない頂点数で作成するべきです。３～５頂点あたりでポリゴンを作成します。そのポリゴンのサーフェイス名を “WALLPLANE” とし、プラグインを起動します。出力先を聞いてきますので、適宜指定します。OK を押せば、指定したディレクトリに拡張子が .sx であるファイルを吐き出します。このファイルはテキストファイルであり、データが意味するものは、頂点の座標を羅列しただけの単純なものですので、利用方法は色々あると思います。
このような独自の手法を取らない場合でも、メッシュを、“ レンダリング用メッシュ ” と ” 判定用メッシュ “ に分けて作成し（もちろん、判定用メッシュは低ポリゴンで作成します）、衝突判定は、判定用メッシュに対して行えば、かなり処理速度を稼ぐことができます。

衝突後の “ 滑り ”

本章の「衝突後の挙動」で解説した原理に基づいて、障害物に沿って滑るように移動するようにしています。この処理に必要な衝突時のレイは、衝突判定プロセスで既に計算しているので、あとは進入角度に対し不自然にならないような角度で、かつ、障害物の壁面に沿って移動させてやります。