はじめに

パソコンでどのようにしてゲームを作るのか？　もっと素朴な疑問として、モニターにどのようにして絵を表示できるのか？　本書の読者層であれば、少なくともその辺りの知識はあるでしょうが、一般の人は不思議に思っています。ソースコードを見せて、「この“プログラム”というもので絵を表示したり、その絵を操作したりできるんだよ」と教えても、依然として彼・彼女等の顔から疑念の色は消えません。むしろ、より一層不思議な顔をします。彼・彼女たちは、たいてい「どうしてそんな文章みたいなもので自分の作った絵が表示できるの？」と言います。この疑問には正直今でも的確な答えを持っていません。知識が無ければ無い人ほど膨大な説明が必要になります。初心者であればあるほど、教える側の人間は優れていなければならないと筆者は思います。
幸い本書は、ある程度のプログラム経験者を想定しているので、筆者のスキルでもなんとか書くことが出来ましたが、いつかは、そのような一般向けの文字通り“ゲームはどうやって作るのか？”的な書籍も執筆してみたいと思っています。
おっと、先に言ってしまいましたが、本書が想定している読者は、C 又は C++ プログラム経験者です。なお、スキルレベルは問いません。また、当然ですが DirectX の知識も問いません（本書の目的は DirectX の知識を提供することでもあるわけですら）。プログラムとは何か、コンパイラーとは何か程度のことを知っていることを最低条件とし、簡単でもいいから何らかの C/C++ プログラムを作成したことのある人、さらにできれば、ウィンドウプログラムの作成経験があることを望みます。しかし、ウィンドウプログラムと言っても、経験がなければ無いでも構いません。本書のサンプルでは単純なウィンドウを 1 つ作成するだけですから。（マルチビューレンダリングのサンプルではウィンドウを 5 つ作成しますが、たいしたことはありません）

さてここで、本書の特徴を述べさせていただきたいと思います。
本書は、基本的には DirectX の初心者、中級者を想定しています。初心者と中級者の境界線は曖昧ですが、初心者とは文字通り全く DirectX のコーディング経験が無い、あるいは DirectX という言葉すら聞いたことの無い人とし、中級者とは、簡単な 3D ジオメトリのコーディング経験がある人から、簡単な DirectX ゲームを作成したことのある人です。
また、本書は DirectX の上級者にとっても必ず役に立つものだと思っています。もし、今これを読んでいる貴方が上級者であるなら、全体にさらっと目を通してみてください。全項目の中で必ず幾つかの項目には興味を持たれることと思います。というよりも、上級者になればなるほど特定の分野に特化するもので、知っているはずの知識が、実は曖昧だったということも少なくありません。本書がそのような穴を埋める可能性は十分あると考えます。さらに 1 章 8 章 15 章を除く全 19 章ごとに 1 個以上のサンプルコードがあります。コーディングの原理は知っていても、サンプルがあると便利なことは往々にしてあります。実際筆者もそうです。プライベートなゲームプロジェクトに組み込んで使うこともあれば、学校の教材として使うこともあります。筆者が本を作る理由の半分は自分自身のライブラリとして活用するためです。当然、他のゲームスクールの先生方にも、授業で活用できるようなサンプルを意識しています。
とにかく、この本を作成するにあたり、かなり労力を費やしたことは確かな事実です。間違ってもいい加減に作成したものではありません。一般的に考えてそのような成果品が“何の役にもたたない”結果となることは考え辛いと思います。少なくとも筆者はこれからこの本を公私ともに活用するつもりです。

DirectXGraphics と Direct3D

DirectXGraphics は、視覚・ビジュアル部分を担当するコンポーネントです。ディスプレイ画面への出力はこの DirectXGraphics が担当します。つい最近までは DirectShow という主にムービー出力用のコンポーネントがありましたが、April2005 から PlatformSDK に移行しました。ディスプレイへの出力はマルチメディア、特にゲームにとっては最も重要で、言うまでも無くなくてはならない行程であるので、このコンポーネントは DirectX の中でも中心的な存在となっています。ビジュアルはゲームにとって「なには無くとも必要」なことです。DirectSound、DirectMusic によるオーディオ出力、DirectInput による操作の入力、DirectPlay の通信機能、それらは無ければ無いでもゲームとして成り立たせることはできます。ところがビジュアルは絶対的に必要不可欠な部分です。サウンドノベルであったとしても何らかの画面出力は必要なのですから。ビジュアルのこのような重要性とともに Direct3D の背景理論が非常にボリュームのあるものであるということから、どうしても DirectXGraphics の解説が中心となり、なおかつ解説のボリュームも大きくなることになります。DirectX と Direct3D を同義に表現するという間違いや DirectX と言えば DirectXGrapichs を連想するのもある意味仕方の無いことかもしれません。

昔、DirectX7 以前においては DirectXGrapichs というカテゴリーは存在せず、それに相当するものは DirectDraw と Direct3D に分かれていました。DirectDraw は 2 次元的な画面出力用のコンポーネントで、具体的にはビットマップ等のピクセルデータを出力するものです。一方 Direct3D は 2 次元、3 次元両方の画面出力のコンポーネントで、3 次元ジオメトリはもとより 2D イメージも出力することも出来ます。両者にはそれぞれ特徴があり、どちらにもメリット、デメリットがあります。DirectDraw の場合は 3 次元を扱うことは出来ない代わりに超高速な画面出力が可能になります。Direct3D の場合は、画面出力に関わることなら全てが可能ですが速度的には DirectDraw に太刀打ちできません。筆者は DirectDraw と Direct3D の両方でスプライトの単純レンダリング速度を比較したことがありますが、綺麗に 2 倍の速度差が出ました。処理速度で 2 倍という倍率はかなりの差です。したがって、完全な 2D ゲームを作成するのであれば DirectDraw を使用するメリットがあります。その場合、SDK はバージョン 7 以前のものを使うことになります。

DirectX8 から DirectDraw と Direct3D をまとめて DirectXGraphics と呼ぶようになります。しかし、DirectDraw はユーザー（開発者）からは操作できなくなったので、事実上 DirectXGraphics とは Direct3D を意味することになります。今後は DirectXGraphics を Direct3D と表現することにします。ただ、DirectX9 において DirectDraw が全く使用できないことも無く、ごく僅かな機能は使用できます。しかし、DirectX7 ほど完全に DirectDraw を使用できるわけではなく、マイクロソフトが DirectDraw の使用を “ 積極的に否定 “ しているだけあって、あくまで “ 申し訳程度 ” の機能しか使用できません。なぜ、DirectDraw の機能を使用不可にしたのか、その理由はわかりませんが、いずれにしても残念なことではあります。DirectDraw という “ カテゴリーを無くす ” ことはどうでもいいものですが、その機能、たとえば画面のピクセルと一対一に対応した操作だとかピクセルデータの高速ブリット等を Direct3D の中に “ 包含 ” して残してくれれば良かったのですが…

初心者のための簡単運用編（Direct3D）

1 章　必要なもの及び設定

本書をパソコンと併用することなく、あくまでも " 読むだけ " として活用してもらっても、もちろん構いませんが（上級者の場合はそのように活用される方は少なくないと思います）、100% 活用していただくためには、やはりサンプルソースコードを実際にビルド・実行しながらのほうが、何 10 倍も理解を深めることが出来るかと思います。
サンプルソースコードは全て VisualC++ のプロジェクトフォルダ単位で収録しています。
サンプルプロジェクトをビルド及び実行するためには必要なものは次のとおりです。

ビルド・実行に必要なもの

●パソコン
OS が Windows であるウィンドウズ PC（当たり前ですが、必要なものなので…）。
CPU のクロック周波数が 500MHz 以下である場合は、事実上動作しない（動作が異常に遅い）サンプルがあります。2000 年以降に購入したパソコンであれば、まず大丈夫です。
ただ、次で述べるビデオカードの性能が低い場合、CPU が高速でも、同様に動作しないサンプルが出てきます。Direct3D、特に本書のサンプルの実行速度は CPU よりもビデオカードの性能に左右されます。

●ビデオカード（グラフィックボード）
購入時に最初から DirectX9 に対応しているものであれば問題ありません。
一方、DirectX9 のリリース以前に発売されたビデオカードの場合は、DirectX9 に最初から対応できるわけがない訳ですが、その場合ダメなのかというとそうでもありません。DirectX9 以前のカードでも、" デバイスドライバ " をアップデートすることにより DirectX9 で動作させることが出来ます。ただ、あまりにも古い、例えば DirectX5 世代のカードの場合は、メーカーのドライバーサポートが打ち切られているため、不可能です。その場合はビデオカードを新たに購入するしかありません。
サンプルを " とにかく起動させる " という目的において重要なのは、ビデオカードのハードウェアというよりも、ドライバーというソフトウェアがまず、第一です。
次に、サンプルを " 快適に動作させる " という目的において重要なのは、ビデオカード自体、ハードウェアです。古いカードをドライバーで無理やり動作させても、ソフトウェアエミュレーションが効いてしまう確立が高くなり、そのような状況ではかなりの速度低下が起こります。また、発売時期が新しくても極端に安価なカードの場合も同様のことが起こります。
もっとも、本書のサンプルで DirectX9 固有の機能（主にプログラマブル・シェーダー）は使用していないので、DirectX8 世代のカードまでは快適に動作するはずです。

●モニター
16 ビットカラー（65,536 色）で 800 × 600 ピクセルを表示できるもの。

●コンパイラー
次のうち何れか 1 つが必要です。

VisualStudio .NET2003（C++ 言語を含む）
VisualStudio .NET2002（C++ 言語を含む）

VisualStudio 6.0（C++ 言語を含む）
VisualC++ .NET2003
VisualC++ .NET2002
VisualC++ 6.0
全てメーカーはマイクロソフトです。

● DirectX SDK 9.0C（April2005 バージョン）
これは添付ディスクに入っています。

コンパイラーの設定

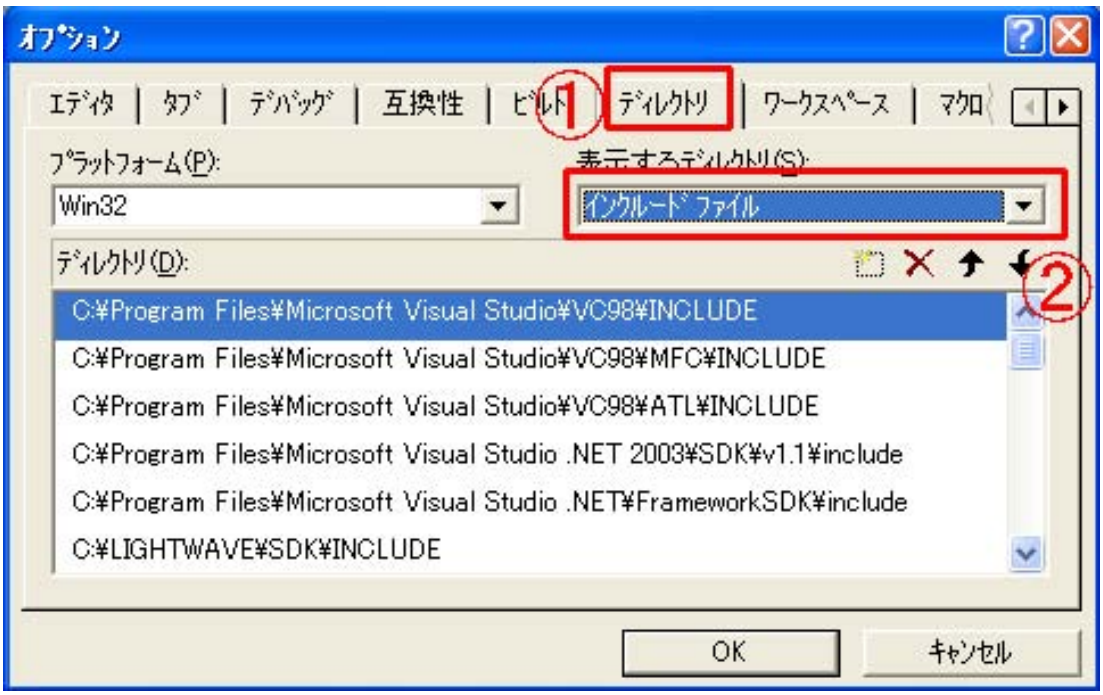
SDK とは Software Development Kit の略であり開発に必要なソフトウェアの一連のセットという意味です。DirectXSDK には DirectX 実行時に必要である DirectX ランタイムとプログラミング時に必要なヘッダーファイル、ライブラリファイル、サンプルプログラム、ヘルプファイルが含まれています。DirectX ゲームをプログラミングする際にはこの SDK が必要です。SDK は添付ディスクに最新バージョンのものがありますので、それをパソコンにインストールします。そして、インストールするだけではまだ準備ができていません。SDK が機能するようにコンパイラーの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）をコンパイラーに登録する必要があります。
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:\Program Files\Microsoft DirectX 9.0 SDK (April 2005) となっているはずです。そしてヘッダーファイルは C:\Program Files\Microsoft DirectX 9.0 SDK (April 2005)\Include、ライブラリファイルは C:\Program Files\Microsoft DirectX 9.0 SDK (April 2005)\Lib\x86 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:\Program Files\Microsoft DirectX 9.0 SDK (April 2005) の部分が変更したパスになります。例えば、D ドライブに SDK9 など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:\SDK9（ルートパス）D:\SDK9\Include（ヘッダーファイルのパス）D:\SDK9\Lib\x86 (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。

【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。

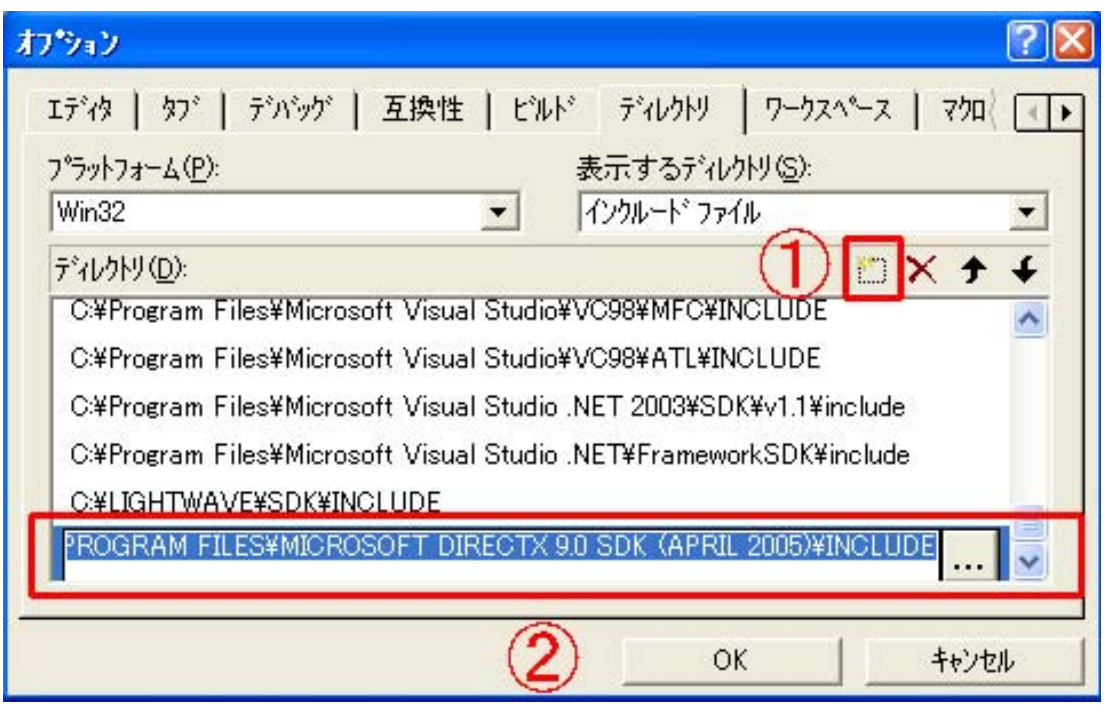
〈ヘッダーファイルパス（インクルードファイルパス）の登録〉

図 1 － 1



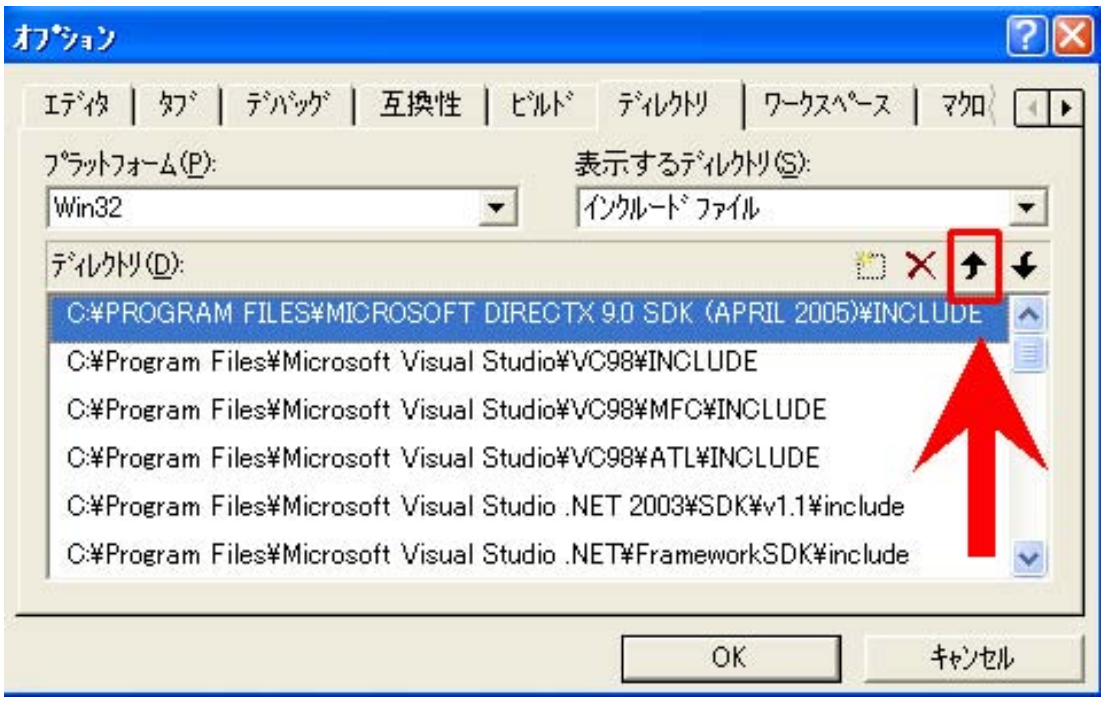
「表示するディレクトリ（S)」コンボボックス内を「インクルードファイル」にします。

図 1-2



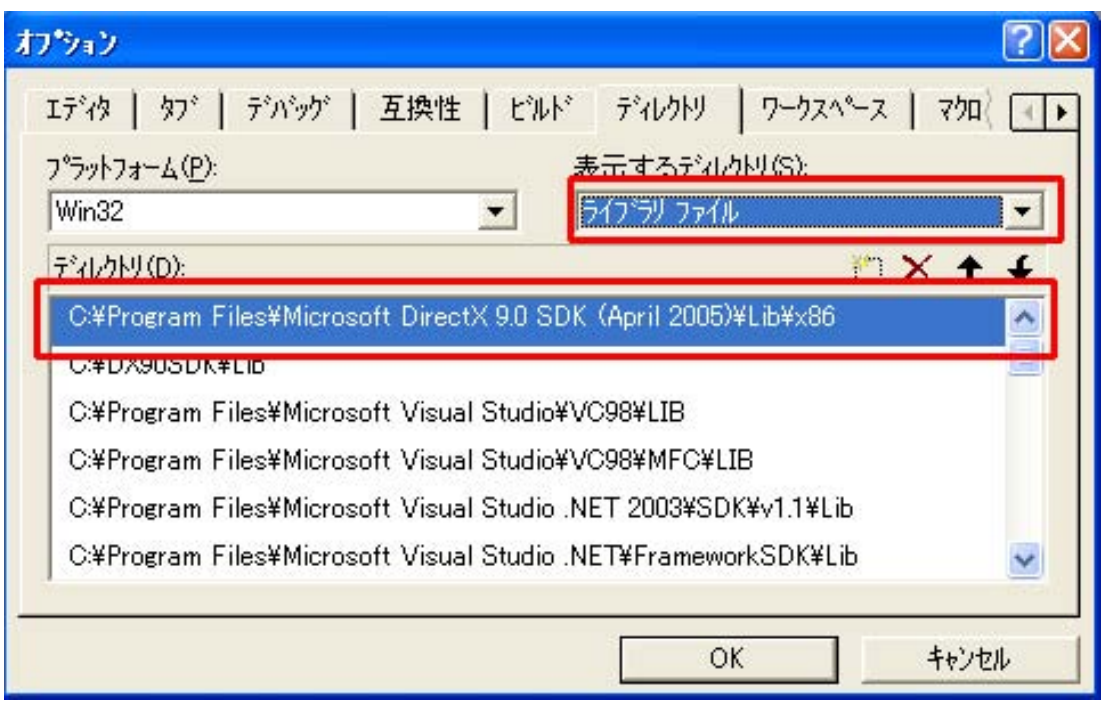
ヘッダーファイルのパスを図のように April2005 のインクルードパスにします。

図 1-3



入力したパスを一番上に移動します。

〈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。
図 1-4



インクルードパスの手順と同様に、ライブラリパスを図のようにします。
これで 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

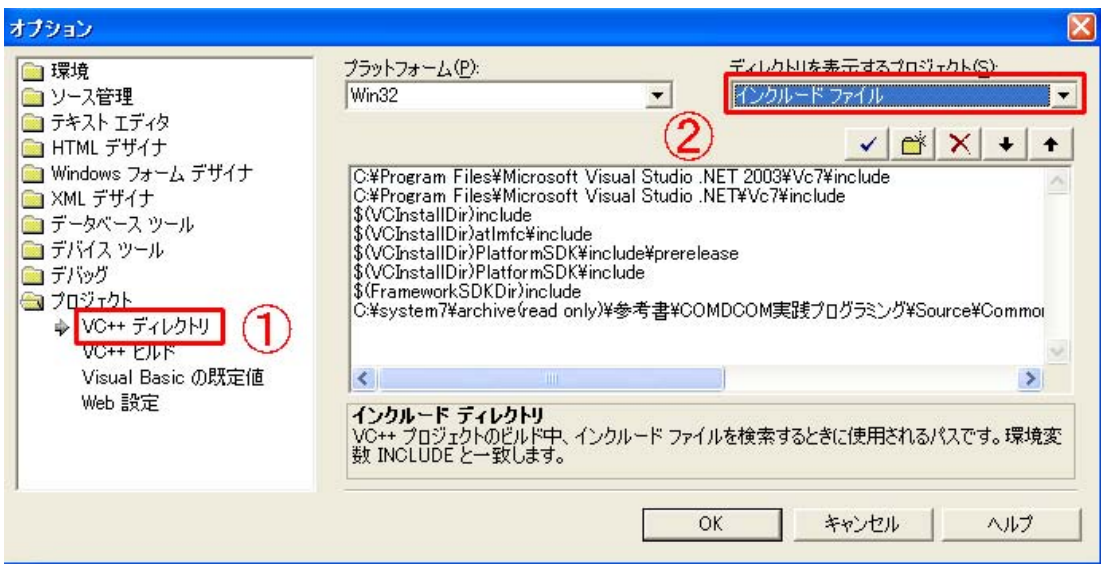
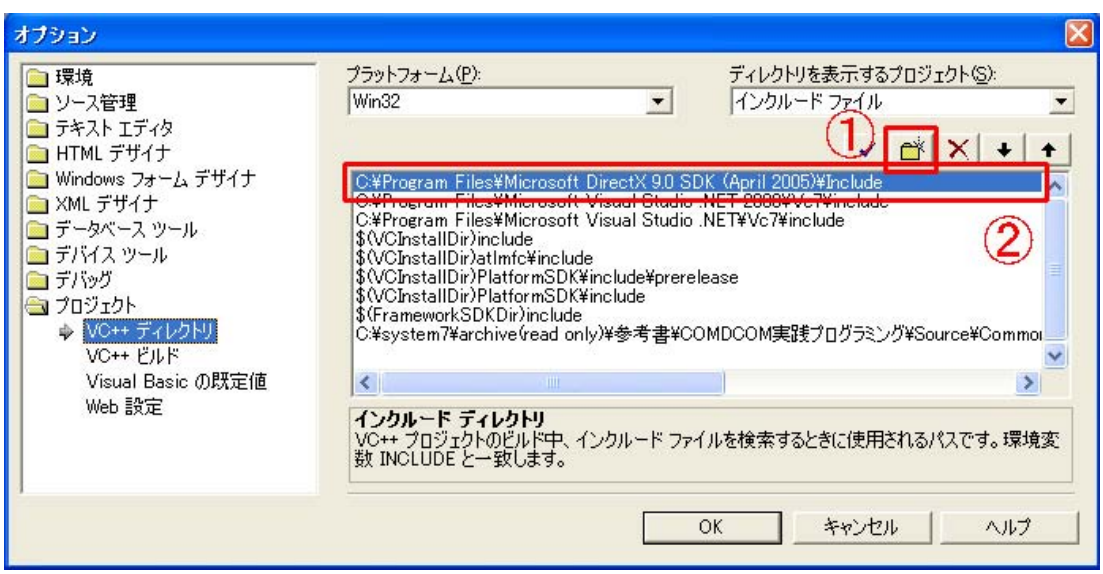


図 1-6



右上の「新しい行」ボタンを押すと、新しいパスを入力するエディットボックスが表示されるので、そこを図のように April2005 のインクルードファイルパスにします。

〈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。

図 1-7

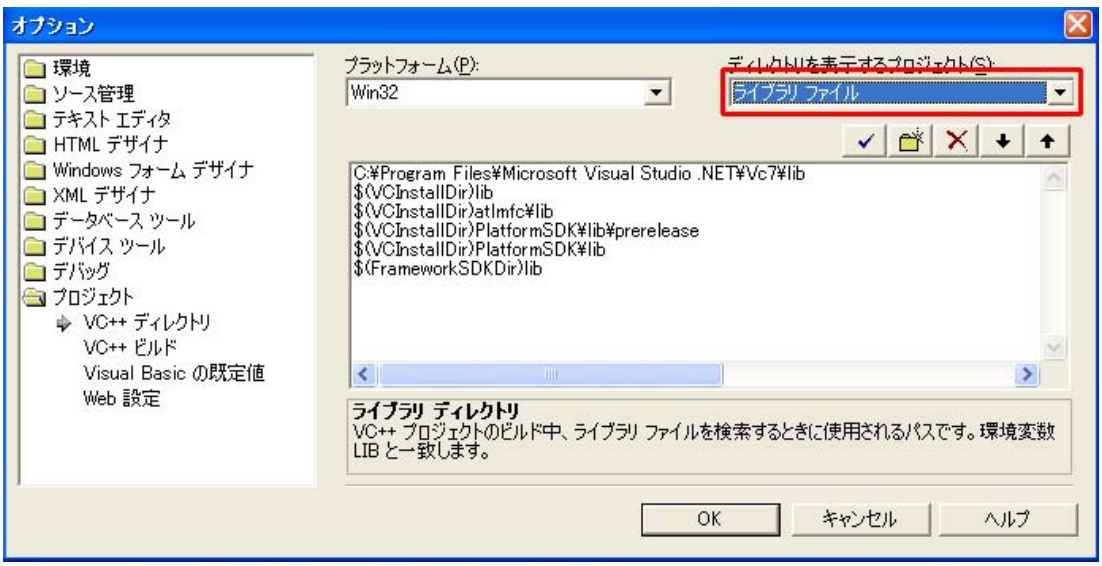
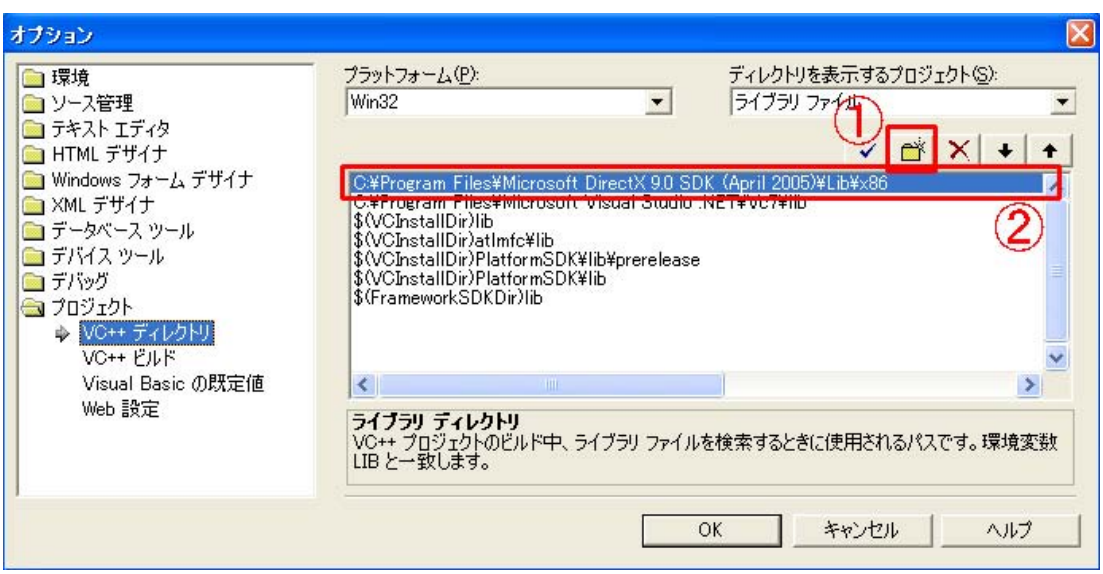


図 1 － 8



ヘッダーファイルパスと同様の手順で図 1-8 のようにします。
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。

図 2-1　スプライト（弁当の絵）を単にレンダリングするだけのプログラム

ゲームプログラミング初心者が「さあ、ゲームを作るぞ！」と思ったとき、最初に直面するのは画面表示の方法ではないでしょうか？　3D 全盛になって久しい昨今、画面への “ 表示 ” は “ レンダリング ” と呼んだほうが相応しいと思います。以降、本書では “ 画面表示 ” はレンダリングと呼びます。
画面に何らかのジオメトリをレンダリングしなくては何も始まりません。ゲームにおいて、レンダリングは最も必要なことです。ジオメトリとは頂点、線、平面（ポリゴン）という 3D 空間上の幾何学形状です。
「そんなジオメトリなんっていう小難しそうなものじゃなくていいから、とにかく “ 絵 ” を、もしくはもっと原始的に “ ピクセル 1 点 ” でもいいから表示させたいんだよ」と思う人もいるかもしれません。しかし、DirectXGraphic における画面表示は “ 事実上 ” すべて 3D 空間で行われます。ピクセルとか絵という一見簡単そうな 2D イメージの表示ですら 3D 空間で考えますので、2D にこだわるとむしろ難しくなります。
キャラクターを表す画面上の 2D イメージは、ゲームの世界では昔から “ スプライト “ と呼ばれます。Direct3D にもスプライトを表示する関数がありますが、もちろん、本来 2D であるスプライトも Direct3D では 3D ジオメトリです。ただ、Direct3D でのスプライトはあたかも 2D イメージのように振舞いますので、最初に勉強するには良い概念だと思います。ここでは、スプライトをレンダリングする方法を学びます。
Direct3D におけるスプライトは、具体的には 4 頂点から成る 4 角形ジオメトリにテクスチャ（bmp や png などの画像）を貼り付けたものの総称です。スプライトの実体は 3D ジオメトリですが、ペラペラの平面なので、使う分には、“2D 絵 ” として考えても差し支えありません。このように、実体は 3D でも、2D 的に考えることができるような擬似 2D ジオメトリ、および、それらの擬似 2D で作る擬似 2D シーンを “3D 空間内での論理 2D” と言います。
まずは、難しい話は抜きにして。画面に絵をレンダリングしてみましょう。

サンプルプログラム

プロジェクト名「ch02 とにかく絵をレンダリングする」
VisualC++6.0（以降、VC6 と表記）、VisualC++.NET2002（以降、VC2002 と表記）、VisualC++.NET2003（以降、VC2003 と表記）の 3 つのバージョンのコンパイラー用それぞれにプロジェクトを作成していますので、自分のコンパイラーに合ったフォルダのプロジェクトを開いてください。
このサンプルは本書で最も単純なもので、1 枚のスプライトをレンダリングするだけです。起動から終了まで、ず〜っと同じスプライトを同じ位置にレンダリングし続けるだけのものです。スプライトの絵は、なぜか「弁当」です（笑） 深い意味はありませんので気にしないでください。弁当の絵は 3DCG ソフトでレンダリングしたものなので、絵そのものに奥行きがあり、一見すると 3D ジオメトリに見え、2D スプライトのサンプルには相応しくないかもしれませんが、ペラペラの 2D 絵です。
最初から最後まで同じ状態なので、使用方法の説明もありません。“ 起動するだけ ” です。ESC キーで終了します。

コード解説

ここは初心者編であるということ、及び、本書の最初のサンプルであるので、細かく解説していきます。

《 プリプロセッサディレクティブ 》

```
// 必要なヘッダーファイルのインクルード
#include <windows.h>
#include <d3dx9.h>
```

コメントの通り、必要なヘッダーファイルをインクルードしています。
windows.h はウィドウプログラムではれば必須のものです。
同じように d3dx9.h は Direct3D プログラムでは必須のものです。

```
// 必要なライブラリファイルのロード
#pragma comment(lib,"d3d9.lib")
#pragma comment(lib,"d3dx9.lib")
```

これもコメントの通りで、ライブラリファイルをロードするようコンパイラーに指示しています。“ ロード “ と言ってもコンパイル時に行われるもので、プログラムの実行時にロードされるわけではありません。
また、これらのライブラリファイルは Direct3D 本体ではなく単なるインポートライブラリです。Direct3D 本体は d3d9.dll や d3d9x.dll などの DLL として存在します。これは Direct3D に限らず DirectX 全体に言えます。(DirectXAudio などライブラリファイルが不要のものもありますが)。この辺のことについては、22 章で解説しています。
まぁ、何れにしてもビルドに必要なものであるということは言えます。

《 定義とグローバル宣言 》

```
// シンボル定義及びマクロ
#define WINDOW_WIDTH 800
#define WINDOW_HEIGHT 600
```

ウィンドウサイズは 800 × 600 としました。

#define SPRITE_WIDTH 200
#define SPRITE_HEIGHT 246
スプライトのサイズを記号定数として定義しています。この値から明らかなように、スプライトにする画像は 200 × 246 のビットマップということです。

#define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
Direct3D のインスタンスをメモリから開放するためのマクロを定義しています。このようなマクロは本書全体を通して使用しています。

// グローバルなインスタンスを宣言
LPDIRECT3D9 pD3d;
Direct3D オブジェクトインターフェイスのポインターです。インターフェイスはオブジェクトへのポインターなので、言い換えると、Direct3D オブジェクトのポインターのポインターとなります。
…と厳密に表現すると混乱する人のほうが多いと思いますので、今後は単に Direc3D オブジェクトと表現します。他のインターフェイスポインターについても同様に OO オブジェクトと表現します。オブジェクトとインターフェイスの詳細についても 22 章で解説していますが、今は深く考えないで、Direct3D プログラムでは、必ず Direct3D オブジェクトが必要なのだと考えてください。

LPDIRECT3DDEVICE9 pDevice;
Direct3D デバイスインターフェイスのポインターを宣言しています。Direct3D プログラムでは、Direct3D オブジェクトと、この Direct3D デバイスオブジェクトは必ず必要です。

LPDIRECT3DTEXTURE9 pTexture;
テクスチャオブジェクトです。

LPD3DXSPRITE pSprite;
スプライトオブジェクトです。

// 関数プロトタイプの宣言
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
HRESULT InitD3d(HWND);
VOID DrawSprite();
VOID FreeDx();
関数はアプリケーションエントリー関数（WinMain 関数）を含めると 5 つあることになりますが、ウィンドウ関連のものは、解説から除きます。
InitD3d 関数は、Direct3D の初期化をする関数です。DrawSprite 関数は、スプライトをレンダリングする関数、Free 関数は、プログラム中で生成した Direct3D の各インスタンスをメモリから開放する関数で、一番最後に 1 回だけ実行します。
24 行〜 71 行までは、ウィンドウの作成とメッセージプロシージャーなので、解説は割愛します。もし読者が “ ウィンドウプログラミング ” においても初心者であるならば、当面はこのようにするものだとして考えてもいいと思います、支障はさほどありません。実際、本書のサンプルにおいて、この部分は全て同じです。

《　InitD3d 関数　》
Direct3D 関係のオブジェクトを作成し初期化する関数です。初期化関数なので、最初に 1 度だけ実行されます。

if(NULL == (pD3d = Direct3DCreate9(D3D_SDK_VERSION)))
Direct3D を使用するためには、なによりも先にこの Direct3D オブジェクトを作成しなくてはなりません。
Direct3DCreate9 関数は関数です。なんか辺な言い回しですが、これは案外本質を突いています。他のオブジェクトは、全て、自分以外の何らかのオブジェクトのメソッドにより作成されます。（例えば下の 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ます。）ところが、Direct3D オブジェクトだけは、他のオブジェクトから作成されるものではありません。Direct3D オブジェクトは全てのオブジェクトの親であり創造主です。したがって、任意のオブジェクトの作成元のオブジェクトを逆に辿ると必ず Direct3D オブジェクトにたどり着きます。例えるなら、人間は必ず親がいて、親にはさらに親が存在するわけですが、先祖をどんどんさかのぼっていくと “ アダムとイブ ” にたどり着きます。アダムとイブは、他の人間とは素性が全く違います。彼ら、言いかえれば “ 最初の親 “ は、何らかの外部のもので創造しなくてはなりません。Direct3D オブジェクトはアダムとイブです。アダムとイブは神が、Direct3D オブジェクトは Direct3DCreate9 関数が作成しなくてはなりません。一旦アダムとイブが創造されれば、それ以降の人間は全て彼らの子供という形で生まれます。同様に、一旦 Direct3D オブジェクトが生成されれば、その他のオブジェクトは外部関数の助けがなくともオブジェクト間で生成できます。ちなみに、このように生成を手助けする外部関数は文字通り ” ヘルパー関数 “ と呼ばれます。ヘルパー関数は、最初の親用に 1 つあれば十分なのはお分かりでしょう。実際、Direct3D の ” 関数 “ は、この Direct3DCreate9 関数ただひとつしかありません。

113 行〜 135 行
Direct3D オブジェクトを作成したら、次は Direct3D デバイスオブジェクトの作成です。
Direct3D デバイスオブジェクトとは平たく言えば、ディスプレイ、モニター及びビデオカードのオブジェクトです。Direct3D デバイスオブジェクトも Direct3D オブジェクトと同様に必須のオブジェクトです。モニターとビデオカードを使用しない Direct3D プログラムは有り得ないのですから。
デバイスオブジェクトは、IDirect3D9::CreateDevice メソッドにより作成します。IDirect3D9 は Direct3D オブジェクトであり、CreateDevice はそのメソッドです。このように、Direct3D オブジェクト以外は、なんらかの親オブジェクト内のメソッドにより作成されるわけです。
ビデオカードは、各 PC で様々な性能のものが考えれれるので、デバイスオブジェクトの作成時には、どうしてもこのように if 文が多くなってしまいます。if 文が 4 重にもネストしているのは、様々なビデオ環境に対応できるコードを目指した結果ですが、実はこれでも足りないくらいです。ただ、これでもかなりの環境をカバーしているので、現実的には十分であると思います。
CreateDevice の第 2、第 4 引数を変えたものが 4 個あります。それぞれの意味は次のとおりです。
一番外側の if 文
D3DDEVTYPE_HAL
D3DCREATE_HARDWARE_VERTEXPROCESSING
HAL モードで、かつ、頂点の処理もハードウェアで行う能力があるビデオカードのデバイスを作成できるか？　HAL とはハードウェア側のサポートを前提とした高速なモードです。
もし出来なければ、次のモードで作成を試みる。

2 番目の if 文

D3DDEVTYPE_HAL
D3DCREATE_SOFTWARE_VERTEXPROCESSING
HAL モード、頂点の処理をソフトウェアで実行するビデオカードのデバイスを作成できるか？
これも出来なければ、さらに内側の if 文に移る。

3 番目の if 文
この if 文まで来たということは、ビデオカードが Direct3D をハードウェアで処理できないということです。HAL ハードウェア処理できないビデオカードのために Direct3D では REF“ リファレンス・ラスタライザ ” という機能があります。ただし、かなりパフォーマンスは落ちます。
D3DDEVTYPE_REF
D3DCREATE_HARDWARE_VERTEXPROCESSING
REF モードではあるものの、頂点処理はハードウェアで行えるビデオカードか？
そうでなければ、最後の if 文に移る。

4 番目の if 文
D3DDEVTYPE_REF
D3DCREATE_SOFTWARE_VERTEXPROCESSING
REF モード、かつ、頂点処理をソフトウェア的に行うしかないビデオカードでのデバイスオブジェクトを作成します。これでもダメなら、本サンプルでは “ デバイス作成失敗 ” として、アプリケーションを終了します。（この if 文が失敗することは、まずないとは思います）

//「テクスチャオブジェクト」の作成
if(FAILED(D3DXCreateTextureFromFileEx(pDevice,"Sprite.bmp",SPRITE_WIDTH,SPRITE_HEIGHT,0,0,D3DFMT_UNKNOWN,D3DPOOL_DEFAULT,D3DX_FILTER_NONE,D3DX_DEFAULT,0xff000000,NULL,NULL,&pTexture)))
スプライトを作成する前に、テクスチャを作成しておく必要があります。
テクスチャとは､なんらかの画像ファイル（bmp png dds 等）のインスタンスです。テクスチャ＝ “ 絵 “ と考えても外れてはいません。
スプライトは、このテクスチャを張ること（テクスチャ・マッピングと言います）により、スプライトとして完成します。
引数は、このようにするものだと考えてください。（このセクションは “ 初心者の簡単運用編 ” です）第一引数に画像ファイルのパスを指定する程度に考えましょう。
なお、FAILED() は括弧内の処理が成功したか失敗したかのマクロです。失敗した場合に FAILED() は真になります。この逆で成功した場合に真となる SUCCDEEDED() マクロもあり、２つとも本書全体で使用しています。

//「スプライトオブジェクト」の作成
if(FAILED(D3DXCreateSprite(pDevice,&pSprite)))
スプライトのオブジェクトを作成しています。引数はデバイスオブジェクトと、空（から）のオブジェクトポインターです。メソッドが成功すると空のポインターには生成されたスプライトオブジェクトを指すことになります（生成されたスプライトオブジェクトのアドレスが入ります）。

《　DrawSprite 関数　》
スプライトをレンダリングする関数です。プログラムの起動中は常に繰り返し実行されます。

pDevice->Clear(0, NULL, D3DCLEAR_TARGET,D3DCOLOR_XRGB(30,30,30), 1.0f, 0);
画面に何かをレンダリングする前には、通常、画面をクリアします。更（さら）の状態にしてから、レンダリングしていくわけです。
クリアとは、厳密に言うと単色で画面を塗りつぶすことです。ここでは、やや灰色っぽい黒で塗りつぶしています。
ゲームは多くの場合、1 秒間に 60 回の画面更新をしますが、その 1 回を 1 フレームと言います。1 フレームの最初には通常クリアを実行します。クリアしないと以前のフレームでのレンダリングが残像にように残ってしまいます。それはそれで面白いかもしれませんが、レンダリングしたピクセルが消されないので画面上にどんどん堆積していき分けが分からなくなります。

if(SUCCEEDED(pDevice->BeginScene()))
クリアはしなければならないものですが、クリアするかしないかは開発者次第であって、しなくともエラーは出ません。それに対し、この BeginScene メソッドは、それをしなければレンダリングが出来ません。レンダリングメソッドは全てエラーを返します。
あらゆるレンダリング処理の前には、必ず BeginScene メソッドをコールします。

RECT rect={0,0,SPRITE_WIDTH,SPRITE_HEIGHT};
RECT は矩形を表す構造体で、windef.h ヘッダーファイル内で次のように定義されています。
```
typedef struct tagRECT
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```
成分から明らかなように、スプライト用に用意した画像（Sprite.bmp）のサイズで初期化しています。

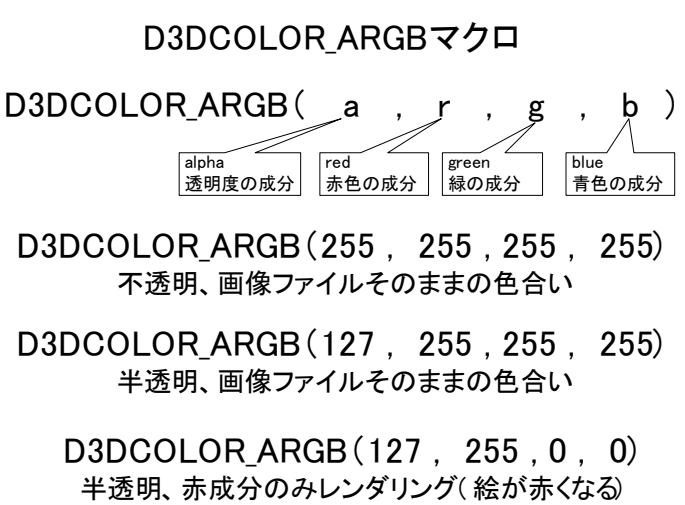
D3DXVECTOR3 vec3Center(0,0,0);
D3DVECTOR3 は 3 次元ベクトルを表す構造体です。D3DVECTOR3 は d3d9types.h ヘッダーファイルで次のように定義されています。
```
typedef struct _D3DVECTOR {
    float x;
    float y;
    float z;
} D3DVECTOR;
```
D3DXVECTOR3 は D3DVECTOR3（X が付いていない）を継承した派生型です。D3DVECTOR3（X が付いていない）は通常の構造体なので、A+B などと書くことはできません。しかし、D3DXVECTOR3 のほうは構造体であるにも関わらず加減算や乗算を直感的に行えるようにオペレーター定義されているので便利です。
```
typedef struct D3DXVECTOR3 : public D3DVECTOR
{
    ( 長いので省略 )
}
```
3 次元ベクトルとして vec3Center を宣言しています。vec3Center はスプライトの中心座標として機

能します。中心座標は絶対的なスクリーン座標ではなく、そのスプライト内でのローカル座標でありウィンドウ上では相対座標です。
中心を (0,0,0) とするとスプライトの左上の点を意味します。今回は、(0,0,0) としていますが、通常は (スプライトの横幅半分 , スプライトの縦幅半分 ,0) とします。こうしないと、スプライトを回転させる際、回転軸がズレているように回るからです。

D3DXVECTOR3 vec3Position(210,110,0);
これも 3 次元ベクトル vec3Position を宣言しています。vec3Position はスプライトの位置（スクリーン座標）として使用します。ウィンドウ上では絶対座標です。この座標がスプライトの左上に対応し、(0,0) とすればウィンドウに左上からレンダリングされますし、(400,300) とすれば（ウィンドウが 800 × 600 の場合)、ウィンドウのちょうど真ん中からレンダリングされます。

pSprite->Begin(D3DXSPRITE_ALPHABLEND);
スプライトをレンダリングする時は、必ず最初に、ID3DXSprite::Begin メソッドを実行します。これを実行しないとスプライトのレンダリングは失敗します。
ここでの引数の意味は、アルファブレンド（透明処理）が効くようにしているということです。

pSprite->Draw(pTexture,&rect,&vec3Center,&vec3Position,0xffffffff);
先に作成しておいたテクスチャを第 1 引数に渡すことにより、スプライトにテクスチャ（絵）が貼られます。第 2 引数は、スプライトのサイズを保持した矩形です。第 3 引数は中心座標、第 4 引数は位置の絶対座標です。第 5 引数は、スプライトの透明度と色合いです。
第 5 引数について、説明します。
第 5 引数の型は D3DCOLOR 型です。D3DCOLOR 型といっても、なんのことはない符号無し 32 ビット整数、つまり DWORD のことです。ここでは、そのまま 32 ビット値を入れるのではなく、D3DCOLOR_ARGB() というマクロを使用しています。このマクロは、4 つの成分から 32 ビット符号無し整数を生成してくれます。マクロをい使用した理由は､透明度と色合いが分かり易いように､また､変更しやすいようにするためです。
4 つの値の意味は次のとおりです。
図 2-2



今は全て 255（0xFF）ですが、値を変更すると、それに対応した色合いになります。

たとえば、透明度を半分の 127 にして実行してみてください。絵が透けます。
また、赤成分と緑成分を 0 にして実行してみてください。不気味な青い弁当が出来上がります。

pSprite->End();
スプライトの全てのレンダリングの最後には必ず、ID3DXSprite::End メソッドを実行します。レンダリングメソッドの成否には関わりませんが、これを実行しないとレンダリングが画面に反映されません。

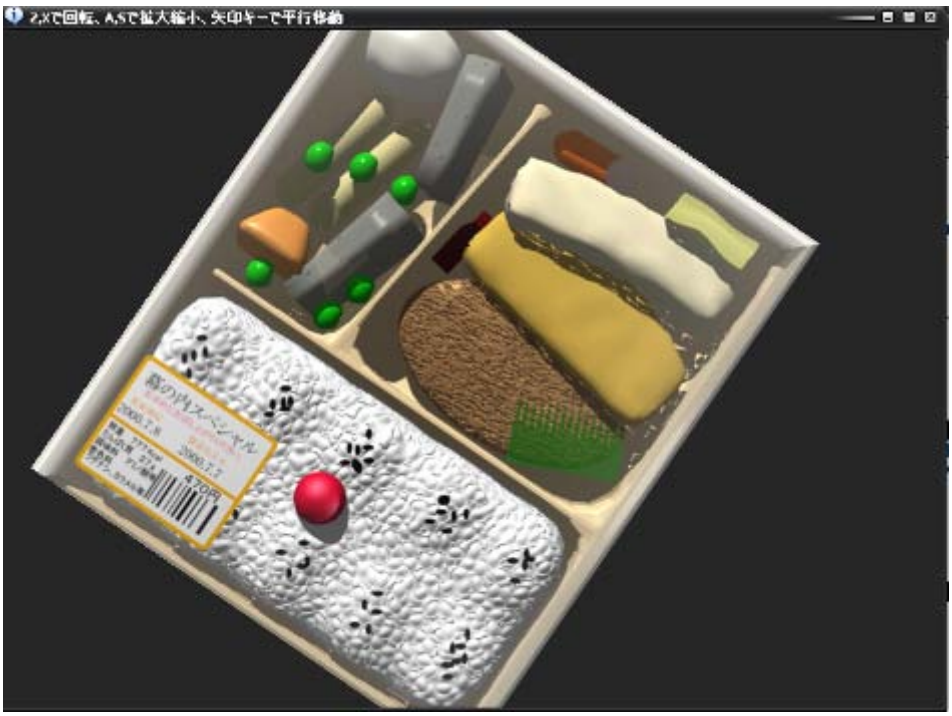
pDevice->EndScene();
これは、スプライトを含む全てのレンダリング処理の後に必ず実行します。

pDevice->Present(NULL, NULL, NULL, NULL);
IDirect3DDevice9::: Present は、画面を更新する関数と覚えましょう。これも、実行しないと画面更新されないためレンダリング結果が見えません。

《　Free 関数　》
プログラム内で作成したオブジェクトをメモリから開放する関数です。プログラムの終了時に 1 度だけ実行されます。
開放は生成した順番と逆に行います。

3章　絵を操作する

図 3- 1　移動、回転、拡大縮小をキーボードで操作



ここでは、レンダリングされているスプライトを操作する方法を学びます。操作とは、ユーザーのなんらかの入力により、移動、回転、拡大縮小（スケーリング）を行うこという意味です。本サンプルではキーボードの押下情報を入力として使用します。

サンプルプログラム

プロジェクトフォルダ名「ch03 絵を操作する」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。
使用方法
平行移動：矢印キー
回転：Z キーと X キー
スケーリング：A キーと S キー
ESC：終了

コード解説

操作のコーディング（ソースコードの作成）の本質は “ 入力情報を取得する部分 ” と “ 操作をスプライトに反映させる部分 ” なので、その 2 つを分けて解説します。

《　入力情報を取得する部分　》
ユーザーのキーボード押下は、DirectInput を使用せず、Win32 イベントメッセージを拾うことにより検知しています。ですので、ウィンドウのメッセージプロシージャー内での処理です。
switch(iMsg)
イベントメッセージをスイッチして…

```
case WM_KEYDOWN:
switch((CHAR)wParam)
```
キー押下のイベントメッセージであれば、さらに wParam が表すキーの種類でスイッチします。
```
case VK_LEFT:
        fPosX-=5;
break;
case VK_RIGHT:
        fPosX+=5;
break;
case VK_UP:
        fPosY-=5;
break;
case VK_DOWN:
        fPosY+=5;
break;
```
この 4 つの case ブロックは、スプライトの位置座標を増減しています。上からそれぞれ、スプライト座標の、X 成分の減少（左に移動）、X 成分の増加（右に移動）、Y 成分の減少（上に移動）、Y 成分の増加（下に移動）、を行っています。

```
case 'Z':
        fAngle-=0.1;
break;
case 'X':
        fAngle+=0.1;
break;
```
この 2 つの case ブロックは、スプライトの回転を行っています。fAngle は回転角を保持するグローバル変数です。Z キーのブロックは、回転角を 0.1 ラジアンだけ減少（反時計回り）、X キーのブロックは 0.1 ラジアンだけ増加（時計周り）させています。
ラジアンとは弧度法による角度の単位です。詳しくは 8 章で解説しています。

```
case 'A':
        vec3Scale*=0.95;
break;
case 'S':
        vec3Scale*=1.05;
break;
```
拡大・縮小を行っています。A キーブロックでは縮小を、S キーブロックでは拡大を行っています。
スケーリングは、x,y,z の成分ごとに指定できますが、ここでは 3 つの成分を均等に増減させています。
均等に増減させた場合は、均等にスケーリングされます。
たとえば、X と Y 成分を異なる比率で増減させるとスプライトが縦横に伸びることになります。
3 つの成分が必要なので、ここでのように 3 次元ベクトルを係数にすると便利です。

《　操作をスプライトに反映させる部分　》

操作をスプライトに反映させるとは、新たな姿勢や縮尺でスプライトをレンダリングするということです。前章でのレンダリングは、毎フレーム常に同じ姿勢、縮尺で単純にレンダリングしていました。プログラムの起動から終了まで常に同じ状態で絵をレンダリングしていたわけです。
レンダリングしている部分は前章と同じ DrawSprite 関数で処理しています。
レンダリングそのものに関しては前章で解説しているので、操作をどのようにレンダリングに反映させているかという部分だけ解説します。

D3DXVECTOR3 vec3Center(SPRITE_WIDTH/2,SPRITE_HEIGHT/2,0);
中心をスプライトの左上隅ではなく、まさしく絵の中心にします。スプライトの横幅の半分を X 成分、縦幅の半分を Y 成分とすることにより中心が求まります。この中心により回転をかけた場合に直感的にまわすことができます。なお、論理 2D ですから、Z 成分は通常は固定値、ゼロなどとします。

D3DXVECTOR3 vec3Position(fPosX,fPosY,0);
スプライトの位置を固定値ではなく、動的に変化する “ 変数 ” で初期化します。

D3DXMATRIX matWorld,matScale,matRotation,matTranslation;
最終的なワールド行列、スケーリング行列、回転行列、移動行列を用意します。

D3DXMatrixIdentity(&matWorld);
“ 念のため ” ワールド行列を単位行列にしておきます。本サンプルでは、この下の処理により必ずワールド行列に適正な値が入るので不要ではありますが、良い慣習としては最初に単位行列にします。単位行列とは、数字の 1 のように、何も効果を及ぼさない行列です。もしこれをしない場合、不定の値が入るのでおかしな結果になる場合があります。

D3DXMatrixScaling(&matScale,vec3Scale.x,vec3Scale.y,vec3Scale.z);
スケーリング行列を、動的に変換する比率で算出します。

D3DXMatrixTranslation(&matTranslation,fPosX,fPosY,0);
平行移動行列を作成します。

D3DXMatrixRotationZ(&matRotation,fAngle);
Z 軸周りの回転行列を作成します。
論理 2D の場合は、回転は全て Z 軸周りです。モニターの正面から奥に向かう軸上（Z 軸）で回るからです。

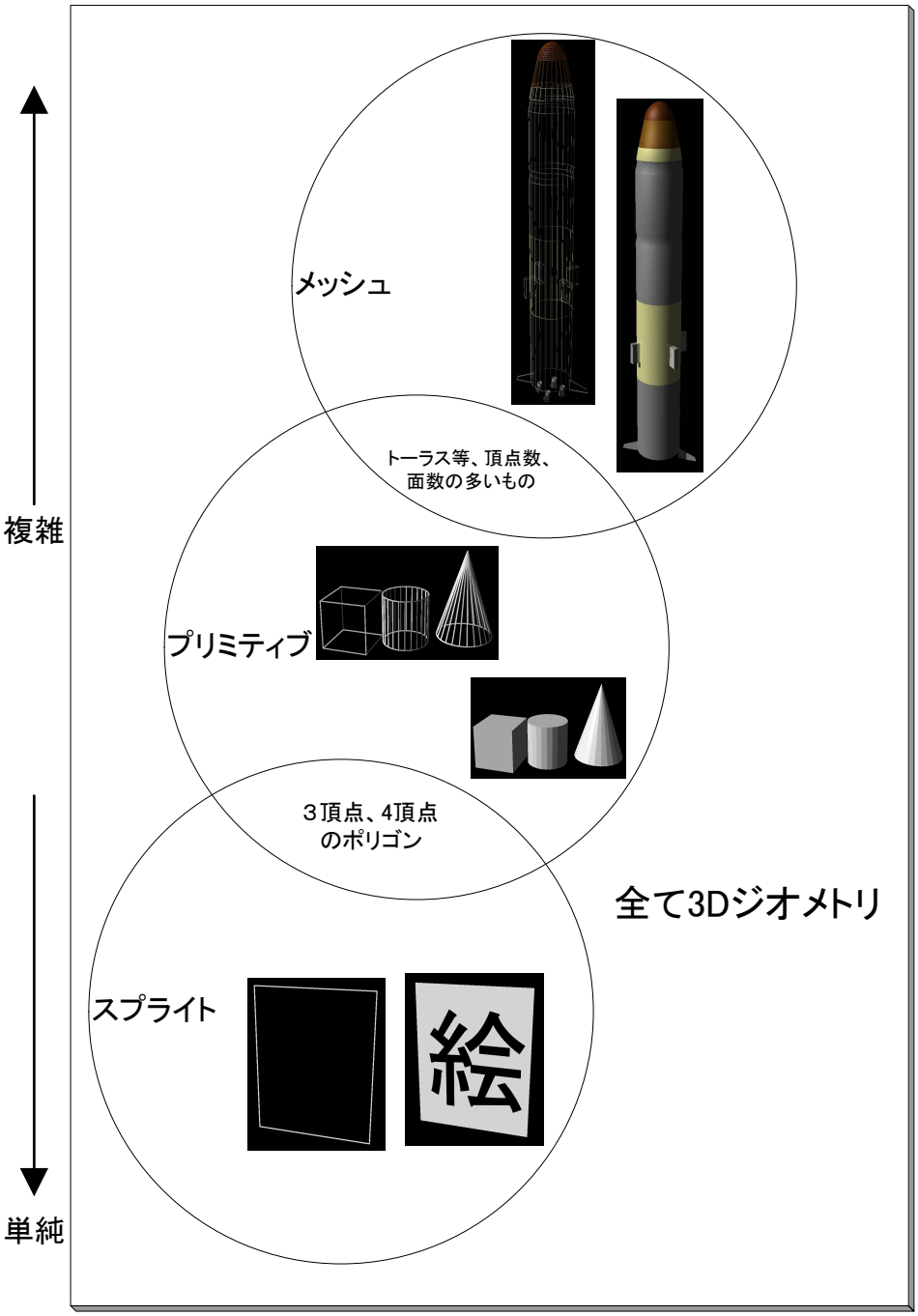
matWorld=matScale*matRotation*matTranslation;
スケーリング、回転、平行移動の順番で行列を合成し、ワールド行列にその結果を代入します。

pSprite->SetTransform(&matWorld);
スプライトのレンダリングの流れ行程に、ワールド行列を渡します。これ以降にレンダリングされるあらゆるスプライトは、新たなワールド行列が渡されるまで、同じ姿勢・スケーリングで変換されます。

図 4-1　完全 3D ジオメトリ、“ ビタミン剤ボックス ” ・メッシュの単純レンダリング

スプライトも、その振る舞いは 2D であるものの実態は 3D ジオメトリです。しかし、ペラペラなので、やはり 3DCG という印象はないでしょう。
ここでは（そして今後も）、スプライトではなく、見た目にも完全な 3D ジオメトリを扱います。
3D ジオメトリは、原理的には頂点と面（ポリゴン）の集合です。面（ポリゴン）には、色や光沢を設定し、そしてさらにリアル性を高めるために、テクスチャを貼ることも多くあります。点と面が非常に少ないジオメトリは “ プリミティブ ” と呼び、頂点や面が多く比較的複雑なものは “ メッシュ “ と呼ばれます。比較的と言ったように、プリミティブとメッシュの境界線は曖昧ですが、一般的にプリミティブとは 3 角形か 4 角形のポリゴン、及び単純な立方体、球、円錐、円柱までででしょう。トーラス（ドーナツ形状）までに至ると頂点とポリゴンが多く必要となるので、プリミティブと呼ぶかは微妙なところです。
何れにしても Direct3D はその名が示すとおり “3D” なのですから、レンダリングするもの全ては 3D 空間内の頂点および面です。スプライトさえも 4 頂点のポリゴンですから例外ではありません。
図にすると次のとおりです。

図 4-2　全て 3D ジオメトリ



全ては程度の差こそあれ 3D ジオメトリです。文章で表現すると、
複雑なものがメッシュ、
比較的単純なものがプリミティブ、
プリミティブの中でも、単一ポリゴンで、絵を貼り付けることが前提であるものがスプライト。
ということになります。
スプライトとプリミティブは、単純なためプログラム的に実行時に動的に作成できます。それに対し、メッシュは複雑なため、普通はプログラムにより動的に作成するよりも X ファイルという別ファイルからデータを読み込み作成することが殆どです。本サンプルでもメッシュは X ファイルから読み込みます。ただし、X ファイル = メッシュではないので、概念は正確に捉えましょう。X ファイルはメッ

シュを作成する手段の一つです。ただ、非常に便利なため事実上メッシュの作成はXファイルを使用します。
では、Xファイルに落とす（変換する）前のメッシュはどうやって作成するのでしょう？
それには3DCGソフトを使用します。筆者はLightwave3Dというソフトを使用しています。その他にもMAYA、3DStudioMax、SoftImage（3D及びXSI）、Cinema4Dなどがあります。ただし、難点としては全て高価です（20万円台から高いもので100万円近くするものもあります）
そして、これらのソフトはメッシュの作成のためだけにおいて、特に初心者には、幾分オーバースペックです。3DCGソフトには、これら以外にも六角大王やメタセコイアといったフリーウェア及びかなり安価なシェアウェアもあります。初心者が単にメッシュを作成するだけの目的であればそれらでも十分かもしれません。ただやはり機能は限定されます。また、同じことをするのに手間がかかります。フリーのソフトは、長い目で見ると効率の面から、特にプロにとってはかえって高くつきます。筆者は、フリーソフトを一切使いません。これは偏見とかポリシーとかの類ではなく、効率対費用を考えた結果であり、逆に極めて費用上の理由からです。もっとも、大昔なら1人でゲーム1本作成するのは当たり前でしたが、現在の日本では数えるくらいしかいません。製品レベルであれば、誰か別のグラフィッカーが作成してくれるので、高価なCGソフトを購入する必要はすくなくとも仕事上はあまりないでしょう。日本にも、トータルなクリエイターがもっと増えることを願っています。

サンプルプログラム

プロジェクトフォルダ名「ch04 メッシュのレンダリング」
VC6、VC2002、VC2003、3つのフォルダそれぞれにあります。

使用方法
起動して眺めるだけです。ESCキーで終了します。

コード解説

スプライトのサンプルと異なるのは、InitD3d関数内にXファイル読み込みルーチンを追加したことと、Render関数です。

InitD3d関数　147行目以降

D3DXLoadMeshFromX関数により、メッシュを読み込みます。この関数をコールすれば大半の処理はしてくれますが、全ての処理を行ってくれるわけではなく、ここでのコードのようにアプリケーション側で書かなくてはならないコードも結構あります。アプリ側で行う処理はマテリアルとテクスチャの処理ですが、今はこのようにするものだと思ってください。
また、自身でロード関数を作成する機会があるとすれば、この部分をコピー＆ペーストすれば大抵はこと足りると思います。

Render関数
Render関数には、3Dにおける基礎が詰まっています。
まず必ず憶えておかなくてはならないのは、3つのトランスフォーム（座標系変換）です。
197行目～201行目

①まず。ジオメトリのローカル座標からワールド座標（絶対座標）へ変換し、
203 行目～ 208 行目
②次に、ワールド座標をさらにカメラ座標に変換し、
210 行目～ 212 行目
③最後にカメラ座標からスクリーン座標に変換します。

この辺の詳細については、「はじめての 3D ゲーム開発」で詳細に解説していますが、ここではなんとなくで構いません。とにかく、その 3 つに変換があって初めてモニターに表示されるのだと憶えてください。

214 行目～ 228 行目
ここでは、“ ライト “ を作成しています。ライトが無くてもレンダリング結果は見えますが、いまひとつ見栄えがよくないので、多少コードは長くなりますが見栄えのほうを優先させました。

スプライトの場合ではレンダリングは 1 行で済みましたが、メッシュの場合はマテリアルとテクスチャをセットするコードが必要になります。メッシュは 1 個以上のマテリアルから成り立っています。マテリアルごとにマテリアル（当たり前）とテクスチャが異なるのが普通ですので、マテリアルの数だけループさせ、ループ 1 回転ごとにマテリアルとテクスチャを当該マテリアルのものでセットしなくてはなりません。

図 5-1　最初は離れているトマト・メッシュとメロン・メッシュ

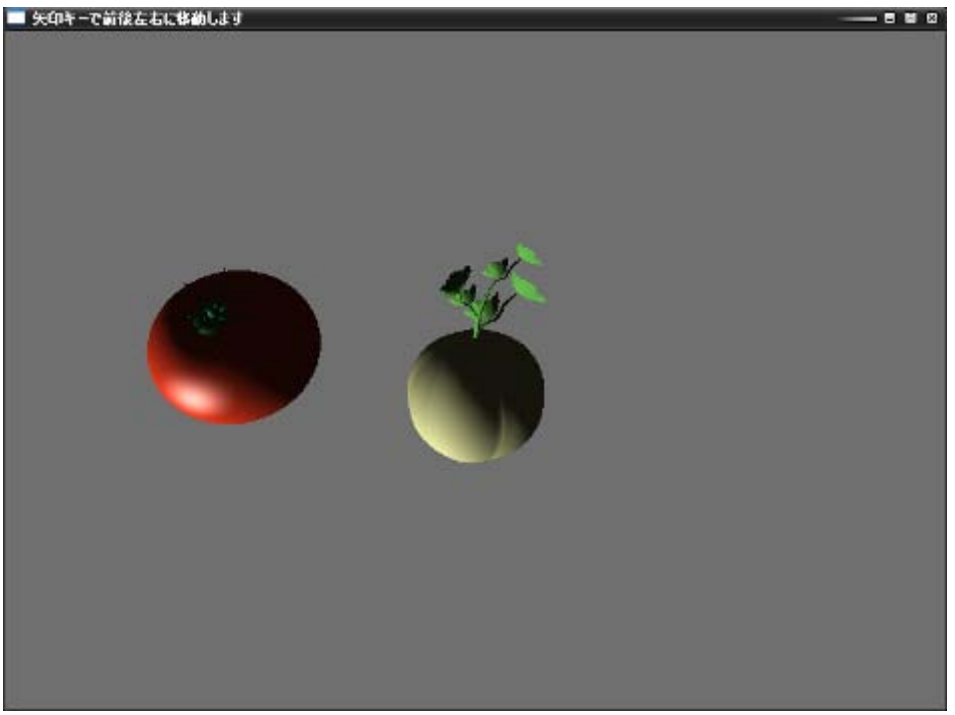
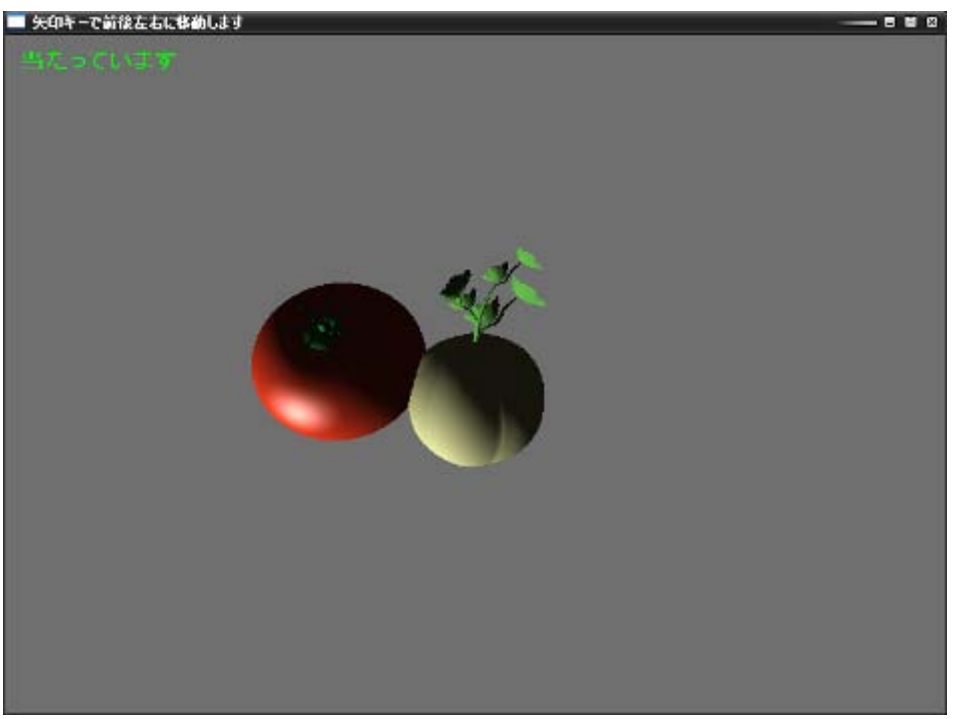


図 5-2　キーボード操作によりトマトが移動したたメロンと “ 当たった ” ところ。



さぁ、スプライトを含む 3D ジオメトリのレンダリングは出来るようになりました。単にレンダリン

グするだけではなくユーザーの操作を反映させてこそ画面表示は完成したと言えます。3D 空間でどうやって物体を移動したり回転させたりするのかを考えてみましょう。
それから、ゲームにおいてレンダリングと同じくらい大切なものは「当たり判定」です。“ 応用編 ” では当たり判定を「衝突判定」と呼んでいますが、初心者にはこの “ 当たり ” ほうが馴染むのでそう呼ぶことにしました。
当たり判定とは呼んで字の如く、物体が当たっているか離れているかの判定です。ここでの物体はメッシュになります。当たり判定の無いゲームはほとんどありません。必ずなんらかの当たり判定は必要になります。シューティングゲームでは弾ジオメトリと敵ジオメトリの当たり判定をしないことにはゲームになりません。

3D 空間における操作の仕組み

3D 空間での位置、つまり物体の場所は、位置ベクトルとして扱います。“ ベクトル ” と言うと難しく感じるのであれば、“ 座標 ” と考えても差し支えありません。差し支えないというよりも、位置ベクトルは座標なのですから、全く問題ありません。もちろん、ここでの座標は成分が 3 つの 3 次元座標です。
Direct3D は全て 3D 空間で考えます。先述のとおり 2D 的スプライトも然りです。3D 空間での座標(位置ベクトル）は、(x,y,z) の形をとります。例えば、(2,3,5) という座標は、X 軸プラス方向（左）に 2、Y 軸プラス方向（上）に 3 の座標をさらに､Z 軸プラス方向（奥）に 5 だけ押しやった位置になります。3D ジオメトリを移動するには、その座標の成分を増減させてやればいいのです。奥に移動させる場合は Z 成分を増加させるという具合です。
ただ、Direct3D に限らず 3DCG では座標をそのまま利用しません。座標だけではなく回転やスケール（縮尺）を一度に処理できる便利な論理ツールがあり、それを使用します。それが “ 行列 ” です。位置ベクトルおよび回転角やスケーリング係数は最終的に行列に置き換えます。この部分が 3D の敷居を高くしているのは確かです。しかし、行列に変換するのは、それなりのメリットがあるからであり大切なことです。行列やベクトルの概念及び仕組みは 8 章で詳しく解説しますが、とにかく、3D ジオメトリの操作はベクトルと行列で行うということを、今はなんとなく頭に留めておいて下さい。

当たり判定の仕組み

当たり判定と一口に言っても、その方法及び実装形態は様々です。高度な当たり判定及び詳細解説は 12 章から 17 章で行っています。
本章での判定は､原理及び解説がもっとも “ お手頃 “ な方法を採用しました。原理としては ” 境界球 “ というものですが、難しく考える必要はありません。単に、お互いの距離の大小関係で判断しているだけです。
判定は、“ 距離の大小 “ で行います。メッシュ A とメッシュ B があったとして、それぞれのメッシュは自分の中心から一定の ” 長さ ” を知っています。その長さより短い距離に相手が入ると ” 当たる “ 可能性があるような長さです。言い換えると、その長さより長い距離である場合、絶対当たらないような長さです。この長さを判断することにより、” 当たり ” と “ 当たらない ” という 2 つの状態の境界が出来ます。その境界の形状はちょうど、そのメッシュをすっぽり包みこむような ” 外接球 ” となり、その長さはその外接球の半径と言えます。これが境界球と呼ばれる所以であり、長さで判断するということは、境界球の内か外かを判断することに他なりません。

サンプルプログラム

プロジェクトフォルダ名「ch05 3D 物体の操作と当たり判定」

VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

使用方法
トマトをキーボードで操作します。メロンは動きません。トマトがメロンに重なる時、つまり、見た目に “ ぶつっている ” 状態を検出して “ 当たっています ” という文字を表示します。
トマトの前後左右移動：矢印キーで行います。
ESC キーで終了します。

コード解説

当たり判定を行っている部分は次のようにわずか数行しかありません。

```
// 当たり判定
 D3DXVECTOR3  vecDistance=Thing[1].vecPosition-Thing[0].vecPosition;
 FLOAT fDistance=D3DXVec3Length(&vecDistance);
 if( fDistance <  (Thing[0].fRadius+Thing[1].fRadius)  )
 {
          RenderString(" 当たっています ",10,10);
 }
```

```
D3DXVECTOR3  vecDistance=Thing[1].vecPosition-Thing[0].vecPosition;
FLOAT fDistance=D3DXVec3Length(&vecDistance);
```

Thing は、独自に定義した構造体です。メッシュが複数ある場合に、すべてグローバルに用意すると煩雑になるため用意しました。この構造体にはメッシュと、メッシュの位置、回転などの情報を格納できるようにしています。そして、本サンプルのポイントである境界球を形成する “ 半径 ” も格納しています。
Thing[1] は、メロンで、Thing[0] はトマトです。２つのメッシュの位置ベクトルの差（の絶対値）はメッシュ間の距離になります。ベクトル同士の差を求めた段階ではマイナスが付く可能性があるのはお分かりでしょう。この段階では、まだ絶対値である必要はありません。なぜなら、次の行にある D3DXVec3Length 関数が自動的に絶対値にしてくれるからです。D3DXVec3Length はベクトルの長さを計算します。長さなので、マイナスが付く場合はマイナスを取り除いてくれます。
ここで、fDistance には２つのメッシュ間の距離が入ることになります。

```
if( fDistance <  (Thing[0].fRadius+Thing[1].fRadius)  )
```

もし “2 つのメッシュ間の距離 fDistance” が、“2 つのメッシュそれぞれの半径を足したもの “ より小さい場合、それぞれの境界球は ” 当たっている ” と言えます。
詳しくは、12 章で解説しています。12 章の図 12-2 〜図 12-4 を見てみると理解が深まるでしょう。

ここまでが本章のポイントですが、本サンプルはその他に、今までのサンプルには無い機能を実装しています。それは “ 文字の描画 ” です。当たり状態を確認する目的で画面にメッセージを出すために実装したものです。今後も殆どのサンプルに実装される機能なので、すこし触れておきます。
文字の描画
Direct3D には文字描画用の ID3DXFONT というインターフェイスが存在します。
31 行目では、次のようにそのインスタンスを作成しています。

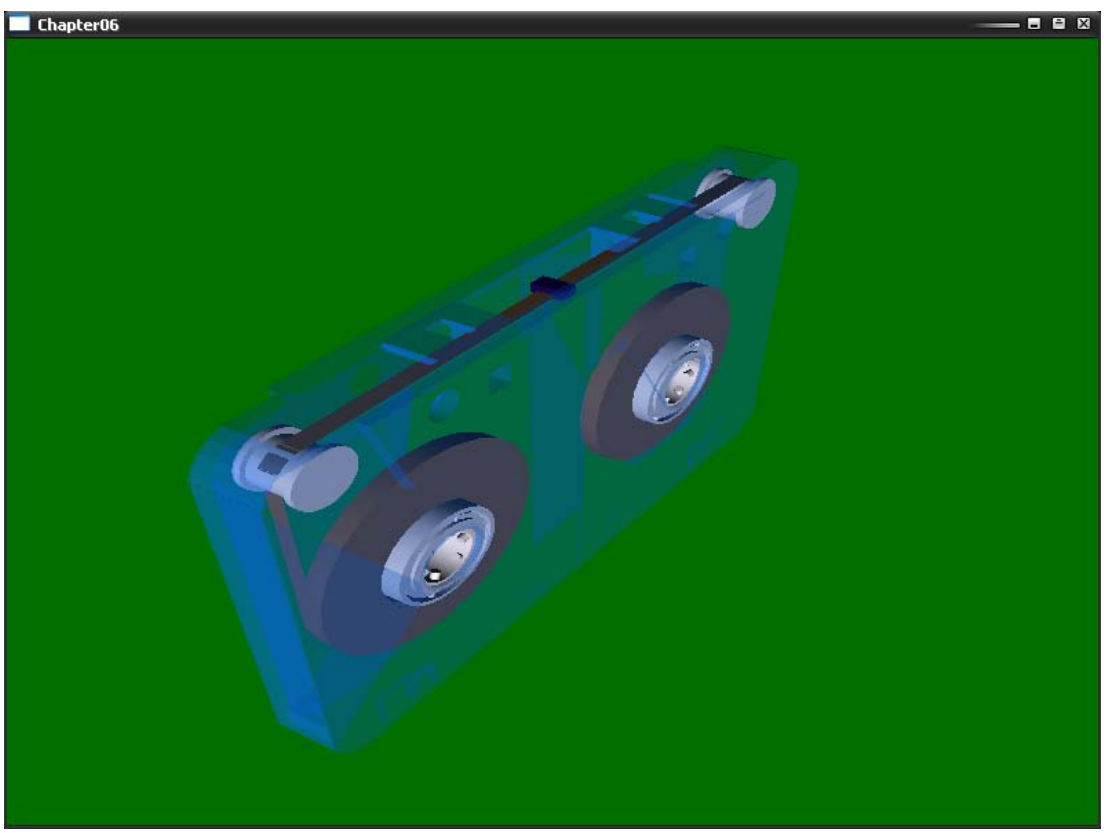
```
LPD3DXFONT pFont = NULL;
```

文字を描画できるようにするためには、インスタンスを適正に初期化しなくてはなりません。初期化コードは次のように簡単なものです。
195 行目～ 197 行目
// 文字列レンダリングの初期化
if(FAILED(D3DXCreateFont(pDevice,0,10,FW_REGULAR,NULL,FALSE,SHIFTJIS_CHARSET,
OUT_DEFAULT_PRECIS,PROOF_QUALITY,FIXED_PITCH | FF_MODERN, "tahoma", &pFont))) return E_FAIL;
これは、このようにするものだと思ってもいいでしょう。というよりも、むしろそう思ったほうがいいとさえ思います。原理性が無く仕様の問題ですから。

実際に文字をレンダリングしているコードは RenderString 関数です。この関数も数行であり、これも深入りしないことにします。また、他のプロジェクトにほぼそのまま使用することができます。

6 章　半透明にする

図 6-1　カセットテープのほとんどは半透明ですよね。



ここでは、メッシュの背後のものが透けて見えるような処理、つまりメッシュを半透明にするということをやってみましょう。
半透明処理はアルファブレンディングと呼ばれます。アルファブレンディングの詳細については、19 章で解説しています。

ここでは、簡単にアルファブレンディングを実践してみます。

サンプルプログラム

プロジェクトフォルダ名「ch06 半透明にする」
VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

使用方法
起動して眺めるだけです。ESC キーで終了します。

コード解説

あまりにも簡単なので解説することはありません。ポイントは次の３行だけです。
184 行目～ 187 行目

```
pDevice->SetRenderState ( D3DRS_ALPHABLENDENABLE, TRUE );
pDevice->SetRenderState ( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
pDevice->SetRenderState ( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

これは丸暗記とはいかないまでも、そのまま覚えるなり使うなりしてください。
これだけです。
この章は、本書の最小文字数をマークしました（笑）。

7 章　アニメーションメッシュ

図 7-1　ミサイルラウンチャーです。普通のメッシュではなく、“ アニメーション・メッシュ ” です。

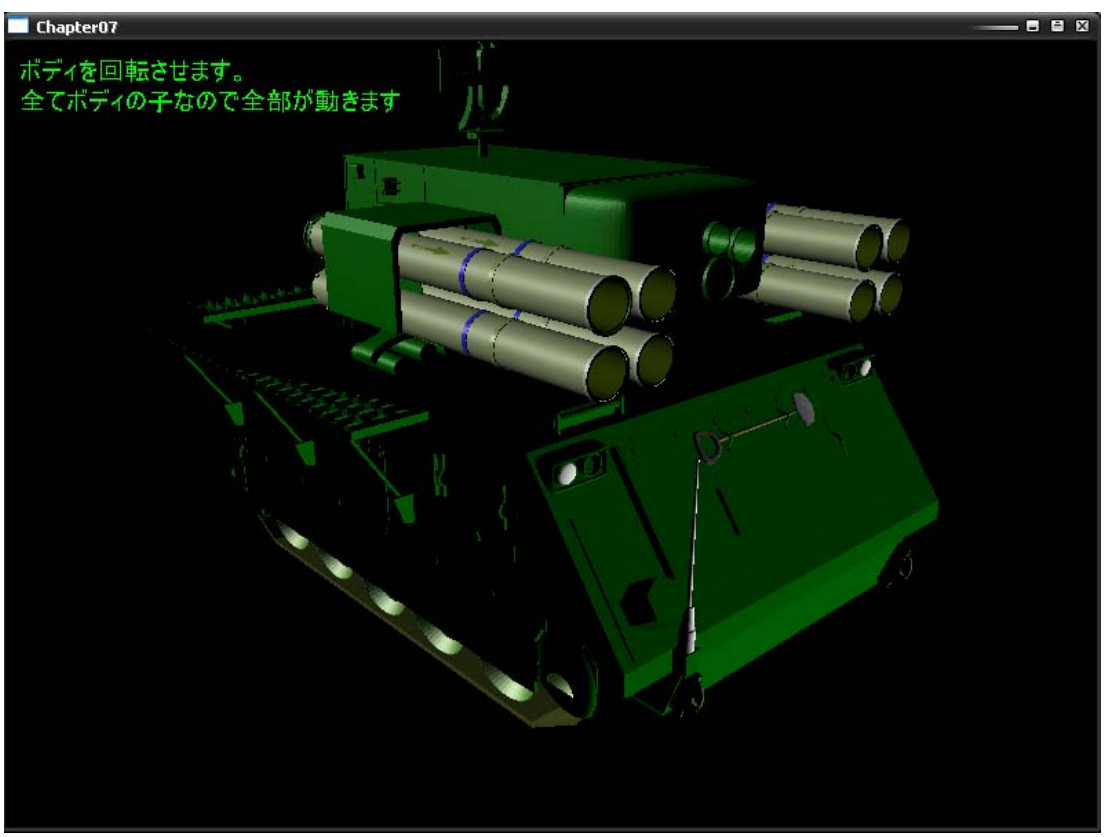


図 7-2　ボディ部分（下部分）が回転しています。

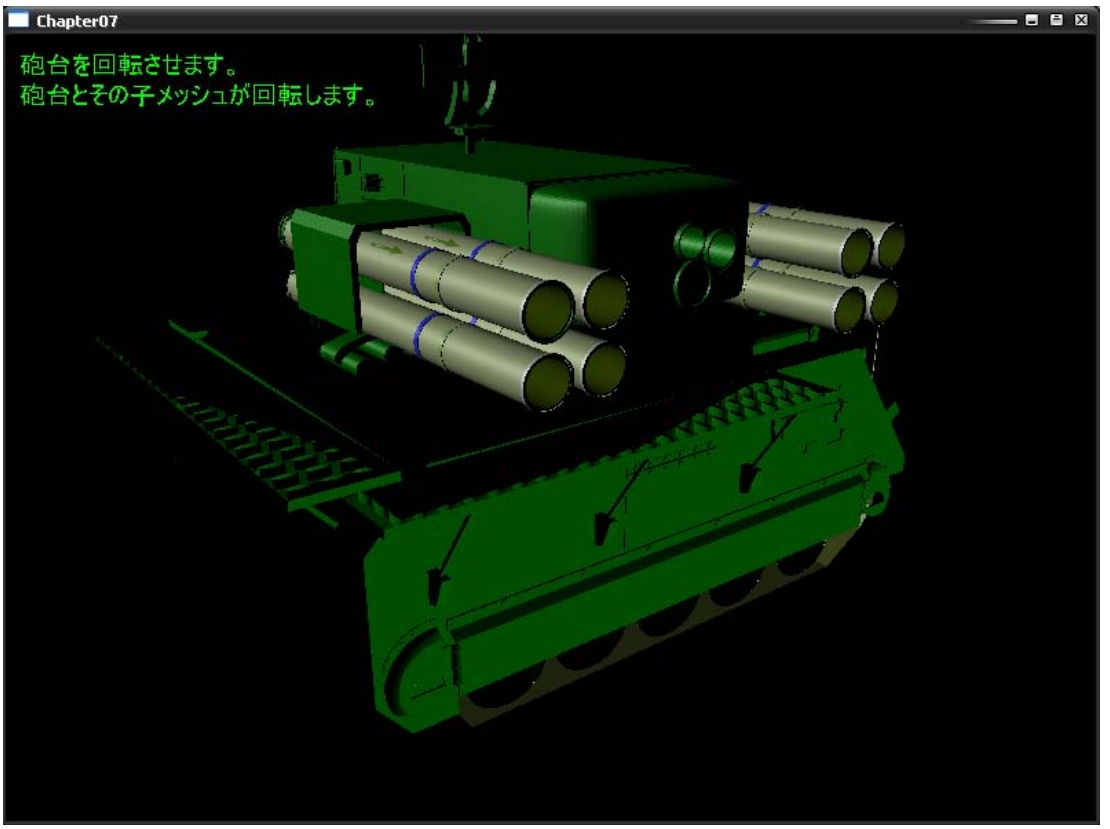
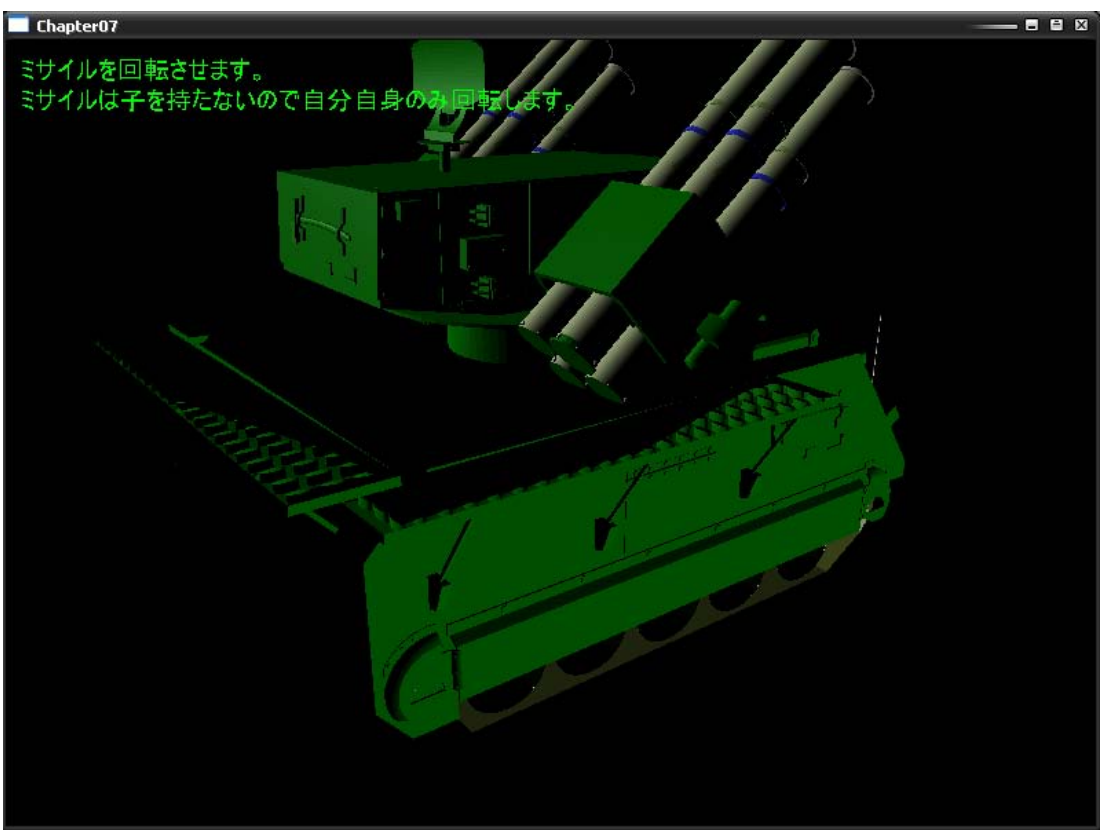


図 7-3　ミサイル部分が回転しています。



アニメーションは、複数のメッシュを用いて、その相対位置や姿勢を変化させて実現します。その仕組みは戦車に例えると分かり易いので戦車アニメーションメッシュを用いました。一見すると 1 つのメッシュに見えますが、各パーツ（ボディ、砲台、ミサイル部分、レーダー部分）は別々のメッシュです。それぞれは階層関係で結合され 1 つの “ アニメーションメッシュ ” として 1 つの X ファイルに収録されます。
アニメーションの詳細については、ここではレベルを超えているので解説しません。
ここでのアニメーションメッシュにおいて、個々のパーツとなるメッシュそのものは変形しません。ちょうど現実の戦車と同じです（全体としては変形すると言ってもいいですが、個々の部位は硬い剛体です）。
これに対し、メッシュ自体を変形させることができるメッシュがあり、そのようなメッシュはスキンメッシュと呼ばれます。スキンメッシュはメッシュ自体を時間とともに変形させることができるので、1 つのメッシュでもアニメーション可能ということになります。スキンアニメーションは有機物体や流体物の表現に向いています。たとえば、爬虫類や恐竜、アメーバー、筋肉の動き、あるいは液体などです。それらは、通常のメッシュでは表現が不可能とさえ言えます。スキンメッシュによるスキンアニメーションについては、18 章で解説しています。

サンプルプログラム

プロジェクトフォルダ名「ch07 アニメーションメッシュ」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
基本的に眺めるだけですが、スペースキーを押すことによりアニメーションを一時停止することができます。再開するには、もう一度スペースキーを押します。このように 1 つのキーやボタンで “ オン・オフ ” スイッチとして機能するものはトグル（スイッチ）と呼び、そのように機能させることをトグリングとか “ トグルする “ と言います。

コード解説

ここではじめてファイルを複数使用します。初心者には分かり辛いので、努めて 1 つのファイルになるようにしているのですが、アニメーションはクラスの継承コードを書かなければならないので、1 つのファイルに全て詰め込むと逆に見辛くなってしまうので、やむを得ず分けました。
各ファイルの内容は次のとおりです。
Chapter07.h
全体で使用する定数、マクロ、データ型の定義をしています。

Chapter07.cpp
メインとなる実装ファイルです。ウィンドウの作成、Direct3D の初期化、レンダリング、メッセージループ等。

Hierarchy.cpp
Direct3D では、アニメーション用の読み込みインターフェイスはあるのですが、そのインターフェイスの主要な 4 つのメソッドは仮想関数として定義されているので、実装はアプリケーション側で行わなければなりません。このファイルでは、その実装を行っています。
中身は、本編のレベルを超えている感が多分にあるので、深入りしないようにしましょう。

中・上級者のための応用編（Direct3D）

8章　必要な数学的知識

2D のみのゲームを作る場合、数学的知識はほとんど必要無いと思います。もちろん 2D と言えども内容によっては複雑な数学を使うこともあるでしょうが、少なくとも必須となる数学は 3D と雲泥の差があります。稀な場合を除いて多くの実装は、数学的なアルゴリズムはせいぜい線の方程式や面積などの中学数学程度か、あるいは全く数学理論を用いない場合も多いと思います。逆に言えば、2D は数学を特別意識しなくとも実現できるということにもなります。

しかし、3D の場合はそうはいきません。2D が言ってみれば “ ペラペラ ” の 2 次元デカルト座標系で考えればいいのに対し、3D の場合は 3 次元ユークリッド “ 空間 “ で考えなくてはなりません。3 次元ユークリッド空間とは我々が日々身を置いている現実の 3 次元空間のことだと思っても不都合はありません。さらに特殊なエフェクトを意図した場合にはユークリッド空間よりも広いアフィン空間内での変換を行う場合もありますが、通常は我々が直感的・経験的に想像するノーマルなユークリッド空間上で考えます。（なお、2D 平面は 2 次元ユークリッド空間とも言います）

さらに、3D の場合はハードウェア自体（ビデオカード）が、数学というインターフェイスを通さないと運用すら出来ない構造になっているので、嫌でも関連する数学を理解しなくてはなりません。つまり、ゲームコーディングに数学理論を適用するかしないかではなく、適用しなければコーディングすら出来ないようになっています。

3D で使用する数学は幾何学と線形代数学です。

3D 空間や 2D 平面を幾何的に考えるのが 3DCG であるので、学問云々以前に幾何的な発想は自明の条件です。幾何学といっても 3DCG においては古典的なユークリッド幾何学であって、さほど複雑なものではありません。重要なのは線形代数学のほうです。重要というより、線形代数学は初等幾何学の知識を前提にしているので幾何的な考察では当然幾何学の知識が必要になるので、線形代数学は幾何学を包含しています。“ 幾何学 ” という学問を独自に勉強する必要があるとすれば。Direct3D にとっては三角関数くらいです。

3DCG は線形代数から成っているといってもいいでしょう。3DCG、Direct3D における線形代数学の分野はベクトル、行列です。計算式だけを見ると無味乾燥とも思える線形代数が幾何的処理に本当に有効なのかどうか、行列が何の役に立つのだろうと思った人は少なくないと思います。筆者が高校生だった頃、行列とベクトルは「代数幾何」という数学の 1 選択科目でした。現在それらがどのような科目名になっているかは知りませんが、当時正直こう思いました、ベクトルの説明を受けても「だからなんなんだ」、行列の解説を聞いても「それが何の役に立つんだ？」と…。代数幾何は最も掴みどころの無い “ 単なる計算 “ としか写りませんでした。訳の分からない定義が先行し、ただ無意味な計算をしているような気分でした。それは教える側の説明方法によるところも大きかったと思うのは、いくぶん自分勝手でネガティブな考えかもしれませんが、少なくとも半分は当たっていると今では思えます。しかし、3DCG の原理を勉強するにしたがって行列やベクトルという概念がどれほど重要で便利なものであるかということに驚きました。3DCG は線形代数学の幾何的な定理をディスプレイ上で実際に “ 証明 “ している格好の例ではないかと思います。我々は線形代数の幾何的な成果をディスプレイ上に見ていることに他ならないとも思います。3DCG は線形代数を、その意義を知らなかった筆者にとって、まさに ” 生きた数学 “ にしてくれたのは事実です。高校当時 3DCG が浸透していて、かつ教師の側にその素養があったなら、3DCG は格好の教材になったのではと思ったりします。

3DCG ひいては Direct3D が線形代数そのものだということを述べた次に、ではその中でどのような概念が登場するのか、具体的には次の 3 つです。

三角関数
ベクトル
行列

この３つの概念を駆使して、考え、実装していきます。
嫌な予感がした読者も少なくないことと思いますが、安心してください。これらは掘り下げて勉強すれば膨大な時間と労力が必要ですが、最初の入り口は思ったよりずっと入り易いと思います。しかも、レンダリング結果というこれ以上ないほどの納得材料があるのです。
さて、これら 4 つを個々に解説するかどうか、悩ましいところです。ベクトルはもっとも頻繁に登場する基本的な概念です。行列はベクトルのスーパーセットのようなもの、クォータニオンの虚数部は 3 次元ベクトルのように振る舞いうという具合に他の概念のプリミティブ的な存在でもありすから。個別にそれぞれの定義から定理の証明までをずらずらと書くよりも、それぞれがどのような状況でどのように 3D に関わっているのか、どのような効果があるのか、Direct3D と絡めて解説していくアプローチをとります。
とは言っても最低限度の定義くらいは書いておかないと何のことか分からないのでそれはそれぞれ示します。

なぜ数学？結論から先に

最初に言っておいたほうが良いと思われることが 1 つだけあります。
なぜ数学が必要なのか？ということです。
一言で言えば「楽だから」ということになると思います。これは “ 数学を勉強する苦労 ” よりもその “ 楽さ ” のほうが大きいという事実も含みます。
数学は、定理の蓄積で成り立っています。その定理の 1 つ 1 つは、先人が何十年・何百年とかけて発見、検証してきた貴重な知的財産です。
コンピューターに何かしらの処理を行わせる際、目的を実現させるための実装はさまざまな方法があり、数学を利用せずとも出来る場合も多いでしょう。しかし大抵の場合、苦労して考えたアルゴリズムであっても、数学の定理を適用した思いもよらない新たなアルゴリズムと比較して、スピード・簡潔さの面で勝ることはありません。考えて見れば当たり前です。多くの学者達が膨大な時間（場合によっては生涯）と労力をかけて確立した発見に、我々がちょっとやそっと考えただけのものでは太刀打ちできるわけがありません。
ベクトルや行列、クォータニオンと言ってもコード的には、単なる FLOAT 型の単純な構造体でしかなく、それ自体で複雑な処理をしてくれるクラスのようなものではありません。それらがエレガントに処理をこなすのは定理があってこその話です。

3D ゲームの基本処理

そもそも 3D ベースのゲームにおいて、どのような処理をコーディングするのか？誤解と反論を恐れず書くと 3D ゲームにおいて必須の処理は次の 2 つだけです。
１．「ジオメトリの操作」
２．「ジオメトリ同士の衝突判定」
ジオメトリとは幾何（学）という意味です。3DCG において具体的にジオメトリとは点、線、面、及びそれらから成るオブジェクトの総称です。“ 操作 “ とはジオメトリの移動、回転、拡大縮小を意味します。そして衝突判定とはジオメトリ同士の重なり・交差を検知してその後の挙動を与えること、つまり ” ぶつかり “ を検知して自然に見えるようにすることを意味し、当たり判定とも言われ、その挙動を物理力学に沿ってより現実に近づけた高度なものは ” ダイナミクス “ と呼ばれます。次元による複雑さは異なりますが、これらは 2D ゲームを作成した経験がある人なら当たり前のことと思います。3D の場合では、この 2 つの処理に 4 つの概念がどう関わってくるかを理解しコーディングできれば 3D ゲームを作成するスキルの 90% は獲得したと言って良いでしょう。2 つの処理に付け加えるとすれば、３．「座標系の操作」がありますが必須ではありませんので、この章では 2 つの処理に的

を絞りましょう。座標系の操作は主に回転系のサンプルコード部分で解説することになります。
ゲームに係る処理全体で見ると、この他にジオメトリのレンダリング処理がありますが、これはDirect3D側でほとんどのことを行うものなので通常は意識しないでコーディングできます。また、様々なゲーム仕様それぞれにおいて、これらの処理以外も考えられます。たとえばAIキャラの移動や振る舞い、会話もできるような高級なゲームでは自然言語ルーチンなど色々ありますが、それらは3Dコーディングの理解という観点からすれば本質ではありません。
数学がどのように関係しているのか、Direct3Dのコードを例にして説明していきます。

8-1 三角関数（Trigonometric Function）

サイン、コサイン、タンジェントと言えば、数学を知らない人でも聞いたことのある言葉だと思います。ほとんどの人はこれらが学校の授業で登場したことを憶えているのではないでしょうか。それらは三角形の辺の比率です。この３つのほかにさらに３つの比率がありますが、そちらはあまり馴染みが無いと思います。

サイン（sine） 正弦
コサイン（cosine） 余弦
タンジェント（tangent）正接
コタンジェント（cotangent）余接
セカント（secant）正割
コセカント（cosecant）余割

直角三角形の辺の比率をあらわすものを総称して三角比といいます。辺同士の比率はその長さに依存せず、比較する辺同士の成す角度によって求まることから、通常、三角比は角度からその値を求めます。三角関数とは任意の角度における三角比を導く関数であり、それぞれ次のように表記します。

サイン sin θ　正弦関数
コサイン cos θ　余弦関数
タンジェント tan θ 正接関数
コタンジェント cot θ 余接関数
セカント sec θ 正割関数
コセカント cosec θ 余割関数
θ は任意角

また、それぞれに逆関数があり、頭にアークまたはインバースが付きます。サインであればアークサイン、インバースサインと呼びますが、インバースＯＯとはあまり言わないようです。通常の関数が角度から辺の比率を求めるのとは逆に、辺の比率から角度を求めたいときに使用します。
アークサイン arcsin(x)　逆正弦関数
アークコサイン arccos(x)　逆余弦関数
アークタンジェント arctan(x) 逆正接関数
アークコタンジェント arccot(x) 逆余接関数
アークセカント arcsec(x) 逆正割関数
アークコセカント arccosec(x) 逆余割関数
(x) は、辺の比率、例えば arcsin(x) は arcsin(sin θ) となる。

関数 6 個と逆関数 6 個、合計 12 個の関数を総称して三角関数または円関数と呼びます。
ここでは、サイン関数、コサイン関数、タンジェント関数、及びそれぞれの逆関数に注目します。

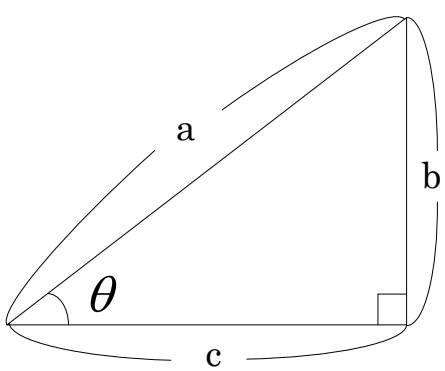
関数の定義
辺の比率を y、その辺同士の成す角度を x とすると y=sin(x)、y=cos(x)、y=tan(x) となるような関数が三角関数です。
直角三角形の直角でない一方の角を θ とし、θ に対応する斜辺を a、対辺を b、隣辺を c とすると、それぞれの関数は、“ θ →辺同士の比率 ” の写像を次の組み合わせで行います。

$$\sin\theta=\frac{b}{a}$$

$$\cos\theta=\frac{c}{a}$$

$$\tan\theta=\frac{b}{c}$$

図 8-1

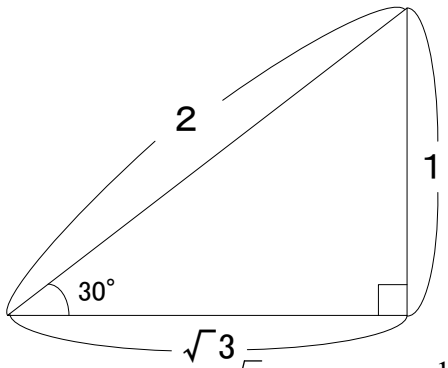


$$\sin\theta=\frac{b}{a}\qquad\cos\theta=\frac{c}{a}\qquad\tan\theta=\frac{b}{c}$$

$$\arcsin\frac{b}{a}=\theta\qquad\arccos\frac{c}{a}=\theta\qquad\arctan\frac{b}{c}=\theta$$

関数は、θ が既知の場合において、未知である辺の比率を写像するものですが、反対に、辺の比率が既知であり、角度 θ が未知の場合は逆関数を使用します。例を挙げると次のようになります。

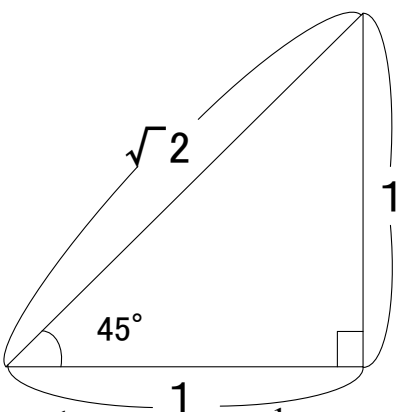
図 8-2



$$\sin 30° = \frac{1}{2} \quad \cos 30° = \frac{\sqrt{3}}{2} \quad \tan 30° = \frac{1}{\sqrt{3}}$$

$$\arcsin \frac{1}{2} = 30° \quad \arccos \frac{\sqrt{3}}{2} = 30° \quad \arctan \frac{1}{\sqrt{3}} = 30°$$

図 8-3



$$\sin 45° = \frac{1}{\sqrt{2}} \quad \cos 45° = \frac{1}{\sqrt{2}} \quad \tan 45° = 1$$

$$\arcsin \frac{1}{\sqrt{2}} = 45° \quad \arccos \frac{1}{\sqrt{2}} = 45° \quad \arctan 1 = 45°$$

組み合わせは小文字筆記体！
偶然なのかどうかは分かりませんが、sin、cos、tan の辺の組み合わせは、それぞれの頭文字の小文字筆記体の書き順に一致しています。

図 8-4

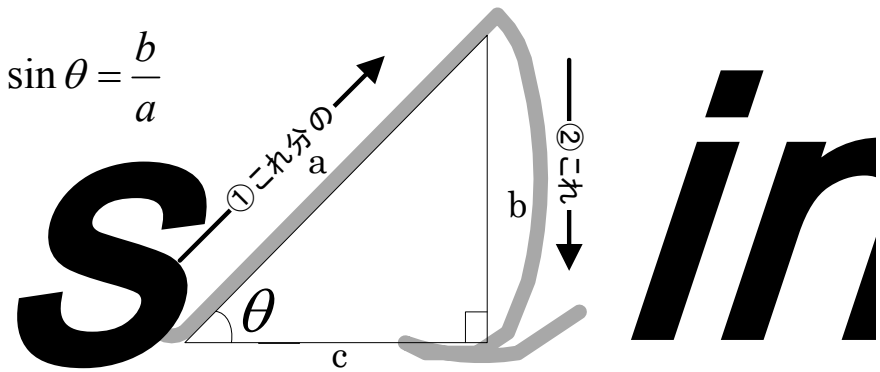


図 8-5

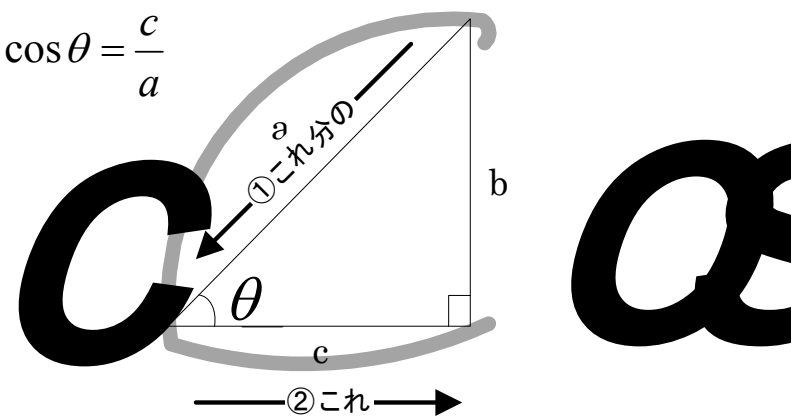
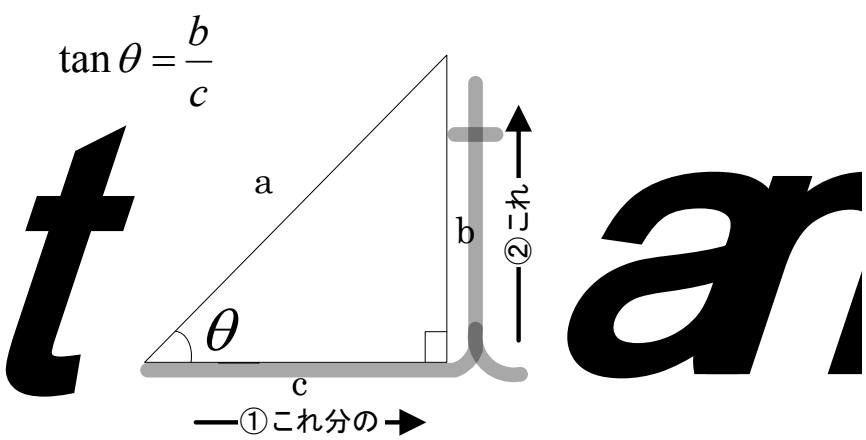


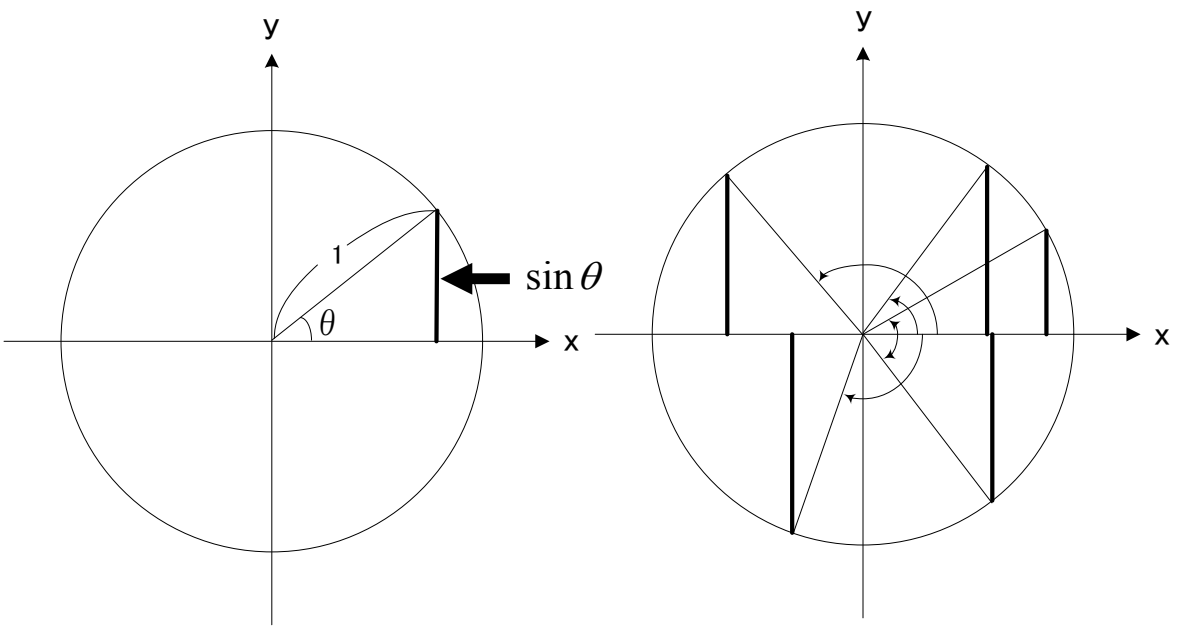
図 8-6



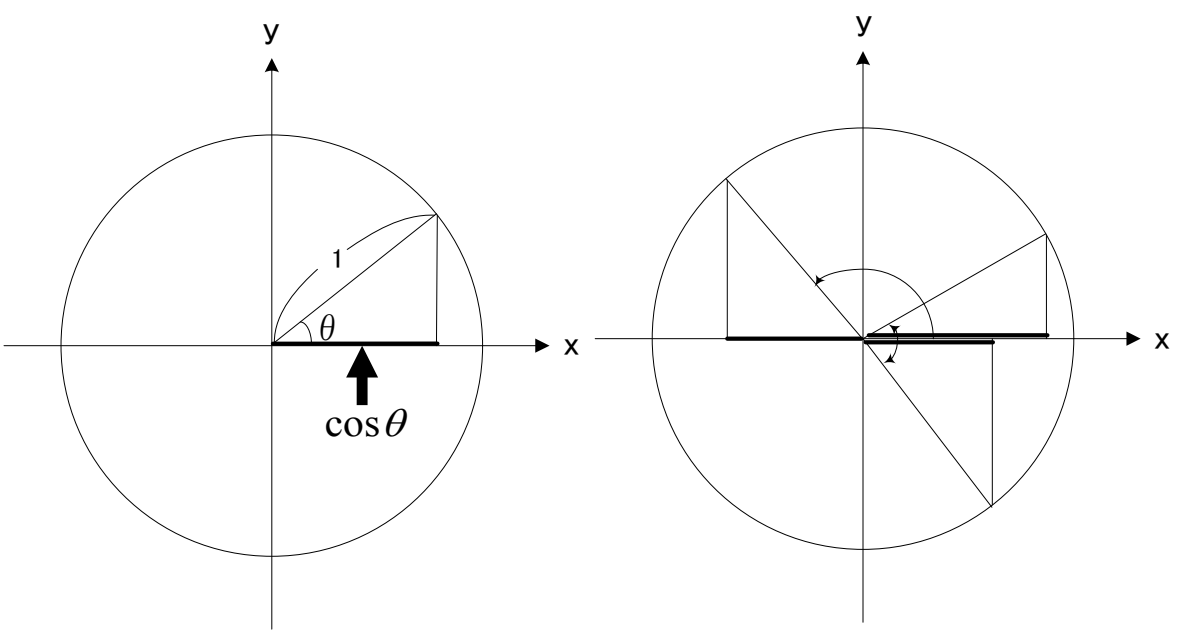
単位円で考える
三角関数は円関数とも呼ばれます。全ては単位円（半径が 1 の円）の各象限に内接する直角三角形で説明が付きます。

三角形の斜辺が 1 ですから、sin の値は x 軸におろした垂線の長さになります。

図 8-7



同様に、cos の値は三角形の底辺の長さになります。
図 8-8

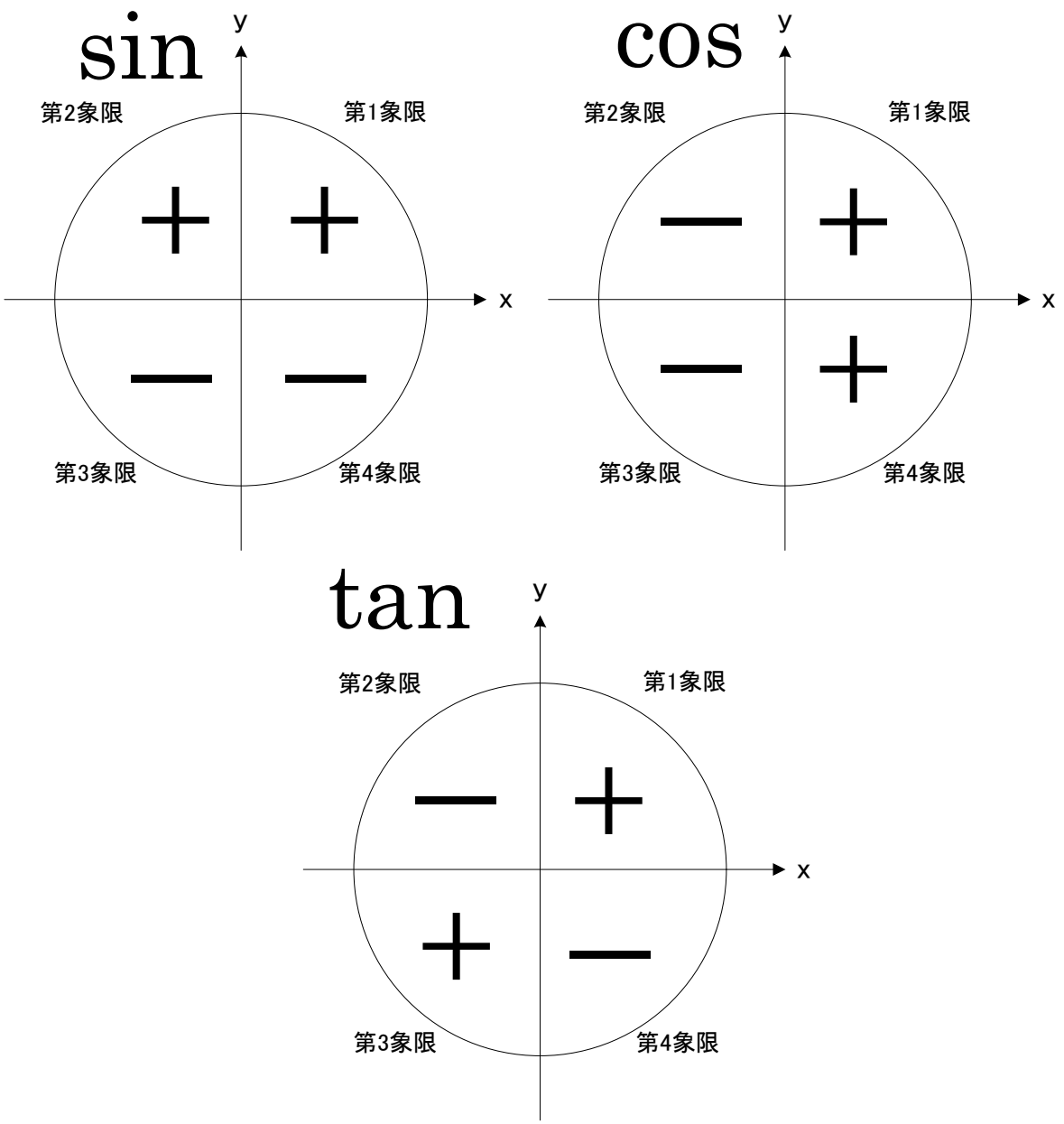


斜辺を時計の針のように回転させるようなアニメーションを頭の中でビジュアル的に想像すると、三角比を簡単に推測できたりします。

4 象限それぞれの符号

先の図から明らかなように、4 つの象限でのそれぞれの符号は次のようになります。
sin は第 1 と第 2 象限でプラス、第 3 と第 4 象限ではマイナス。
cos は第 1 と第 4 象限でプラス、第 2 と第 3 象限でマイナス。
tan は sin と cos の符号が同じ象限ではプラス、異なる象限ではマイナス。

図 8-9

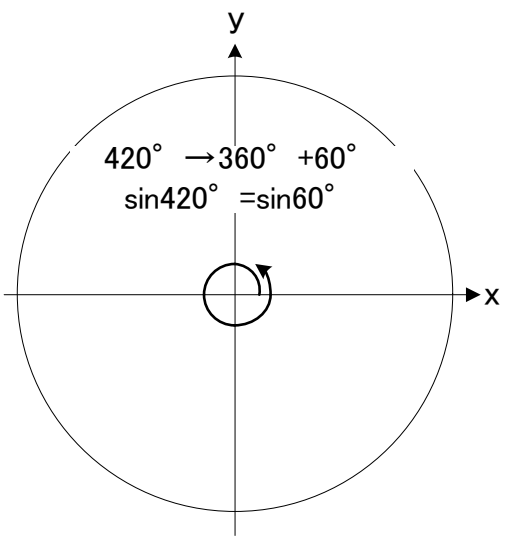


360 度以上の角度

3D では特に、回転に関わる部分で 360 度以上の角度の三角比を求める場合があります。三角関数では、360 度以上の角度は、360°で割った余り部分を角度とします。例えば sin1100°の場合、1100 は (360 × 3)+20 であり 360 の倍数部分は無視して、sin20°とします。回転角として捉える場合は何周したかという情報も大切ですが、三角関数では 360 の倍数部分は無視します。

また、マイナスの角度は 360 からその角度引いた角度で計算します。たとえば sin-90 の場合、sin(360-90) = sin270 となります。

図 8-10



60 分法と弧度（ラジアン）法

我々が日常的に慣れている角度の単位は 60°など「°（度）」でしょう。これは 60 分法と呼ばれます。しかし、3DCG ではほとんどの角度は 60 分法ではなく弧度（こど）法で表されます。弧度は別名ラジアンと呼ばれ、弧度法による角度の単位はラジアンとなります。

弧度法において、半径と同じ長さの円弧の円周角度は 1 ラジアンと定義されています。したがって、

長さが a の円弧の円周角 θ は $\frac{a}{r}$ ラジアンとなります。

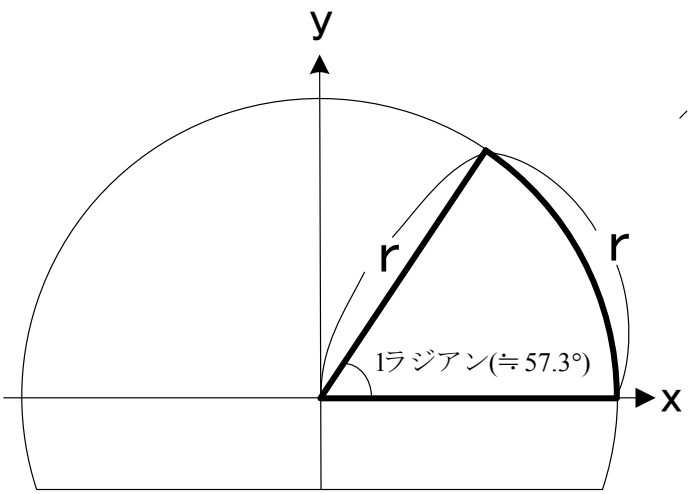
全円周は直径×円周率ですから 2 π r です。これを上の式に当てはめると全円周の角度は $\frac{2\pi r}{r}$ すなわち 2 π となるわです。

60 分法では全円周は 360°であるのに対し弧度法では 2 π なので、60 分法と弧度法の比率は 360：2 π、180：π となるので、度数とラジアンの変換は次のようになります。

θ（ラジアン）= $\frac{\pi}{180}$ × 度数

θ（度）= $\frac{180}{\pi}$ ラジアン値

図 8-11



θ（ラジアン）$=\frac{\pi}{180}\times$度数（ラジアン）

θ（度）$=\frac{180}{\pi}\times$ラジアン値（度）

$360°=2\pi$ラジアン　$180°=\pi$ラジアン

$90°=\frac{\pi}{2}$ラジアン　$30°=\frac{\pi}{6}$ラジアン

三角関数の重要な定理
加法定理

$$\sin(\alpha\pm\beta)=\sin\alpha\cos\beta\pm\cos\alpha\sin\beta$$
$$\cos(\alpha\pm\beta)=\cos\alpha\cos\beta\mp\sin\alpha\sin\beta$$

よく使用する公式も加法定理より導かれます。

$$\sin 2\alpha=\sin(\alpha+\alpha)=2\sin\alpha\cos\alpha$$
$$\cos 2\alpha=\cos(\alpha+\alpha)=\cos^2\alpha-\sin^2\alpha=2\cos^2\alpha-1=1-2\sin^2\alpha$$

8-2 ベクトル（Vector）

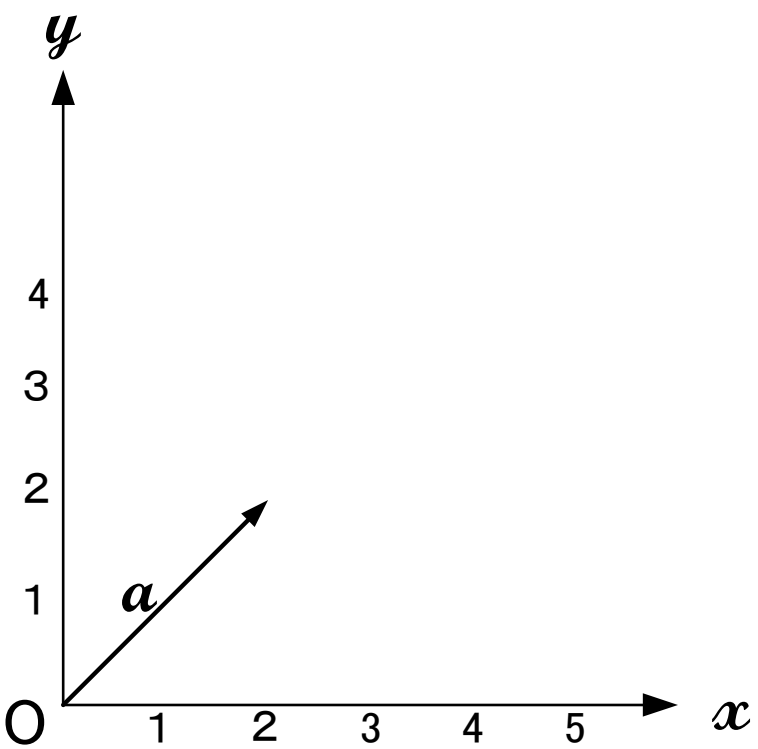
ベクトルという概念が速度や力を表す際に有効であることを最初に示したのは「メビウスの帯」の発見者である数学者（独）オーギュスト・フェルディナント・メビウス（August Ferdinand Mobius 1790-1868）と言われています。
ベクトルは 3D において、というより 2D も（n 次元も）ひっくるめて線形代数の世界では " 数 " のように無意識的に使用する概念なので、いざそれは何なのかということを説明する場合、逆に説明し辛いものです。
言葉での定義として、ベクトルとは「向きと長さをもった量」です。
幾何的に表現すると、その方向に沿ってその長さだけ " 矢印 " を引く（書く）こととなります。
代数的に表現すると、たとえば 2 次元であれば（x,y）、3 次元であれば (x,y,z) と表現します。この場合の x,y,z をそのベクトルの成分と呼びます。
しかしこれでは何のことやら分かりませんね。
小中高校の授業でお馴染みの x 軸と y 軸からなるデカルト座標系で幾何的に解説していきます。

ベクトルの演算

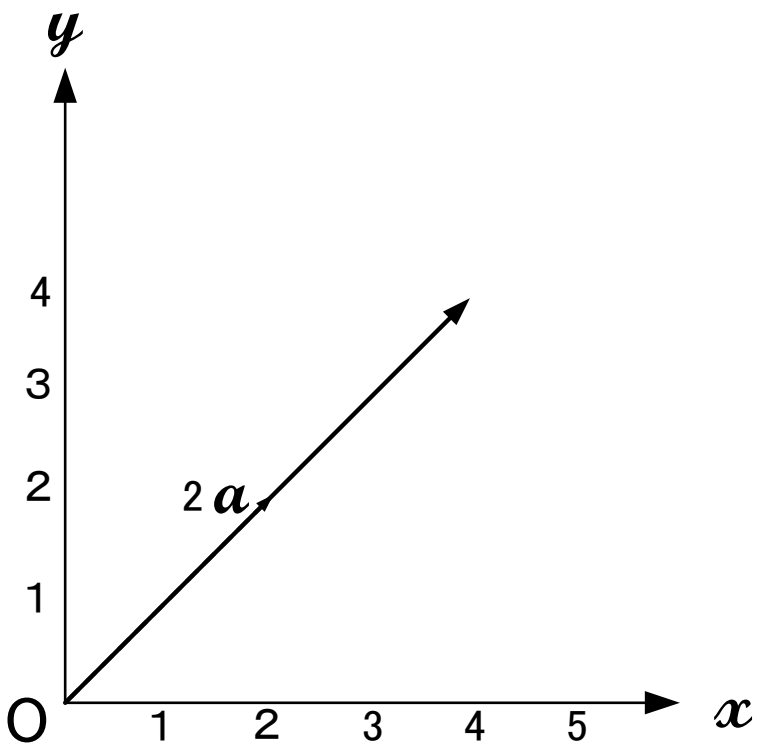
ベクトルの演算はスカラー倍、加算と減算（一方をマイナスとしてときの加算）があり、この 2 つ（3 つ）が直感的な数の演算と同じなので簡単です。それからちょっと変わった掛け算的な演算が 2 つあります。
それぞれを見ていきましょう。

スカラー倍

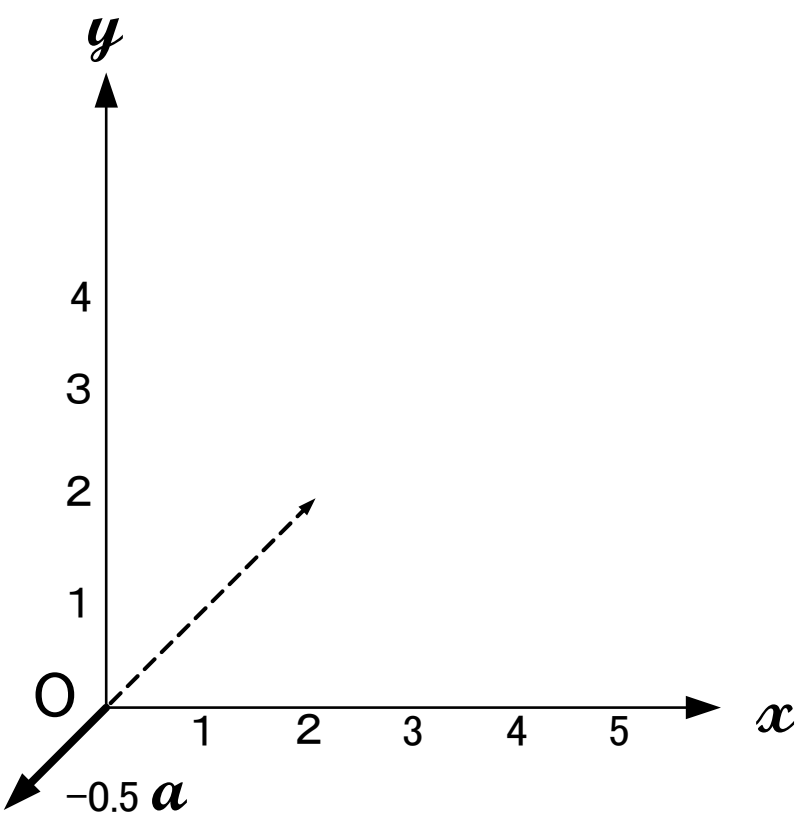
今、a というベクトルがあったとします。a の成分は a=(2,2) です。
図 8-12



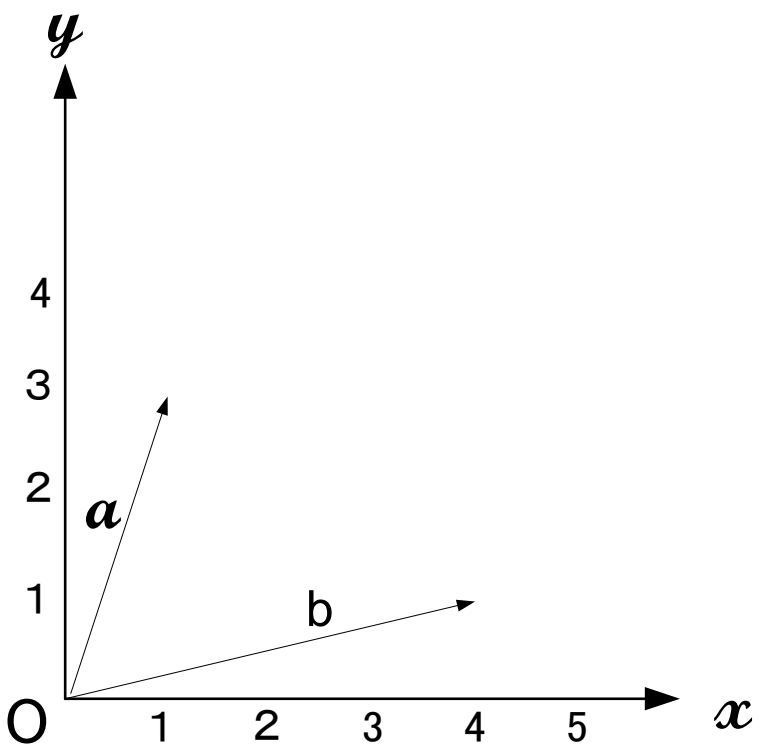
ベクトル a にスカラー（定数）を掛けるということは、単純に各成分にそのスカラーを掛けることです。たとえば、2 を掛けると、2 × a=(2 × 2,2 × 2)=(4,4) となります。幾何的には次のように長さが 2 倍になります。
図 8-13



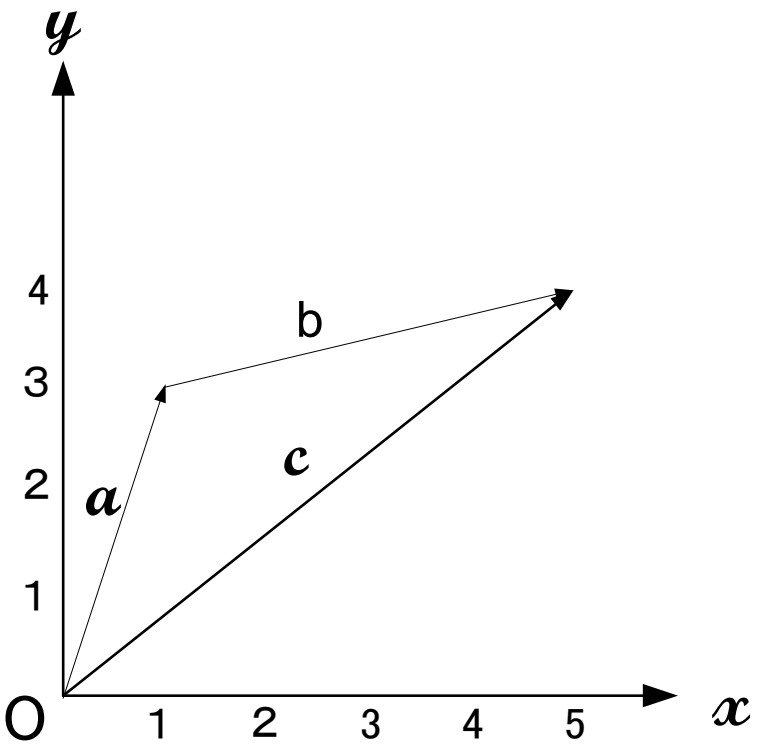
同じように、-0.5 を掛けると -0.5 × a=(-0.5 × 2,-0.5 × 2)=(-1,-1) となります。
図 8-14



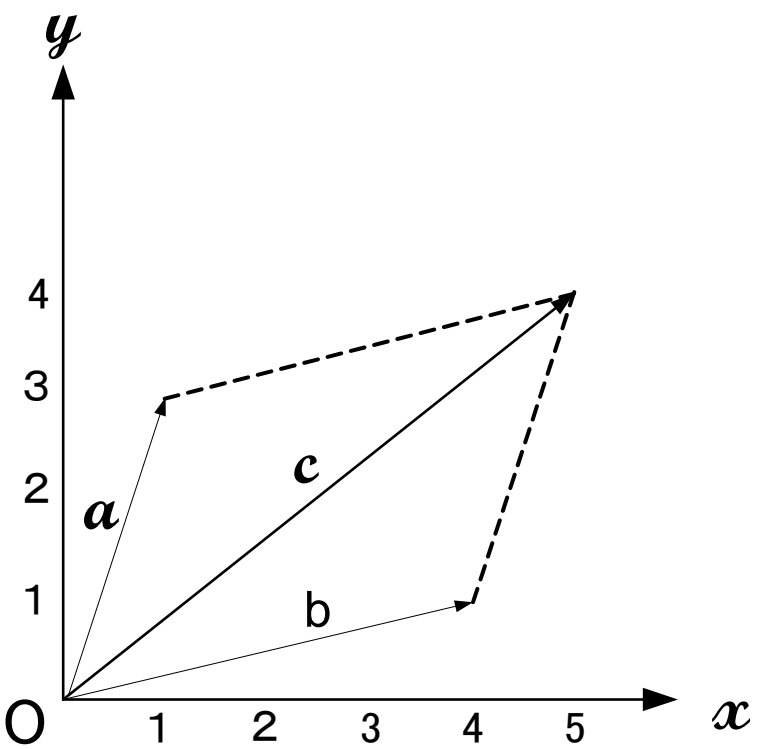
加減算
ベクトル a=(1,3)、ベクトル b=(4,1) である 2 つのベクトルの足し算と引き算について。
図 8-15



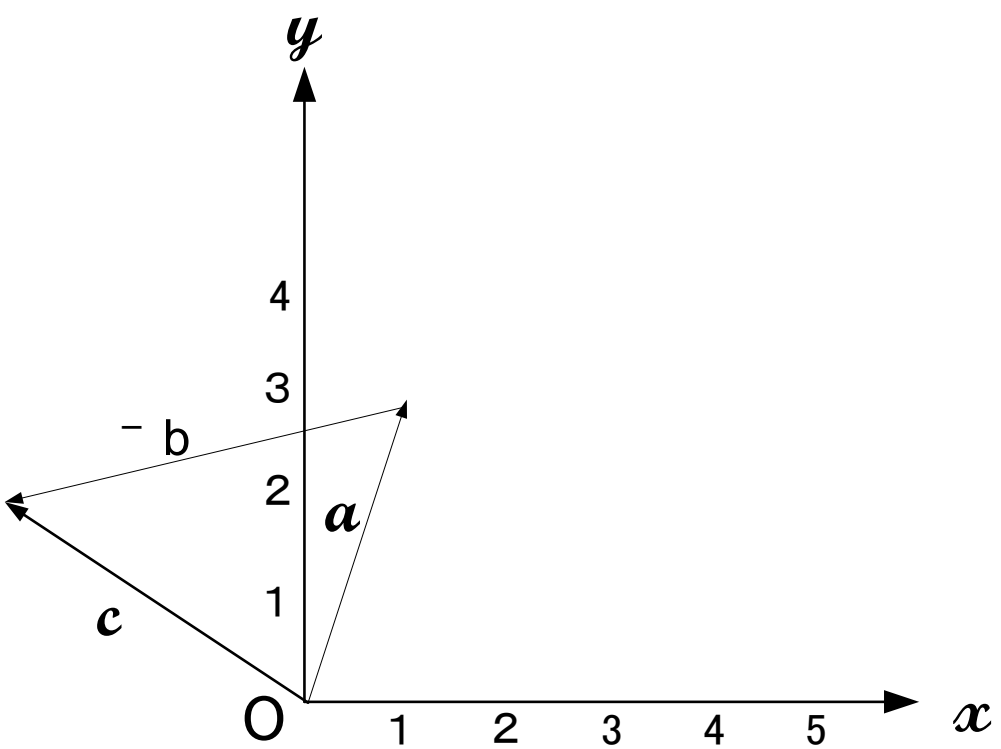
a と b を足したベクトルを c=a+b とします。
代数的には、単純に a と b の成分を足すことになります。c=(1+4,3+1)=(5,4)
幾何的に求める場合は次のように考えることもできます。
右辺の左側のベクトル（ベクトル a）の終点に、右側のベクトル（ベクトル b）の始点を持っていきます。つまり b を平行移動します。なお、ベクトルは平行移動しても成分は変わらず、同じベクトルとなります。
b を平行移動したあと、a の始点から b の終点を結ぶようなベクトルが a と b を足したベクトル c=(5,4) となります。
図 8-16



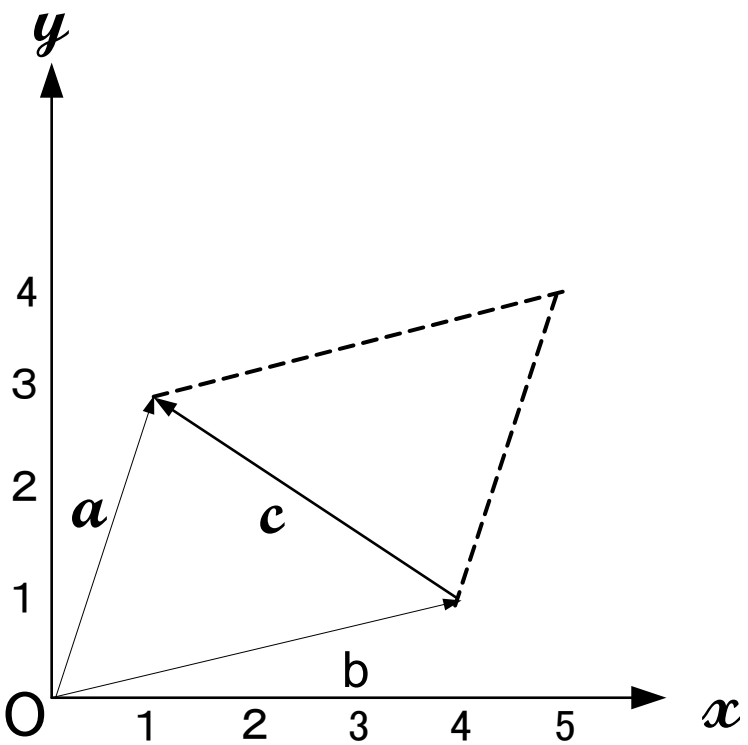
幾何的には、次のように考えることもできます。
a と b を隣り合う辺とする平行四辺形を作り、a と b の始点を起点とする対角線を引きます。この対角線が a と b を足したベクトル c となります。
図 8-17



次は引き算です。引き算と言っても、-b との足し算と考えることができます。a-b=a+(-b)
b をマイナスにします。マイナスにするということは、長さが同じで方向を真逆にすることです。代数的には各成分を単純にマイナスにすればいいだけです。b=(4,1) → -b=(-4,-1)
足し算であるので、同じように b の始点が a の終点となるように b を平行移動し、a の始点と b の終点を結ぶベクトルが演算結果となります。
図 8-18



引き算の場合も、平行四辺形の法則で考えることができます。ただし、足し算の時の対角線と逆の対角線となります。
図 8-19



内積

内積は、演算結果がスカラーになることからスカラー積とも呼ばれますが、スカラー倍と間違いやすいので内積に統一します。

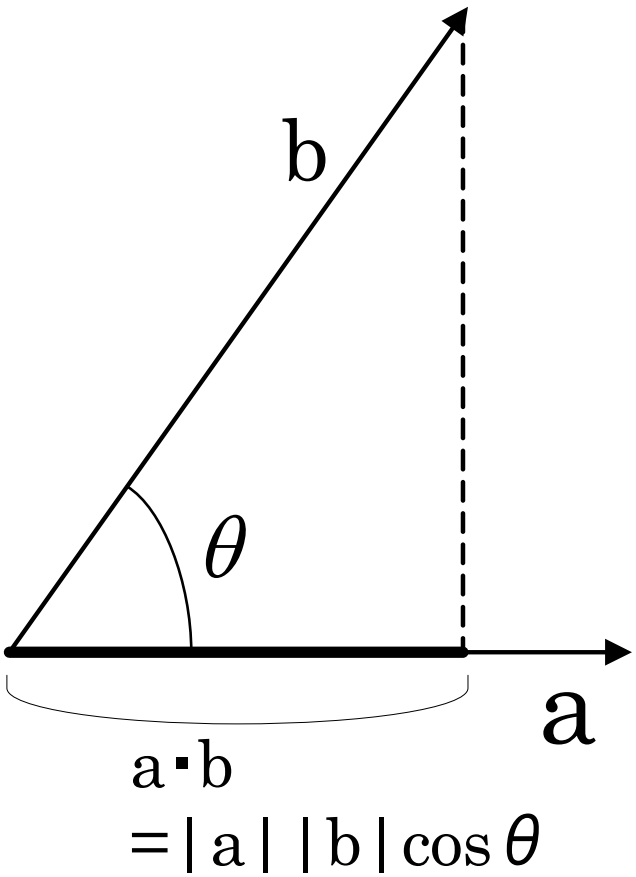
内積は

$a \cdot b = |a||b| \cos\theta$

と定義されます。a と b の間の “・”（ドット）は内積の演算子記号です。ドットではなく、(a,b) あるいは <a,b> と表記されることもあります。|a| は a の絶対値（長さ）です。|b| も同様です。θ は、ベクトル間の角度です。

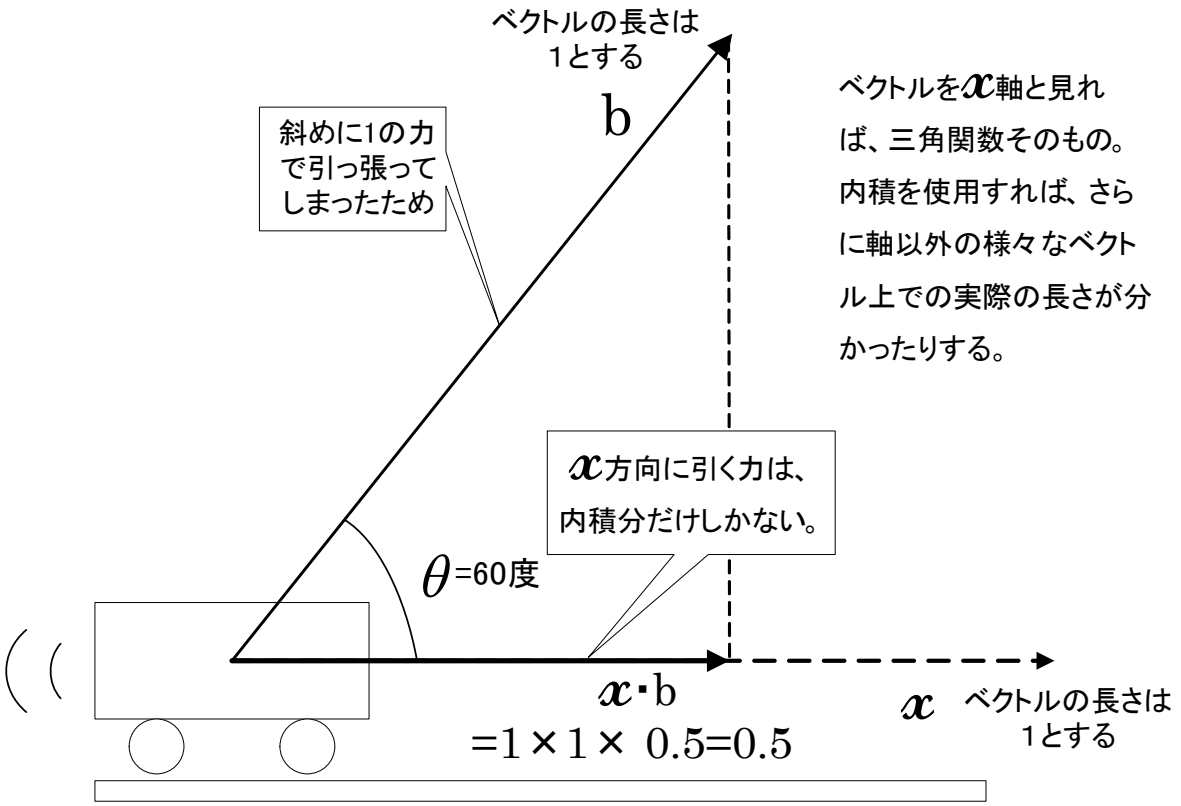
演算結果である内積の値は幾何的には次のようになります。

図 8-20



太線で強調している部分が、内積の演算結果となります。三角関数に似ていると感じることと思います。実際、a が基底軸に平行で長さが 1 であるなら、三角関数と同じと考えることもできます。内積は三角関数をさらに拡張し、より柔軟な軸上（ベクトル上）での三角関数と考えることもできます。内積がどのような時に有効であるのか 1 例を示すと、次のようにレール上の台車を斜めに引っ張る場合の正味の力を算出する場合があります。

図 8-21



外積
外積は、演算結果がベクトルになることからベクトル積とも呼ばれます。
3 次元空間上の 2 つのベクトル a=(x1,y1,z1)、b=(x2,y2,z2) とするとき
外積 A × B=(y1z2-z1y2)X+(z1x2-x1z2)Y+(x1y2-y1x2)Z
と定義されます。A と B の間の“ × ”（クロス）は外積の演算子記号です。クロスではなく、[A,B] と表記されることもあります。
X、Y、Z は基底軸ベクトルを意味します。直感的な 3 次元ユークリッド座標系（正規直交座標系）では基底軸が X=(1,0,0)、Y=(0,1,0)、Z=(0,0,1) なので、もっと簡単に次のように書けます。
A × B=(y1z2-z1y2 ,z1x2-x1z2 ,x1y2-y1x2)

演算の性質は
A × B = -B × A（A × B+B × A=0）
A × (B+C)=A × B+A × C
kA × B=A × kB=k(A × B)
(A × B)・A=(A × B)・B=0

演算結果となる外積ベクトル N=A × B は a と b の両方に垂直なベクトルとなり、幾何的に表すと次のようになります。

図 8-22

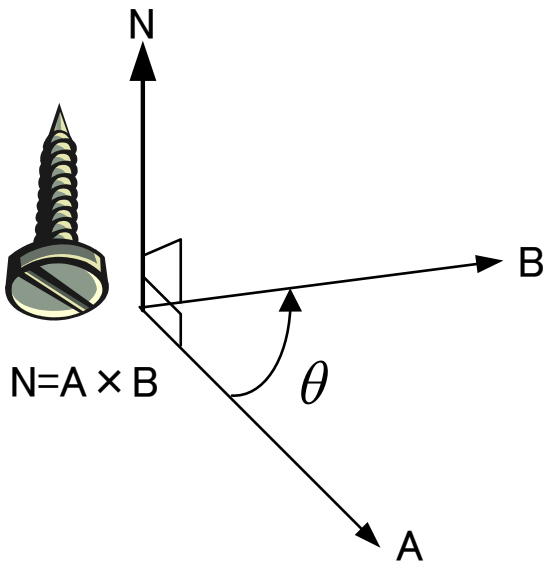
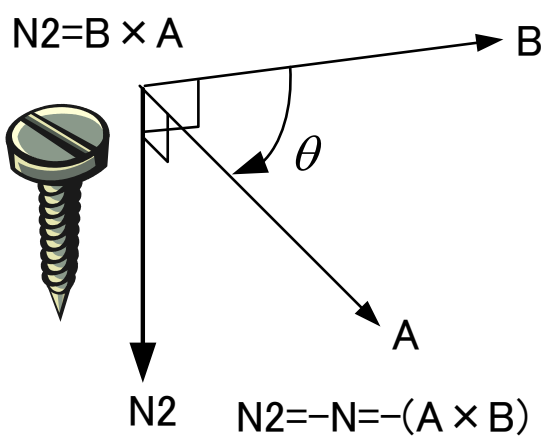


図 8-23



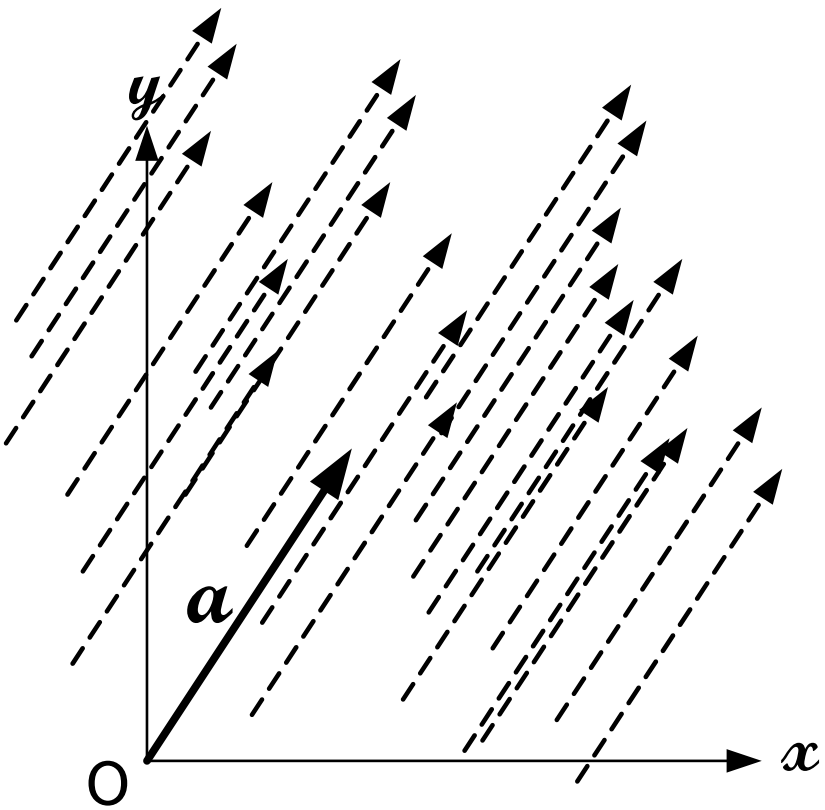
ベクトル A とベクトル B 両方に垂直なベクトルは上下 2 本存在しますが、
N=A × B において、右辺の左側のベクトル (ベクトル A) から右側のベクトル (ベクトル B) への方向が “ 右ねじの方向 ” となるベクトル 1 本が外積ベクトルです。したがって、B × A とすると、逆向きのベクトルが外積ベクトルとなります。

また次のような性質も導かれます。
A と B が作る平行四辺形の面積を S とすると、
$S=|A||B|\sin\theta=|A\times B|$（外積ベクトルの長さ）

ベクトルの同一性
ベクトルはそれ自体に位置はありません。“ 図中での位置 “ はベクトルにとって意味を持ちません。” 図中 “ で異なる位置にあるベクトルの成分をいくつか計算してみると分かりますが、全て同じになります。（なお、成分の計算は始点と終点の差で簡単に計算できます。位置ベクトル同士の引き算）

ベクトルの性質は「向きと長さ」だけの相対的な量なのですから当然のことです。向きと長さが同じベクトルを“等しい”と言い、等しいベクトルは無数に描けます。しかし、通常はそれらのベクトルの中で都合の良いものを“代表”として描きます。

図 8-24



向きと長さが同じベクトルは無数に描ける 。

ベクトル$\vec{a}$は、その無数のベクトルの”代表”と考える 。
通常、図示するのは代表1本で十分 。

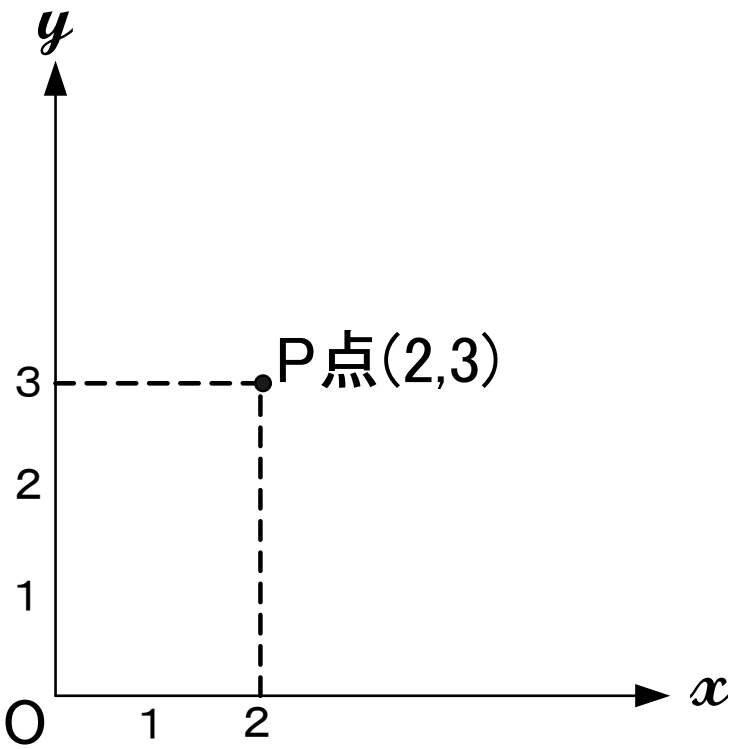
位置ベクトルと方向ベクトル

ベクトルは、その「捉え方」によって位置ベクトルあるいは方向ベクトルと“便宜上分けて呼ばれる”ことがあります。その２つは“捉え方”の違いであるので、あるベクトルが位置ベクトルになる時もあれば方向ベクトルとなる時もあります。どうゆうことなのか考えていきましょう。

位置ベクトル（Position Vector）

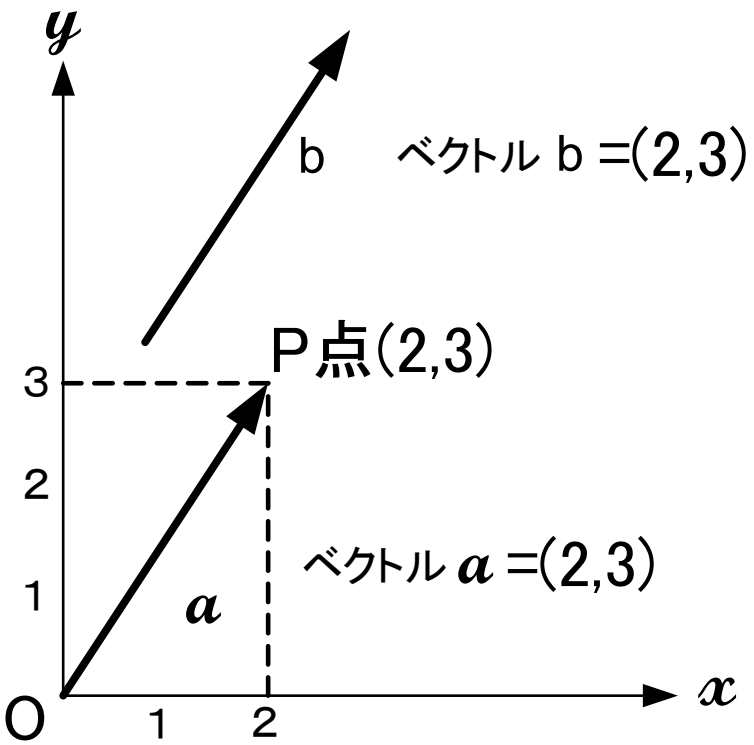
ベクトルは相対的な量です。ベクトル自体に位置があるわけではありません。これは位置ベクトルといえども同じです。というより「ＯＯベクトル」と様々に呼ばれるものは全てベクトルには違いないのですから“相対的な量”です。位置ベクトルは、“原点という絶対的な位置からの相対位置”です。ベクトル自体が相対的でも、絶対位置と組み合わせれば（絶対位置を始点とすれば)、終点は絶対的な位置を表すことができるという仕組みで、それがちょうどその系での一意な場所（なんらかの点の位置）を表すので位置ベクトルと呼ぶわけです。位置ベクトルは、ベクトルに位置があるわけではなくベクトルで位置を表している時の呼び名です。
さて、ある物の位置を座標で表すということは直感的で説明の必要はないでしょう。

図 8-25



たとえば点 P の位置はと聞かれたら X 成分が 2 で y 成分が 3 の位置、すなわち座標 (2,3) と表現するでしょう。

この「座標」と言っていたものが「位置ベクトル」です。
図 8-26



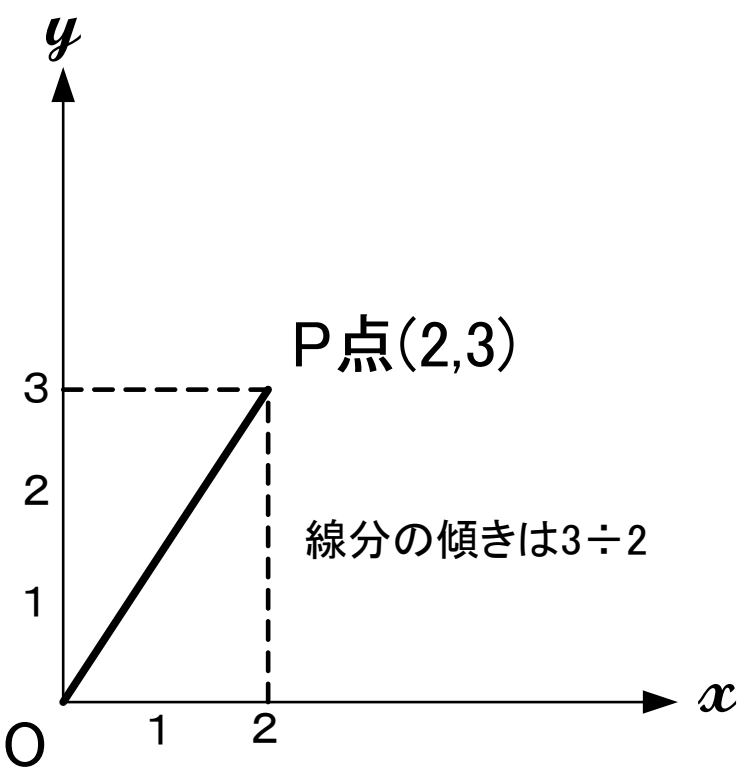
ベクトル a は点 P の位置ベクトルと呼べます。
したがって「点 P の座標は (2,3) です」と「点 P の位置ベクトル（の成分）は (2,3) です」は同じことを言っていることになります。原点を始点とするという条件を付けることによって「座標」と同義になるわけです。
成分が (2,3) であるベクトルは無数に描けますが、位置ベクトルを描いてくださいと言われたら原点から伸びる一本しか描けません。逆に言えば、無数にある同じベクトルの中で特定のベクトル（始点が原点）に着目し名前付けしたに過ぎないとも言えます。
このように始点を固定したベクトルは一般的に「束縛ベクトル（Bond Vector)」と呼ばれ、位置ベクトルは束縛ベクトルの一種であり、束縛ベクトルの中でたまたま固定始点が原点であるベクトルとも言えます。
ここで疑問を持つ読者もいるかと思います。ベクトルは向きと長さだけを表す量ということは図中(座標系上）のどこに描いても同じはず、だとしたらベクトル b も点 P の位置ベクトルなのではないかと。確かに、ベクトル a とベクトル b の成分は同じであり、同じベクトルです。それは間違いありません。この考えは実際、次に解説する方向ベクトルでは成り立つ議論ですが、ここでは残念ながら成り立ちません。なぜなら、それらは同じベクトルというだけであって同じ束縛ベクトルではないからです。位置ベクトルは束縛ベクトルの定義によって限定されるので、ベクトル a とベクトル b は束縛ベクトルの定義上では異なるベクトルとなり、ベクトル b を点 P の位置ベクトルと呼ぶことは出来ません。

位置ベクトル以外の束縛ベクトル、つまり始点を原点以外に固定した束縛ベクトルは力や流量を表現するのに都合が良く、物理学の世界で考える場合が多いのですが、当然その束縛ベクトルもまたベクトルの定義に反した特別なベクトルでもありません。束縛ベクトルといってもベクトル自体に位置情報を持たせることなど出来ないのですから。物理の力ベクトルで言えば、なんらかの基準点を始点として、あたかもベクトル自体に位置があるように考えているだけです。そこではベクトルは始点の情

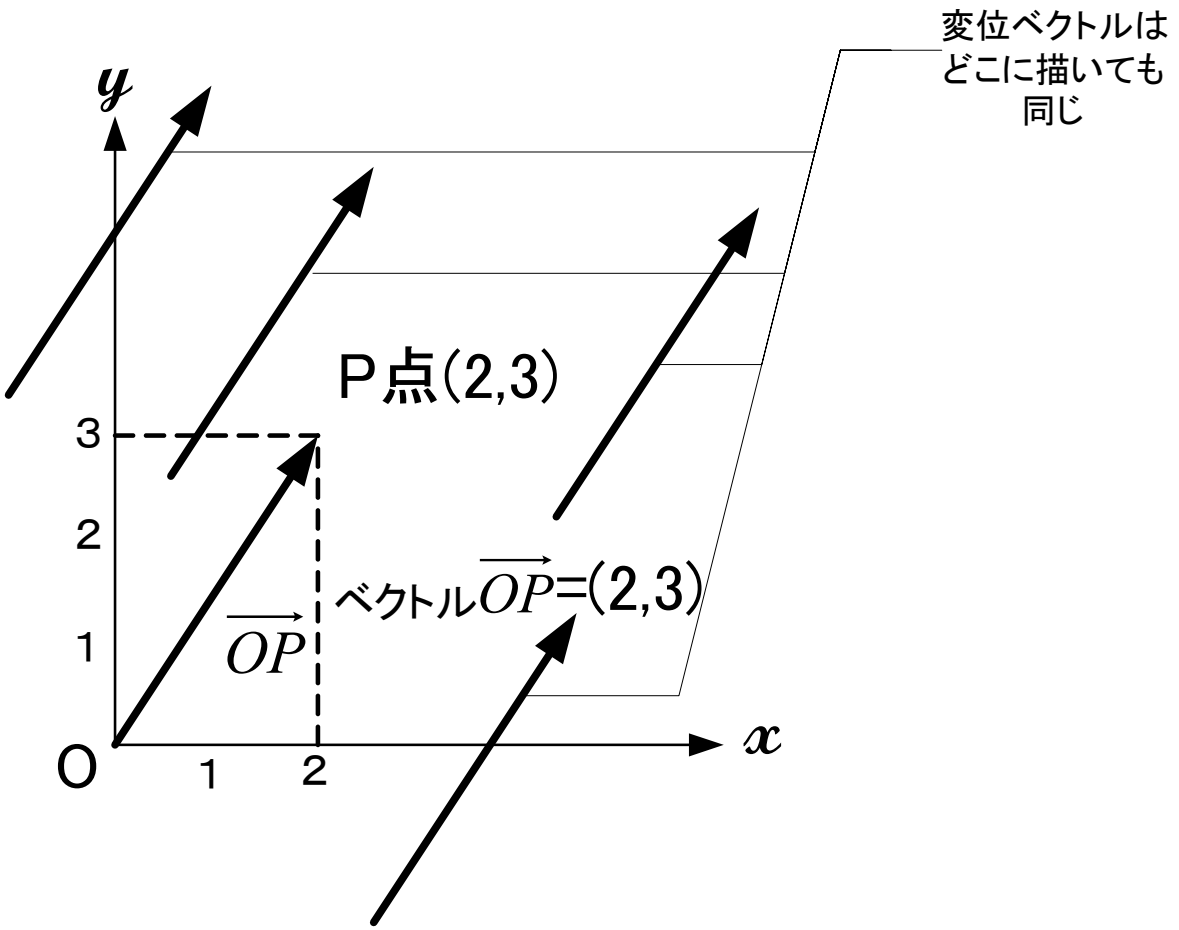
報と対に (P 点 ,a) などと表現されます。それぞれの分野で都合が良いようにベクトルという共通の概念に様々な縛り（条件）を設けたＯＯベクトルとしての定義があるわけですが、ベクトル自体の定義が異なるわけではありませんし、ベクトル自体に何か別の性質があるわけでもありません。ＯＯベクトルとしては～というようにそれぞれの定義上で考えたときに当てはまらないベクトルが出てきます。なんでもそうですが縛りをかければかけるほど範囲が限定されるのは当たり前の話です。

数学だろうが物理だろうが、位置ベクトルだろうが変位ベクトルだろうが、ベクトルの性質はただ 1 つ「向きと長さだけの相対的な量」です。
位置ベクトルに限らずＯＯベクトルは、ＯＯであるベクトルではなく、ベクトルでＯＯを表現する時、及び、何らかの条件を課した際にベクトルがＯＯである時の都合の良い呼び方であって、必ず独自の定義（条件）に縛られています。

方向ベクトル（Direction Vector）
2 次元デカルト座標系上での直線は最も馴染みのあるジオメトリでしょう。原点と点 P を端点とする線分の傾きを表現してくださいと言われれば多くの人が傾き =3 ÷ 2 と答えることができると思います。
図 8-27



この「傾き」と言っていたものが「方向ベクトル」です。
図 8-28



方向ベクトルという言葉が数学的に地位を確立した用語かどうかは別にして、3DCG という学問の中では方向ベクトルは定着した用語です。

位置ベクトルは束縛ベクトルでしたが、束縛ベクトルと区別するために「自由ベクトル（Free Vector)」という言葉があります。自由ベクトルは束縛ベクトルのような縛りが無いため、始点はどこにとっても良く、言いかえれば図形上どこに描いても良く、幾何的な場所は意味を持ちません。ベクトルと言った場合通常は自由ベクトルを指します。

方向ベクトルは自由ベクトルです。ベクトルはそもそも方向（と長さ）を表すものなのでわざわざ方向ベクトルと呼ばなくてもいいのですが（実際、方向ベクトルを単にベクトルと呼ぶ人もいます）方向パラメーターとしての機能を強調するため、および位置ベクトルと区別するためにそう呼んだほうがいいでしょう。

繰り返しになりますが、位置ベクトル・方向ベクトルと言っても、ベクトル自体は同じものです。先述の位置ベクトル (2,3) は同時に、P 点の原点からの方向ベクトル (2,3) としても機能します。例えばコード上で D3DXVECTOR2 vecX(2,3) とあった場合、それが位置ベクトルなのか方向ベクトルなのかはそれだけでは分かりません。コードを書いた人が、それを座標として使用していれば位置ベクトル、それを傾き（方向）として使用していれば方向ベクトルということになるでしょう。

ベクトルは図中のどこに描いても方向と長さが同じであれば同じベクトルになります。通常は方向ベクトルを図中に一本しか描きませんが、位置ベクトルと異なり、描こうと思えば図が真っ黒になるくらい無数に描けます。試しに何本かのベクトルを“図中での位置”を変えて書いてみて、それぞれの成分を計算してみてください。長さと向きが同じベクトル（つまり同じベクトル）であれば“図中の場所”に限らず全て同じになるはずです。言うまでも無く、図が真っ黒になっては図解する効用がな

いので一本しか描きません。

ベクトル、実際の使用例

コード上においてベクトルは単なる単精度実数変数３つ(FLOAT 型の変数が３個)から成る構造体です。位置ベクトルとして捉える局面としては、例えばゲーム内でよく “ キャラ “ と呼ぶジオメトリの位置、すなわち x,y,z の 3 成分から成る座標は位置ベクトルとして扱います。D3DXVECTOR3 というデータ型が Direct3D におけるベクトル（これは 3 次元ベクトル）なので、キャラの位置がワールド座標系(全てのジオメトリ共通の座標系)原点から (2,2,3) の位置はコード上は D3DXVECTOR3(2,2,3) となり、実際は何らかのインスタンスを作成するので、たとえば D3DXVECTOR3 vecPosition とインスタンスを作った場合 vecPosition ={2,2,3} と初期化したり、X 軸方向に移動する場合は vecPosition.x+=1 などとします。これについては問題ないでしょう。
今度は方向ベクトルとして使用する場合です。D3DXVECTOR3 vecDirection などと方向ベクトル用インスタンスを用意して、例えば上に 1 単位移動する場合、位置ベクトル vecPosition の Y 成分にプラス 1 してもいいのですが、方向ベクトルを (0,1,0) を用意してそれを位置ベクトルに加算する vecPosition+=vecDirection という方法もあります。別の例で今度は位置ベクトルの成分を直接いじくる方法に馴染まない局面を考えます。
敵キャラを主人公キャラのほうに向かって進ませたい場合がそれです。両方とも原点ではない別々の座標に居るとして、敵キャラの位置ベクトルを vecPosEnemy(2,3,0) 主人公キャラの位置ベクトルを vecPosHero(-3,1,0) であるとすれば、敵から主人公へと向かう方向ベクトルはベクトルの定理（定理というほどのものではありませんが）で A から B へ向かうベクトル X は X =B-A というのを適用し、vecDirection = vecPosEnemy – vecPosHero なので、vecDirection(5,-2,0) となります。この時点での方向ベクトルを敵の位置ベクトルに加算すると敵が一瞬で主人公にたどり着いてしまうので、方向ベクトルの成分同士の比率をそのままで値を適当に小さく正規化して、だんだんと近づいてくるようにすれば自然です。
このようにベクトル及び定理を知っていれば、複雑なコードを書かなくとも簡潔に目的を実現できます。

8-3 行列（Matrix）

行列は線形代数学の中心的な “ テクニック ” です。個人的には線形代数学は行列の学問なのではないかとも思えます。
行列はベクトルを効率的に操作するために考案されたものですが、その構造は “ ベクトルを横（行）と縦（列）に羅列した ” ものであり、行あるいは列が 1 つの場合はベクトルそのものと捉えることができるという、云わばベクトルのスーパーセットのような存在です。
プログラマー的に表現すれば、行列はベクトルになんらかの変換を施す効率的なアルゴリズムとも言えると思います。

書式

行列は縦・横に数（実数あるいは複素数）を羅列した形になります。行と列それぞれに何個の数を配置するかは任意なので、無数のパターンがあります。例えば次の 3 種類のような行列が考えられます。
図 8-29

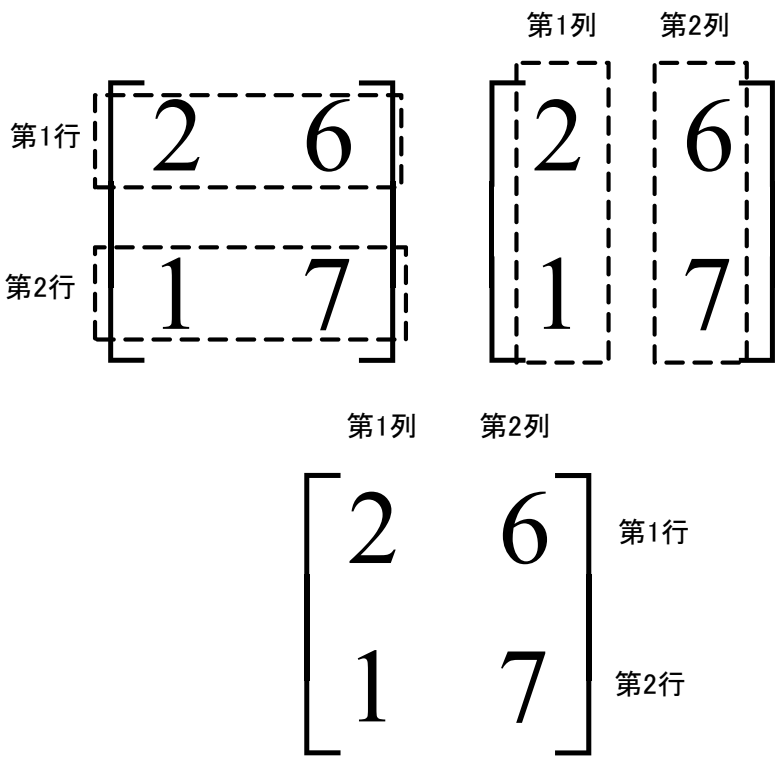
$$\begin{bmatrix} 2 & 6 \\ 1 & 7 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 2 & 0 \\ 1 & 5 & 7 \\ 0 & 7 & 6 \end{bmatrix}$$

$$\begin{pmatrix} 6 & 1 & 7 & 2 \end{pmatrix}$$

行列は、数字を縦・横に並べて括弧で囲みます。括弧に大括弧（角が直角のもの）を使うか、小括弧（角が無いもの）を使うかは任意です。今回は上と真ん中の行列に大括弧、一番下の行列に小括弧を使いました。
中身の数は、要素あるいは成分と呼びます。今回は適当に決めました。

図 8-30



数字の横のラインを行、縦のラインを列とし、行と列から成るので行列と言うわけです。
この例での行列は、「2 行 2 列の行列」または 22 行列と呼びます。一般的に m 行 n 列の行列を mn 行列と呼び、この例のように行数と列数が同じものを特に正方行列と呼ぶので、この場合は 2 次の正方行列となります。Direct3D における行列は正方行列しか使いません。

図 8-31

$$\begin{bmatrix} 2 & 6 \\ 1 & 7 \end{bmatrix}$$ 22行列

$$\begin{bmatrix} 5 & 2 & 0 \\ 1 & 5 & 7 \\ 0 & 7 & 6 \end{bmatrix}$$ 33行列

$$\begin{pmatrix} 6 & 1 & 7 & 2 \end{pmatrix}$$ 14行列、4次の行ベクトル

先の３つの行列は、それぞれ 22 行列、33 行列、14 行列と呼び、特に一番下の行列は 4 次の行ベクトルというようにベクトルとして捉えることもできます。ベクトル表記では普通、数の間にカンマを付けますが行列表記ではカンマは付けません。

演算ルール

行列の演算について、加算、減算、スカラー倍、乗算を見ていきます。当たり前ですがプログラムは行列の演算を自動的に行ってくれますし、コーディング段階においても行列の演算を手作業で行うことはまずありませんが、演算ルールは知っておかなければなりません。
まずは、加算と減算及びスカラー倍をみてみましょう。スカラーとは定数のことであり、定数と行列の掛け算をスカラー倍と言います。図を見てあきらかなように、この３つの演算は直感的で理解にはなにも問題ないでしょう。加減算では同じ場所の要素どうしを演算すればいいだけです。そしてスカラー倍は、全ての要素に一律そのスカラーを単純に掛ければいいだけです。

図 8-32

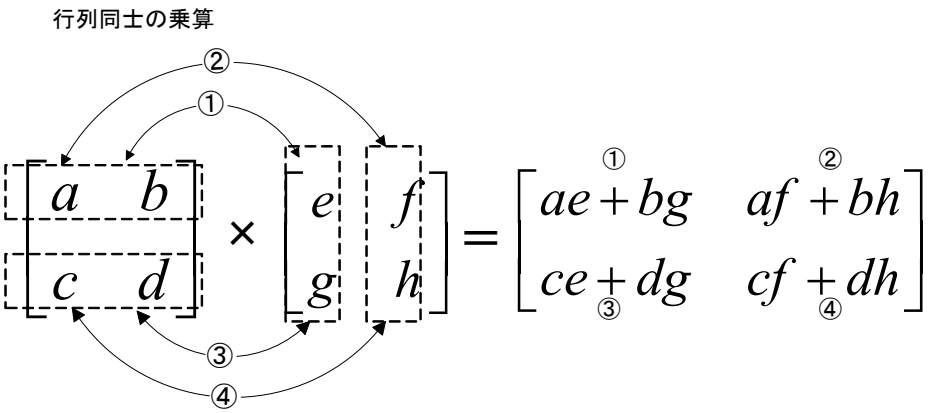
加算 例）
$$\begin{bmatrix} 2 & 6 \\ 1 & 7 \end{bmatrix} + \begin{bmatrix} 5 & 3 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 2+5 & 6+3 \\ 1+8 & 7+9 \end{bmatrix} = \begin{bmatrix} 7 & 9 \\ 9 & 16 \end{bmatrix}$$

減算 例）
$$\begin{bmatrix} 2 & 6 \\ 1 & 7 \end{bmatrix} + \begin{bmatrix} 5 & 3 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 2-5 & 6-3 \\ 1-8 & 7-9 \end{bmatrix} = \begin{bmatrix} -3 & 3 \\ -7 & -2 \end{bmatrix}$$

スカラー積 例）
$$2 \times \begin{bmatrix} 5 & 3 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 2\times5 & 2\times3 \\ 2\times8 & 2\times9 \end{bmatrix} = \begin{bmatrix} 10 & 6 \\ 16 & 18 \end{bmatrix}$$

問題は行列同士の乗算です。これは、加減算のように単純に同じ場所の要素を掛け合わせるというものではありません。要素同士は図のように演算します。

図 8-33



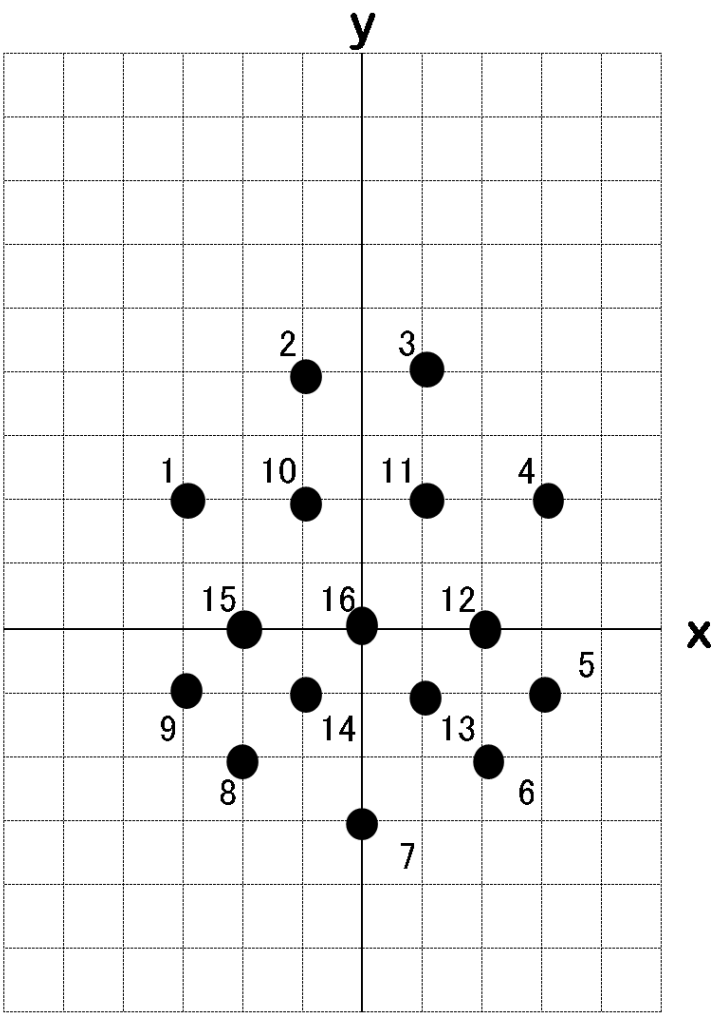
例）$\begin{bmatrix} 2 & 6 \\ 1 & 7 \end{bmatrix} \times \begin{bmatrix} 5 & 3 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 2\times5+6\times8 & 2\times3+6\times9 \\ 1\times5+7\times8 & 1\times3+7\times9 \end{bmatrix} = \begin{bmatrix} 58 & 60 \\ 61 & 66 \end{bmatrix}$

少々ややこしいですが、考え方としては“左の行列は行単位”、“右の行列は列単位“で扱うということになるでしょうか。
なぜ、こんなややこしい演算手順なのかと思うかもしれませんが、それには当然それなりの理由があるからです。具体的に言うと変換の合成はこの演算手順でなければ上手くいきません。単純に要素を掛け合わせるだけでは、それこそ単なる数字の無意味な羅列となってしまい何の役にも立ちません。行列は云わば“ツール”ですから、ベクトル操作に役立って何ぼのものです。これ以降読み進めていくうちに乗算の効果が明らかになっていくと思います。

ジオメトリの回転と拡大縮小が出来る！

3DCG での行列の演算は、行列と行列の乗算、ベクトルと行列の乗算、というように乗算が大半を占めます。
ベクトルと行列の演算はベクトルを変換するために、そして、行列と行列の乗算は変換を合成するために行います。
今、図のような 16 個の頂点から成るジオメトリがあるとします。それぞれの位置ベクトルは右側に書いています。
さて、この頂点群をぱっと見て、なんらかのイメージが浮かんだ人はいるでしょうか？

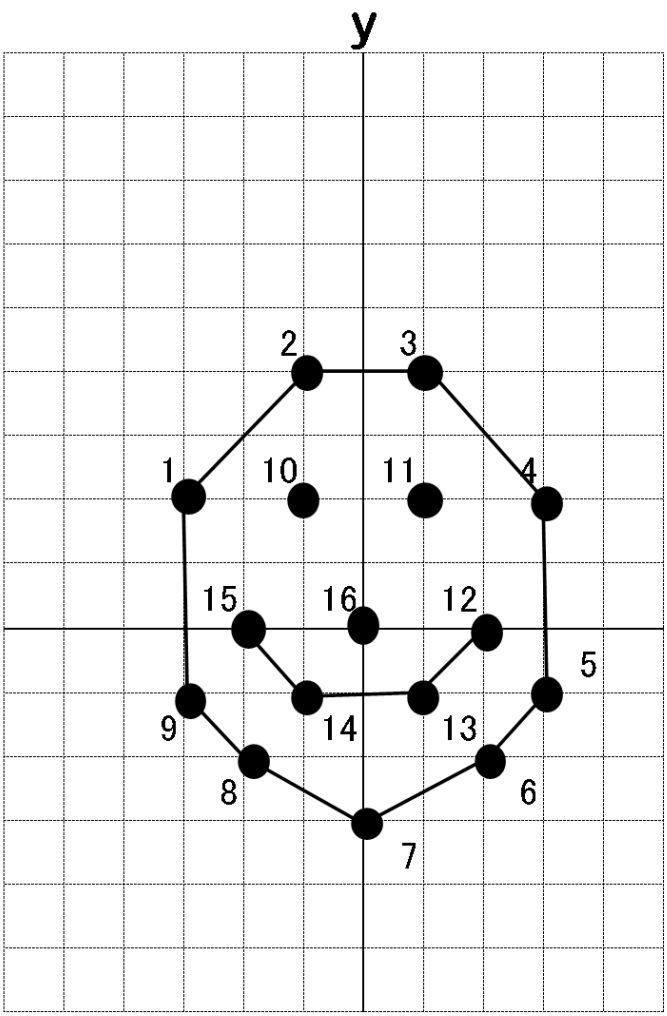
図 8-34



1（ −3， 2）
2（ −1， 4）
3（ 1， 4）
4（ 3， 2）
5（ 3， −1）
6（ 2， −2）
7（ 0， −3）
8（ −2， −2）
9（ −3， −1）
10（ −1， 2）
11（ 1， 2）
12（ 2， 0）
13（ 1， −1）
14（ −1， −1）
15（ −2， 0）
16（ 0， 0）

ジオメトリの意図をはっきりさせるため、頂点同士を線で結んだものが図です。

図 8-35



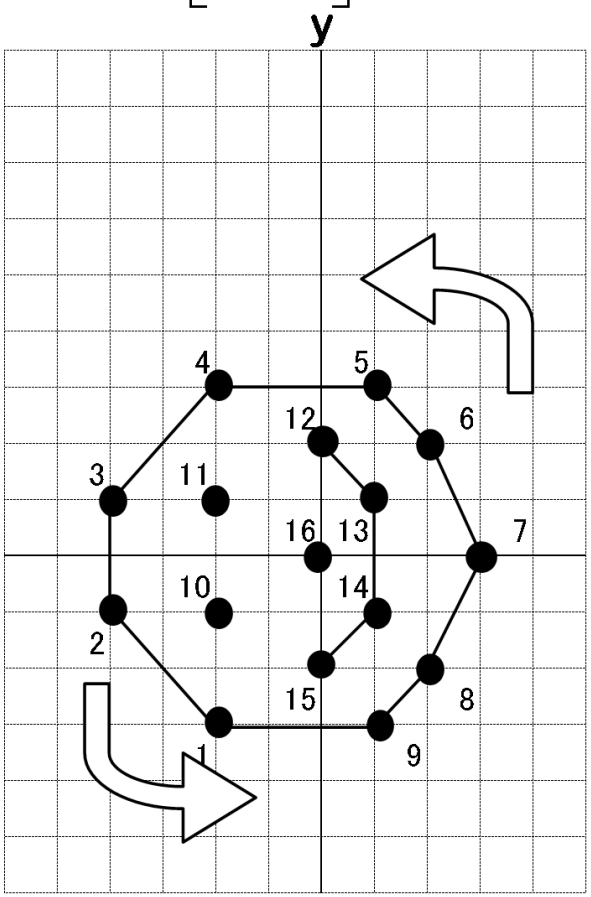
1（ −3， 2）
2（ −1， 4）
3（ 1， 4）
4（ 3， 2）
5（ 3， −1）
6（ 2， −2）
7（ 0， −3）
8（ −2， −2）
9（ −3， −1）
10（ −1， 2）
11（ 1， 2）
12（ 2， 0）
13（ 1， −1）
14（ −1， −1）
15（ −2， 0）
16（ 0， 0）

顔に見えますか？　これは 16 頂点からなる顔ジオメトリだったのです。Direct3D 的に言うと顔メッシュです。ちなみにどうでもいいことではありますが、いちおう名前は “ スマイリー山田 ” といいます。筆者が担当するゲーム学科では 18 頂点のスマイリー鈴木を使用しましたが、その従兄弟にあたります。
…さて、全頂点に次の行列を掛けて、頂点がどのように変化するか見てみましょう。
“ 掛ける “ とは行列における乗算ルールを適用するという意味です。2 次元ベクトルと 22 行列の乗算は、12 行列と 22 行列の乗算です。行列の乗算ルールはすでに示していますが、念のため示しておきます。
図 8-36

$$\begin{pmatrix} x & y \end{pmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{pmatrix} ax + cy & bx + dy \end{pmatrix}$$

図 8-37

行列 $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ を全ての頂点（位置ベクトル）に掛ける



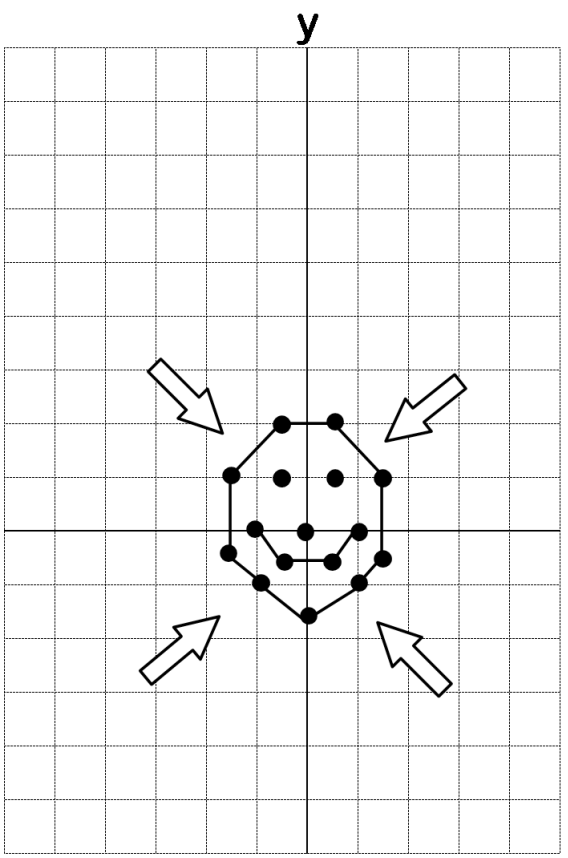
1（ −3， 2）⟶（ −2，−3）
2（ −1， 4）⟶（ −4，−1）
3（ 1， 4）⟶（ −4， 1）
4（ 3， 2）⟶（ −2， 3）
5（ 3，−1）⟶（ 1， 3）
6（ 2，−2）⟶（ 2， 2）
7（ 0，−3）⟶（ 3， 0）
8（ −2，−2）⟶（ 2，−2）
9（ −3，−1）⟶（ 1，−3）
10（ −1， 2）⟶（ −2，−1）
11（ 1， 2）⟶（ −2， 1）
12（ 2， 0）⟶（ 0， 2）
13（ 1，−1）⟶（ 1， 1）
14（ −1，−1）⟶（ 1，−1）
15（ −2， 0）⟶（ 0，−2）
16（ 0， 0）⟶（ 0， 0）

行列 $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ を掛けて、それぞれの頂点は変化します。変化後の頂点をプロットして同じように線で結ぶと、なんと綺麗に “ 反時計回りに 90 度回転 “ しました。もうお分かりのように、この行列は ” 回転行列 “ です。

同じように、今度は $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ という行列を全頂点に掛けると、それぞれ次のように変化し、結果的に顔ジオメトリは原点を中心に縮小します。この行列は縮小行列です。

図 8-38

行列 $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ を全ての頂点（位置ベクトル）に掛ける



1 (−3 , 2) ⟶ (−1.5 , 1.0)
2 (−1 , 4) ⟶ (−0.5 , 2.0)
3 (1 , 4) ⟶ (0.5 , 2.0)
4 (3 , 2) ⟶ (1.5 , 1.0)
5 (3 , −1) ⟶ (1.5 , −0.5)
6 (2 , −2) ⟶ (1.0 , −1.0)
7 (0 , −3) ⟶ (0.0 , −1.5)
8 (−2 , −2) ⟶ (−1.0 , −1.0)
9 (−3 , −1) ⟶ (−1.5 , −0.5)
10 (−1 , 2) ⟶ (−0.5 , 1.0)
11 (1 , 2) ⟶ (0.5 , 1.0)
12 (2 , 0) ⟶ (1.0 , 0.0)
13 (1 , −1) ⟶ (0.5 , −0.5)
14 (−1 , −1) ⟶ (−0.5 , −0.5)
15 (−2 , 0) ⟶ (−1.0 , 0.0)
16 (0 , 0) ⟶ (0.0 , 0.0)

どうでしょう、面白くはありませんか？

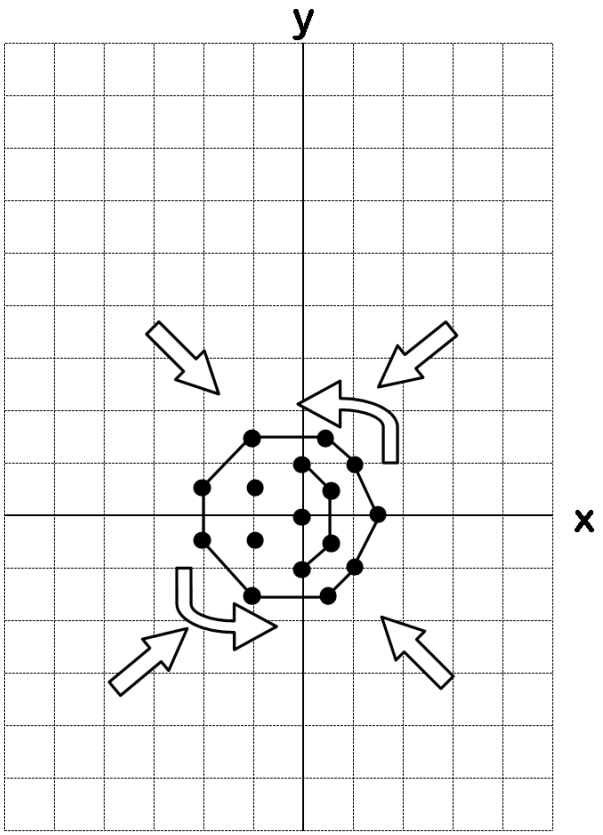
変換を合成出来る！

行列で回転と縮小という変換をそれぞれ別個に行いました。では今度は、それらを一回の変換で行うことを考えてみます。行列の乗算の性質により、これは簡単に実現できます。変換行列同士を掛け合わせた行列を頂点に掛ければ良いのです。

図 8-39

$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 0 & 0.5 \\ -0.5 & 0 \end{bmatrix}$ 回転行列とスケーリング行列を乗算し、

その行列$\begin{bmatrix} 0 & 0.5 \\ -0.5 & 0 \end{bmatrix}$を全ての頂点（位置ベクトル）に掛ける



1	(	−3	,	2	) ⟶ (	−1.0 , −1.5)
2	(	−1	,	4	) ⟶ (	−2.0 , −0.5)
3	(	1	,	4	) ⟶ (	−2.0 , 0.5)
4	(	3	,	2	) ⟶ (	−1.0 , 1.5)
5	(	3	,	−1	) ⟶ (	0.5 , 1.5)
6	(	2	,	−2	) ⟶ (	1.0 , 1.0)
7	(	0	,	−3	) ⟶ (	1.5 , 0.0)
8	(	−2	,	−2	) ⟶ (	1.0 , −1.0)
9	(	−3	,	−1	) ⟶ (	0.5 , −1.5)
10	(	−1	,	2	) ⟶ (	−1.0 , −0.5)
11	(	1	,	2	) ⟶ (	−1.0 , 0.5)
12	(	2	,	0	) ⟶ (	0.0 , 1.0)
13	(	1	,	−1	) ⟶ (	0.5 , 0.5)
14	(	−1	,	−1	) ⟶ (	0.5 , −0.5)
15	(	−2	,	0	) ⟶ (	0.0 , −1.0)
16	(	0	,	0	) ⟶ (	0.0 , 0.0)

まず、回転行列と縮小行列を乗算します。すると、新たに$\begin{bmatrix} 0 & 0.5 \\ -0.5 & 0 \end{bmatrix}$なる行列が出来上がります。

その行列を全頂点に掛けた結果、顔ジオメトリは、縮小して尚且つ回転もします。２つの変換が一度に行えました。この行列は縮小と回転の両方の性質を持つ変換行列だったのです。
たまたま 90 度回転を例にとりましたが、任意の角度θで回転させたい場合の一般式は次のようになります。

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

今回の回転行列は回転角度が 90 度だったので、sin90 で 1、-sin90 で -1、cos90 で 0 となったわけです。

スケーリングも縮小のみを例にとりましたが、任意の比率で拡大縮小させる場合の一般式は次のようになります。

$$\begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix}$$

今回のスケーリング行列は Sx、Sy に 0.5 なので半分の縮尺に縮みましたが、Sx と Sy の部分を 2 としていたら 2 倍の縮尺に拡大したことになります。また、Sx と Sy が同じ値だったので縦横均等にスケーリングしましたが、たとえば Sx を 2 に Sy を 1 などとすると、横長に伸びることとなります。

同次変換なら平行移動も出来る！

回転とスケーリングの時のような 22 行列では平行移動は出来ません。なぜなら、16 番目の頂点は (0,0) ですから、何を掛けても (0,0) です。16 番目の頂点は、まるで釘で固定したかのようにビクとも動かないのです。ベクトルの加算であればもちろん平行移動できますが、2 次ベクトルと 22 行列は加算できません。

ベクトルと行列の乗算で平行移動させるためには、同次座標空間内で考えると上手くいきます。同次座標（homogeneous coordinates ホモジーニアス・コーオーディネイト）は斉次（せいじ）座標あるいは射影座標とも呼ばれ、一言で言えば、元の次元よりも次元の数が多い座標系であるものの、座標が元の系と 1 体 1 に対応しているのでいつでも元の次元に戻して考えられるような座標系のことです。ここでの場合は要するに、2 次元より 1 つ次元の多い 3 次元同次座標に一旦置き換えて、3 次元上でベクトルに行列を掛け、また元の 2 次元に戻すということです。2 次元と 3 次元を 1 体 1 に対応させるのですから、当然のごとく通常の 3 次元ではありません。普通の 3 次元上には任意の 2 次元座標は無限に存在してしまいます。3 次元同次座標は名前上は “3 次元 ” と付いていますが､限定的な 3 次元、もっと分かり易く表現すると、3 次元空間上の 2 次元平面座標系ということになります。3 次元同時座標では点（座標）は、ある特定の平面上でしか自由度が無いことになります。

言葉だけではピンと来ないかもしれません。実際に計算してみましょう。

顔ジオメトリの全頂点を同次座標にします。次元を 1 つ追加し、追加した成分は全て 1 にします。

掛ける行列側も同次行列にします。33 行列にして対角上の成分を全て 1 にします。そして平行移動行列は追加した次元の成分が移動量になります。この場合は 3 行目のラインが移動量となります。

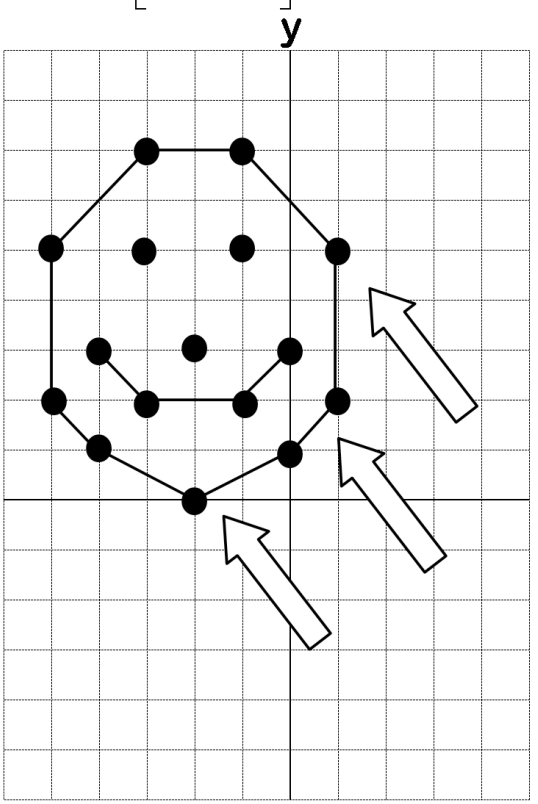
3 次ベクトルと 33 行列は言い換えれば 13 行列と 33 行列の乗算であり、乗算の演算規則は最初に示した 22 行列同士の乗算で全てを表していますが、念のため記しておきます。

図 8-40

$$\begin{pmatrix} x & y & w \end{pmatrix} \times \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{pmatrix} ax + dy + gw & bx + ey + hw & cx + fy + iw \end{pmatrix}$$

図 8-41

同次行列$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 3 & 1 \end{bmatrix}$を全ての頂点（同次ベクトル）に掛ける



1（ -3， 2， 1）→（ -5， 5， 1）
2（ -1， 4， 1）→（ -3， 7， 1）
3（ 1， 4， 1）→（ -1， 7， 1）
4（ 3， 2 1）→（ 1， 5， 1）
5（ 3，-1， 1）→（ 1， 2， 1）
6（ 2，-2， 1）→（ 0， 1， 1）
7（ 0，-3， 1）→（ -2， 0， 1）
8（ -2，-2， 1）→（ -4， 1， 1）
9（ -3，-1， 1）→（ -5， 2， 1）
10（ -1， 2， 1）→（ -3， 5， 1）
11（ 1， 2， 1）→（ -1， 5， 1）
12（ 2， 0， 1）→（ 0， 3， 1）
13（ 1，-1， 1）→（ -1， 2， 1）
14（ -1，-1， 1）→（ -3， 2， 1）
15（ -2， 0， 1）→（ -4， 3， 1）
16（ 0， 0， 1）→（ -2， 3， 1）

この成分は無視すれば良い

ビクともしないはずの 16 番頂点が動きました。これは、同次座標系上で 2 次元平面が第 3 成分の分だけ（1 だけ）浮いた形になり、16 番目の頂点がもはや原点ではなくなるからです。変な例えかもしれませんが、3 次元同次座標は 2 次元平面のガンコな汚れを浮かす界面活性剤のようなものと考えても面白いかもしれません。
これが同次座標の旨みです。増やした部分の成分は無視すればいいだけのことです。
今回は 2 次元に対する 3 次元同次座標を考えましたが、増やす次元はいくらでも構いません。例えば、2 次元に対する 4 次元の同次座標も考えられますし、100 次元の同時座標すら考えられます。
なお、同時座標系の作り方にはルールがあります。増やした成分はゼロにして、最後の成分のみ 1 にします。今回の場合は増やした成分が最後の次元でもあったので 1 だけが出てきてゼロは出てきませんでした。
2 次元→ 4 次元同次座標　(a,b) →　(a,b,0,1)
2 次元→ 6 次元同次座標　(a,b) →　(a,b,0,0,0,1)
2 次元→ n 次元同次座標　(a,b) →　(a,b, 0, …,0,1)　n 次元目（最後の成分）は 1。それ以外の追加成分はゼロ。

Direct3D は全て 44 行列（同次行列）

Direct3D において、ベクトルは 2 次元ベクトル、3 次元ベクトル、4 次元ベクトルの 3 種類あります。それに対し行列の型は 1 種類しかありません。それは 44 行列です。
なぜ行列が 1 種類で、しかも 44 行列なのでしょう？

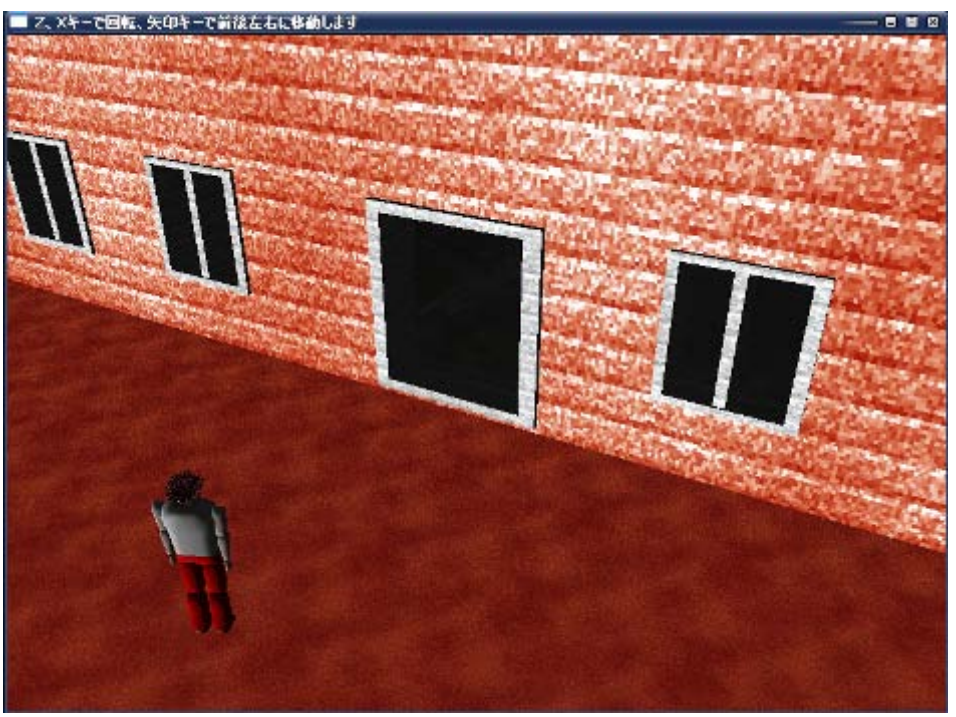
それは、次元の異なる 3 種類のベクトルの変換は同次座標として考えれば最も大きい 4 次元に合わせた 44 行列 1 種類でこと足りますし、むしろ 1 種類のほうが都合が良いという積極的な理由もあるからです。なぜかというと、ベクトルの型に合わせて行列の次元を使い分ける必要が無く、画一的な処理ができますし、なにより、変換の合成の際、異なる型の行列同士を掛け合わせることができなくなります。回転行列やスケーリング行列と平行移動行列を合成できないのでは、実務上使い物になりません。

9 章　自転と公転

9-1 向いている方向に進ませる

9-1-1 まずは 3 人称視点

図 9-1　人間メッシュとビルディングメッシュ､地面メッシュ（と空（そら）メッシュも隠れています）



まずは、3 人称視点（TPV：Third Person View）バージョンのサンプルから見ていきましょう。
キャラクター（メッシュ）の位置ベクトル用として D3DXVECTOR3 ＯＯＯなどと用意することが多いと思います。キャラを前後左右に移動させたい場合、その成分を増減させた位置ベクトルから移動行列を作成して、それをワールドトランスフォーム行列に掛ければ移動してくれます。
ただ、それだけでは単純に前後左右に移動するだけです。ゲームではキャラが現在向いている方向に進ませたい場合がありますが、それだけではキャラが横や斜めを向いているにも関わらず、無条件に前方に移動してしまい不自然になってしまいます。これはキャラがワールド軸上でしか移動できないため、絶対的な前後左右で動いてしまうためです。
キャラが横を向いているときの前進は、ワールド系から見た時には横になり、斜めに向いている時の前進はワールド系での斜め方向に移動させたほうが自然な時もあります。それは、そのキャラ独自の軸、すなわちローカル軸を持たせてやれば簡単に実現できます。
なお、ゲームのフィールド（舞台）が宇宙空間のような場合は、3 軸を持たせることになり複雑になりますが、ここではヨー回転だけなので、この手のサンプルとしてはもっとも簡単でしょう。（回転

に関する詳細な解説は 10 章で行っています）

サンプルプログラム
プロジェクトフォルダ名「ch09-1-1 向いている方向に進ませる（TPV）」
VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

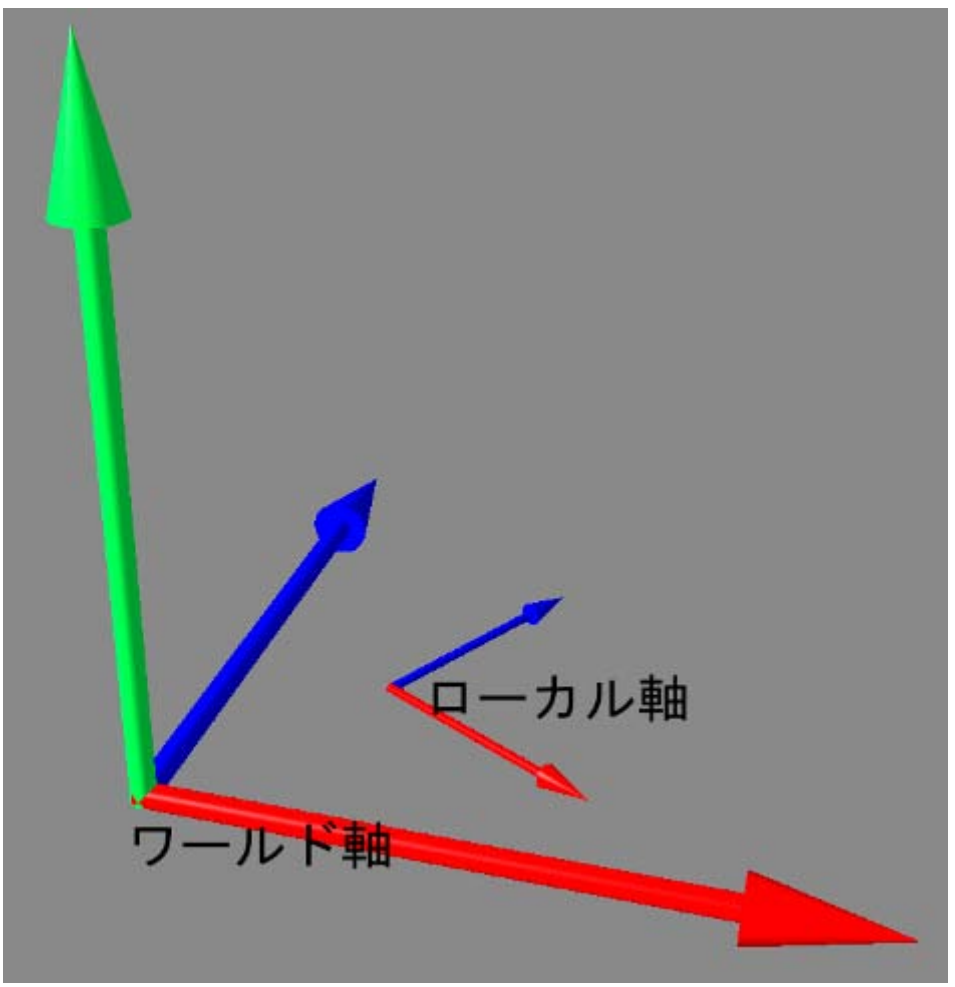
なにも処理しないで、普通に位置ベクトルの成分そのままでワールド行列を作成してしまうと、キャラの向きに関係なく、絶対的な X 方向、Z 方向に進んでしまいます。このサンプルにおいてのキャラは、前進のキーを押せば顔が向いている方向に進み、後退のキーを押せば背中方向に進みます。また、左キーを押せば左手方向、右キーで右手方法に進みます。したがって、顔が真横に向いている時に前進すると “ 画面上は横に ” 進むことになります。
使用方法
平行移動：矢印キー。
回転：Z キーと X キー。
ESC キーで終了。

コード解説
さて、このサンプルのコード上でのキーポイント部分は、VOID StepMove 関数です。コードを追って解説していきます。
このサンプル及び、その他ほとんどの 3D サンプルでは THING という構造体を定義し、そこにメッシュや姿勢パラメーターを格納するようにしています。
StepMove 関数は、引数に THING のポインターを受け取ります。

D3DXMATRIX matPosition;
THING 構造体には、位置行列が無いので、関数内で用意しています。

D3DXMatrixIdentity(&pThing->matScaling);
D3DXMatrixIdentity(&pThing->matRotation);
D3DXMatrixIdentity(&pThing->matWorld);
THING のスケーリング行列、回転行列、ワールド変換行列を単位行列に初期化します。
このサンプルでは、関数内で毎回、各変換行列をそれぞれのパラメーターにより作成する形態なので、関数が呼ばれる度に最初にまず初期化します。逆に、行列の成分に直前の値が残っていると回転しっぱなしなんてことになるので、それを防ぐ意味もあります。
D3DXMatrixScaling(&Thing[2].matScaling,pThing->fScale,pThing->fScale,pThing->fScale);
スケーリング行列を THING の fScale パラメータにとり作成します。当初このサンプルにはスケーリング機能を持たせていなかったのですが、Building メッシュ（建物）が大きすぎたのでスケーリング機能を持たせ、それに伴って THING 構造体に fScale メンバを追加することになりました。
D3DXMatrixRotationY(&pThing->matRotation,pThing->fYaw);
ヨー回転（Y 軸周りの回転）のみの回転行列を fYaw メンバにより作成します。fYaw は、ウィンドウプロシージャー内でキー入力により更新されている変数であり、現在の回転量が格納されています。
 D3DXVECTOR3 vecAxisX(1.0f,0.0f,0.0f);
 D3DXVECTOR3 vecAxisZ(0.0f,0.0f,1.0f);
D3DXVec3TransformCoord(&vecAxisX,&vecAxisX,&pThing->matRotation);
D3DXVec3TransformCoord(&vecAxisZ,&vecAxisZ,&pThing->matRotation);

この 4 行がミソの 1 つです。
まず、キャラ固有の軸ベクトルを 2 つ用意します。今回は X 軸と Z 軸です。
X 軸は素直に正規直行座標系での基底 X 軸とするので成分は（1,0,0)、同様に Z 軸は (0,0,1) とします。
この 2 軸をキャラ固有の軸ベクトル、言い換えればキャラのローカル軸として機能させることになります。
2 軸を共に現在の回転行列で回転させます。この段階で、2 つの軸ベクトルはワールド軸と異なる方向に導くローカル軸として機能します。
図 9-2




```
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;
```

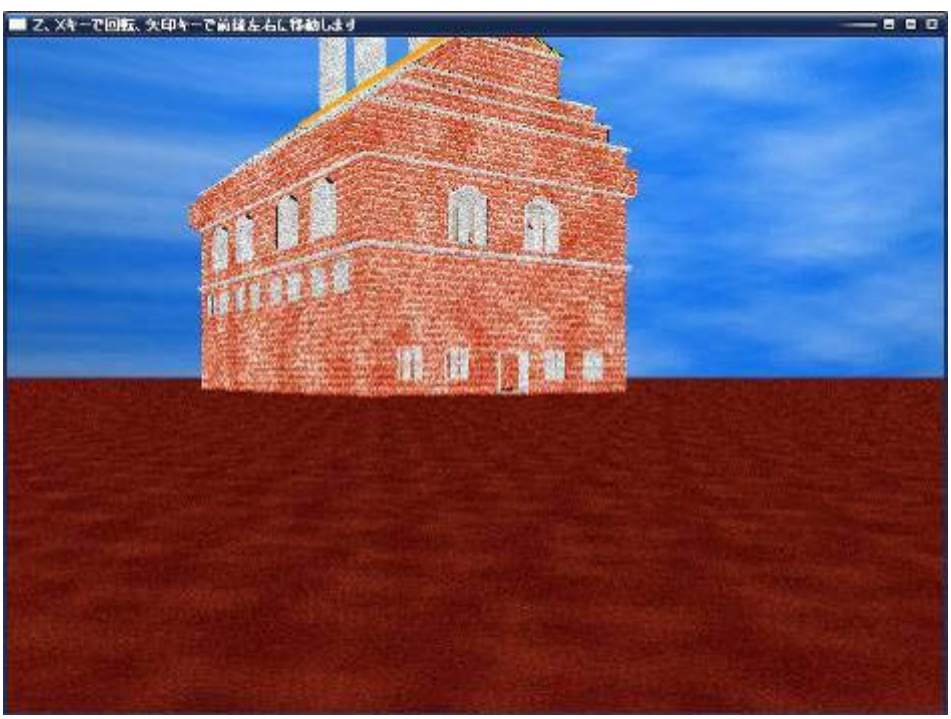
```
        case BACKWARD:
                pThing->vecPosition-=vecAxisZ*0.1f;
        break;
}
```
あとは簡単です。キー入力の方向により、そのローカル X 軸あるいはローカル Z 軸上を移動させるだけです。

```
 pThing->Dir=STOP;
```
Dir メンバは、キー入力がどの方向なのかを格納する列挙型で、この段階で役割は終了したので、最後にニュートラルな状態にしておきます。

```
D3DXMatrixTranslation(&matPosition,pThing->vecPosition.x,pThing->vecPosition.y,pThing->vecPosition.z); D3DXMatrixMultiply(&pThing->matScaling,&pThing->matScaling,&pThing->matRotation);
D3DXMatrixMultiply(&pThing->matWorld,&pThing->matScaling,&matPosition);
```
スケーリング、回転、位置の 3 つの変換行列を合成し、ワールド行列に格納して完了です。

9-1-2 1 人称視点にするには？

図 9-3　1 人称視点になって、隠れていた空（そら）メッシュが見えます。



サンプルプログラム
プロジェクトフォルダ名「ch09-1-2 向いている方向に進ませる（FPV）」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

コードにちょこっと手を加えれば、すぐさま 1 人称視点（FPV：First Person View）にすることができます。FPV を FPS と言う人がいますが、あまりお勧めできません。FPS のフルネームは First

Person Shooter です。名前から明らかなように、シューティングゲームにしか当てはまりません。なにより、フレームレート（FPS：Frame Per Second）と混同しやすく紛らわしい用法です。
さて、ここで大切なのは、変更を丸暗記的に憶えるのではなく、なぜ簡単に 1 人称視点にできるのかその仕組みを理解することです。
使用方法
キー操作は、TPV バージョンと同一です。変更したのは、ビュー変換部分の数行なのですから。

コード解説
結論（コード変更箇所）から先に示すことにしましょう。
1 行の書き換えと 2 行の追加、計 3 行の変更だけで 1 人称視点への変更完了です。
次のビュートランスフォーム部分の、
D3DXVECTOR3 vecEyePt(5.0f, 6.0f,-8.0f); // カメラ（視点）位置
D3DXVECTOR3 vecLookatPt(0.0f, 0.0f, 1.0f);// 注視位置

vecEyePt を書き換えて、2 行を追加するだけです。
D3DXVECTOR3 vecEyePt(Thing[3].vecPosition.x,Thing[3].vecPosition.y,Thing[3].vecPosition.z);// カメラ（視点）位置
D3DXVECTOR3 vecLookatPt(0.0f, 0.0f, 1.0f);// 注視位置
D3DXVec3TransformCoord(&vecLookatPt,&vecLookatPt,&Thing[3].matRotation);
vecLookatPt+=Thing[3].vecPosition;

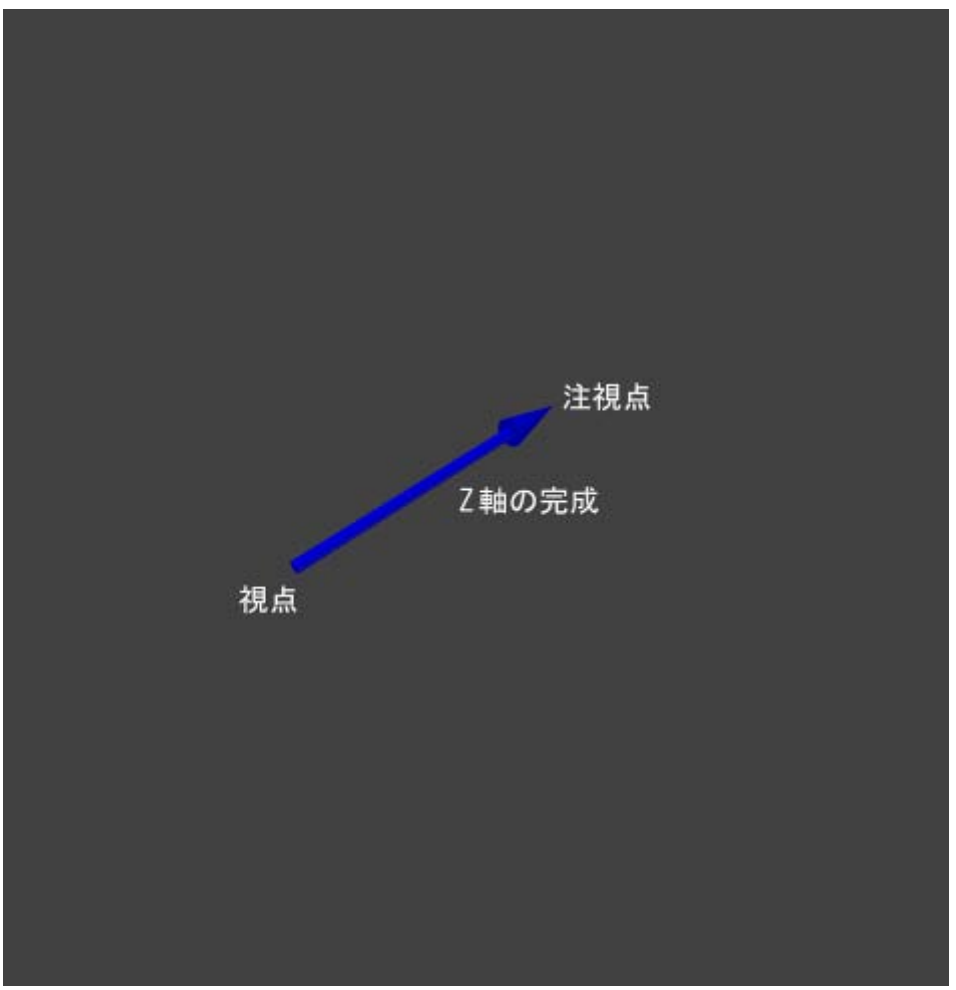
ビルドして実行してみてください。見事に 1 人称視点になっています。

仕組み
ではどのような原理で 1 人称視点になったのか、その仕組みを解説します。
仕組みを一言で言うなら、カメラ位置をキャラにシンクロさせれば、それだけで 1 人称視点を実現できるということになりますが、詳しく見ていきましょう。
さて、ビュー行列は大抵の場合、D3DXMatrixLookAt 関数で作成します。この関数は 3 つのベクトルからビュー行列を作成します。
① カメラ位置ベクトル
② 注視位置ベクトル
③ 上方ベクトル
本題に入る前に、3 つのベクトルからどのようにビュー行列を作成しているのか、その作成プロセスから明らかにしていきます。
次の擬似コードは SDK のヘルプドキュメント内からの抜粋で、これは D3DXMatrixLookAt 関数の内部アルゴリズム、つまりビュー行列作成原理を示しています。
zaxis = normal(At - Eye)
xaxis = normal(cross(Up, zaxis))
yaxis = cross(zaxis, xaxis)

xaxis.x	yaxis.x	zaxis.x	0
xaxis.y	yaxis.y	zaxis.y	0
xaxis.z	yaxis.z	zaxis.z	0
-dot(xaxis, eye)	-dot(yaxis, eye)	-dot(zaxis, eye)	1

Eye：カメラ位置ベクトル、At：注視位置ベクトル、Up：上方ベクトルの３つのベクトルを受け取り、xaxis、yaxis、zaxis の３つの軸ベクトルを内部で作成していることが分かります。
また、normal、cross、dot が次の機能を持つ関数であることは、アルゴリズムから明らかです。
normal() ベクトルの正規化関数
cross() ベクトルの外積関数
dot() ベクトルの内積関数

処理の流れは次のとおりです。
zaxis = normal(At - Eye)
まず、注視位置ベクトルとカメラ位置ベクトルの差（変位）ベクトルを求め、それを正規化（長さを 1 にする）します。要するにカメラの向きです。これをカメラ座標での z 軸と決めます。
図 9-4



xaxis = normal(cross(Up, zaxis))
求まった z 軸と上方ベクトルの外積ベクトルを x 軸とします。外積とは、２つのベクトルに垂直なベクトルを求めること、言い換えると２つのベクトルが作る平面に垂直なベクトルを作ることでした。上方ベクトルをとりあえずの y 軸と想定して、z 軸と仮の y 軸が作る zy 平面に垂直なベクトルが x 軸というわけです。

図 9-5

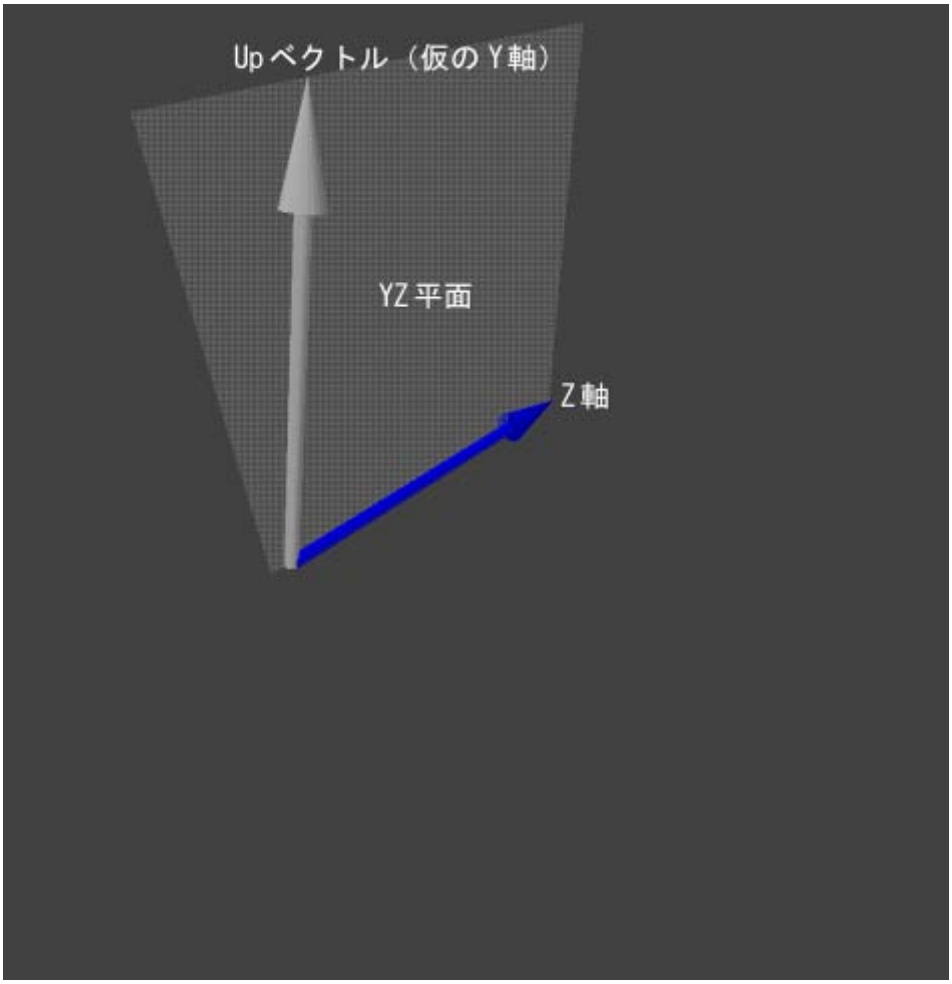
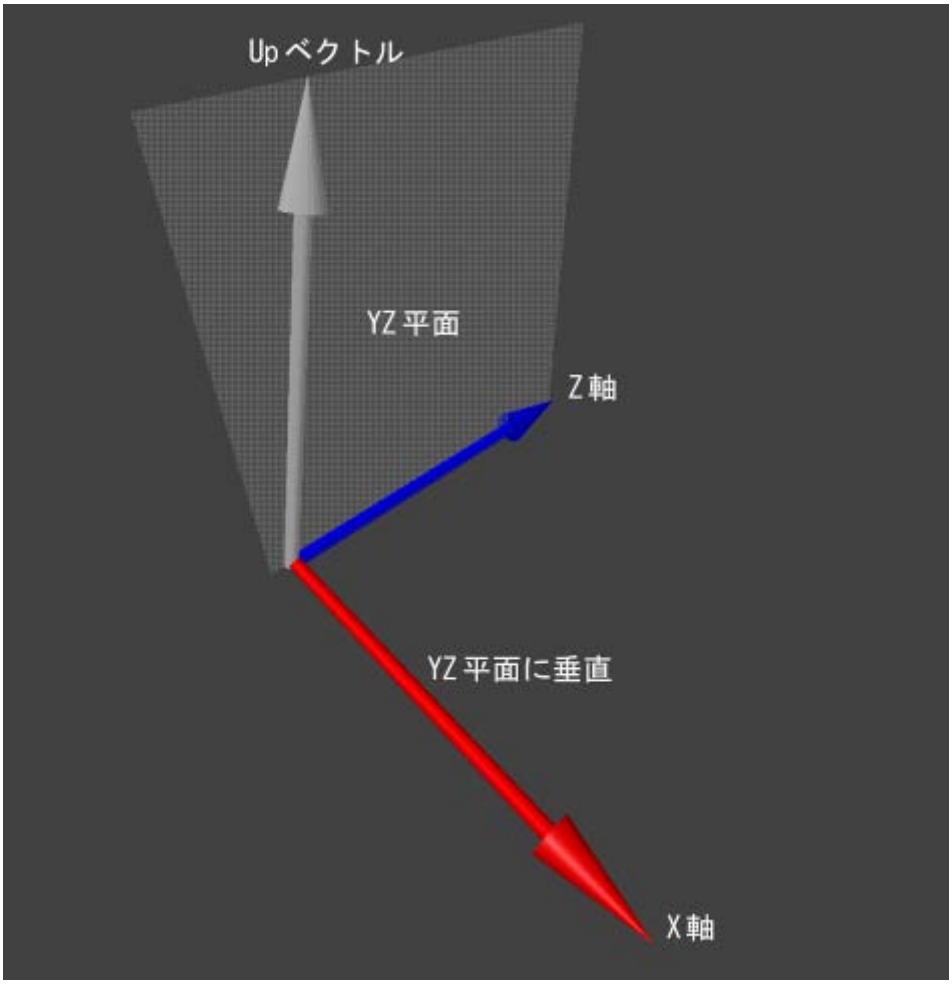
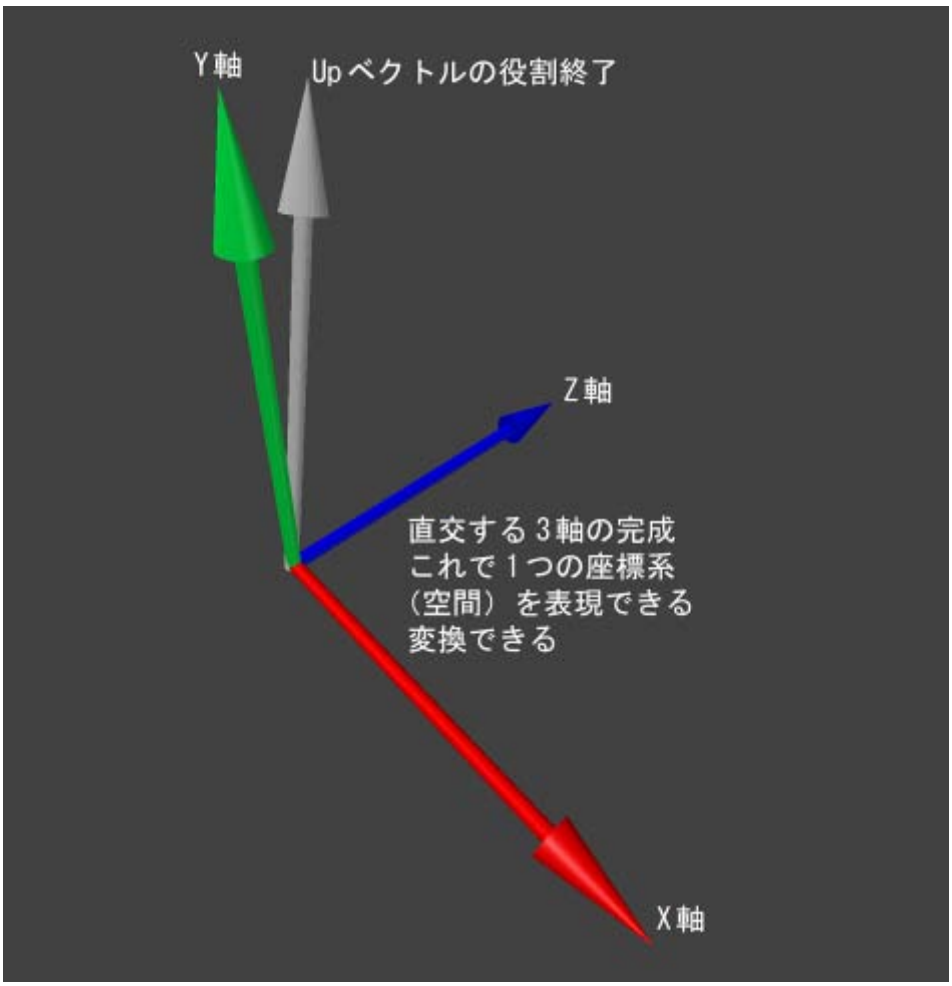


図 9-6



yaxis = cross(zaxis, xaxis)
z 軸、x 軸が求まれば、残りの y 軸はそれらの積で求まります。ここで、z 軸と x 軸両方に直交した y 軸が求まります。

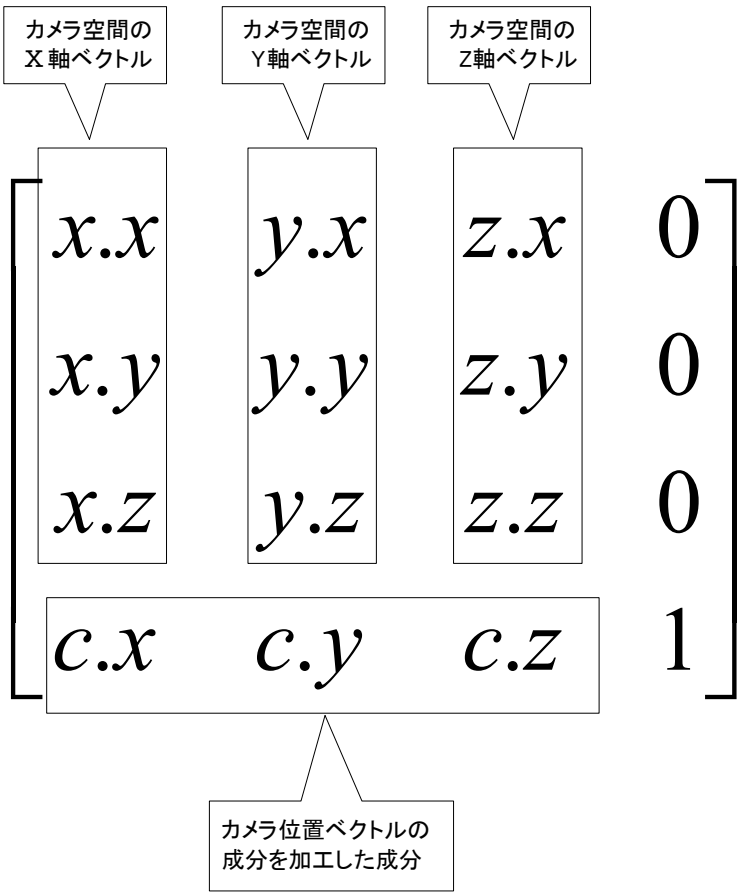
図 9-7



ここまでで、カメラ座標空間を作る基底軸３本が出来ました。
あとは、この３つの基底軸を基に行列の成分を埋めていきます。
行列の成分は、３つの軸ベクトルを配置していることがわかります。軸ベクトルを並べた行列は座標系の変換行列となります。この場合はワールド座標からカメラ座標への変換になります。なお、３軸を縦に配置しているのは、４次の行ベクトル× 44 行列を想定しているからであり、演算順序やベクトルが列ベクトルであれば異なる配置となるので、これが絶対的な配置とは考えないでください。本質は、軸ベクトルを並べた行列が、座標系間の変換行列となることです。
同次行列の一番下のラインは平行移動成分であるということを思い出してください。平行移動成分には、カメラ位置ベクトルと軸ベクトルの内積をとって、軸に射影したあとの成分を入れています。ビュー行列は全ジオメトリに掛けられますから、カメラが動いたときに、シーン全体が動くのはこのラインの効果です。内積により軸へ射影した後の成分を使用しいているということは結果的にはカメラ座標に変換する際に対象とするベクトルを回転させることを意味します。そして、カメラの動きとシーン（全ジオメトリの見かけ）の動きは真逆になるのでマイナスを付けているわけです。

図 9-8

ビュー行列



ちなみに、この手法がビュー行列に限らず任意の直交座標系を作成する際に、そのまま利用できることに気が付いたでしょうか？ D3DXMatrixLookAt 関数はビュー行列作成という目的以外にも重宝しそうです。というよりも、ビュー変換が、ある直交座標系（ワールド）から別の直交座標系（カメラ空間）への座標変換の 1 つに過ぎないといったほうが的を射ているかもしれません。

そろそろ本題に戻り具体的な仕組みを述べます。カメラ位置ベクトル vecEyePt にキャラの位置ベクトル Thing[3].vecPosition をそのまま代入してやれば、カメラ位置はキャラ位置そのものとなり、キャラに追従するようになります。これでカメラの “ 位置 ” の設定は終了です。

“ 位置 ” だけでは不十分なので、次にやることはカメラの “ 向き ” の設定です。注視位置ベクトル vecLookAtPt をとりあえず直感的な前方向ベクトルとして成分を決定します。素直に Z 軸でいいでしょう。別に Z 軸でなくても X 軸でもいいですし、基底ベクトルでなくてもいいのですが、わざわざややこしいベクトルにすることはありません。それをキャラの現在の回転量で回転させます。さらにキャラの位置ベクトルを加算します。“ 回転させてから “ 位置ベクトルを足します。位置ベクトルを足してから回転させると妙な方向を向いてしまうので順番には気を付けてください。

```
D3DXVec3TransformCoord(&vecLookatPt,&vecLookatPt,&Thing[3].matRotation);
```

以上あっという間に終了です。

ちなみに、カメラが常にキャラクターの背中を映すようにしたい場合は、次のようにします。

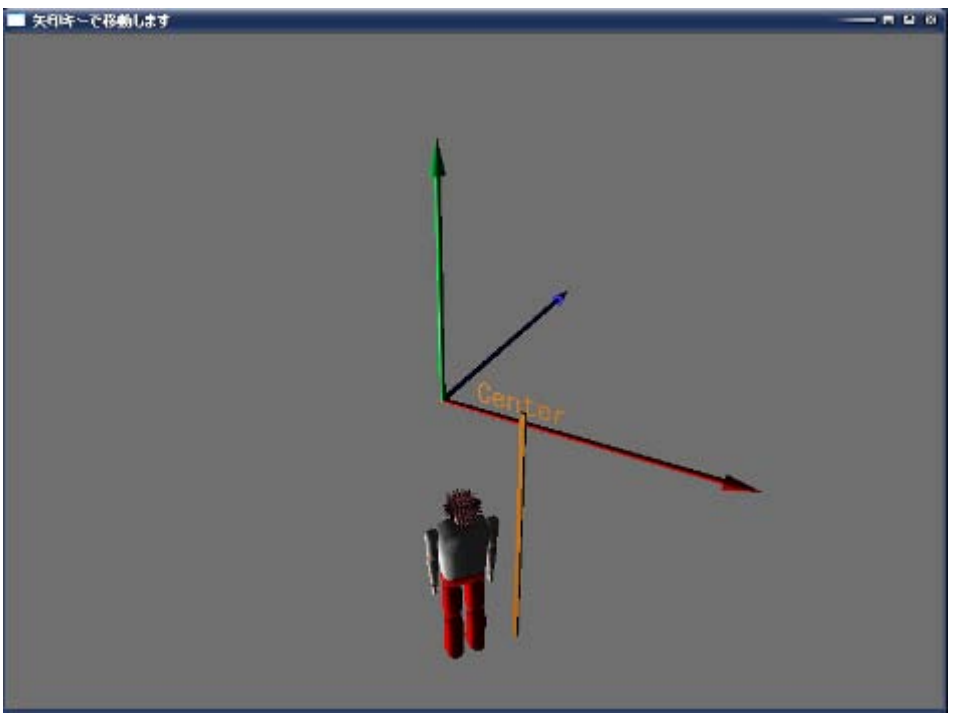
```
D3DXVECTOR3 vecEyePt(0,0,-2 );
D3DXVECTOR3 vecLookatPt(0,0,0 );
```

```
D3DXVec3TransformCoord(&vecEyePt,&vecEyePt,&Thing[3].matRotation);
D3DXVec3TransformCoord(&vecLookatPt,&vecLookatPt,&Thing[3].matRotation);
vecEyePt+=Thing[3].vecPosition;
vecLookatPt+=Thing[3].vecPosition;
```

念のため、この変更を加えたものを「ch09-1-2-b 背中バージョン」というプロジェクトとして収録しておきました。

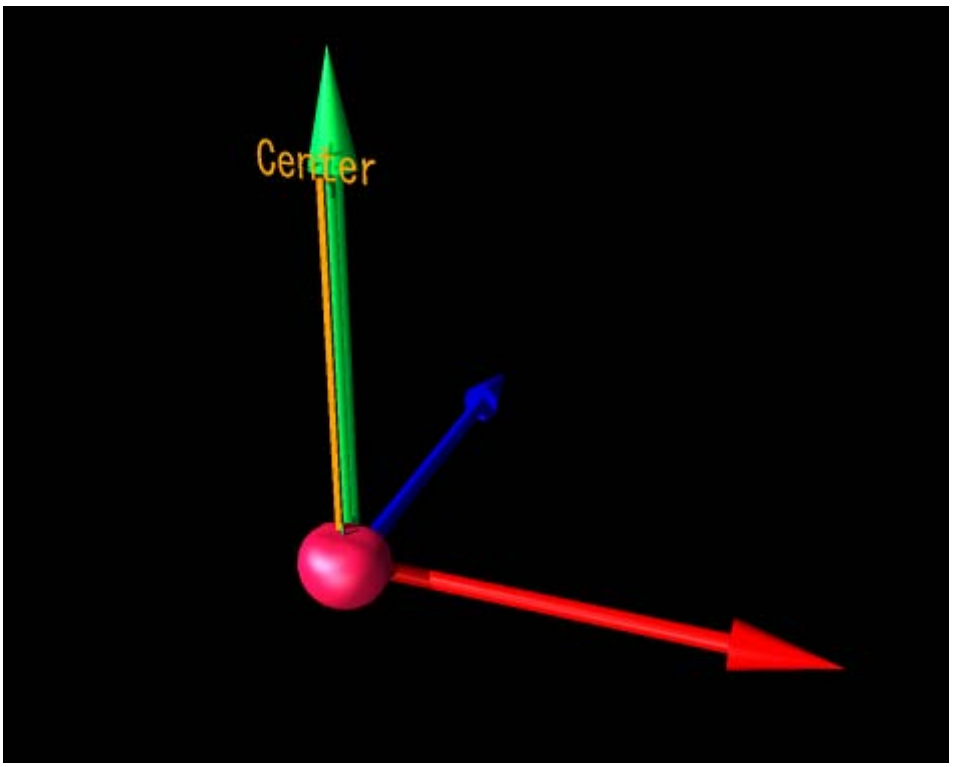
9-2“ 公転 ” させる

図 9-9　Center メッシュの周りを一定の半径で “ 公転 ” します。



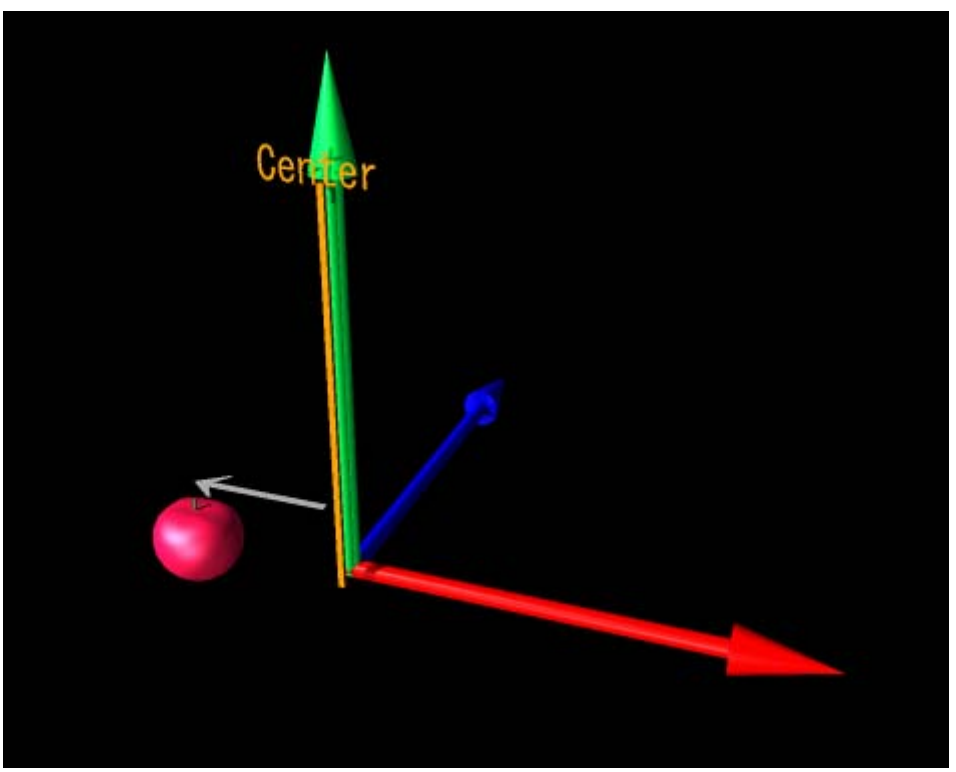
キャラを回転させるということは普通は “ 自転 ” です。自転ではなく時として “ 公転 “ させたい場合があります。公転とは、例えば地球は自分自身の軸で自転していますが、太陽を中心に公転もしています。イメージとしてはそれと同じ公転です。
つまりローカル軸周りではなく、ワールド系のほかの座標周りに回転させたい場合を考えます。
原理の考え方はこうです。
最初は、公転させるジオメトリ、公転の中心軸ともに原点にあるとします。
ジオメトリを公転半径分だけ原点からズラします。原点のままではどんなことをしても自転になってしまうためです。
ずらした後に回転を掛けます。これで原点の周りを公転した格好になります。
あとは、公転軸の移動量とシンクロさせれば、公転軸まわりに公転したように見えます。
この原理の流れを図にすると次のようになります。ジオメトリはリンゴメッシュ、公転軸は Center メッシュになっています。

図 9-10



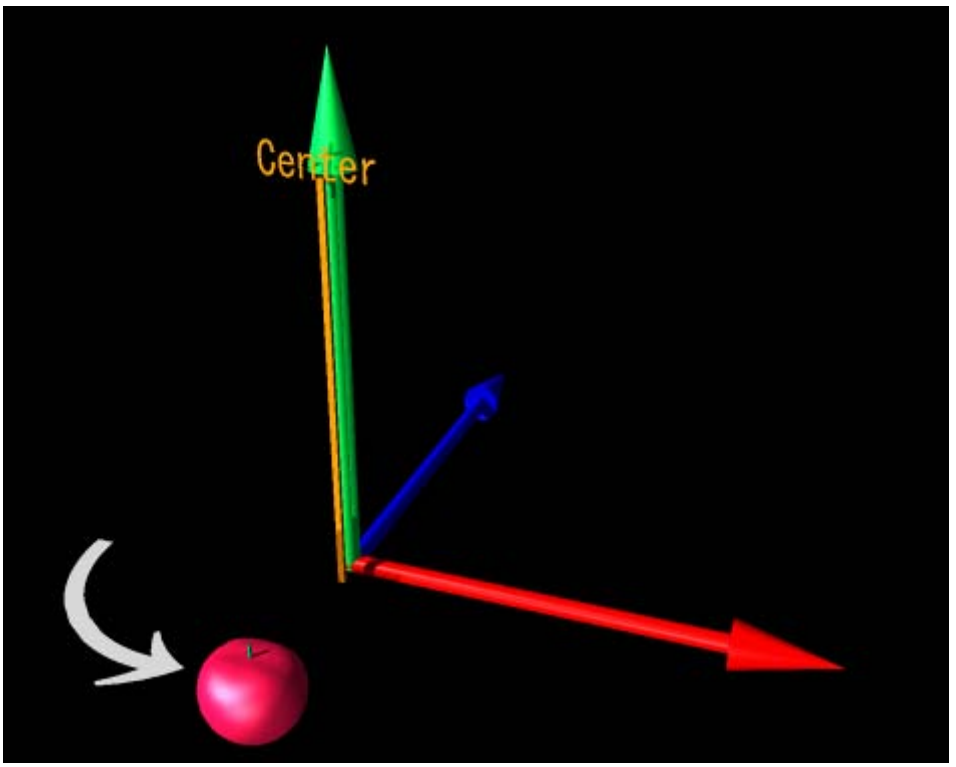
公転軸と林檎メッシュの初期位置です。

図 9-11



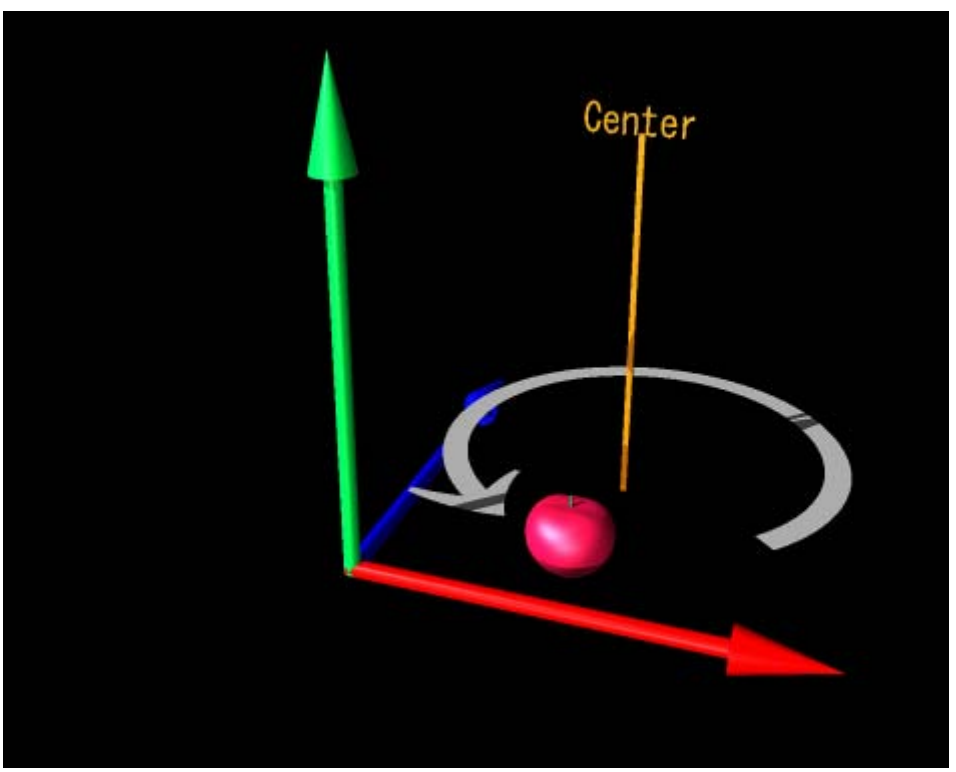
とにかく、任意の方向に公転半径だけズラす。

図 9-12



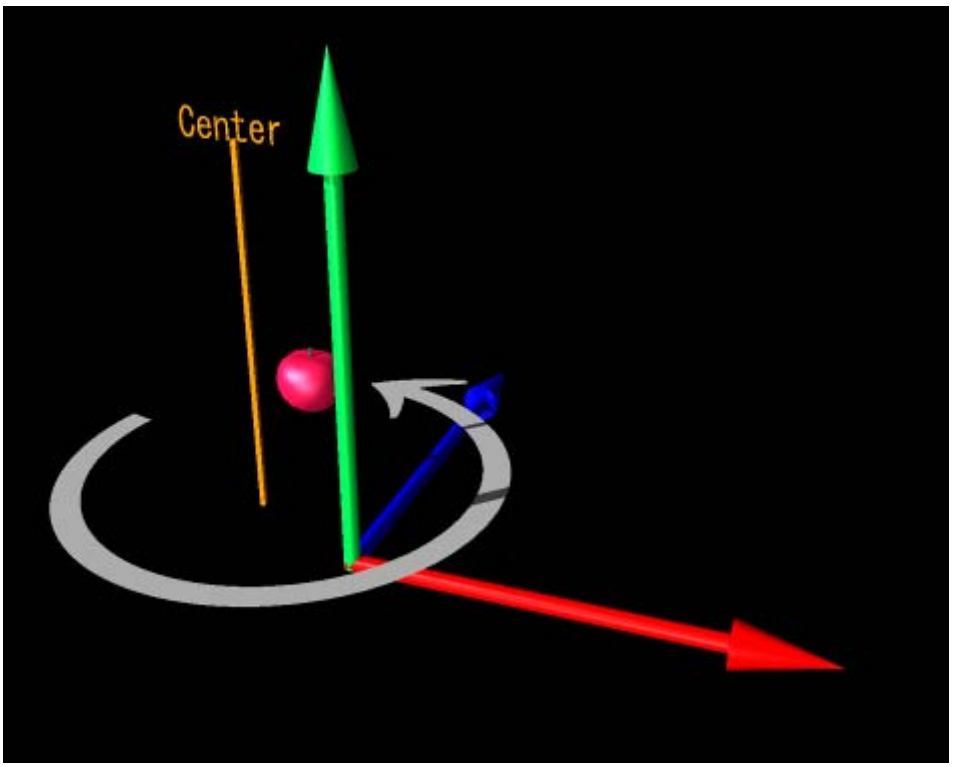
ズラしたあとに、回転を掛けると公転となる。

図 9-13



最後に公転の中心軸の移動量分だけ加えれば、中心軸に追従する。

図 9-14



回転軸が移動しても追従するので、回転軸周りの公転をすることになります。

サンプルプログラム

プロジェクトフォルダ名「ch09-2 ” 公転 ” させる」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

人間メッシュは、Center メッシュを軸として、その周りを一定の半径で回転します。言い換えると、Center メッシュを軸とした “ 公転 “ をします。また、人間メッシュは公転と同時に自分自身の中心で ” 自転 “ もします。自転することにより、公転軌道円の接線の方向を向きながら公転します。

使用方法
人間メッシュの前後左右移動：矢印キー

コード解説

では、実際のコードを見てみましょう。
注目すべき部分は、RotaryMove 関数です。
D3DXVECTOR3 vecTarget(-fRadius,0.0f,0.0f);
vecTarget は、公転させるメッシュの位置ベクトル用のベクトルで関数内部での一時的なベクトルです。このベクトルを公転させていきます。
まずは、原点から公転半径 fRadius 分だけずらしておきます。
図 9-11 がこの段階にあたります。
D3DXMatrixRotationY(&matRotation,timeGetTime()/1000.0f);
Y 軸周りの回転（ヨー回転）行列を作成します。D3DXMatrixRotation 関数で作成される回転行列はすべて原点周りの回転しか生成しません。というか、回転行列だけで公転させることはできません。

D3DXVec3TransformCoord(&vecTarget,&vecTarget,&matRotation);
vecTarget ベクトルにヨー回転行列を掛けると公転します。今は原点のまわりを公転しています。図 9-53 の段階です。
D3DXVec3Add(&vecTarget,&vecTarget,pvecCenter);
最後に vecTarget を公転軸の位置 vecCenter までもっていけば完了です。

pThing->vecPosition.x=vecTarget.x;
pThing->vecPosition.z=vecTarget.z;
pThing->matRotation=matRotation;
あとは、pThing の位置ベクトルに一時的な vecTarget の成分をコピーして、関数から戻るだけです。
なお、ウィンドウプロシージャー内でキー入力にともなって更新しているのは、Human メッシュの位置ベクトルではなく、Center メッシュ（公転軸）の位置ベクトルです。Center メッシュを動かして、Human メッシュを追従させるという発想でコーディングしました。

10 章　回転の誤解一掃！

3D が 2D に比べ難しいのは、“ 回転 ” が桁違いに複雑であるというのがその原因の 1 つだと思います。2D の場合、回転は XY 平面上での 1 種類しか無いのに対し、3D の場合はまず 3 軸周りの回転があり、その 3 つが組み合わさったときには頭の中でも想像し辛いものとなってしまいます。そして、2D の場合では、任意の座標で回転させるといっても同一平面上の回転なので簡単に想像出来るのに対し、2D 平面の点は 3D 空間では線になり、つまり、任意座標は “ 任意軸 ” となってしまうわけですが、任意軸周りの回転は無数に存在し、それもまた 3D 回転の難しさの 1 つでしょう。
ここでは、情報の氾濫や混乱により 3D の回転に関する様々な誤解や不明な点を解消していきたいと思います。

10-1 カルダン角とオイラー角

“ オイラー角 “ という言葉が間違って使用されていたり、間違っていないまでも不適切に使用されている場面を見かけます。
オイラー角（Euler Angle）とは、座標軸周りの回転が 3 回連続して行われるときの “3 つの回転角のセット ” のことを指し、その回転をオイラー回転（Euler Rotation）と言います。したがって、1 つの回転のみの回転角はオイラー角ではありませんし、異なる軸周りの回転であっても回転の合成が 2 つだけの場合はオイラー回転とは言えません。
オイラー回転は、X 軸、Y 軸、Z 軸周りの回転を 3 回行った結果のいわば合成回転です。聡明な読者は予想されたことと思いますが、軸と順序の組み合わせによりオイラー角の種類は複数存在します。軸が 3 種類、そして、回転が 3 回なので、単純に考えれば、27 種類（3、3 の重複順列）あるように思えますが、実際はそのうち、同じ軸が連続している組み合わせは意味が無いので除かれます。ちょっと想像すると分かりますが X 軸に回転した直後にまた X 軸に回転させたのでは回転を分ける意味はありません（笑い話としては面白いですが）。結果として、オイラー角は 12 種類存在します。そして、その組み合わせのことをコンベンション（Conventions）と言います。
回転する軸の順序（コンベンション）は次の 12 通りです。
X-Y-X
X-Y-Z
X-Z-X
X-Z-Y

Y-X-Y

Y-X-Z
Y-Z-X
Y-Z-Y

Z-X-Y
Z-X-Z
Z-Y-X
Z-Y-Z

12 種類ではありますが、特に Z-X-Z が最も使用頻度が高いため、これのみをオイラー角という人もいます。また、1 回目と 3 回目の回転軸が異なる組み合わせはオイラー角ではないという人もいますが、オイラー角は 12 種類です。
この 12 種類のうち、3 つが全て異なる軸である組み合わせは、特にカルダン角とも呼ばれます。したがって、カルダン角であればオイラー角ですが、オイラー角であればカルダン角であるとは限りません。
X-Y-Z
X-Z-Y

Y-X-Z
Y-Z-X

Z-X-Y
Z-Y-X

せっかくカルダン角という言葉があるのですから、3 つとも異なる組み合わせをカルダン角、それ以外の場合にはオイラー角と呼ぶのが妥当でしょう。
オイラー・カルダン角の 3 つの角度は、厳格な資料では順に（θ（シータ）、ϕ（ファイ）、ψ（プサイ））などと分けていますが、分かりにくいので（α、β、γ）あるいは単に 3 つとも全て θ と表記しても構いません。
角度の一般名とは別に、3 つの回転の呼称は、（Yaw ヨー、Pitch ピッチ、Roll ロール）とか、（Heading ヘディング、Pitch ピッチ、Bank バンク）などと、航空学、航空宇宙学、自動車工学…etc その他関連分野によって様々です。LightWave にどっぷり染まった筆者としては後者が馴染むのですが、Direct3D では前者を採用していますので、本書でもそれに合わせます。回転軸と回転呼称は独立しているので、例えば Z-X-Z コンベンションでは、同じ Z 軸周りの回転が異なる回転呼称になります。
なお、オイラー回転とは別に「ロール・ピッチ・ヨー回転」（？）なるものがあると思っている人がいますが、それは “XYZ コンベンションであるオイラー回転 “ でありオイラー回転の 1 つの呼称に過ぎないということを念のため申し添えておきます。回転の名前は他にも色々あり、全てを書くことはできませんし、どこかの研究所で全く独自の呼称があるやもしれません。中でも有名な呼称は次のものがあります。
（アジマス Azimuth、エレベーション Elivation、ティルト Tilt）
（プレセッション precession、ニューテーション nutation、スピン spin）
オイラー・カルダン回転は、1 つの軸回転が他の 2 つの軸に影響を及ぼしますので、各軸回転を複合的にイメージする必要があります。ただ、コンベンションと逆の順序で回転させると、客観的に見て、各軸が影響していなかったように見えますが、1 つの回転が他の 2 軸を回転させてしまうのは全く同

じです。
図 10-1

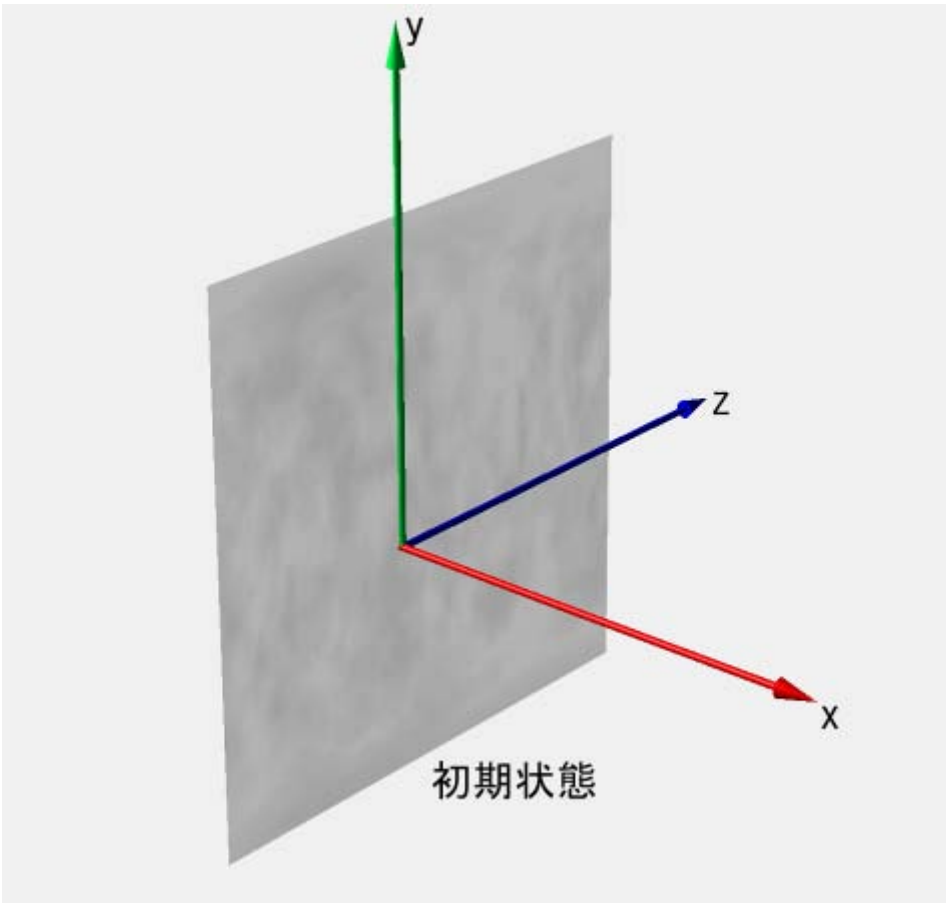


図 10-2　1 個目の回転

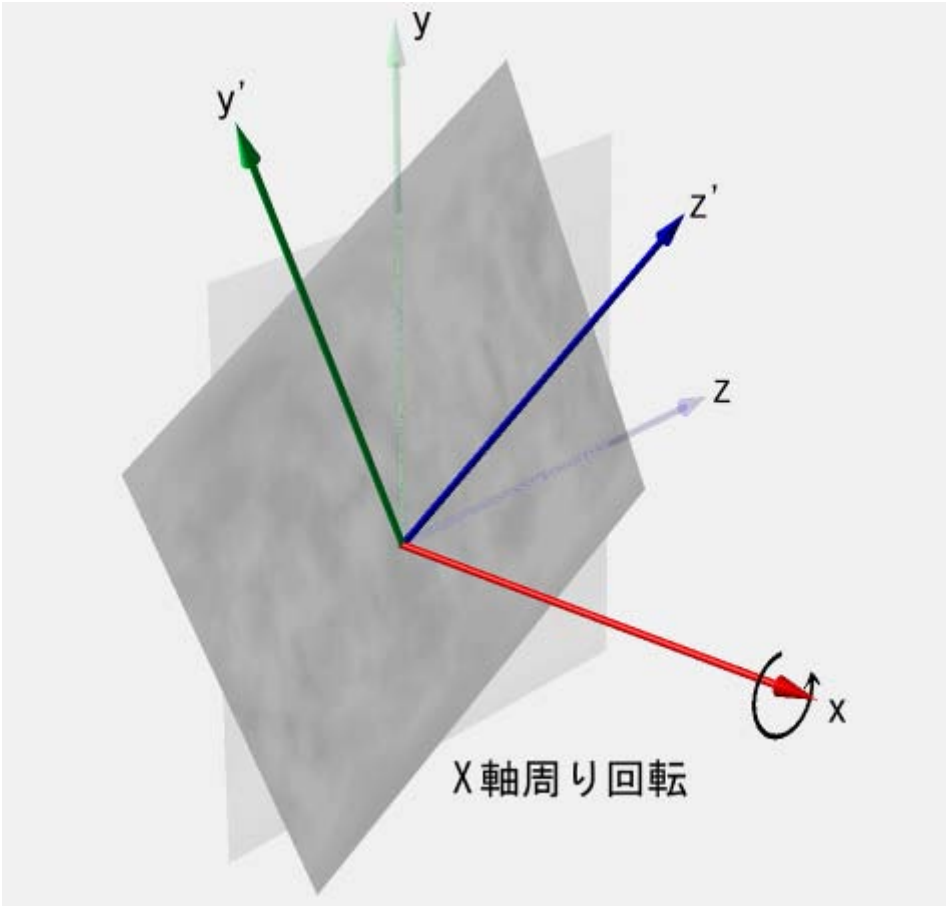


図 10-3　2 個目の回転（y' 軸周り）

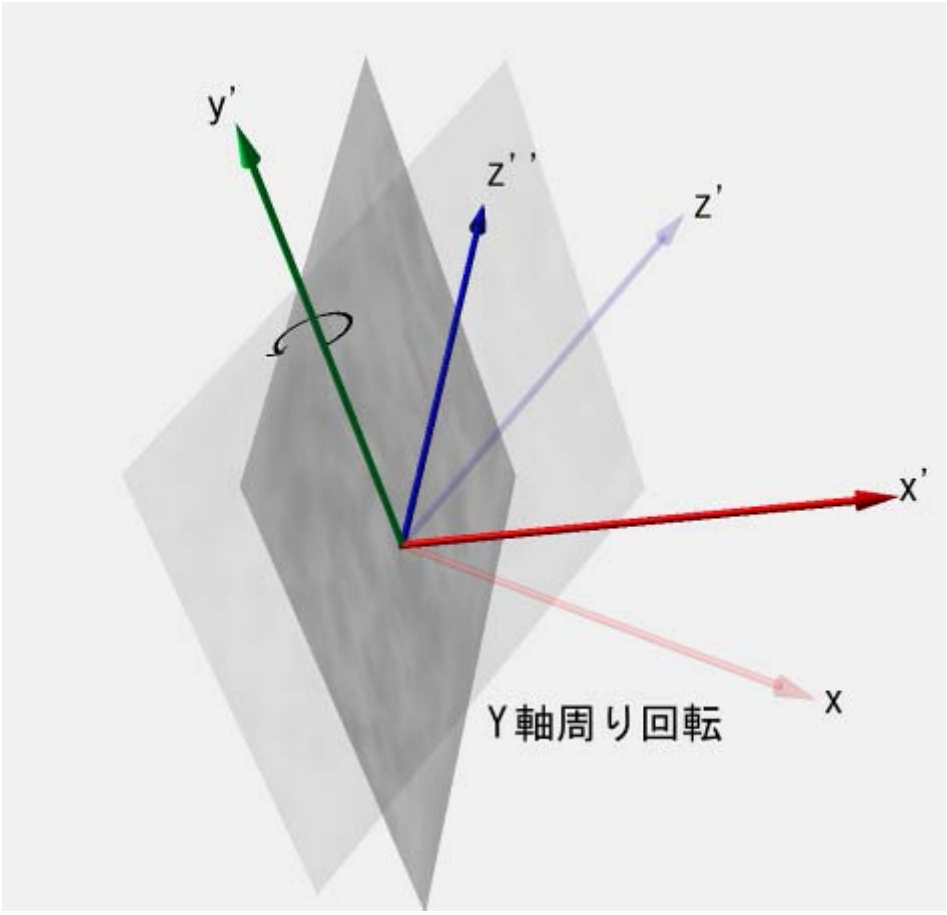
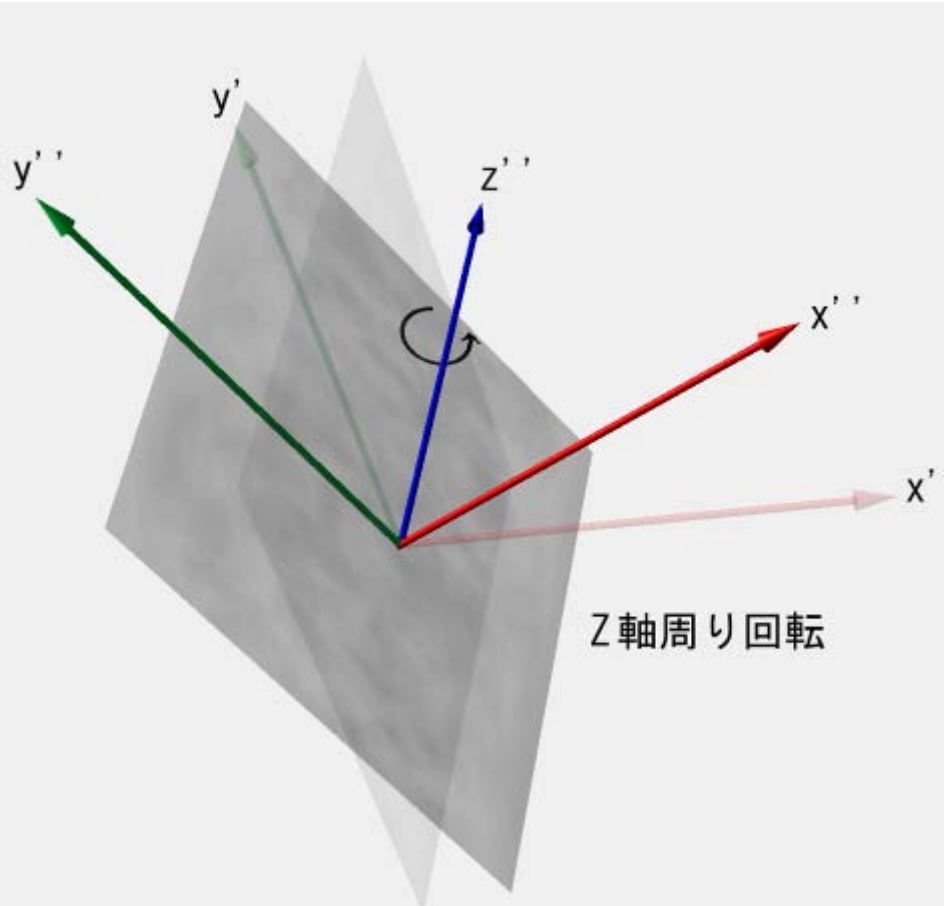


図 10-4　3 個目の回転（Z" 周り）



10-2 特異点とジンバルロック

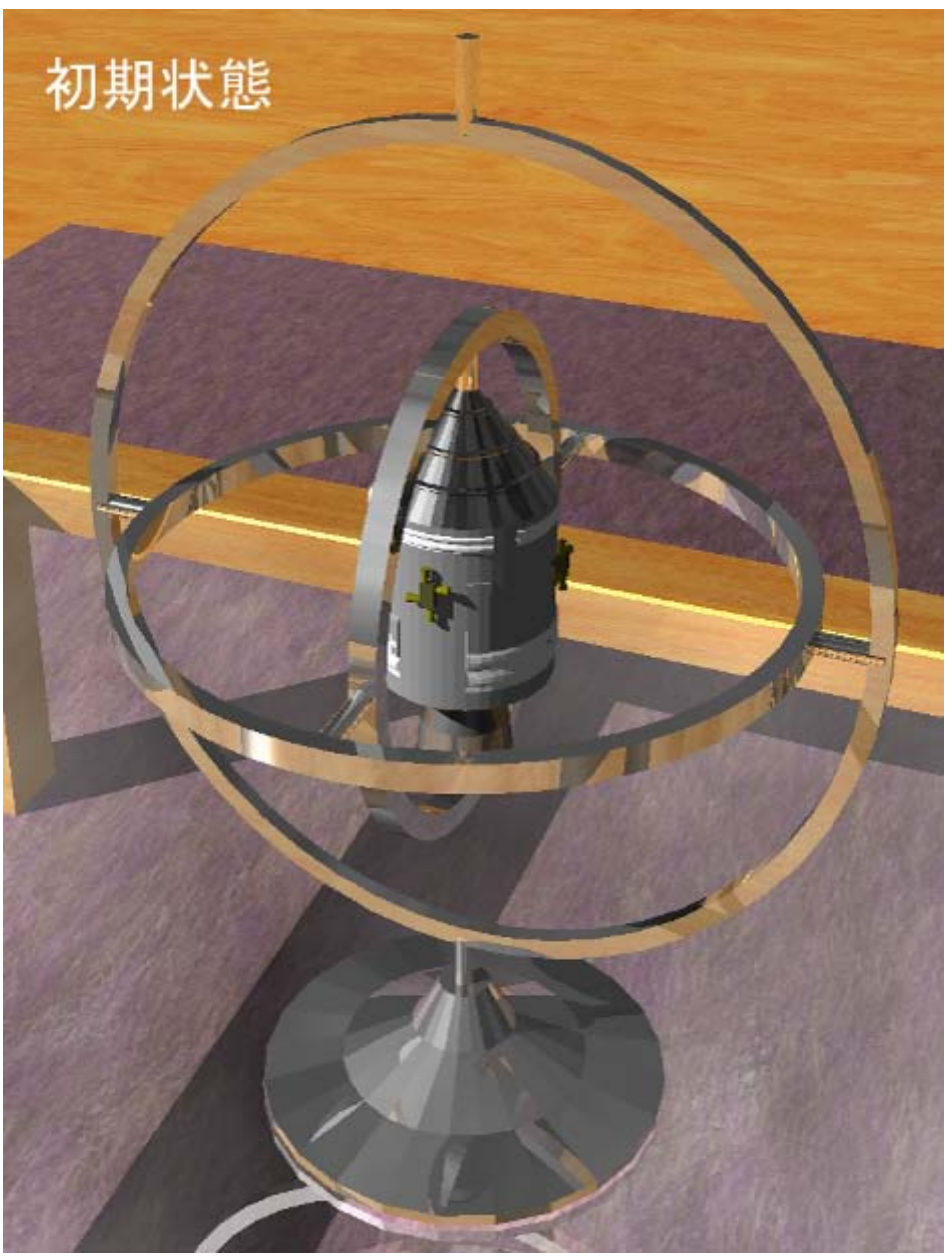
3DCG を少しでもかじったことがある人、特に 3DCG ソフトを勉強したことのある人は、ジンバルロックという言葉を聞いたことがあるかもしれません。筆者は 3DCG ソフトでは Lightwave しか知らないので他のソフトのことは何とも言えませんが、Lightwave ユーザーであれば聞いたことあるというよりも殆どの人は実際に体験していると思います。

ジンバルロックとは、回転軸のうち 2 本が重なったときに、思うような回転が出来なくなる状態のことで、もともとはジャイロスコープという姿勢安定装置のジンバルという部品がロックしてしまう現象からきています。オイラー・カルダン角には特異点という（悩ましい）角度があり、ジンバルをオイラー角により制御すると特異点でジンバルロックが起こり、特異点ではなくともその付近で好ましくない挙動をします。全てのコンベンションにおいて場所は異なれど特異点は必ず存在します。つまり、オイラー・カルダン角である限りジンバルロックからは逃れられないということです。

昔の宇宙船や人工衛星はジャイロの回転パラメーターにオイラー角を使っていたため、常にジンバルロックに気をつける必要があったようです。映画のアポロ 13（1995 年）を観たことがあるでしょうか？ケビン・ベーコンが液体酸素を拡散した際に起こった爆発により、アポロ 13 がきりもみ状態に陥り、トム・ハンクスが必死に姿勢を制御しようと奮闘している場面があります。その中で「このままではジンバルロックが起こる…」、「ジンバルロックが起こったら宇宙の迷子になる…」なんて調子で「ジンバルロック」という言葉が幾度となく使われていました。爆発で電力の大半を失ったため、3 人のクルーはやむなく宇宙船から LEM（あの独特な形で有名な月面着陸船です）に急遽乗り移りますが、コンピューターの姿勢制御計算プログラムは宇宙船側の重心を想定して作成されたものだった

ために、きりもみ状態は長く続きます。コンピューターが使い物にならないので、映画の中ではトム・ハンクスが紙と鉛筆でオイラー角を手計算し、その検算を地上の NASA 職員に頼むなんていう興味深いシーンもありました。関係ない話ではありますが、コンピューターさえ一般的ではなかった当時（1971 年）で、“ システムを再起動 ” だとか、“ 地上からプログラムをダウンロード ” なんていうやり取りを観て、「アメリカって凄いなぁ」なんて思ったものです（笑）。
なにか映画のレビューみたいになってきたので、この辺で止めておきますが、どうでしょう、興味が沸いてきませんか？まぁ、主観的な感想は置いておいても、ジンバルロックのイメージは掴めたのではないでしょうか。
現在はジャイロの制御にオイラー角より 1 つ成分の多い 4 つの角度から成る “ オイラーパラメーター ” という角度セットを使用しているため、パラメーターが原因でジンバルロックが起こることは無いそうです。ちなみに、このオイラーパラメーターは数学の世界で “ クォータニオン（四元数）” と呼ばれている数です。

図 10-5



それぞれのリングがジンバルです。一番外側から順に、アウタージンバル、ミドルジンバル、インナージンバルと呼びます。ジンバルはそれぞれシャフトで連結され、シャフトを軸として回転します。
図 10-6

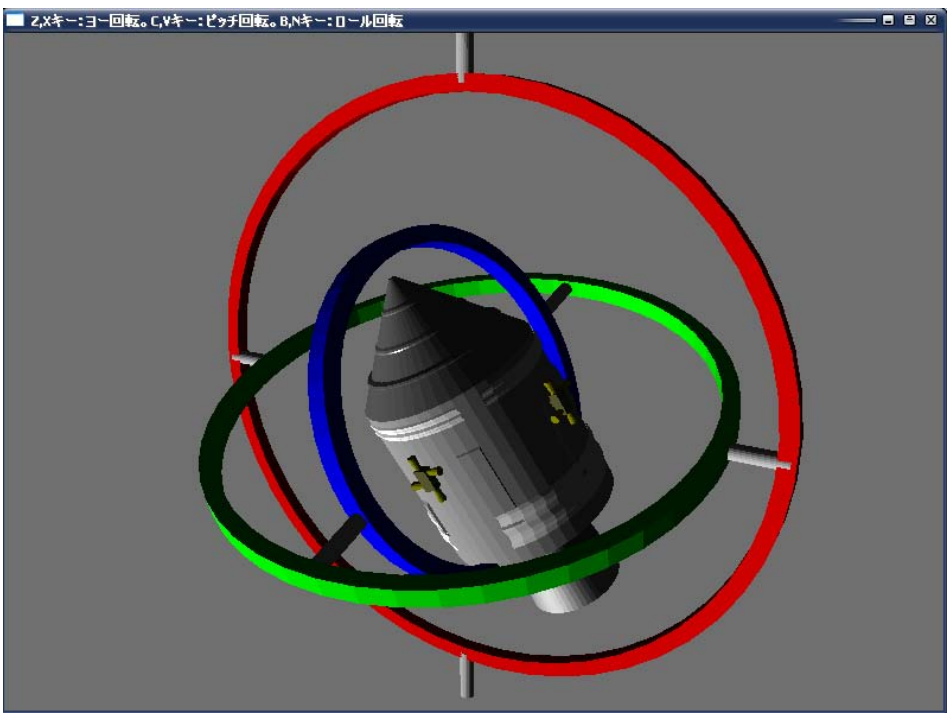
ミドルジンバル（真ん中のジンバル）を 90°回転させると、アウタージンバル（一番外側のジンバル）と重なります。
この状態では、ロール回転ができません。図を見れば明らかなようにロール回転の軸（シャフト）が無いのですから、回転できるわけがありません。言い換えれば 3 つの平面 XY、YZ、ZX のうち XY 平面上の回転が出来ないということであり、XY 平面を失っていると言えます。これを “ ディメンションを失う ” と言います。
「物理的な装置だから動かなくなるんだよ」とは思わないでください。この装置は忠実にカルダン角を再現しており、異なる部分は 1 つもありません。（ファイ、シータ、プサイ）がそれぞれのリングに対応しているわけですが、1 つの角度（1 つのジンバルリング）で 2 方向の回転を表現できるわけがありません。

このようにそれぞれのジンバルリングが 1 方向にしか回転しないようなモデルは、まさにカルダン角そのものと言えます。

10-2-1 ジンバルロックの実践？

先の図 -60、図 -61 と全く同じことをサンプルで実際に操作してみましょう。

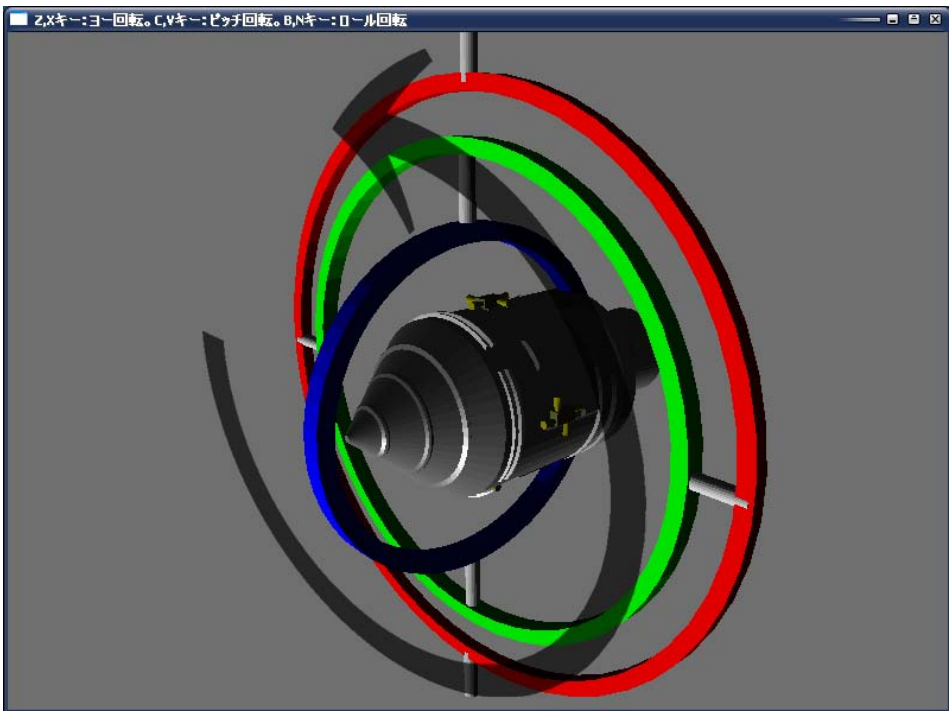
図 10-7　ジャイロの基本モデル



サンプルプログラム
プロジェクトフォルダ名「ch10-2-1 ジンバルロック」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

このサンプルは実際にジンバルロックを体験するために作成しました。
プログラムを起動したら、まずピッチを 90° 上げるか下げるかしてください。ピッチを上げ下げすると言うことはミドルジンバル（真ん中の緑色ジンバル）を回転させることです。90 度回転させるとミドルとアウターの 2 つのジンバルが重なるはずです。
その状態で宇宙船をロール回転出来ますか？逆立ちしても無理です。これがオイラー・カルダン角の限界です。

図 10-8　ロール回転は無理。“ 軸が無い ” のですから



使用方法
Z キーと X キーでヨー回転。C キーと V キーでピッチ回転、B キーと N キーでロール回転をします。
ESC キーで終了します。

補足
オイラー・カルダン角で、全ての回転状態（姿勢）を表現することは可能ですが、問題はその姿勢にたどり着くまでの操作だったり計算を工夫しなければならないことです。
オイラー・カルダン角を使う現実の各現場ではピッチを 90°にしないか、あるいは、なるべく 1 軸のみの入力にするか、いずれにしてもなんらかの工夫をして操作しているようです。

コード解説
一応、コードを解説しておきます。
ウィンドウプロシージャー内で、キー入力により各ジンバルの回転量を更新しています。
3 つのジンバルを回転させることは、カルダン角（ϕ、θ、ψ）を決定することと同義です。
Thing[0] はアウタージンバル（外側の赤ジンバル）なので ϕ 角度
Thing[1] はミドルジンバル（真ん中の緑ジンバル）なので θ 角度
Thing[2] はインナージンバル（内側の青ジンバル）なので ψ 角度
となります。
アウタージンバルは Z と X キーで、ミドルジンバルは C と V キーで、インナージンバルは B と N キーで、それぞれ操作します。
D3DXMatrixRotationY(&Thing[0].matRotation,Thing[0].fAngle);
D3DXMatrixRotationX(&Thing[1].matRotation,Thing[1].fAngle);
D3DXMatrixRotationZ(&Thing[2].matRotation,Thing[2].fAngle);
このサンプルではアウタージンバルは Y 軸周り回転、ミドルジンバルは X 軸周り回転、インナージンバルは Z 軸周り回転としています。

Thing[1].matRotation*=Thing[0].matRotation;
ミドルジンバルは、アウタージンバルの影響を受けます。影響とはアウタージンバルの回転行列を掛けて回転を継承することを意味します。
Thing[2].matRotation*=Thing[1].matRotation;
インナージンバルは、ミドルジンバル（したがってアウタージンバルも）の影響を受けます。
Thing[3].matRotation=Thing[2].matRotation;
宇宙船は、直接的にはインナージンバルの影響を受けるので、間接的に全てのジンバルの回転の結果を反映することになります。
コード的なポイントはこれくらいです。

10-2-2 いまいちなカメラの回転

図 10-9　宇宙ドームメッシュ　これは視点がカルダン角の特異点にある状態

回転量をオイラー・カルダン角で保存すると、どうゆうことになるかはもうお分かりでしょう。そうジンバルロックが必ず起こります。
さらに、オイラー・カルダン角の特異点がもたらす障害はジンバルロックだけではありません。特異点ピッタリの角度じゃなくても、その付近では角速度が上がるため、見た目にも変ですし操作するのも難しくなります。

サンプルプログラム
プロジェクトフォルダ名「ch10-2-2 いまいちなカメラの回転（FPV）」
VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

このサンプルではそれらを実際に体験するためのものです。
本サンプルは、この後の「いまいちなオブジェクトの回転」サンプルとほとんど同じですが、本サンプルが 1 人称視点（宇宙船から見た視点）であるのに対し、次のサンプルが 3 人称視点（宇宙船を外

から見ている視点）になっているところが違います。
まず、実際に体験するために、操作方法から説明します。
使用方法
Z キーと X キーでヨー回転。C キーと V キーでピッチ回転、B キーと N キーでロール回転をします。
特異点まで操作するのが面倒な場合は、F1 キーでショートカットできます。
ESC キーで終了します。

起動すると、1 人称視点から宇宙空間が広がります。矢印キーの左右で視点が左右に回転、矢印キーの上下で視点が上下に回転、X,Z キーで視点が画面に対して時計（反時計）回りに回転します。…と言いたいところですが、カルダン角の制御の難しさにより、このキーと回転の対応が当てはまるのは、姿勢が初期状態の時だけです。無限にある回転状態（姿勢）のうち、ただ 1 箇所で瞬間的にしか当てはまりません。これは、角度の更新を単純に行っているというアプリケーション側の問題もありますが、このサンプルは、直感的なコーディングで発生するカルダン角の問題を体験するためのものでもあるため、奇妙な回転はむしろ好都合です。
たとえば、ピッチがゼロじゅないときにヨー回転を意図しても綺麗にヨー回転をしてくれません。他方で、ヨーがゼロじゃなくともピッチは綺麗に回転したりします。これは、回転の合成順番とキー入力の順番が関係しているもので、この順番をうまくすれば 2 軸のみの回転で済むようなアプリケーションでは上手くいくかもしれません。しかし、それでも本質的に特異点から逃れられるわけではありません。

コード解説
では、コードの解説をします。

定義とグローバルインスタンス

#define NormalizeWhithin2PI(radian)
与えられた角度を 2 πラジアン（360°）の範囲内に変換するマクロを定義しました。例えば、430°の場合は 70°に変換します。
#define NormalizeWhithinPI(radian)
同様に角度をπラジアン（180°）の範囲に収まるような正規化をします。
#define NormalizeWhithinHalfPI(radian)
同様に角度を 1/2 πラジアン（90°）の範囲に収まるような正規化をします。

```
struct CARDAN_ANGLE// カルダン角
{
	FLOAT phi;// φファイ
	FLOAT theta;// θシータ
	FLOAT psi;// ψプサイ
};
```

カルダン角を構造体にしました。3 つのフロート値であれば D3DXVECTOR3 型と同じなので、D3DXVECTOR3 を利用すればオペレーターも定義されていることだし便利だったのですが、要素名が x,y,z というのは混乱する恐れがあると思い、独自に定義しました。3 つの要素ファイ、シータ、プサイは軸周りの角度を意味し、これら 3 つで 1 つのカルダン角を表現します。

enum CARDAN_CONVENTION
これはあまり重要ではありません。カルダン角のコンベンションを識別するための列挙定数です。無くても良かったのですが、IsGimbalLock 関数の見栄えを良くするために用意しました。

CARDAN_ANGLE CardanAngle={0};
独自に定義したカルダン角型のグローバルインスタンスを作成し、ゼロで初期化しています。

ウィンドウプロシージャー
ウィンドウプロシージャー内では、キー入力によってカルダン角を調整する処理を行っています。

```
case VK_RIGHT:
        CardanAngle.phi-=0.05f;
        break;
case VK_LEFT:
        CardanAngle.phi+=0.05f;
        break;
case VK_DOWN:
        CardanAngle.theta-=0.05f;
        break;
case VK_UP:
        CardanAngle.theta+=0.05f;
        break;
case 'X':
        CardanAngle.psi-=0.05f;
        break;
case 'Z':
        CardanAngle.psi+=0.05f;
        break;
```

特異点までもっていく操作が面倒な場合には F! キーでショートカットできるような機能も付けました。

```
case VK_F1:// このアプリケーションにおけるカルダン角の特異点はピッチ =90 度
        CardanAngle.phi=D3DXToRadian(0);
        CardanAngle.theta=D3DXToRadian(90);
        CardanAngle.psi=D3DXToRadian(0);
        break;
```

カルダン角的に表現すると、θ が 90 度である時には ϕ と ψ の区別が付かなくなり、両者を分ける意味を失ってしまうのです。

Render 関数

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

本サンプルは、1 人称視点なのでメッシュの移動の必要はありません。というか移動すると困ります。

ワールド変換行列は、無害な単位行列にして、そのままパイプラインに渡します。

CardanRotation(&matView,CardanAngle.phi,CardanAngle.theta,CardanAngle.psi);
CardanRotation 関数は、第２引数のカルダン角を基に回転行列を作成して、第１引数の行列に格納します。

```
if(S_FALSE==IsGimbalLock(XYZ))
{
        …
}
```
IsGimbalLock 関数は現在の姿勢がジンバルロック状態かどうかを判断する関数です。
ジンバルロックの場合は、メッセージを表示します。

NormalizeWhithin2PI(CardanAngle.phi);
NormalizeWhithin2PI(CardanAngle.theta);
NormalizeWhithin2PI(CardanAngle.psi);
独自に定義しておいたマクロを使用して、カルダン角を２πラジアン（360°）以内に正規化しています。
正規化する理由は、次のように画面に角度情報を表示する際の視認性のためです。

sprintf(szInfo," カルダン角（φ，θ，ψ）=(%3.1f ,%3.1f,%3.1f)",D3DXToDegree(CardanAngle.phi),
D3DXToDegree(CardanAngle.theta),D3DXToDegree(CardanAngle.psi),D3DXCOLOR(0,1,0,1));
ラジアン単位よりも度単位のほうがさらに見易いので、度数に換算してから表示します。

CardanRotation 関数
関数がやっている処理は極めて簡単です。このような処理は最もよく行われているのではないでしょうか。

D3DXMatrixRotationY(&matYaw,Phi);
D3DXMatrixRotationX(&matPitch,Theta);
D3DXMatrixRotationZ(&matRoll,Psi);
x,y,z 軸周りの回転行列を作り

matRotation=matYaw*matPitch*matRoll;
その３回転を合成しているだけです。
もっとも素直で単純な処理ですね。皆さんもこのような３軸周りの回転をこのように合成することが多いのではないでしょうか？それはカルダン回転行列を作成していたのです。

IsGimbalLock 関数
この関数は、本質的には必要ないのですが、確認しやすいように便宜上の付加機能として書きました。
現在の姿勢がジンバルロック状態（あるいはそれに近い）かどうか判断し、そうであれば S_FALSE を、大丈夫であれば S_OK を返します。

NormalizeWhithinPI(fAngleYaw);
NormalizeWhithinPI(fAnglePitch);

NormalizeWhithinPI(fAngleRoll);
まず、角度がπラジアン（180°）以上になっている可能性もあるので、πラジアン（180°）内に収まるように正規化します。ジンバルロックの角度ピッタリの瞬間だけ判断するのであれば 1/2 πラジアン（90°）以内に正規化すればいいのですが、ここでは 95°までの範囲を判断しているのでπラジアン（180°）以内で正規化します。

```
fAngleYaw=D3DXToDegree(fAngleYaw);
fAnglePitch=D3DXToDegree(fAnglePitch);
fAngleRoll=D3DXToDegree(fAngleRoll);
```

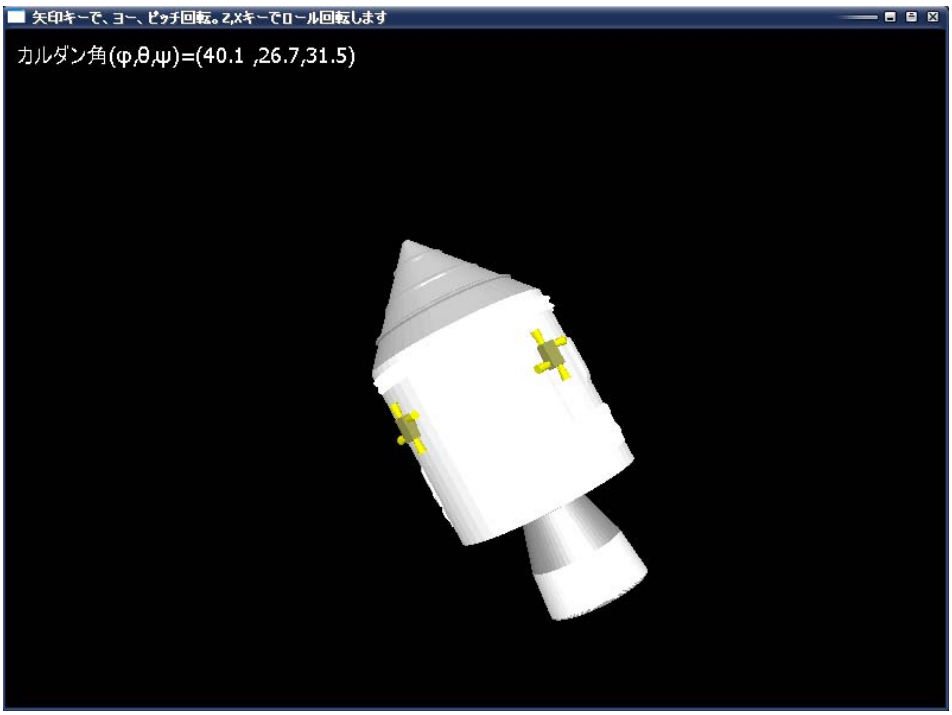
正規化した角度を 60 分法の度数単位に換算します。この換算の目的は、次の if 分を見やすくするためだけに行っています。
各軸の回転角が 85°～ 95°の場合、フラグを立てます。

```
if(fAnglePitch>85 && fAnglePitch<95  )
{
        dwFlag = 1;
}
if(fAngleYaw>85 && fAngleYaw<95   )
{
        dwFlag |= 2;
}
if(fAngleRoll>85 && fAngleRoll<95   )
{
        dwFlag |= 4;
}
```

ピッチが 85°～ 90°の場合を A
ヨーが 85°～ 90°の場合を B
ロールが 85°～ 90°の場合を C
とすると、パターンは 7 種類で、それぞれのときの dwFlag の値は次のようになります。
A　dwFlag=1
B　dwFlag=2
C　dwFlag=4
AB　dwFlag=3
AC　dwFlag=5
BC　dwFlag=6
ABC　dwFlag=7
このように、場合が複数組み合わさる状況では、それぞれの場合に 2 のべき乗の論理和をとっていくと、すべてのパターンを重複することなく表現できます。
全てのコンベンションについての特異点を判断すると膨大なソース量になってしまうので、今回は 1 つのコンベンションについて判断します。その場合のフラグ値は 9,11,13 の 3 つです。

10-2-3 いまいちなオブジェクトの回転

図 10-10　アポロ 13 メッシュ　操作しづらい…



プロジェクトフォルダ名「ch10-2-3 いまいちなオブジェクトの回転（TPV)」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

本サンプルは直前のサンプルの 3 人称視点バージョンであり、コード上の相違点は次の 2 点だけです。
① 回転をワールド変換行列に反映する（前サンプルはビュー変換行列）
② （当然ですが）カメラ位置は固定。つまりビュー行列は一定。

これ以外は全く同じなので、解説することは何もありません。
3 人称視点なので、オブジェクトの回転がどのように奇妙であるかが分かり易いかもしれません。

10-3 クォータニオンでもジンバルロック？

クォータニオンにより回転量を保存すればジンバルロックは起こりません。ただ、あまりにもクォータニオンがもてはやされていることから、初心者はそれがまるで魔法の数であるかのような錯覚を起こし、クオータニオンを使いさえすればジンバルロックや妙な回転を回避できると考えてしまい、クオータニオンを使っているのに回転が改善されない…という錯誤に陥ることがあります。最も良くあるケースとしては、クォータニオンと行列を単純にすり替えただけのコードがあります。これは明らかに意味がありませんので、初心者でもすぐに気付きます。そこで次に、任意の軸を用意してその軸周りでの回転を考えたところまでは良かったものの、その軸自体の回転にカルダン角を使ってしまっているという状況もあります。笑い話としては面白いですが、もちろんこれではジンバルロックや不規則な角速度というカルダン角の負の遺産を全額相続してしまいます。
初心者は、クォータニオンと行列を、それぞれの代替ツールとして同じ土俵で考えることが多いですが、クオータニオンは数であって、どちらかといえばベクトルや角度と比較すべきものです。クオータニオンはそれ自体、なにか別の概念の組み合わせではなく、ベクトルのようにある意味プリミティブです。それに対し、行列はベクトルやクオータニオンから構成されるセットですから、次元（素性という意味です）が同じではありません。

ただ、コーディング上での利用において、たしかに行列のごとく振舞うことは事実です。行列に回転量を蓄積するということはよくやります。3 次元ベクトルの複素数バージョンであるクオータニオンが運良く 3 次元空間での姿勢を表現できることから、その部分で行列をクォータニオンに置き換えるということもよくやります。そのような代替を客観的に見ると両者はツールとして同次元かもしれません。しかし、両者に互換性が存在するのは " 姿勢 " を扱うときだけであって、回転処理においてたまたま相互交換が可能であるだけです。
まぁここまで哲学的な分類は必要ないかもしれませんが、心のどこかに留めておくのも悪くはないはずです。

10-4 CG におけるクォータニオンのメリット

図 10-11　クォータニオンベースに切り替えると 38fps が 74fps になりました。

サンプルプログラム

プロジェクトフォルダ名「ch10-4 マトリクス vs クォータニオン？」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

「行列 vs クォータニオン」というのも意味が良く分かりませんが（自分で書いておいてスイマセン）要するに、" 姿勢保持においてどちらが有用 " であるかということを言いたかったのです。
ここで断っておきますが、
ジンバルロックが起きない
任意軸の回転ができる
アニメーション補完ができる
これらは行列とベクトルでも出来ることであって、クォータニオンの専売特許ではありません。となるとクォータニオンのアドバンテージ（優位性）は " 処理速度 " ただそれだけに尽きます。
Direct3D における行列が 16 個の FLOAT 値に対する演算なのに対し（実際に計算する回数はもっと少ないでしょうが）、クォータニオンは 4 つの FLOAT 値しかないわけですから、考えるまでもなく演

算が速いのは明らかでしょう。
そこで、「本当に速いのか？」を実証することが本サンプルの趣旨です。
結論から先に言うと、「やっぱり速かった」です。（笑）
どのような測定をしたかと言うと、
回転行列の作成…a
回転クォータニオンの作成…b
a と b を比較するという単純なものです。
そのままでは、どちらも 1ms オーダーでは計測できないほど速いので、適当に負荷を掛けています。
負荷とは単純に同じ処理をループにより無駄に繰り返しているだけです。
行列とクォータニオンをスペースキーでトグルし、矢印キーで適宜負荷を上げ下げしながら、それぞれのフレームレートによって差を視認できるようにしました。
行列の計測対象は RotateWithMatrix 関数で、クォータニオンは RotateWithQuaternion 関数です。
コードの解説は不要でしょう。
なお、本来クォータニオンの場合は、最終的に行列に変換するというオーバーヘッドも加味するべきなのかもしれません。ただ、様々な実装形態によりそのオーバーヘッドが無視できるようなものだったり、非常に大きくなったりする（これはプログラマーの責任のような気もします）ので、ここでは除外しました。
使用方法
マトリクスとクォータニオンのトグリング：スペースキー
負荷の増減：左矢印キーと右矢印キー
ESC キーで終了します。

11 章　直感的回転の実現

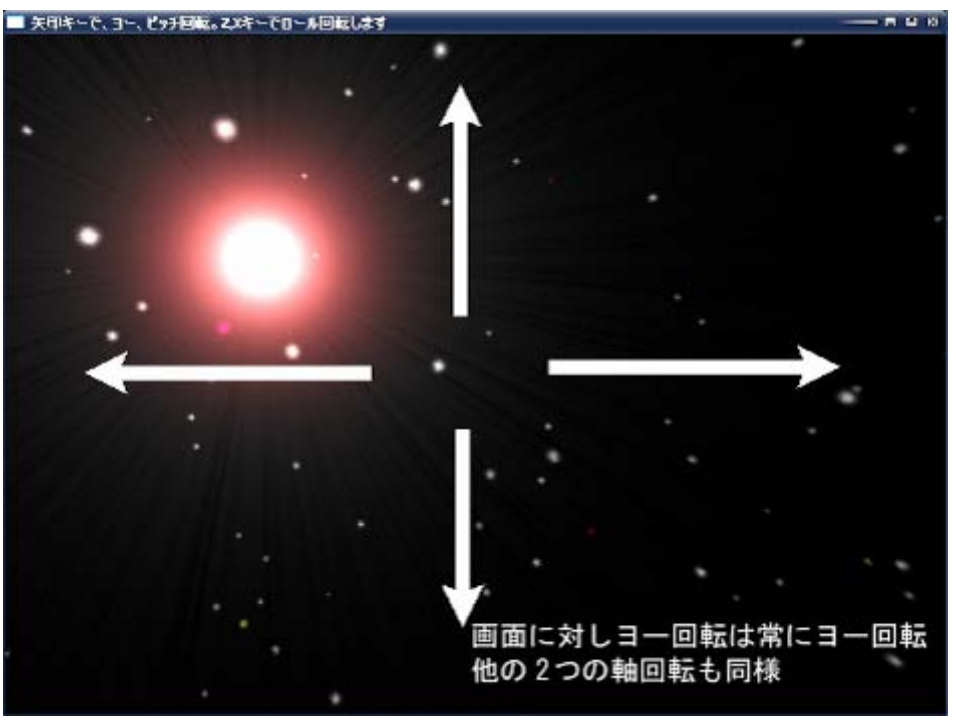
11-1 カメラの直感的回転

これまでの回転は、画面に対してヨー回転がピッチ回転になったりするなど、直感的な操作がし辛いものでした。ここでの直感的回転とは、画面に対してヨー回転は常にヨー回転、つまり常に左右の回転、同様にピッチ回転は常に上下の回転、ロール回転は常に画面手前から画面奥伸びる Z 軸周りの回転ができる操作です。
画面に対して絶対的なヨー、ピッチ、ロールを維持するのにオイラー・カルダン角を使用する場合、入力を工夫する必要があり面倒です。ここでは、オイラー・カルダン角を使用しないで直感的回転を実現するアプローチを採ります。

サンプルプログラム

プロジェクトフォルダ名「ch11-1 カメラの直感的回転（FPV）」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

図 11-1　画面に対して絶対的な縦横方向に回転できる



直感的回転を実現するために、行列に現在の姿勢を保存します。このような行列は一般に姿勢行列などと呼ばれています。
キー入力により増減される回転量は、オイラー・カルダン角ではなく、x,y,z のうち何れか 1 軸周りの回転とします。それをその時の姿勢行列に掛け合わせていきます。このような回転ではいかなる角度でも特異点は存在せず、直感的な回転操作を持続できます。
オイラー・カルダン角を保持（保存）するのではなく姿勢を保持するという発想です。
使用方法
画面に対して常に横方向の回転（絶対的なヨー回転）：左右矢印キー

画面に対して常に縦方向の回転（絶対的なピッチ回転）：上下矢印キー
画面に対して常に時計回り回転（絶対的なロール回転）：Z キーと X キー
ESC で終了。

コード解説

コードを見てみましょう。
ウィンドウプロシージャー
キー入力によりヨー回転角 fYaw、ピッチ回転角 fPitch、ロール回転角 fRoll を増減させています。ただし、1 回の姿勢更新で使用するのは 1 軸についてのみです。

Reder 関数
特に注目すべき箇所はありません。

CurrentAttitude 関数
この関数が本サンプルのキーポイントです。
3 軸の回転角を受け取るわけですが、それをカルダン角として使用しているわけではありません。使用するのは 1 軸についてのみです。なぜなら、3 軸全てを使用してしまうと、いくら姿勢行列を保存していうるからといっても回転同士の非可逆性が顔を出してしまい、カルダン角の問題が生じてしまうからです。もっとも、保存してある現在姿勢に “ 回転の増分 ” を積み重ねていく場合には、非可逆性が表面化するほど増分が大きくはないですが、それは大小の問題であって、本質的に問題をはらんでいることに変わりはありません。

```
static BOOL  boOneTime=FALSE;
if(!boOneTime)
{
	boOneTime=TRUE;
	D3DXMatrixIdentity(pmatAttitude);
}
```

関数が最初に呼ばれたときに 1 度だけ、姿勢行列を単位行列にしておきます。どこか別の場所で姿勢行列を初期化している場合は、この部分は不要です。

```
if(fYaw !=0)
{	D3DXMatrixRotationAxis(&matAxis,&D3DXVECTOR3(pmatAttitude->_21,pmatAttitude->_22,pmatAttitude->_23),fYaw);
}
```

わざわざ if 文でくくっているのは、1 軸のみの回転をさせるためです。他の 2 軸の回転コードはその下の else ブロック内に置き、2 軸以上の回転処理を行わないようにしています。
1 人称視点の場合は、言い替えるとカメラを回転させる場合は、x,y,z のような基底軸ではなく任意軸を使います。任意軸といっても、カメラのローカル座標系での基底軸 (ローカル x,y,z 軸) です。行列の行ラインは、それぞれ基底軸ベクトルになっています。姿勢ベクトルの 1 行目の横ラインはカメラのローカル座標系における X 軸ベクトル、同様に 2 行目は Y 軸、3 行目は Z 軸を意味します。ヨー回転は Y 軸周りなので、2 行目のベクトルを回転軸として、fYaw 分だけ回る回転行列を作成しています。以下同様に、fPitch、fRoll についても処理しています。

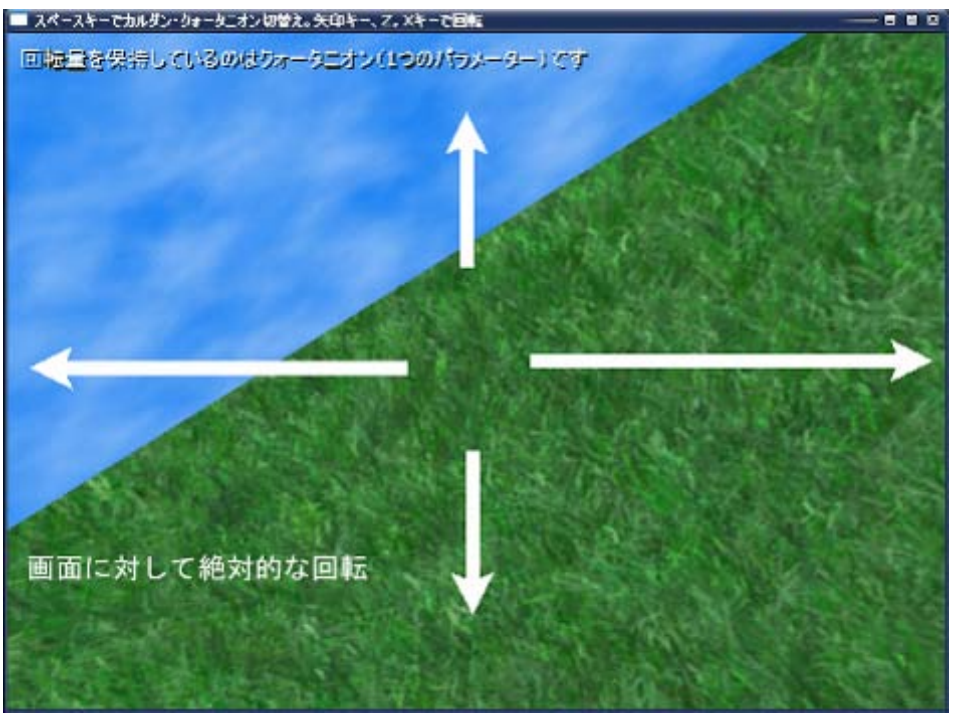
pmatAttitude=matAxis;
関数内で一時的に用意したローカルインスタンスを呼び出し元の行列にコピーします。
最初から呼び出し元の行列を使用する場合は不要です。

11-2 カルダン角ではなくクォータニオンを使う

サンプルプログラム

プロジェクトフォルダ名「ch11-2 カルダン角ではなくクォータニオンを使う」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

図 11-2　クォータニオンを使用した場合。モニターに対し絶対的な回転



直前のサンプルでは、姿勢を保持しているインスタンスは行列でした。ここでは、行列の代わりにクォータニオンを使用します。行列インスタンスを使用する場合と同じく、ここでも、姿勢を保持するインスタンスはカルダン角としての 3 つの FLOAT 値ではなくクォータニオンというインスタンス 1 つなので、非可逆性（掛ける順番によって結果が違うという性質）を考える必要もありません。
使用方法
画面に対して常に横方向の回転（絶対的なヨー回転）：左右矢印キー
画面に対して常に縦方向の回転（絶対的なピッチ回転）：上下矢印キー
画面に対して常に時計回り回転（絶対的なロール回転）：Z キーと X キー
カルダン角とクォータニオンのトグリング：スペースキー
ESC で終了。

コード解説

ほとんど直前のサンプルと同様ですが、キー入力によりクォータニオンを更新す部分は説明の必要が

あるでしょう。

ウィンドウプロシージャー
いままでと同じくキー押下メッセージハンドラで回転のパラメーターを増減させています。
クオータニオンの更新は、カルダン角（fYaw,fPitch,fRoll）の時のように単純に値を増減させることは出来ません。
コードを理解するにはあたり、多少の数学的知識を仕入れる必要があります。数学と言っても証明無しの結果だけなので、たいしたことはありません。
クオータニオンは、次のように表すことができる数です。

Qをクオータニオン、θ　を回転角、Vを回転軸ベクトルとす　ると

$$Q = \cos\left(\frac{1}{2}\theta\right) + \sin\left(\frac{1}{2}\theta\right) \times V \cdots ①$$

クオータニオン Q は実数と虚数から成る複素数です。複素数とは a+bi（i は虚数単位）と定義されるる数です。クォータニオンも Q=w+(x,y,z)i 正確に書くと Q=w+xi+yj+zk（i,j,k は虚数単位）というように実数部 + 虚数部とう形になっています。①式で言うと

$\cos\left(\frac{1}{2}\theta\right)$ の部分が実数部

$\sin\left(\frac{1}{2}\theta\right) \times V$ の部分が虚数部に対応します。

コード的には、D3DXQUATERNION の w メンバが実数部、x,y,z が虚数部に対応しています。
虚数部といっても x,y,z は “ 虚数部の実数 ” なので実数です。

Q=w+　ix+jy+kz　i,j,k は虚数単位

実数部　　虚数部

x,y,z は虚数単位の係数、すなわち実数値です。クォータニオンの虚数部の係数はたまたま 3 次元ベクトル空間での回転軸を表します。そして、実数部は回転量（回転角）を表します。
したがって、例えば X 軸周りにπラジアン（180°）回転させる場合は、
回転軸 V の成分は V=(1,0,0) なので、

$$Q = \cos\left(\frac{1}{2} \times \pi\right) + \sin\left(\frac{1}{2} \times \pi\right) \times (1,0,0)$$
$$= 0 + 1 \times (1,0,0)$$

D3DXQUATERNION.w = 0
D3DXQUATERNION.x = 1
D3DXQUATERNION.y = 0
D3DXQUATERNION.z = 0

となります。

同様に、1/6 π ラジアン（30°）回転させる場合は、

$$Q = \cos\left(\frac{1}{2}\times\frac{1}{6}\pi\right) + \sin\left(\frac{1}{2}\times\frac{1}{6}\pi\right)\times(1,0,0)$$
$$= 0.9659 + 0.2588\times(1,0,0)$$

D3DXQUATERNION.w = 0.9659
D3DXQUATERNION.x = 0.2588
D3DXQUATERNION.y = 0
D3DXQUATERNION.z = 0
となります。
ウィンドウに標準で付いている関数電卓（普通の電卓の場合は表示メニューで関数電卓になります）で簡単に計算できるので検算してみてください。（1/2 π =90、1/12 π =15 で入力すると簡単です）

キー入力のメッセージハンドラでは、これと全く同じことを行っています。例えば、右矢印キー用のハンドラを見てみましょう。

```
case VK_RIGHT:
        fYaw-=DELTA_ANGLE;
        //sin(1/2 θ )*(0,1,0)
        qtnDelta.x=0;
        qtnDelta.y=sin(-DELTA_ANGLE/2.0f);
        qtnDelta.z=0;
        qtnDelta.w=cos(-DELTA_ANGLE/2.0f);
        qtnAttitude*=qtnDelta;
        break;
```

qtnDelta は、1 回のキー入力に対する回転の増分を格納するクオータニオンです。
右矢印キーはヨー回転なので、回転軸は Y 軸ベクトル =(0,1,0) です。回転角は適当に 0.05 ラジアンとし、DELTA_ANGLE として定数定義しています。
これを①式通りに計算して、結果を qtnDelta に格納しています。回転軸が基底軸なので、ここでは Y 成分以外はゼロになりますし、その他のハンドラ内では他の成分以外がゼロになります。

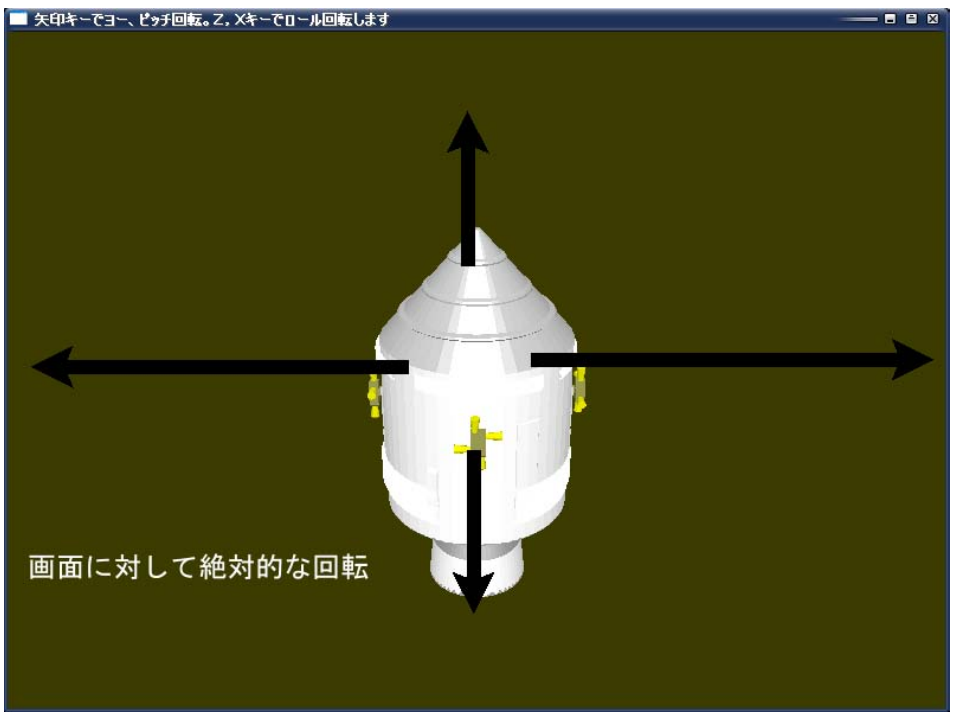
qtnAttitude*=qtnDelta;
増分の回転クォータニオンが求まったら、そのクォータニオンと現在姿勢を保持しているクォータニオン qtnAttitude を掛け合わせます。（姿勢クォータニオンをマスタークォータニオンと言う人もいます。プログラマー用語です）
これが、1 つのハンドラの説明です。その他 5 つのキーメッセージハンドラも同様です。
なお、fYaw-=DELTA_ANGLE;
これは、カルダン角モードのための処理なので、クォータニオンの処理には関係ありません。

Render 関数

ポイントは次の 1 行だけです。
D3DXMatrixRotationQuaternion(&matView,&qtnAttitude);
姿勢クオータニオンを行列に変換しています。レンダリングパイプラインは行列ベースなのでクォータニオンの形で渡すことは出来ないという理由から、この 1 行が必要になります。
本サンプルでは、カルダン角と比較できるようにするためのコードが多いように見えますが、その部分は重要ではないので割愛します。

11-3 オブジェクトの直感的回転

図 11-3 今度のアポロ 13 は操作し易い♪



サンプルプログラム

プロジェクトフォルダ名「ch11-3 自然なオブジェクトの回転（TPV）」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。
使用方法
画面に対して常に横方向の回転（絶対的なヨー回転）：左右矢印キー
画面に対して常に縦方向の回転（絶対的なピッチ回転）：上下矢印キー
画面に対して常に時計回り回転（絶対的なロール回転）：Z キーと X キー
ESC で終了。

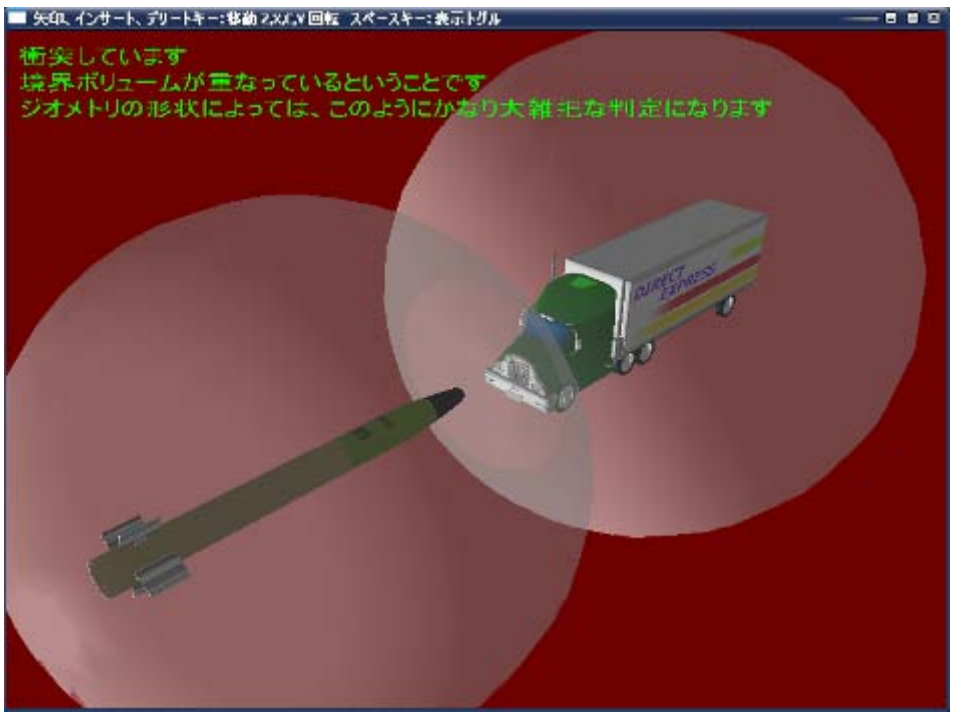
コード解説

ch11-1 カメラの直感的回転（FPV）サンプルと全く同じと言ってもいいでしょう。
3 人称視点になっていることに伴って異なる部分は次のとおりです。

CurrentAttitude 関数
1 人称視点の場合は、カメラのローカル軸を回転軸にしました。今回の場合は、基底軸周りでいいので、より単純に D3DXMatrixRotation 関数を単純に使用します。

12 章　境界球による衝突判定

図 12-1　ボリュームにかなりのデッドスペースがあります。特にミサイルのほうは酷い



3D ゲームの必須処理はレンダリングと衝突判定です。本書では、境界領域（バウンディング・ボリューム）判定と光源（レイ）判定の 2 つのアプローチで衝突判定を見ていきます。境界ボリュームは境界球（バウンディング・スフィア）と境界立方体（バウンディング・ボックス）に分けて解説します。バウンディング・ボックスはさらに、回転しない軸平行バウンディング・ボックス（AABB: Axis Aligned Bounding Box）と回転可能な有向バウンディング・ボックス（OBB: Oriented Bounding Box）に分けて解説します。

コードの解説方針について、ここまでくるともはや、重複しているようなルーチンには一切触れません。たとえば、InitD3d 関数、Render 及び RenderThing 関数などのサンプル間でほとんど変わらない部分です。あまりにも異なる部分が多い場合は別ですが、基本的に把握しているものとして解説を進め、キーポイントとなる関数やルーチンのみを解説していきます。

境界球は、半径と 2 つの物体の距離の「大小関係」だけで衝突を検出することができるため、最も高速な判定が出来ます。次章で解説する軸平行バウンディングボックスも、大小関係で判断できる高速な判定ではありますが、境界球ほどではありません。

他方、有向バウンディングボックスとレイ判定の場合は、単純な大小関係だけでは判断できず、ベクトルの内積と外積及び三角関数を使用することになるので、必然的に乗算や浮動少数点数が絡みます。当然、処理速度は格段に遅くなります。

境界球は非常に高速な手法なのですが、予想を裏切らずやはりそこには “ 処理速度とフィット率 ” のトレードオフがあります。境界球は最も高速であると同時に、もっともフィット率が悪い、言いかえれば「大雑把」な判定でもあります。もし、境界球にトレードオフが当てはまらなければ、境界ボックスはとっくにこの世から姿を消しています。

図 12 － 1 のキャプチャー絵を見れば分かると思いますが、実際のメッシュ形状との間にかなりの隙

間がありフィット率が悪いことが分かります。ただし、ジオメトリ自体が球形に近いような場合は、境界球のフィット率が最大になります。しかしそれはレアなケースであって、運が良いだけのことだと言えるでしょう。

判定原理
球でどのように衝突を判定するのか、その原理はいたってシンプルかつ簡単です。
２つの物体の半径の合計 R1+R2 と、２つの物体間の距離 L の大小を比較するだけです。
２つの物体 A と物体 B があるとして、物体 A の半径 R1、物体 B の半径を R2、そして物体 A と物体 B の中点間の距離を L としたときに、
L>R1+R2 なら、衝突していない
L=R1+R2　あるいは　L<R1+R2 なら、衝突（接触あるいは交差）していると判断できます。

次の３枚の図を見てください。

図 12-2　衝突していない

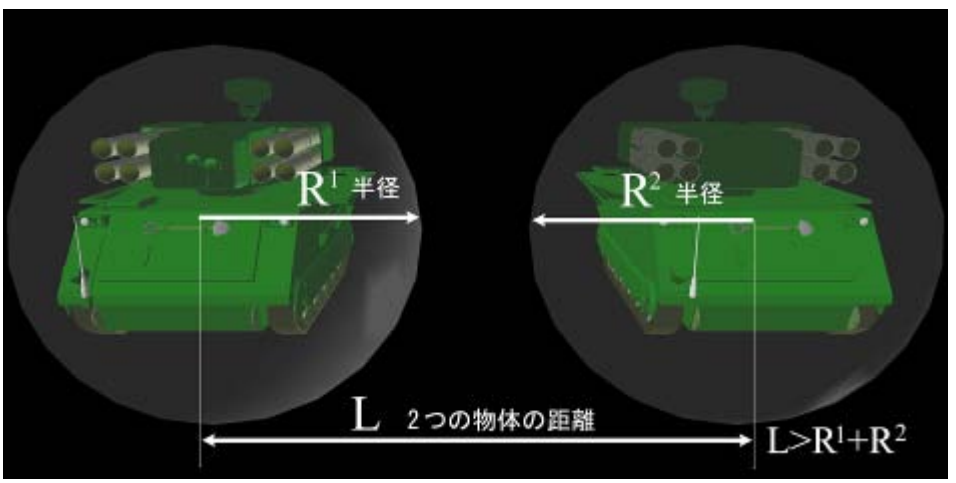


図 12-3　衝突（接触）

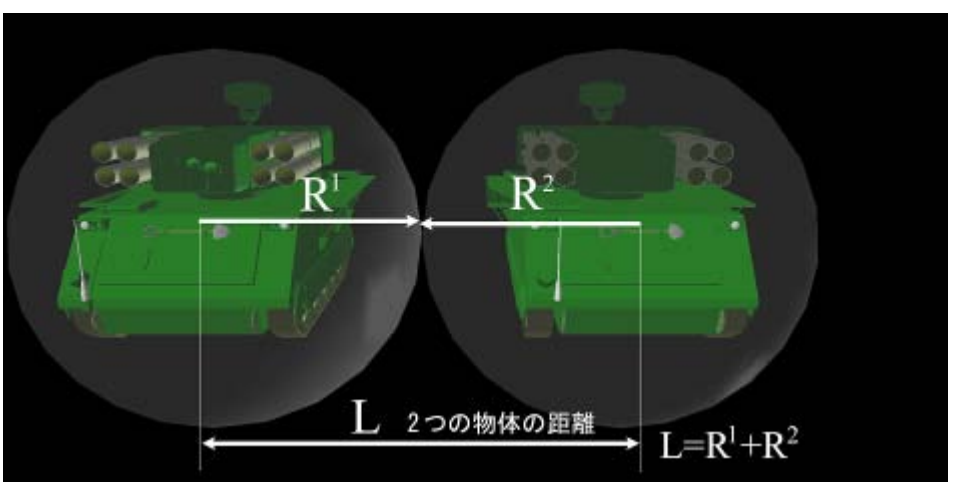
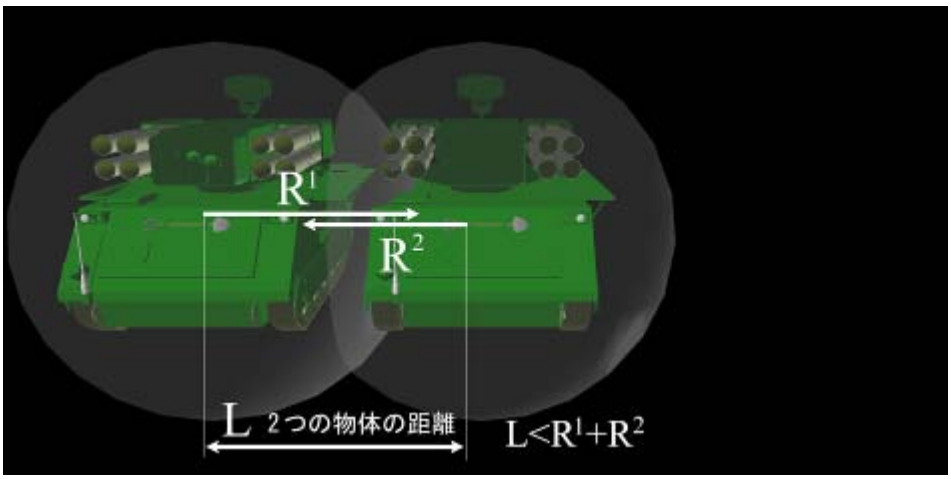


図 12-4　衝突（交差）



球なので、どの半径をとっても同じ長さであること、さらに、中心の距離なのでジオメトリの姿勢の影響を受けないというのも境界球の魅力です。これは姿勢の影響を受ける境界ボックスに対する大きなアドバンテージかもしれません。境界球はそれ自体の高速性に加え、姿勢変化に対する処理が不要であるので全体的な速度向上をもたらします。

サンプルプログラム

プロジェクトフォルダ名「ch12 メッシュと境界球」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
ミサイルを操作します。トラックは動きません。ミサイルとトラックの境界球がぶつかると、メッセージを表示するようになっています。
前進と後進：上下矢印キー
左右の移動：左右矢印キー
上下の移動：インサートキーとデリートキー
ヨー回転：Z キーと X キー
ピッチ回転：C キーと V キー
境界球の表示トグル：スペースキー

コード解説

ではコードを解説していきます。
ほぼ全ての Direct3D サンプルに共通な部分と、本サンプルの本質部分は別ファイルに分けました。
注目するのは、BS.cpp 実装ファイルだけです。

Collision 関数
この関数は、境界球の原理を忠実にコーディングしたものであり、この関数が全てです。
メインループ側（もう一方の cpp ファイル）は、この Collision 関数の結果によって衝突メッセージを表示します。
境界球の原理にそって、半径を足したものと距離を比較するだけなので 12 行しかありません。

あまりにも明確なので解説の必要が無いとは思いますが念のためしておきます。

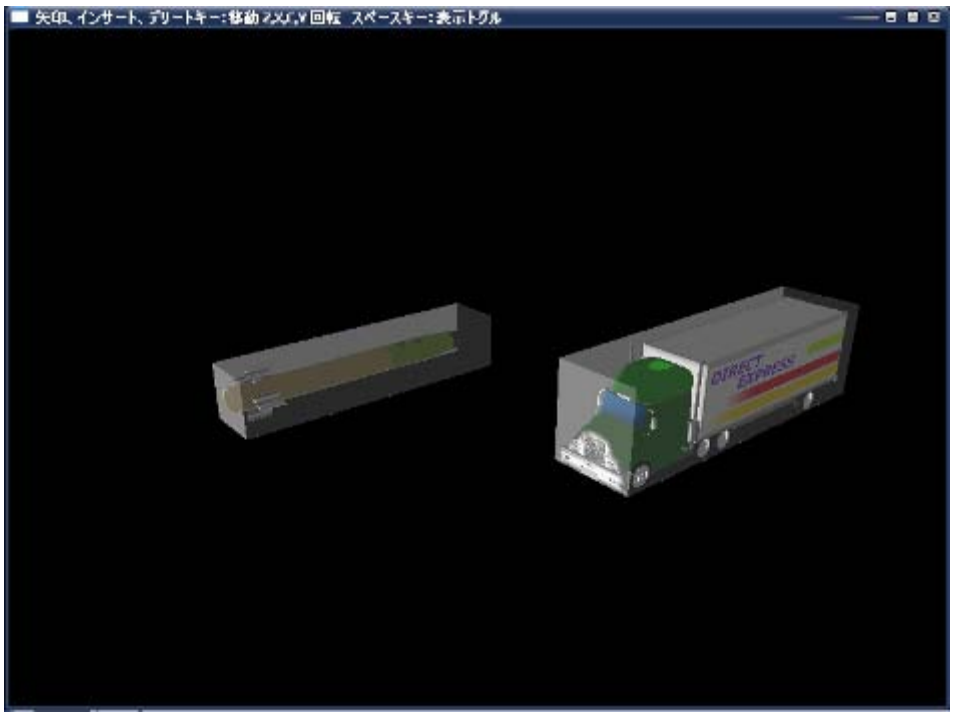
D3DXVECTOR3 vecLength=pThingB->vecPosition-pThingA->vecPosition;
FLOAT fLength=D3DXVec3Length(&vecLength);
２つの物体間の距離（中心の距離）を求め、それを fLength に格納しています。
メッシュの中心（ピボットポイント）は、ジオメトリの中心になるように作成しているので、位置ベクトルをそのまま中心として利用できます。
まず、２つのメッシュの位置ベクトルの差ベクトルを求めます。その差ベクトルの長さが２物体間の距離なので D3DXVec3Length 関数によりさベクトルの長さを得ています。

if(fLength <= pThingA->Sphere.fRadius+pThingB->Sphere.fRadius)
Sphere.fRadius は球の半径です。先ほどの式（L<=R1+R2）と全く同じことを判断しています。
この if 文が真なら、接触あるいは交差しているということなので、TRUE を返します。
この関数は、衝突している場合に TRUE を、離れているときに FALSE を返します。

以上です、境界球は簡単ですよね？

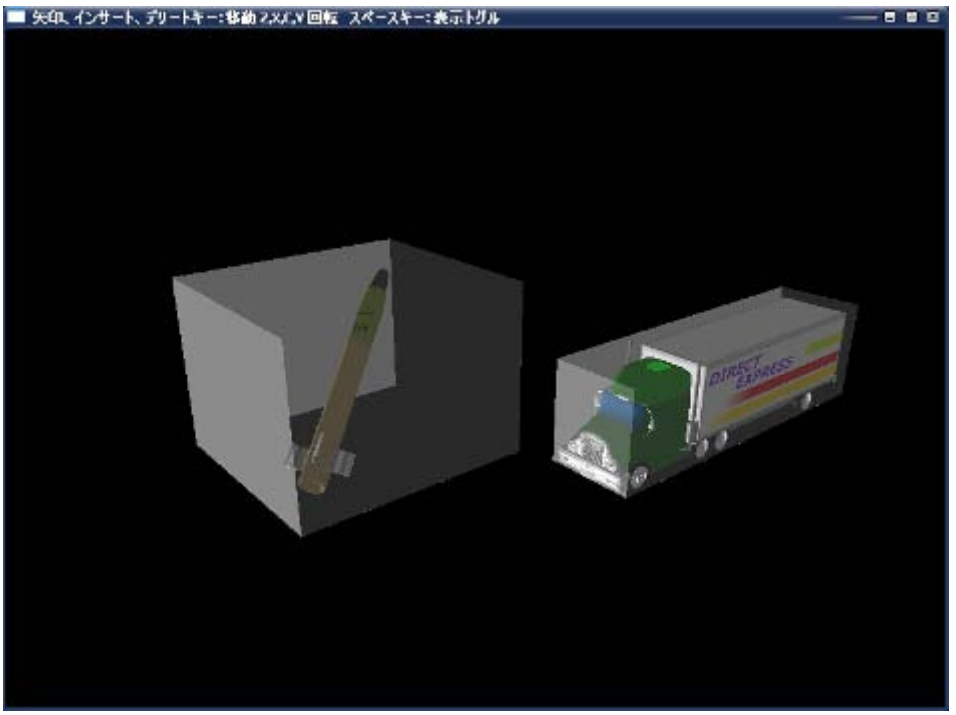
なお、InitSphere 関数は、目でみて分かり易いように境界球をレンダリングするためのものなので本質的には無くてもいい部分です。解説は割愛します。

図 13-1　ボックスが軸と平行な場合には最適



バウンディングボックスは境界球よりフィット率が高いものの、その代償として処理速度は遅く、しかもコーディングはちょっと厄介です。
バウンディングボックスは 2 種類あります。
1 つは、絶対座標軸に平行な軸平行バウンディングボックス、
2 つ目は、ボックス自体が回転する有向バウンディングヴボックスです。
それぞれ軸平行バウンディングボックスは AABB（Axis Aligned BoundingBox アクシス・アラインド・バウンディングボックス）そして有向バウンディングボックスは OBB（Oriented Bounding Box　オリエンティッド・バウンディングボックス）と呼ばれます。
どちらが優れているかは状況に依るので明言できませんが、ちゃんとコーディングした場合には総合的に見て OBB に軍配が上がります。その理由はこれから明らかになります。

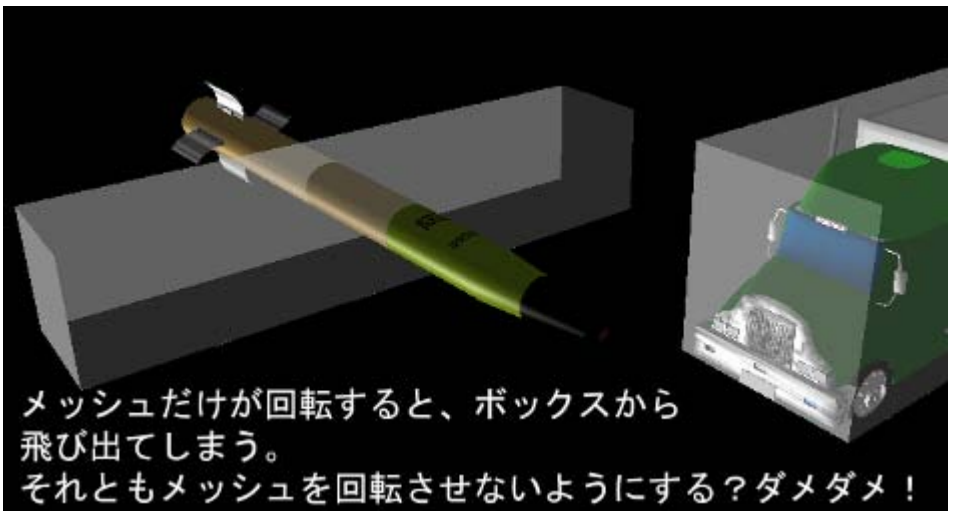
図 13-2　中身が回転するとボックスは大きくなってしまう



境界球の解説では、カギとなる関数は 1 つだけだったので解説が 1 瞬で終わり、楽だったのですが、この AABB（及び OBB）はそうは行きません。
カギとなる関数は 2 つあり、さらにレンダリング関数側でも解説が必要となります。

判定原理
AABB も境界球のように、「大小関係」で判断します。具体的には、物体同士の頂点全ての大小を比較することにより、一方が他方の物体に接触しているか、重なっているか、及び含んでいるかということを判断します。
基底軸に平行ということは、向きが変わらない、姿勢が変わらないということであり、言い換えると回転をしないボックスを意味します。その性質により、OBB に比べ、単純な頂点同士の大小比較のみで接触や交差及び含有をもらさず検出できます。
ただし、AABB には難点があります。ボックス自体は回転しなくとも、メッシュは通常回転します。メッシュが回転しているのにボックスが回転しない場合どうゆうことになるかは予想ができるでしょう。次の図のようになり、まともな判定など出来ません。

図 13-3　中身の回転にボックスをシンクロさせていない場合



そこで、ボックスにもメッシュと同じ回転をかけるアプローチが次項で解説する OBB 有向バウンディングボックスですが、AABB は回転させずにあくまでも向きは一定の平行移動だけのバウンディングボックスですから回転させるわけにはいきません。
では、どうしたらこの問題を解説できるか考えてみましょう。
悪い例
メッシュがどのように回転しても、すっぽり収まるような大きさのボックスを作る。もちろんその条件を満たす範囲で最小のボックスです。回転による頂点の移動を考慮して全ての姿勢に対しての最大頂点と最小頂点を結ぶボックスというわけです。
このようなボックスは、外接円をさらに囲む外接立方体と言えます。実際このようなボックスを作成するとしたら、その理屈に基づいてコードを書くでしょう。
もう気付いたかもしれませんが、外接円はすなわち境界球ですから、これでは境界球よりもさらにフィット率の悪い代物を作ることになります。それなら、最初から境界球を採用すればいいという話になってしまいます。この発想には良い所が全くありません。踏んだり蹴ったり（意味不明）ですね。

サンプルプログラム

プロジェクトフォルダ名「ch13 メッシュと AABB」
VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

解決方法
軸平行を維持するために、回転の度に最大頂点と最小頂点を再計算します。つまり、バウンディングボックスを回転の度に動的に作り直します。バウンディング・ボックスの作成のために、頂点バッファをロックするのはスマートな方法とは言えませんが、軸平行を維持するためには、こうするしかありません。
次の図は本サンプルのキャプチャ画像です。回転に伴い、ボックスが再構築されているのが分かると思います。

図 13-4　中身の回転にボックスをシンクロさせた場合

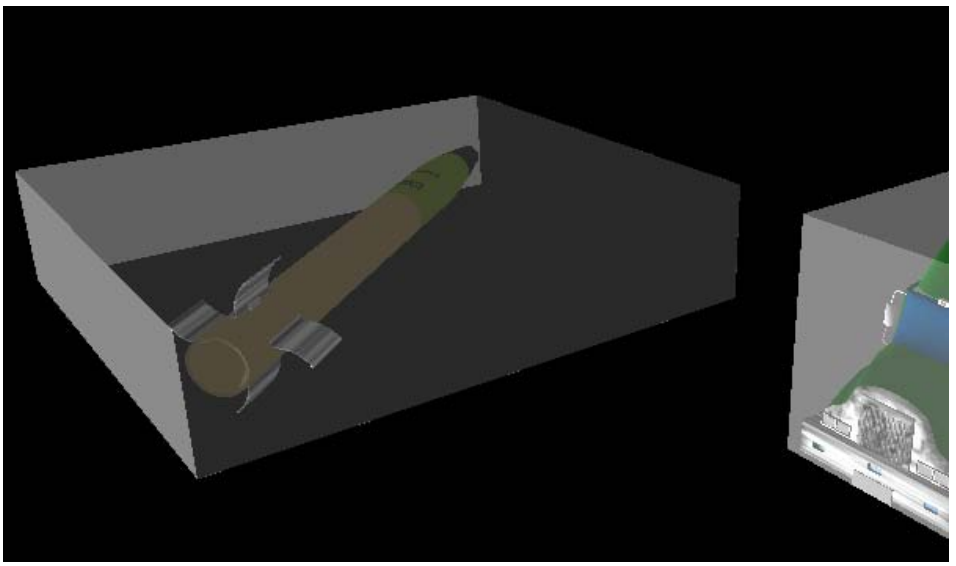
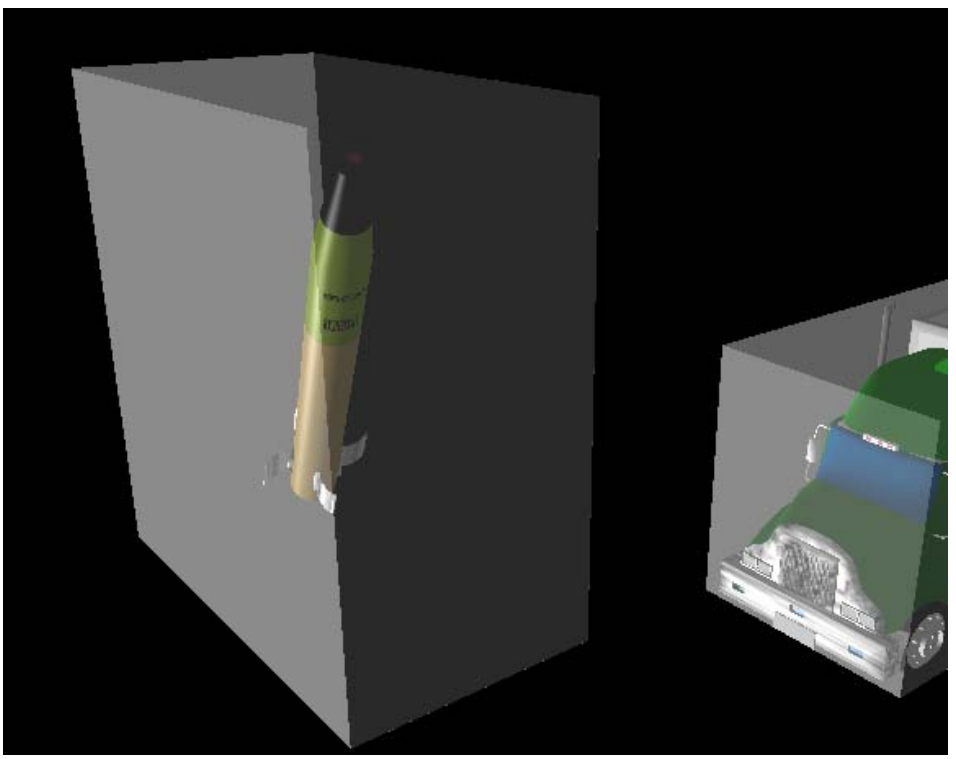


図 13-5　中身の回転にボックスをシンクロさせた場合（2）



フィット率は悪くなりますが、境界球よりは遥かにマシです。
さて、ここで読者は、なぜわざわざそこまでして軸平行にこだわるのか疑問に思うかもしれません。それは、基底軸と平行であるという前提に大きな魅力があるからです。その前提のおかげで判断部分のコードは、ほんの数行の大小判断だけで済みます。AABB の特徴は、判断部分は数行で済むのですが、その判断をお膳立てする “ 平行維持 ” 部分に多くのコードが必要になる点です。
使用方法
ミサイルを操作します。トラックは動きません。ミサイルとトラックの境界ボックスがぶつかると、メッセージを表示するようになっています。
前進と後進：上下矢印キー

左右の移動：左右矢印キー
上下の移動：インサートキーとデリートキー
ヨー回転：Z キーと X キー
ピッチ回転：C キーと V キー
境界ボックスの表示トグル：スペースキー

コード解説

UpdateBBoxMesh 関数
バウンディングボックスの形状を再構築する関数であり、その都度メッシュ内の最大頂点と最小頂点を見つけて、新たなバンディングボックスと確認用のボックスメッシュを作成するという処理を行っています。
具体的には、
① . メッシュの頂点にアクセスするためにメッシュの頂点バッファーをロックする。
②全頂点に現在の回転をかける。
③回転した後は、最大座標と最小座標が別の頂点になる可能性があるため、最大と最小を新たに見つける。
④頂点バッファーをアンロックする。
⑤新たな最大座標と最小座標を保存する。（Thing 構造体に保存しておく）
⑥古いバウンディングボックスを表現するボックスメッシュは破棄して、新たなバウンディングボックスに基づいて新たにボックスメッシュを作成する。

という流れになっています。

バウンディングボックスの計算は全てボックスメッシュの頂点情報を基に行います。
ボックスメッシュ用には 3 つの受け皿を用意しています。1 つは、Thing 構造体の pBBoxMesh、2 つ目は同じく Thing 構造体の pBBoxMeshHold、3 つ目はこの関数内でローカルに用意している pClonedMesh です。pBBoxMesh は常に新しいボックスメッシュデータで更新されます。pBBoxMeshHold は X ファイル読み込み時に作成したボックス形状を最後まで保持し、更新されることはありません。いつでも最初の形状を取り出せるようにしています。pClonedMesh は関数内でローカルに生成される一時的なメッシュ用です。ボックスの形状が動的に変化するので、pBBoxMesh のメッシュデータを更新するのは明らかでしょう。また、一番最初の形状データが無いと上手くいきません。なぜなら、回転行列は最初の姿勢からの回転量を表しているからです。最後 3 つ目の pClonedMesh がなぜ必要かと言うと、関数内で頂点バッファーの全頂点に回転行列を掛けるわけですが、pBBoxMesh や pBBoxMeshHold に直接それを行った場合、それらの頂点を元に戻す必要が出てきてしまうためです。そうしないと、ボタンを 1 回押しただけでボックスの形状が連続的に変化してしまいます。
逆行列を掛けて元に戻すのであれば 3 つ目の pClonedMesh は不要ですが、メッシュのクローンを作成するほうが処理速度的に速いのでそうしました。なお、クローンとは単なるポインターのコピーではなく、メッシュデータを別のメモリ領域にそっくりコピーする文字通りのクローンなので、クローンした後は煮るなり焼くなり自由に弄ることができます。

dwVertexAmount=pThing->pBBoxMeshHold->GetNumVertices();
ボックスメッシュの頂点数を得ます。

dwStride=D3DXGetFVFVertexSize(pThing->pBBoxMeshHold->GetFVF());
ボックスメッシュの頂点ストライドを得ます。ストライドとは日本語で「歩幅」を意味し、プログラム的には、なんらかのデータ間のバイト単位の距離です。この場合はメモリ上にある頂点データと頂点データの間の距離（バイト数）、平たく言えば、1 つの頂点のバイト数です。

pThing->pBBoxMeshHold->CloneMeshFVF(D3DXMESH_MANAGED,pThing->pBBoxMeshHold->GetFVF(),pDevice,&pClonedMesh);
最初のボックス形状 pBBoxMeshHold のデータをクローンし、pClonedMesh がそれを指すようにします。この時点で、ボックスメッシュデータは 3 つ存在することになります。
1 初期の形状、 2 現在の形状、 3 初期の形状のクローン
関数内では 3 番目のクローンボックスメッシュに回転を掛け、新たなボックスに必要な最大と最小を抽出します。

① . メッシュの頂点にアクセスするためにメッシュの頂点バッファーをロックする。
```
if(FAILED(pClonedMesh->GetVertexBuffer( &pVB )))
{
        return E_FAIL;
}
if(FAILED(pVB->Lock( 0, 0, (VOID**)&pVertices, 0 )))
{
    SAFE_RELEASE( pVB );
    return E_FAIL;
}
```
クローンメッシュには、初期状態のボックスメッシュデータと同じものが入っています。その頂点バッファーをロックします。この時点で pVertices は頂点バッファーを指し示します。

②全頂点に現在の回転をかける。
```
for(DWORD i=0;i<dwVertexAmount * dwStride;i+=dwStride)
{
    D3DXVec3TransformCoord( (D3DXVECTOR3*)&pVertices[i],(D3DXVECTOR3*)&pVertices[i],&pThing->matRotation);
}
```
初期状態のボックスメッシュを現在の回転行列で回転させます。メッシュを回線させることと全頂点に回転行列を掛けることは同じです。
ループ 1 回転の増分は 1 ストライド分つまり 1 頂点分、ループカウンタの上限は頂点数×ストライド分となります。

③回転した後は、最大座標と最小座標が別の頂点になる可能性があるため、最大と最小を新たに見つけます。
```
if(FAILED(D3DXComputeBoundingBox( (D3DXVECTOR3*)pVertices, dwVertexAmount,
                    dwStride, &vecMin,&vecMax )))
```

X ファイルを読み込んだ直後にバウンディングボックスを作成したのと同じように、新たにバウンディングボックスを作成します。ただし、回転を掛けた後の頂点群を渡しているので、それに見合った形状になります。

④頂点バッファーをアンロックする。
pVB->Unlock();
SAFE_RELEASE(pVB);
頂点バッファーをアンロックして、開放します。

⑤新たな最大座標と最小座標を保存する。（（ロケットメッシュの）Thing 構造体に保存しておく）
pThing->BBox.vecMax=vecMax;
pThing->BBox.vecMin=vecMin;
本来の判定関数（AABBCollisionDetection 関数内）で使用するので、最大頂点と最小頂点を Thing に保存します。

⑥古いバウンディングボックスを表現するボックスメッシュは破棄して、新たなバウンディングボックスに基づいて新たにボックスメッシュを作成する。
SAFE_RELEASE(pClonedMesh);
pCloneMesh を生成した理由は、頂点に変更を加えるため、および、変更した頂点を元に戻す手間を省くためでした。その目的はすでに達成されているこの段階ではもはや不要なので、ここで開放します。

SAFE_RELEASE(pThing->pBBoxMesh);
この段階ではまだ pBBoxMesh は古い（更新前）ボックスメッシュのポインターです。新しいメッシュデータを作成する前に古いメッシュを開放します。新しいメッシュのポインターになってしまった後では古いメッシュのメモリ位置が分からなくなるので、このタイミング、つまり、古いメッシュのメモリ位置が分かっているうちに開放しておきます。
hr=D3DXCreateBox(pDevice,vecMax.x*2,vecMax.y*2,vecMax.z*2,&pThing->pBBoxMesh,NULL);
開放したあとに、新たなボックスメッシュを作成し、pBBoxMesh がそのメモリを指すようにします。

AABBCollisionDetection 関数
基底軸と平行であるため、衝突は頂点の大小関係だけでもらさず検出できることが保障されます。もし仮に、ボックスがちょっとでも傾いていれば（基底軸に平行でなければ）頂点と辺の両方について吟味する必要が出てきてしまいます。頂点同士では接触・交差していなくても辺同士が接触・交差している場合もあるからです。そのような判定は次項の OBB で行いますが、AABB にはその必要がありません。そのために苦労して平行を保つようにしたのですから。

```
D3DXVECTOR3 vecMaxA,vecMinA,vecMaxB,vecMinB;
vecMaxA=pThingA->BBox.vecMax+pThingA->vecPosition;
vecMinA=pThingA->BBox.vecMin+pThingA->vecPosition;
vecMaxB=pThingB->BBox.vecMax+pThingB->vecPosition;
vecMinB=pThingB->BBox.vecMin+pThingB->vecPosition;
if(vecMinA.x < vecMaxB.x && vecMaxA.x > vecMinB.x && vecMinA.y < vecMaxB.y
&& vecMaxA.y > vecMinB.y && vecMinA.z < vecMaxB.z && vecMaxA.z > vecMinB.z)
```

コードは 7 行ありますが、判定部分は本質的に最後の 2 行だけです。上 5 行部分は、保存してあった最大・最小頂点を短い名前の変数にしているだけなので、省略できる部分です。コードが見辛くなるので、vecMaxA 等を用意しただけの話です。

ここまででが重要な部分の解説です。これ以降の解説は、ボックスメッシュのレンダリング関係なので、さほど重要ではありませんが、AABB であるが故に若干の工夫をしているので解説しておきます。

RenderThing 関数内のボックスメッシュレンダリング部分
pDevice->SetTexture(0,NULL);
処理がここまで来る前に、何らかのテクスチャーが設定されていると、ボックスにテクスチャが張られてしまうので、テクスチャを解除します。

UpdateBBoxMesh(pDevice,pThing);
既に解説した関数です。

D3DXMATRIX matBBoxWorld;
D3DXMatrixIdentity(&matBBoxWorld);
matBBoxWorld._41=pThing->matWorld._41;
matBBoxWorld._42=pThing->matWorld._42;
matBBoxWorld._43=pThing->matWorld._43;
pDevice->SetTransform(D3DTS_WORLD, &matBBoxWorld);
ボックスの位置を Thing（ミサイルメッシュやトラックメッシュ）の位置にシンクロさせます。
ワールド変換行列というか 44 同次行列の 4 行目ラインは位置ベクトルであったことを思い出してください。それぞれの Thing は自身のワールド変換行列を保持しているので、その行列から位置成分だけを取り出します。もしも単純に Thing のワールド行列そのままでボックスをレンダリングしてしまうと、見かけ上ボックスが回転してレンダリングされてしまいます。回転したようにレンダリングされるだけであって、実際のバウンディングボックス形状に影響はありませんが、そもそも確認のためにボックスをレンダリングしているのですから、意味が無くなってしまいます。
pDevice->SetMaterial(pThing->pBBoxMeshMaterials);
ボックスのマテリアルを設定。
pThing->pBBoxMesh->DrawSubset(0);
ボックスをレンダリング。

14 章　有向境界ボックス（OBB）による衝突判定

図 14-1　頂点と辺が他方のボックス内に含まれている

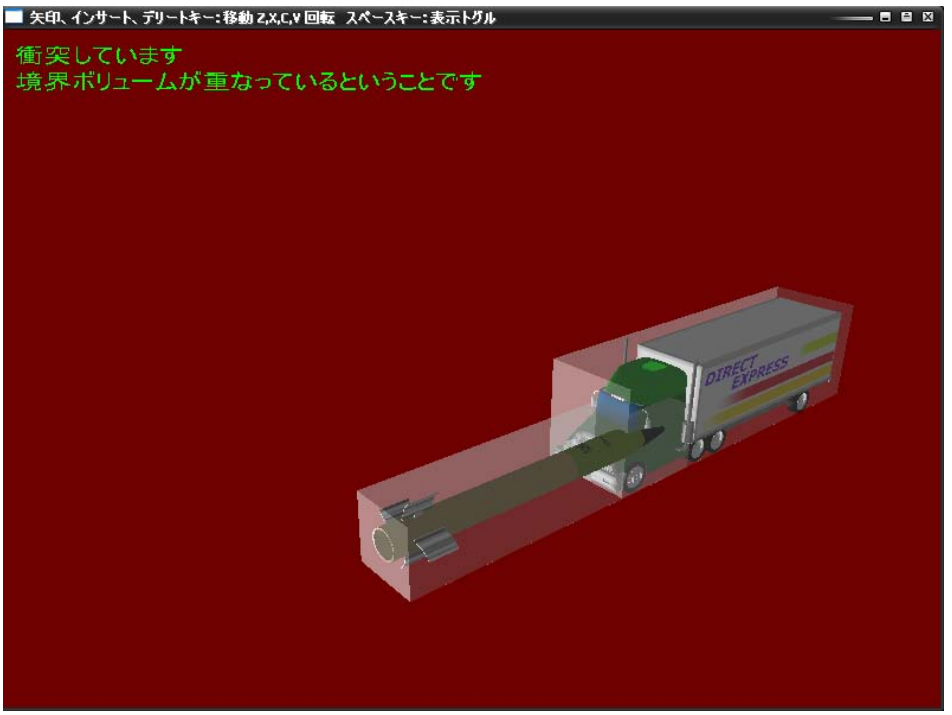


図 14-2　辺が他方のボックスと交差している

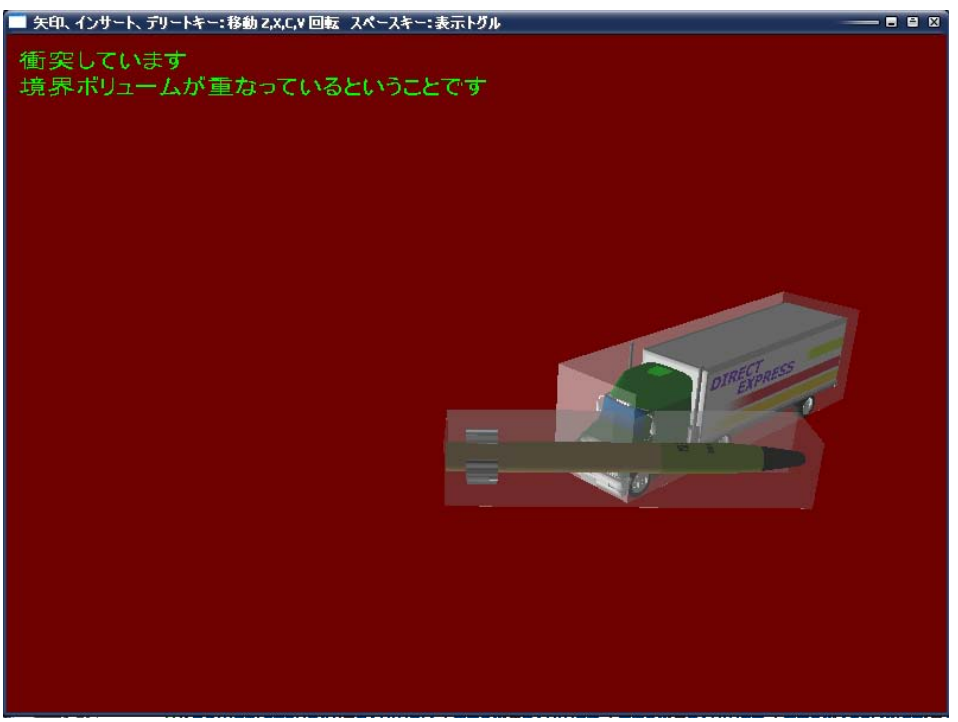


図 14-3　辺が他方のボックスと交差している（2）

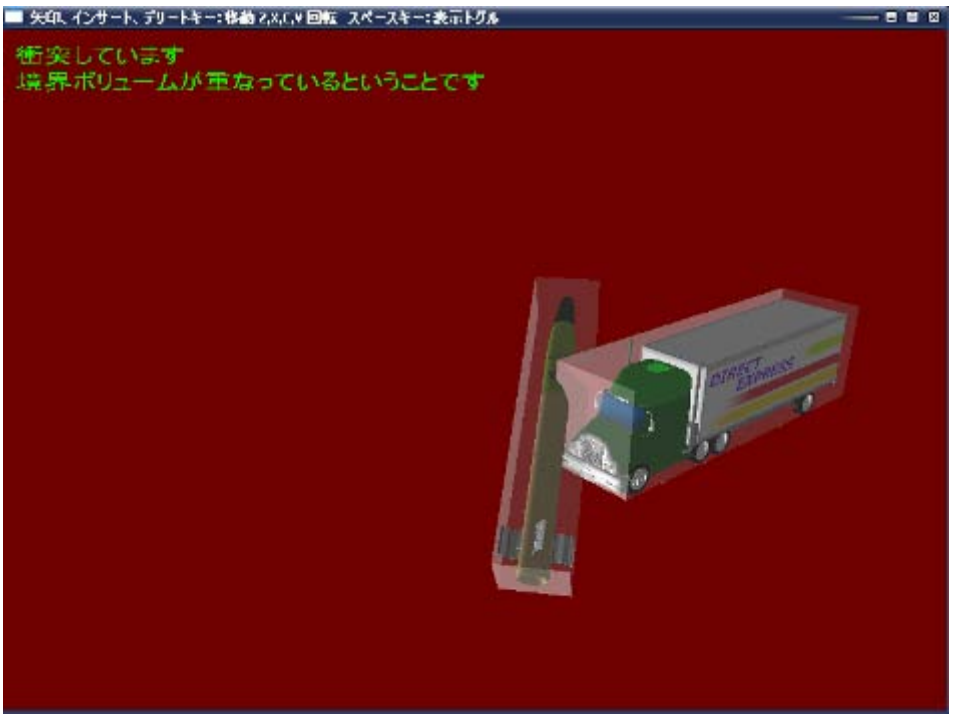
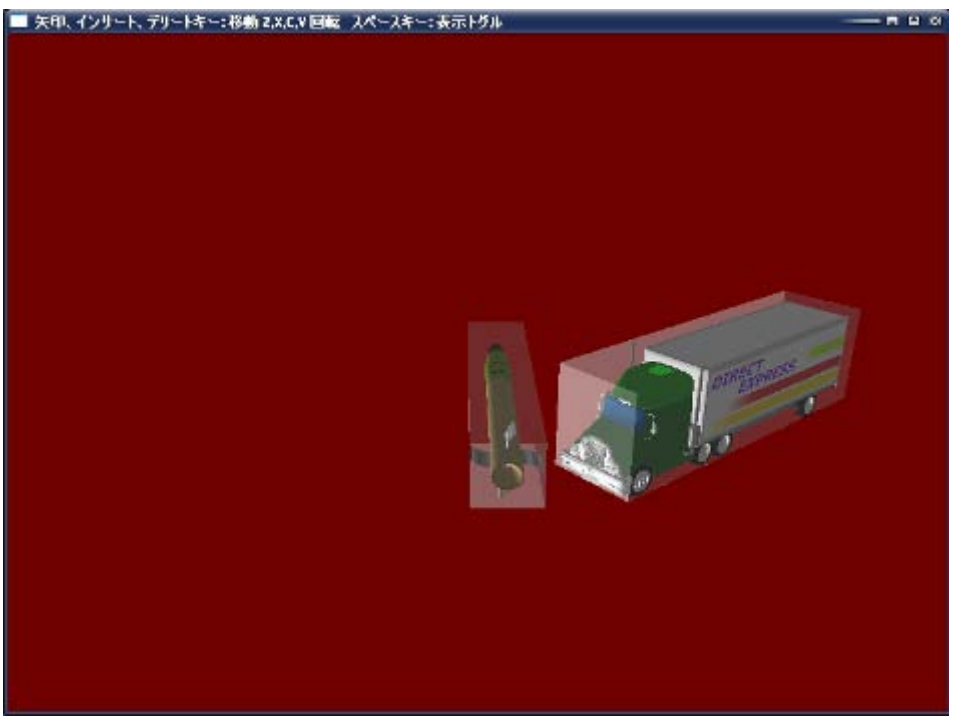


図 14-4　衝突していない。かなり見た目と近い判定ができる



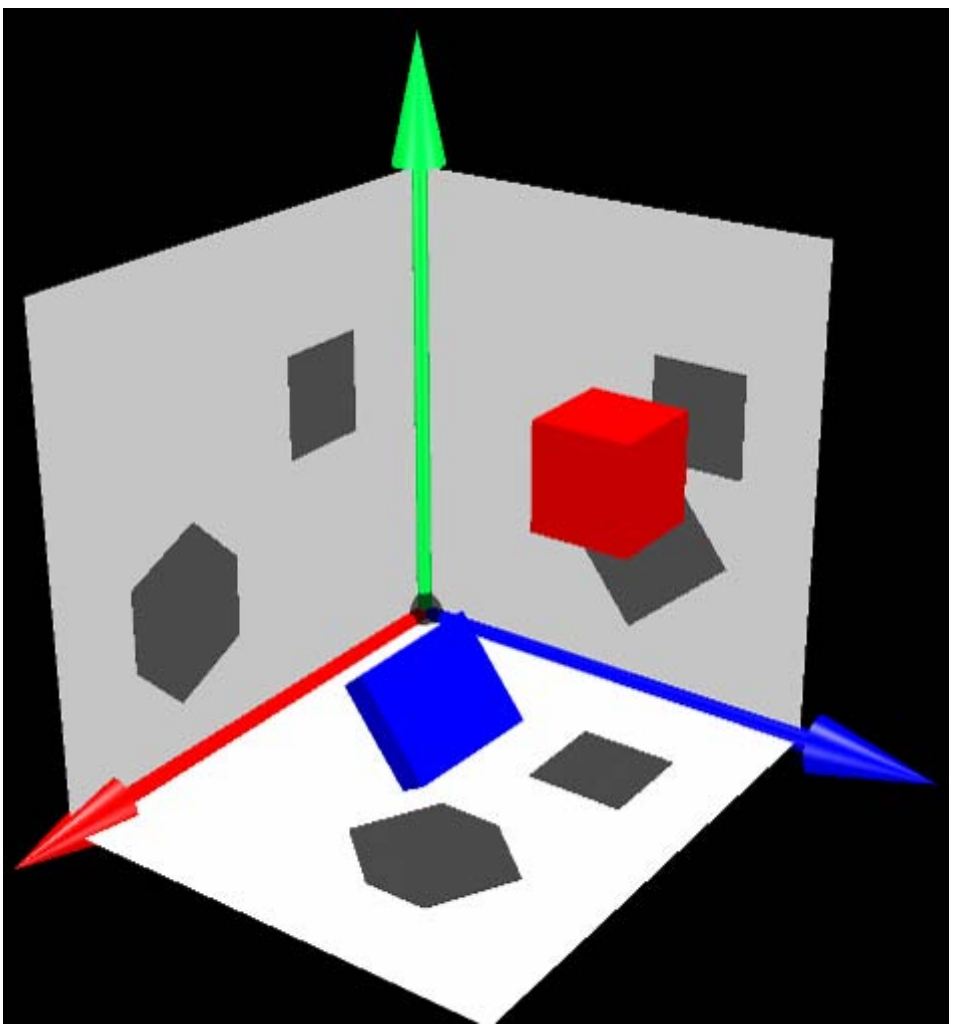
ここで解説する OBB による衝突検出の原理は筆者が考えたものではありません。ACM（Association for Computing Machinery）という世界で最も権威あるアメリカの計算機協会が主催する、これも最も権威のある SIGGRAPH（Special Interest Group on computer GRAPHics：「シーグラフ」と発音）という CG 分化会において 1996 年に発表された論文を参考にさせていただき、その原理を基にアルゴリズムを設計しました。（もちろんソースコードもフルスクラッチしました）

判定原理

原理を簡潔に述べると、ボックスの“影”に着目して、その影同士の交差の有無で判定しようというものです。“影”が離れていればボックスが離れている、つまり衝突していないということになります。ボックスの影とは、3次元のボックスを1次元に投影したものを意味します。OBBに限らず3次元で考えるより2次元のほうがより考え易くなるのは経験的に納得できるでしょう。さらに2次元を1次元（つまり線）で考えると位置と長さだけで判断できるので極めて明瞭になります。

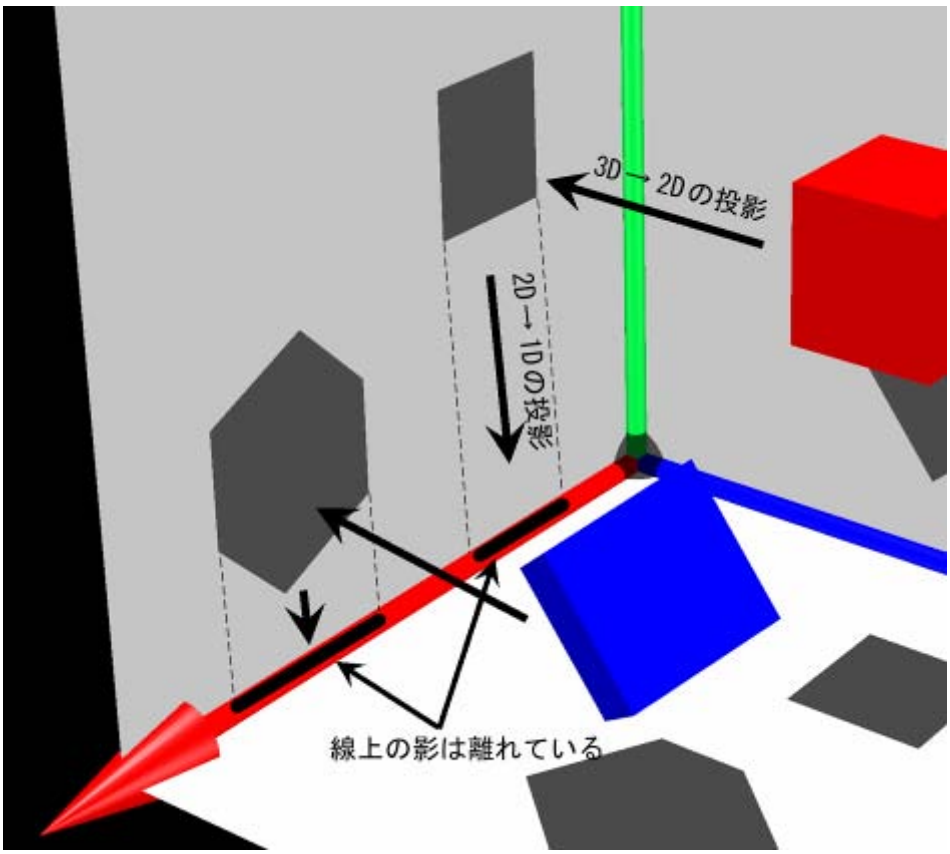
影を求めるということは、投影することです。読んで字の如く“影”を“投げる”のが投影あるいは射影です。3次元を2次元に投影したときの影は次の図のように文字通り“影”です。

図 14-5　ボックスの影。ある次元での影とは、1つ低い次元への投影である

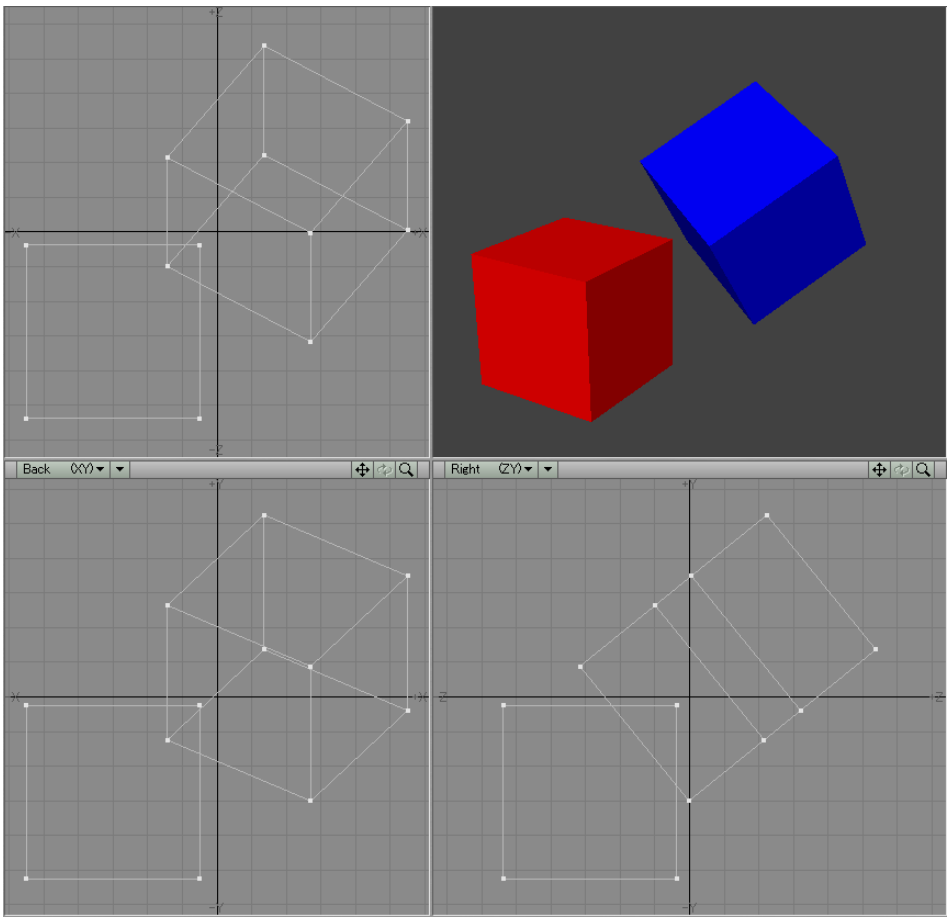


そして、OBB判定で使用（判定の基に）するのは1次元上の影です。1次元なので影は線になります。イメージとしては次の図のようになります。

図 14-6　最終的に1Dの影まで投影した様子



1 次元と言ってもどこの 1 次元に（どこの線に）投影するのかという疑問が浮かぶでしょう。具体的には、ボックス A とボックス B があったとして、
まず、A のローカルな X 軸、Y 軸、Z 軸上に投影します。（3 本）
次に、同じく B のローカルな X 軸、Y 軸、Z 軸上に投影します。（3 本）
そして､A と B のローカル軸ベクトル同士の外積ベクトル上に投影します（3 と 3 の組み合わせ =9 本）
3 本 +3 本 +9 本 =15 本の線上に投影します。そして、15 本のうち一本でも影が離れていれば、衝突していないという判断が出来ます。
ここまでくると頭の中で正確にイメージするのは少々困難かもしれません。ローカル軸への投影まではなんとか想像できるものの、外積ベクトルへの投影となると腰が引ける人も多いのではないでしょうか。
筆者も先の論文を見るまでは、ローカル軸への投影まではアイディアが出たのですが、その理論が正しいのかどうなのか証明することができず悩んでいました。結論を言うと、軸ベクトルへの投影しか行わないその理論は不完全なものでした。衝突していないのに衝突と判断してしまう理論上の穴があったのです。その穴を埋めてくれたのが「外積ベクトル」への投影という発想です。実にうまいことを考える人がいるものです。なぜ外積ベクトルが必要なのかは、次の図で明らかになります。
図 14-7　三面透視図だけでは検出できない “ 隙間 “



この図はライトウェーブ（モデラー）のキャプチャー画像です。左上、左下、右下の各ウィンドウはそれぞれ XZ 平面、XY 平面、YZ 平面への投影図であり、これらはすなわち、ボックス A とボックス B のローカル軸が作る平面への投影と同じ像、すなわちローカル軸 6 本への投影を表しています。それらを見る限り、どれも交差しているので、衝突有りと判断してしまったのが当初の不完全な理論です。ところが、右上のプレビューウィンドウを見てください。2 つのボックスには隙間が存在し、”衝突していません”。
このことから、ローカル軸 6 本への投影だけでは足りないということが分かります。この隙間は 2 つのボックスの辺ベクトル同士の外積ベクトルへの投影でのみ検出されます。これが、全部で 15 本の線が必要な理由です。
なお、このライトウェーブ用のオブジェクトファイルは付属ディスクに収録しているので、ライトウェーブを持っている読者は確認してみてください。

サンプルプログラム

プロジェクトフォルダ名「ch14 メッシュと OBB」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
ミサイルを操作します。トラックは動きません。ミサイルとトラックの境界ボックスがぶつかると、メッセージを表示するようになっています。

前進と後進：上下矢印キー
左右の移動：左右矢印キー
上下の移動：インサートキーとデリートキー
ヨー回転：Z キーと X キー
ピッチ回転：C キーと V キー
境界ボックスの表示トグル：スペースキー

コード解説

原理の解説中は“15 本の線”と表現しましたが、これからは、15 本の分離軸（Separate Axis）と呼びます。影の分離を検出するのが目的なので“分離”軸というわけです。分離軸上に 2 つの物体両方の影を落として“影がくっついているか、分離しているか”で判定をしていきます。 1 つでも分離していれば衝突無しとしてその場で判定は終了します。
注目すべき部分は、OBBCollisionDetection 関数と CompareLength 関数だけです。

OBBCollisionDetection 関数
解説の便宜上、pThingA を物体 A、pThingB を物体 B と呼び、それぞれのバウンディングボックスをボックス A、ボックス B と呼ぶことにします。

D3DXVECTOR3 vecDistance = pThingB->vecPosition - pThingA->vecPosition;
物体間の距離を求めて vecDistance に入れておきます。

D3DXVECTOR3 vecSeparate;
分離軸のインスタンスを作ります。コード上は外積ベクトル 9 本部分にしか使用していません。

pThingA->BBox.vecAxisX=D3DXVECTOR3(1,0,0);
pThingA->BBox.vecAxisY=D3DXVECTOR3(0,1,0);
pThingA->BBox.vecAxisZ=D3DXVECTOR3(0,0,1);
ボックス A のローカル軸ベクトルを、まずは基底ベクトルとして普通に初期化します。

pThingB->BBox.vecAxisX=D3DXVECTOR3(1,0,0);
pThingB->BBox.vecAxisY=D3DXVECTOR3(0,1,0);
pThingB->BBox.vecAxisZ=D3DXVECTOR3(0,0,1);
同様にボックス B のローカル軸も初期化しておきます。

D3DXVec3TransformCoord(&pThingA->BBox.vecAxisX,&pThingA->BBox.vecAxisX,&pThingA->matRotation);
D3DXVec3TransformCoord(&pThingA->BBox.vecAxisY,&pThingA->BBox.vecAxisY,&pThingA->matRotation);
D3DXVec3TransformCoord(&pThingA->BBox.vecAxisZ,&pThingA->BBox.vecAxisZ,&pThingA->matRotation);
物体 A のローカル軸に物体 A の現在姿勢で回転を掛けます。

D3DXVec3TransformCoord(&pThingB->BBox.vecAxisX,&pThingB->BBox.vecAxisX,&pThingB->matRotation);
D3DXVec3TransformCoord(&pThingB->BBox.vecAxisY,&pThingB->BBox.vecAxisY,&pThingB->matRotation);
D3DXVec3TransformCoord(&pThingB->BBox.vecAxisZ,&pThingB->BBox.vecAxisZ,&pThingB->matRotation);
同様に物体 B のローカル軸も回転させます。

if(!CompareLength(&pThingA->BBox,&pThingB->BBox,&pThingA->BBox.vecAxisX,&vecDistance)) return FALSE;
ボックス A のローカル X 軸に影を落として、影が分離しているかどうかを CompareLength 関数により判断しています。CompareLength 関数は、2 つの物体のバウンディングボックスと分離軸（この場合はローカル X 軸）及び物体間の距離を受け取り、それらを元に影を算出し、影がくっついている場合は TRUE を、離れている場合は FALSE を返します。関数内部は後で詳しく解説します。
ローカル Y 軸、ローカル Z 軸についても同様に判定します。
これでボックス A を基準とした 3 本の判定が終了です。
ボックス B についても全く同様に、3 本の判定を行います。
この時点で 6 本の分離軸の判定が終了します。

D3DXVec3Cross(&vecSeparate,&pThingA->BBox.vecAxisX,&pThingB->BBox.vecAxisX);
if(!CompareLength(&pThingA->BBox,&pThingB->BBox,&vecSeparate,&vecDistance)) return FALSE;
外積ベクトルを分離軸とした判定に入ります。この判定を 9 本分行っています。
この場合は、ボックス A のローカル X 軸とボックス B のローカル X 軸の外積ベクトルを分離軸としていることが分かります。そして、それぞれの 3 本の軸を組み合わせると全部で 9 パターンになることも明らかでしょう。以降の 8 本分も組み合わせが異なるだけです。

これら合計 15 回の判定のうち、どれか 1 回でも CompareLength が FALSE を返した場合は、影が離れているということなので、非衝突としてすぐに戻ります。これから分かるようにこのアルゴリズムは衝突を積極的に検出するものではなく、“ 衝突していないこと ” を検出するものです。平たく言えば、“ 隙間 ” を見つけるアルゴリズムであり、一本でも隙間があれば衝突しているわけがないということです。

CompareLength 関数
FLOAT fDistance=fabsf(D3DXVec3Dot(pvecDistance, pvecSeparate));
OBBCollisionDetection 関数のほうで最初に物体間の距離ベクトルを求めましたが、その距離には少々手を加える必要があります。なぜなら、影だけ分離軸上に投影して、それと比較する距離ベクトルのほうをそのまま比較しても正確な結果は得られないからです。両者を同じ分離軸という次元で比較しなければ意味がありません。
このコードは距離ベクトルを分離軸上に投影しています。今まで、投影、投影と言ってきましたが、投影とは内積に他なりません。（内積の解説を思い出してください）
距離ベクトル pvecDistance と分離軸ベクトル pvecSeparate の内積をとり、その値の絶対値が、分離軸上に投影した後の距離です。
距離のほうはこれで OK です。残るは比較するもう一方、2 つのボックスの影です。

FLOAT fShadowA=0;
FLOAT fShadowB=0;
ボックス A とボックス B それぞれの影の長さを格納する変数を用意します。

fShadowA = fabsf(D3DXVec3Dot(&pboxA->vecAxisX, pvecSeparate) * pboxA->fLengthX)+
fabsf(D3DXVec3Dot(&pboxA->vecAxisY, pvecSeparate) * pboxA->fLengthY)+
fabsf(D3DXVec3Dot(&pboxA->vecAxisZ, pvecSeparate) * pboxA->fLengthZ);
1 つの軸に 3 軸分の長さを足し合わせているのは、回転している場合、どのような組み合わせも考えられるからです。各軸との内積をとっているので、たとえば軸に垂直な場合は長さがゼロになってカウントされないので心配はありません。
同様にボックス B の影を算出します。

if(fDistance > fShadowA + fShadowB)
分離軸上での距離が影の合計より大きいということは、2 つの影が離れていることを意味します。

15 章　境界ボリュームとフィッティング

単一のボリュームをジオメトリの形状に合わせること、あるいは、複数のボリュームでジオメトリを（なるべく効率的に）隙間無く覆うことを“フィッティング（Fitting）”と言い、後者の場合はフィリング（Filling）とも言えます。
球、ボックス、有向ボックス、それぞれのフィッティングはどんな特徴があるのか、結論を先に言うと（おそらく読者の予想通り）球が最もフィッティングが難しく、次いでボックス、一番優れているのが有向ボックスとなります。

次の図のようなジオメトリの場合は運の良い場合で、たまたま球のフィット率が高く、むしろ、球フィット率が最高となります。

図 15-1

図 15-2

図 15-3

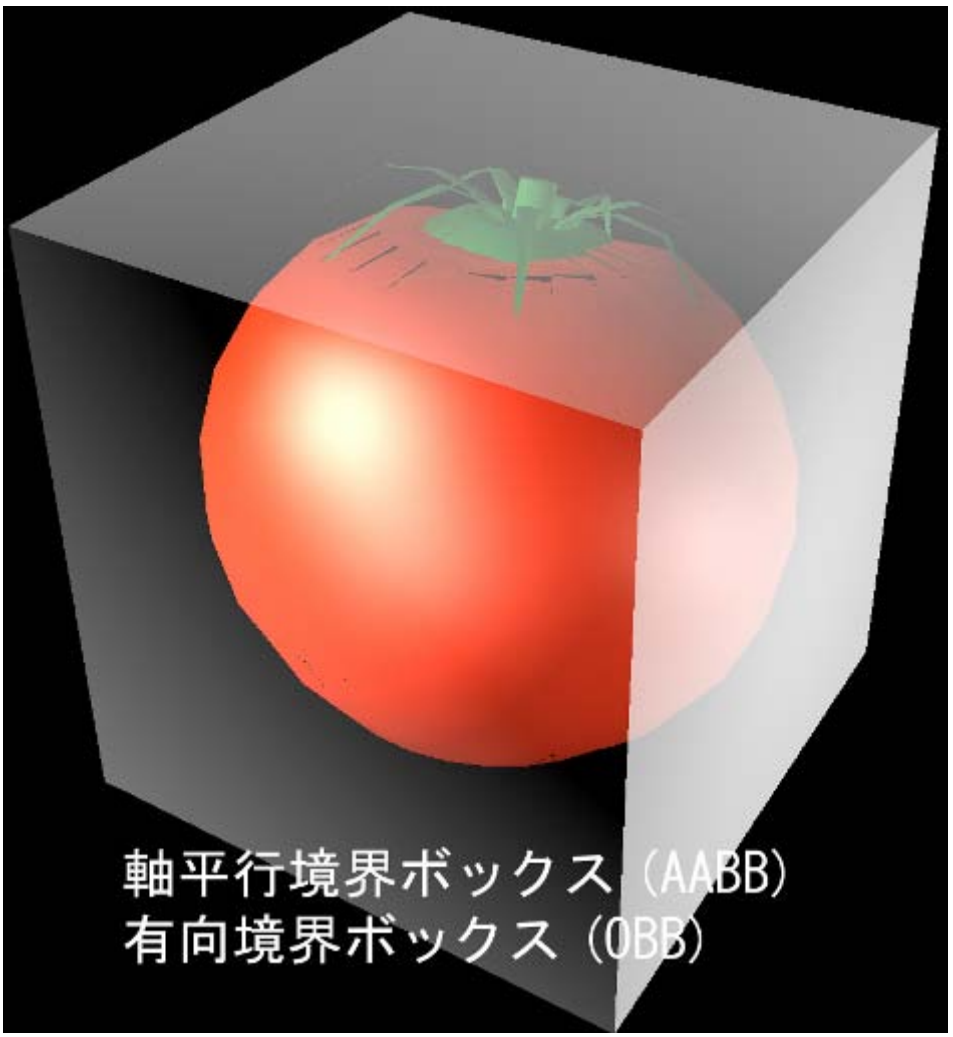


図 15-4

図 15-5

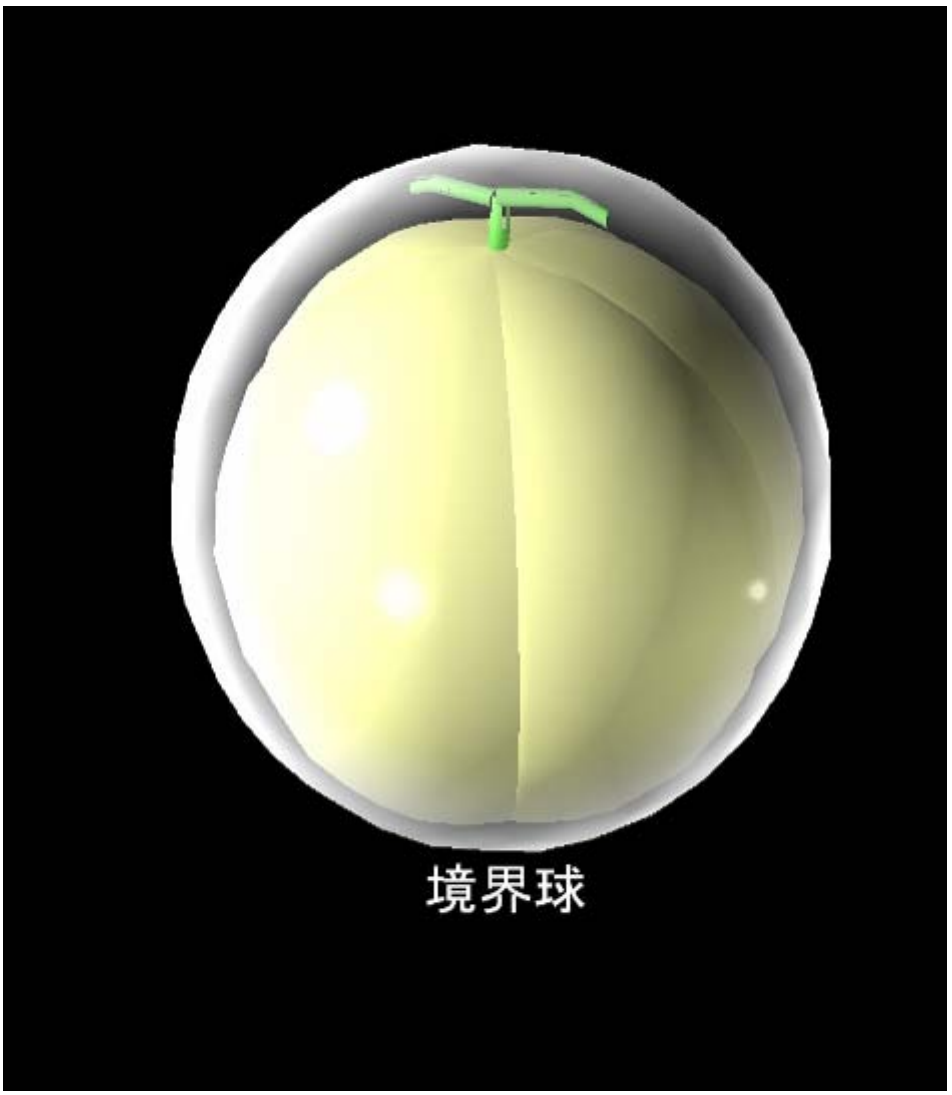
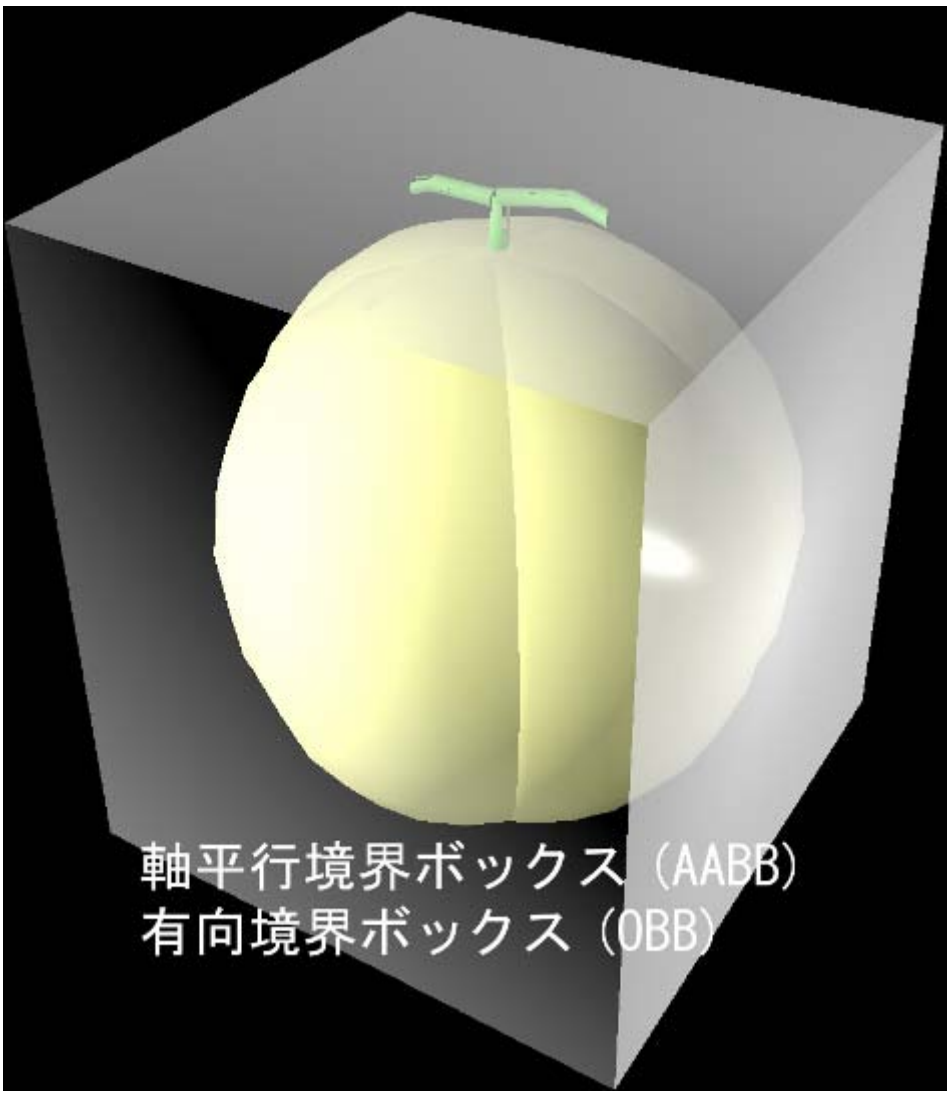
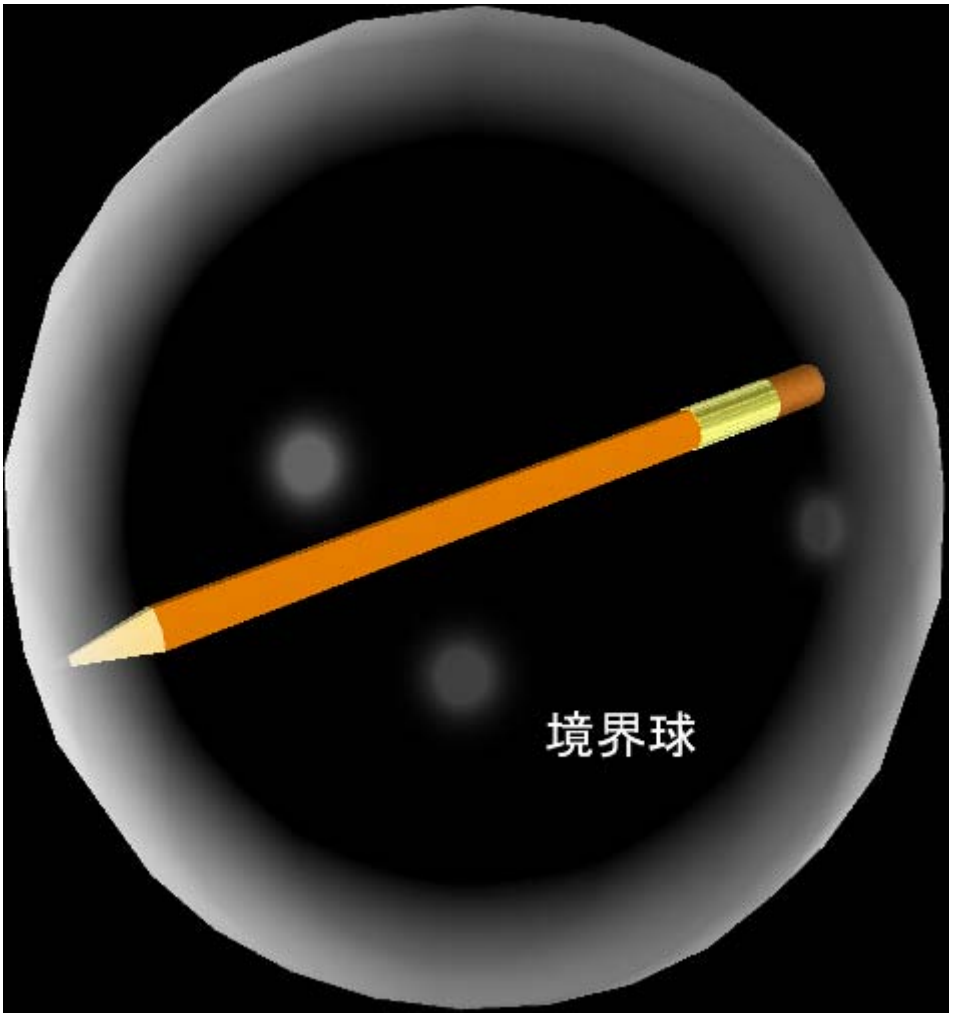


図 15-6



しかし、常に運が良いわけはなく、一般的に球は無駄な空間が生まれてしまいます。
例えば、次の鉛筆ジオメトリを考えます。
図 15-7

図 15-8



そのような場合、ボリュームを複数敷き詰めてデッドスペースを削減します。その場合でも球はボックスに比べて効率が悪く、結局、ボックスよりも多くの個数を必要としてしまいます。
図 15-9

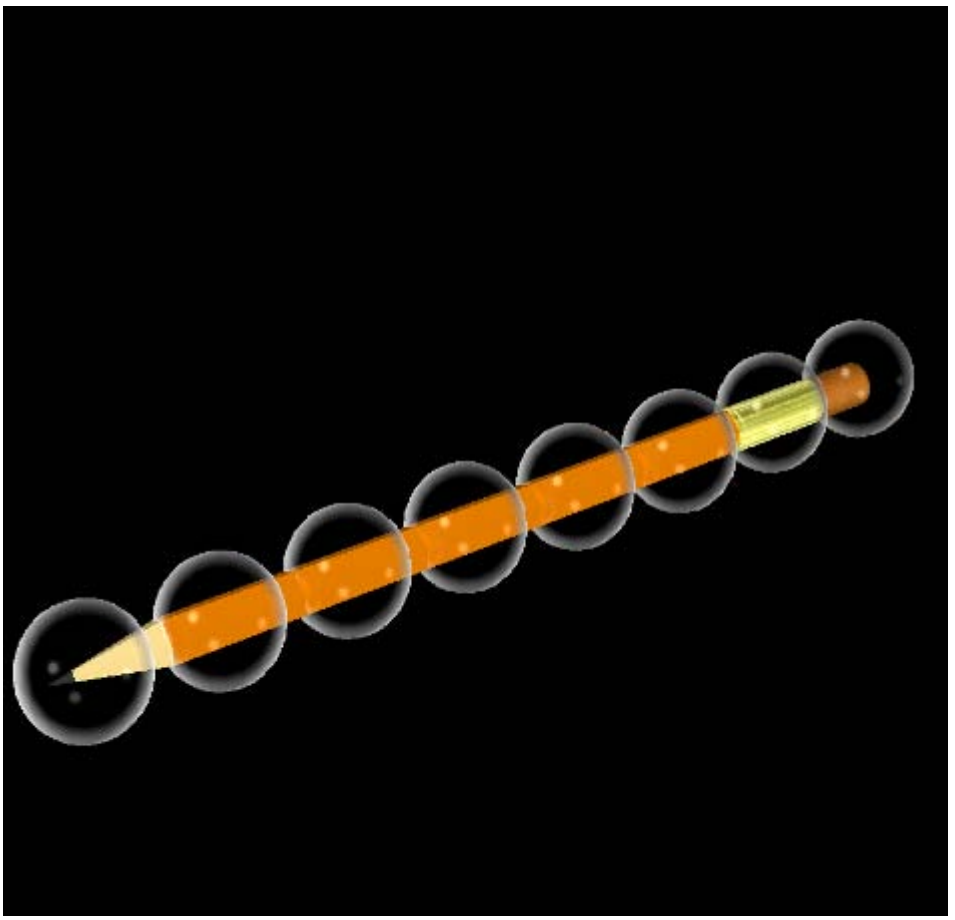


図 15-10

図 15-11

次の図からも OBB が、最もフィット率が高いことがわかると思います。

図 15-12

図 15-13

図 15-14

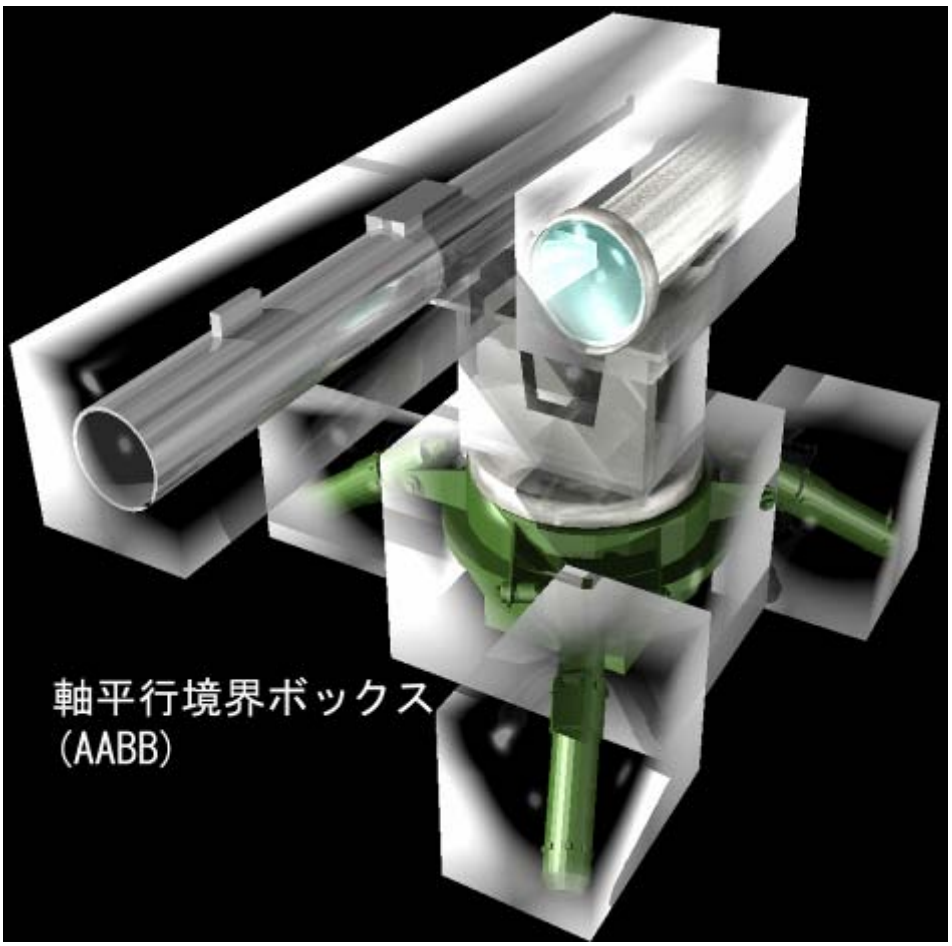


図 15-15

図 15-16

図 15-17

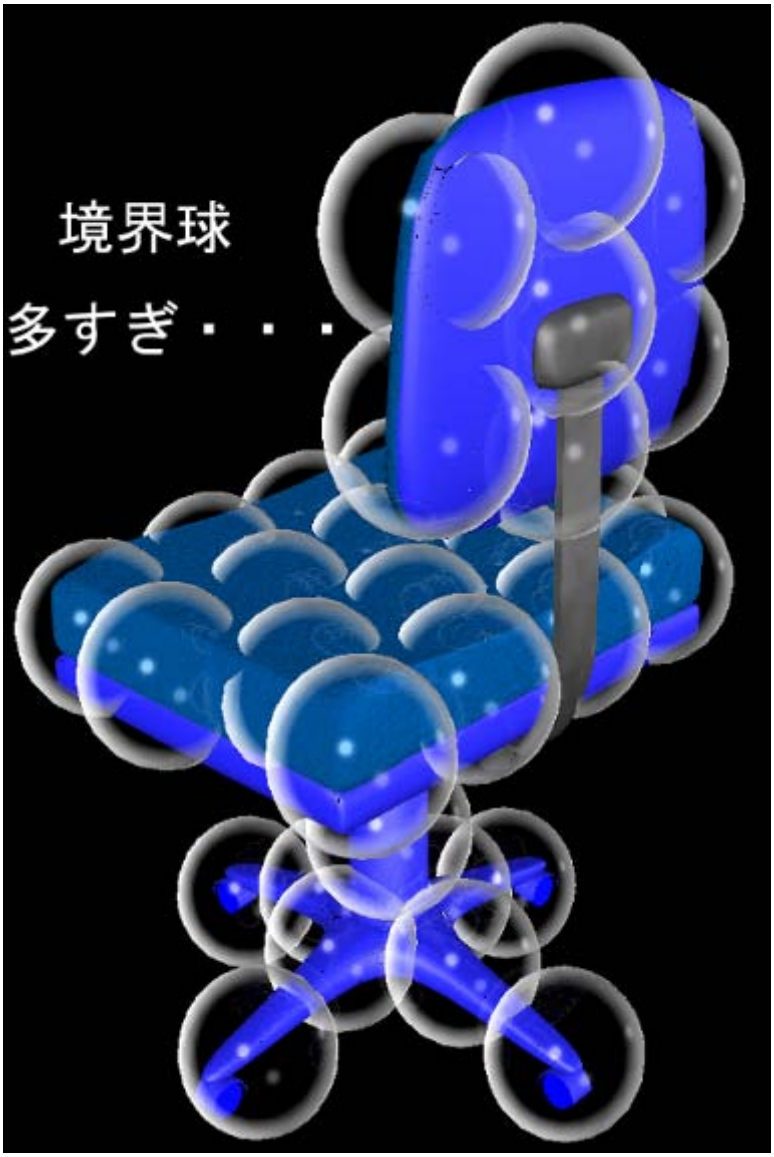


図 15-18

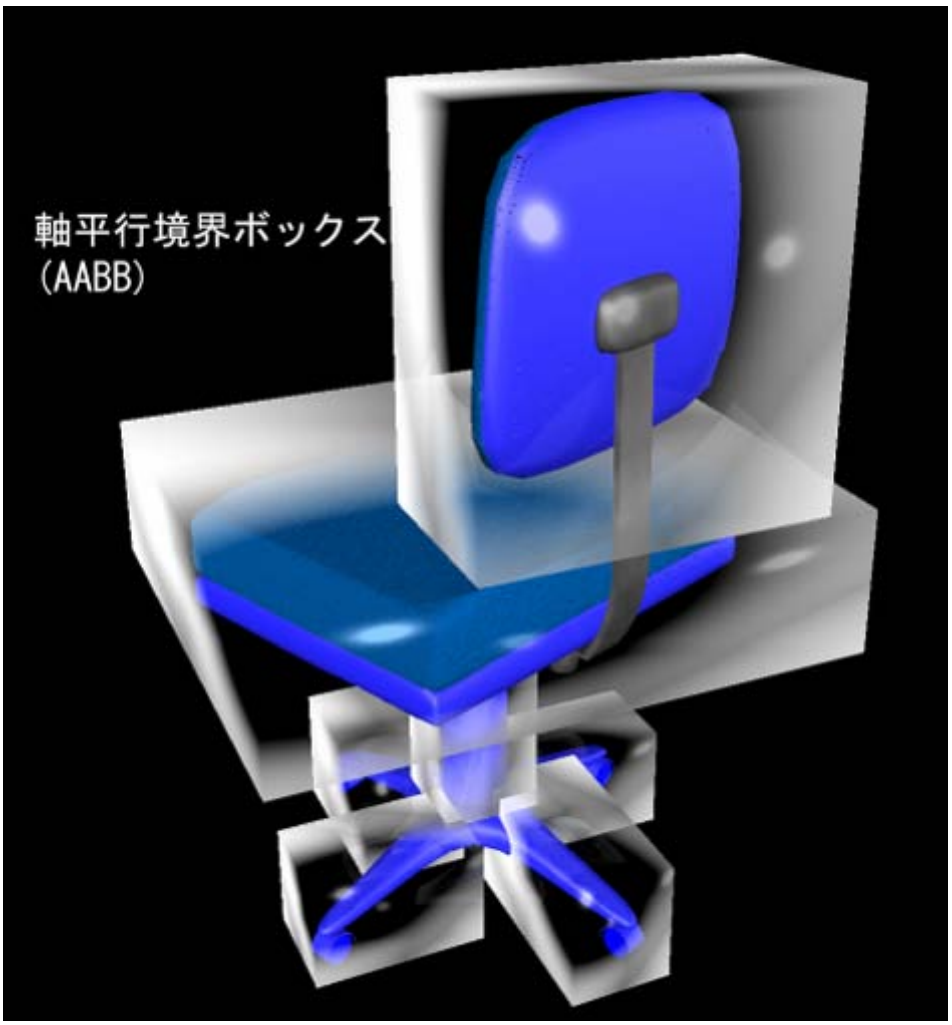
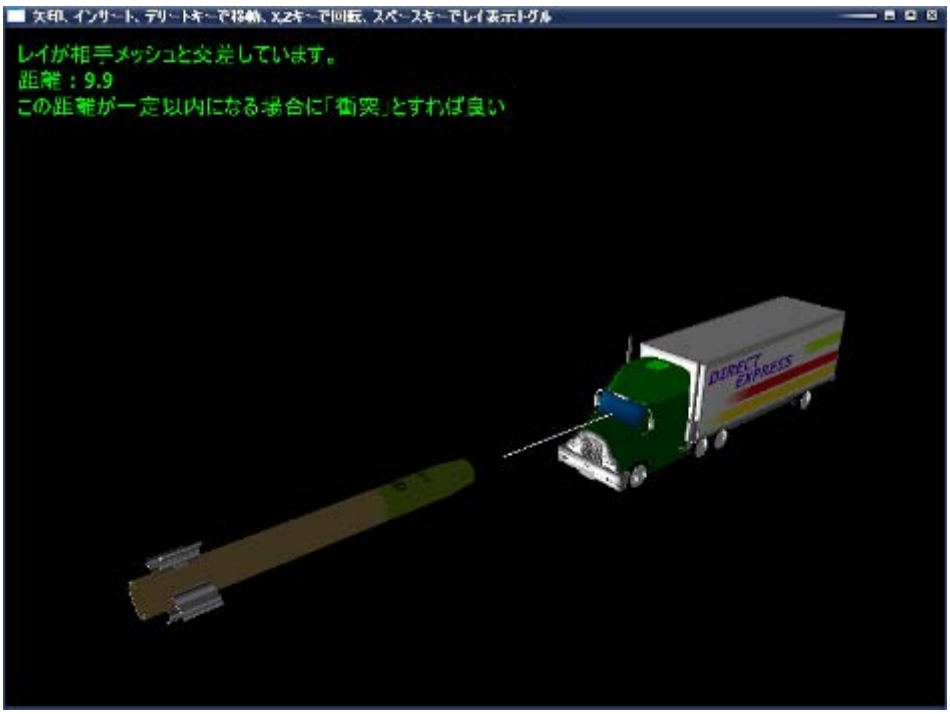


図 15-19

なお、図中での AABB は、かなりひいき目に見た場合です。AABB は対象ジオメトリの回転により形状が変わるので、姿勢によってはもっと惨たらしい形状になります。そういう意味では境界球より使い物になりません。

16 章　レイ（Ray）による衝突判定

図 16-1 レイが相手メッシュと交差している



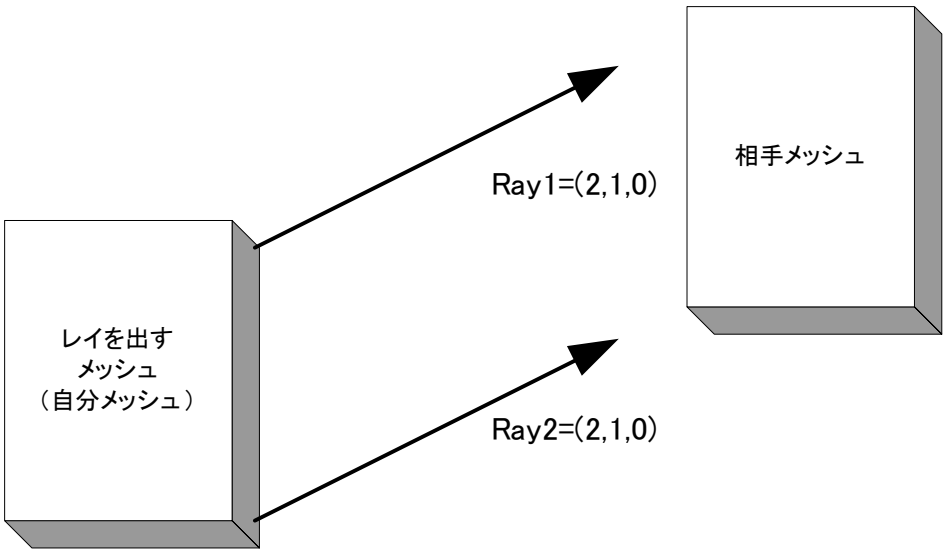
Ray「レイ」とは？

Ray（レイ）の単語としての意味は「光線」です。X-Ray「X 線」、UltraViolet-Ray「紫外線」、UltraRed-Ray「赤外線」、Electromagnetic-Ray「電磁波」などにおける Ray「光線」です。これは 3DCG 分野でのレイ・トレーシング「光線追跡法」におけるレイに由来しているものです。レンダリングという局面においてレイ・トレーシングが文字通り光源からの光を計算する技法であるのに対し、ダイナミクス（衝突判定）局面では、特に光線と捉える必要はなく、単なる線や線分（Line あるいは Segment）として捉えれば十分です。ただ、ラインやセグメントという用語は、あまりにも他の状況で多く使用される用語ということから衝突判定においても、広くレイと呼ばれているものです。もちろん光線と言っても、そのベクトル的性質のみが重要であるので、判定の際にそれ以上の意味はありません。

具体的に言うと、レイは任意の始点から伸びるベクトルであるのでコード上は 3 次元ベクトル D3DXVECTOR3 で表現されます。ベクトルは長さを持つ量なので当然終点も存在することになります。この場合のレイは両端点を持つ有限直線ですから、すなわち「有向線分」と呼んだほうが分かり易いかもしれません。このようにレイは、何らかの位置ベクトル（始点座標）から伸びるベクトルであるので、その位置が意味を持つ束縛ベクトルです。

判定の際は、主体ジオメトリ内の意図する座標から意図する方向に向かうベクトル（レイ）を考えます。レイが相手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、レイの始点と交点の距離により衝突判断するというものです。

図 16-2



レイは束縛ベクトルである。(始点が固定的で意味を持つ)
Ray1とRay2はベクトルとしては同じだが、”束縛ベクトルとしては”異なる 。
「ベクトルとしては同じだが、レイとしては異なる 。」

実際にレイと交差判定するものは、相手ジオメトリを形成しているポリゴンです。相手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。(最適化されているとしても全ポリゴンが調査対象になります。)
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が、Direct3D にはレイとメッシュの交差を判定してくれる D3DXIntersect という便利な関数が存在し、それを使用する限りは、それらの数学的知識は不要です。ここではD3DXIntersect関数により判定します。なお、レイとポリゴンの交差を判定する際の数学的原理は、直線と無限遠平面（ポリゴンと平行な無限遠平面）の交点を求めて、さらにその交点がポリゴンの内部に存在するどうかの判断をするというものです。その詳細については、「はじめての 3D ゲーム開発」（工学社刊）で解説していますので興味のある読者はそちらをどうぞ。

サンプルプログラム

プロジェクトフォルダ名「ch16 メッシュとレイ」
VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

使用方法

ミサイルを操作します。トラックは動きません。ミサイルから出るレイがトラックと交差すると、メッセージを表示するようになっています。
前進と後進：上下矢印キー
左右の移動：左右矢印キー
上下の移動：インサートキーとデリートキー
ヨー回転：Z キーと X キー
レイの表示トグル：スペースキー

コード解説

注目すべ部分は、Intersect.cpp ファイルだけです。ファイル内には 2 つの関数しかありありませんが、その中でも判定に関わるのは RayIntersect 関数だけです。

RenderRay 関数

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->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 " にセットします。

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す。本サンプルでは DrawPrimitive の後に UP が付いています。UP は、Up「アップ：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 を意味していますので、「ユーピー」と発音するのが妥当です。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。

UP の付いていない DrawPrimitive 関数により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 Missile メッシュ）から Z 軸プラス方向に 10 単位（RAY_DISTANCE 定数として定義しています）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 10 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 Missile 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 Missile メッシュの位置ベクトルを始点とし、始点の Z 成分に 10 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、10 単位というのは適当に設定した値です。

RayIntersect 関数

RayIntersect 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 を引くことにより求まります。

D3DXIntersect 関数に渡すレイは、長さが意味を持たない方向ベクトルですが、その代わり D3DXIntersect は始点と交点の距離を返してきます。最終的な判断部分で、その距離の範囲を限定すれば間接的にレイを " 線分 " として捉えることになります。

vecStart=vecEnd=pThingA->vecPosition;

とりあえず、始点も終点も物体 A（この場合は Missile メッシュ）の座標で初期化します。

BOOL boHit=FALSE;

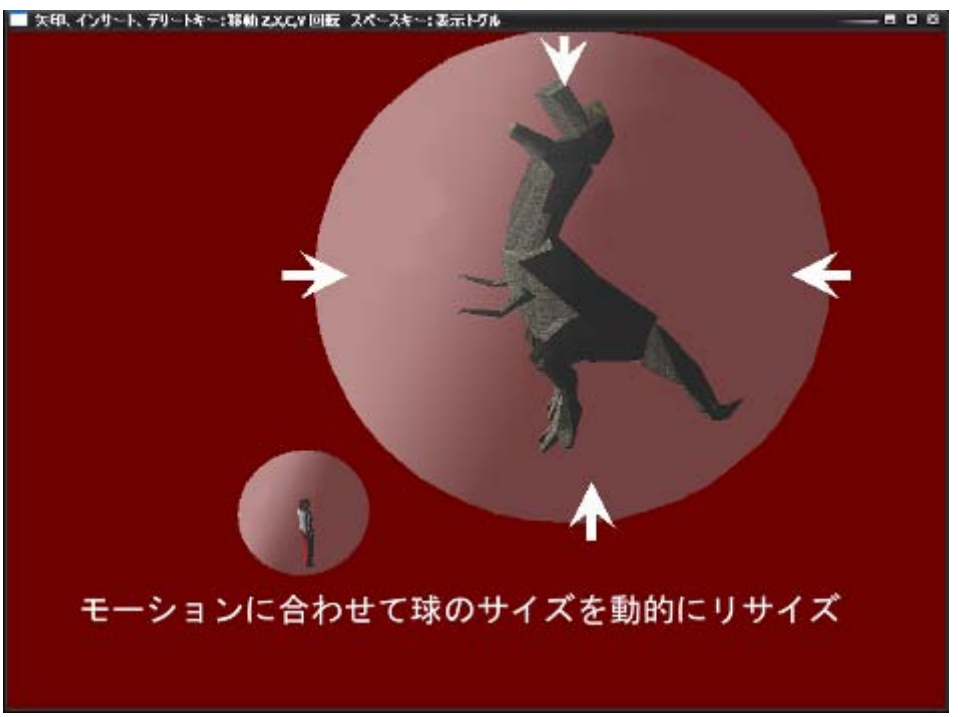
レイと物体 B（Truck メッシュ）が交差する場合の BOOL 値格納用です。
vecEnd.z+=1.0f;
終点の Z 成分を 1 単位増加させます。ここで増加させる値は正の値であればなんでも良く、0.1 単位でもいいですし、1000 単位でもとにかくプラスの値を増加させます。Z 成分を増加させる目的は方向を持たせるためであり、どんな値でもいいのです。ここでのレイは方向ベクトルですから方向が定まればそれで目的は達成されます。
D3DXMATRIX matWorld;
D3DXMatrixTranslation(&matWorld,pThingB->vecPosition.x,pThingB->vecPosition.y,pThingB->vecPosition.z);
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、相手メッシュ（Truck 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 Truck 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
“ 相手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、相手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。レイと言っても単なる束縛ベクトルであって、その実装は D3DXVECTOR3 型のインスタンスであることがお分かりでしょう。

17-1 アニメーションメッシュと境界球

図 17-1　尻尾が後ろに長いため、境界球も大きくなっています。

図 17-2　尻尾を引っ込めたモーションでは境界球も小さくなります。



サンプルプログラム

プロジェクトフォルダ名「ch17-1 アニメーションメッシュと境界球」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

このサンプルは、「バウンディング・スフィア」サンプルと「アニメーションメッシュのレンダリング」サンプルを組み合わせたような構成になっています。
「バウンディング・スフィア」サンプルではスフィアの中身がただのメッシュなのに対し、本サンプルのメッシュはアニメーションメッシュ（階層アニメーション）なので、境界球の作成部分が若干異なります。また、メッシュがアニメーションするということは、その全体形状が時間とともに変形することですので、球のサイズも変化させるべきです。球のサイズを動的に変化させると処理速度が少なからず落ちます。それを嫌って、最初からすべてのモーションの外接球となるようなサイズの球を作成してもいいのですが、フィット率が悪くなります。それは、ちょうど AABB と OBB の関係に似ています。ここでは、球のサイズを動的に変化させて少しでもフィット率を高くすることにしました。

①（階層）アニメーションメッシュのバウンディングスフィアの作成
②バウンディングスフィアをその時のモーションに見合ったサイズにリサイズする
の 2 点に的を絞って解説します。

使用方法
人間を操作します。恐竜は操作できません。両者の境界球がぶつかると、メッセージを表示するようになっています。
前進と後進：上下矢印キー
左右の移動：左右矢印キー
上下の移動：インサートキーとデリートキー
境界球の表示トグル：スペースキー

コード解説

InitSphere 関数

```
D3DXFrameCalculateBoundingSphere(pThing->pFrameRoot,&pThing->Sphere.vecCenter,&pThing->Sphere.fRadius);
```

単なるメッシュにおけるバウンディングスフィアの計算では、D3DXComputeBoundingSphere 関数を使用しました。ここではそれに代わって D3DXFrameCalculateBoundingSphere 関数を使用します。D3DXComputeBoundingSphere 関数は “ メッシュ単位 ” での境界球を算出する関数であるのに対し、D3DXFrameCalculateBoundingSphere 関数は “ ルートフレーム ” 単位で境界球を算出します。フレームとはアニメーションメッシュを構成するパーツメッシュの論理単位です。ルートフレームは、全ての階層の上位に位置するフレームで、そのフレームから作成される境界球は、アニメーション内の全パーツメッシュを包み込むような球になります。
単なるメッシュの場合は、まずそのメッシュの頂点バッファーをロックして、最初の頂点のポインターを取得するなどという作業が必要でしたが、ルートフレームから球を算出する場合は、それらの作業は D3DXFrameCalculateBoundingSphere 関数内部で自動的に行われるので、コード上は逆にシンプルになります。

②バウンディングスフィアをその時のモーションに見合ったサイズにリサイズする
RenderThing 関数（.cpp ファイル）
D3DXFrameCalculateBoundingSphere(pThing->pFrameRoot,&pThing->Sphere.vecCenter,&pThing->Sphere.fRadius);
pThing->pSphereMesh->Release();
D3DXCreateSphere(pDevice,pThing->Sphere.fRadius,24,24,&pThing->pSphereMesh,NULL);
最初にあるこの数行のコード片が、球を動的に再計算している部分です。
ほとんど InitSphere 関数での処理と同じです。ただし、pThing->pSphereMesh->Release(); の 1 行を追加する必要があります。動的に作成される球メッシュは、次々に新たなメモリ領域に格納されますので、古い球メッシュデータを開放しないとどんどんメモリを食ってしまいます。

本サンプルのポイントはこんなところでしょうか。

17-2 アニメーションメッシュとレイ

図 17-3　レイと恐竜の “ 胸パーツメッシュ ” が交差している

サンプルプログラム

プロジェクトフォルダ名「ch17-2 アニメーションメッシュとレイ」
VC6、VC2002、VC2003、３つのフォルダそれぞれにあります。

使用方法
人間を操作します。恐竜は操作できません。人間から出るレイが恐竜に交差すると、メッセージを表示するようになっています。
左右の移動：左右矢印キー
上下の移動：上下矢印キー

ヨー回転：Z キーと X キー
レイの表示トグル：スペースキー

コード解説

前項と同様、本サンプルも「メッシュとレイ」サンプルと「アニメーションメッシュのレンダリング」サンプルを組み合わせたような構成になっています。
本サンプル固有のキーポイントは無いと考えます。強いて挙げるとすれば、やはりアニメーションメッシュの展開でしょうか。アニメーションの詳細については、「はじめての 3D ゲーム開発」で完全な解説をしていることは先述のとおりです。

Collision 関数

```
if(pFrame->pFrameSibling != NULL)
{
	if(br=Collision(pFrame->pFrameSibling,pThingA,pThingB,pboHit,pfDistance,ppContainer)==TRUE)
	{
		return TRUE;
	}
}
if(pFrame->pFrameFirstChild != NULL)
{
	if(br=Collision(pFrame->pFrameFirstChild,pThingA,pThingB,pboHit,pfDistance,ppContainer)==TRUE)
	{
		return TRUE;
	}
}
```

「メッシュとレイ」サンプルにおける Collision 関数と異なるのは、この再帰コードです。再帰させているのは、アニメーションノードの展開をスマートに行うためです。
アニメーションを展開していく過程での各パーツメッシュの衝突判定は、「メッシュとレイ」サンプルと全く同じです。
アニメーションメッシュに関わる処理全般に言えることですが、ノードを展開、つまり、アニメーションメッシュを各パーツメッシュに展開してしまえば、それらは通常のメッシュとして扱うことができます。

図 18-1　筋肉の表現（同じ X ファイルを 2 つのインスタンスに読み込んでいます）

アニメーションの本質は、何らかの階層構造を頼りにジオメトリをレンダリングすることです。アニメショーンは、なんらかの相互作用を伴い、相互作用は階層構造によりスマートにコーディングされます。

階層になるものは、メッシュ、頂点、サブセットであり、もっとも簡単なアニメーションの場合は、いくつかのパーツメッシュに親子兄弟関係を設定して、それらの相対位置、相対姿勢を変化させるもので、フレームアニメーションと呼ばれます。ここで言うフレームとは、フレームレートのフレームとは意味が異なります（フレームレートのフレームはフレームバッファ 1 回分の更新を意味します）。さらに、CG ソフトでのキーフレームとも意味が異なります（キーフレームのフレームはシーンの単位時間を意味します）。ここでのフレームとは、コード上はアニメーションメッシュを形成している各パーツメッシュを格納する上位コンテナー（入れ物）であり、そして、概念的にはパーツメッシュ間の親子関係を保持するローカル座標系でもあります。

通常フレームアニメーションと言った場合における各パーツメッシュは変形しません。パーツメッシュ間の相対位置や相対姿勢により、全体的な形状を変化させるものです。それに対し、メッシュそのものを変形させることをスキニングと言い、スキニングするメッシュはスキンメッシュ、スキニングによるアニメーションはスキンアニメーションと呼ばれます。スキンアニメーションは一個以上のスキンメッシュを含むアニメーションであり、スキンメッシュ 1 つでアニメーションさせることができます。もちろん、スキンメッシュを階層構造にすれば、スキンメッシュのフレームアニメーションとなりますが、通常はスキンアニメーションと呼びます。

フレームアニメーションの理解のポイントは階層構造に尽きます。コーディングにおいて階層の展開と走査の作業は常に伴います。それについては、「はじめての 3D ゲーム開発」（工学社刊）に詳細な解説があります。その解説は完成されているので、再度解説しようとすると同じことを書くしかあり

ません。全く同じことを書くことは筆者のスタイルではありませんので、興味のある方はそちらを参照してください。
ここでは、フレームアニメーションの知識を前提として、スキンメッシュによるアニメーションのコーディングを解説していきます。
スキンアニメーションは、メッシュ自体を時間とともに変形させるので、1つのメッシュでのアニメーションできます。また、メッシュが変形するので、滑らかな動きが表現できるのが特徴です。本サンプルでの筋肉のような動きはメッシュを変形させなければ難しいでしょう。
スキンメッシュは、単体でもアニメーションすることから一見すると階層とは無関係に見えますが、実はフレームアニメーションです。というのは、スキンメッシュはボーン（骨）の階層アニメーション（フレームアニメーション）を内部で行っていることに他ならないからです。そして各頂点は、それぞれのボーンの姿勢行列を掛けることにより、ボーンに引っ張られるように移動します。頂点は、それぞれのボーンにどの程度影響を受けるかのパラメーター（重み）を持っており、滑らかな動きを実現します。（ボーンは目に見えませんが）ボーンをパーツメッシュと捉えると、通常のフレームアニメーションと何ら変わりありません。実際、ボーンのオフセット行列は、フレームの姿勢行列ですし、ボーン行列はフレームの合成行列を使用しています。

サンプルプログラム

プロジェクトフォルダ名「ch18 スキンメッシュアニメーション」
VC6、VC2002、VC2003、3つのフォルダそれぞれにあります。

使用方法
アニメーションのトグリング：スペースキー

コード解説

アニメーション自体が、Direct3D の仕様上どうしてもコード量が多くなってしまう上、さらにスキンアニメーション独自のルーチンを加えているので、全体のコードは結構な分量になってしまいました。本書のコンセプト「1 サンプル 500 行以内」を唯一破ったものとなりますが、本質的な部分は僅かです。

初期化段階（x ファイルからスキンメッシュを作成する段階）とレンダリング段階（スキンメッシュのレンダリング手順）に分けて考えることにしましょう。

初期化段階
本サンプル用に用意したスキンメッシュは単一メッシュであり、サーフェイス（マテリアル、サブセット）もただ 1 つしか持っていません。作成には LightWave を使用しました。LightWave のボーンがそのまま x ファイルに反映され、なおかつ、そのままレンダリングできるのは少し感動しました。Direct3D 側でもボーンという用語がちらちら見え隠れしているのも個人的にはとっつきやすいです。
さて、x ファイルを読み込み、スキンメッシュ作成に関連するコード部分は（InitAnimation.cpp 内）
CreateMeshContainer 関数内の当該部分
AllocateAllBoneMatrices 関数と AllocateBoneMatrix 関数
の 3 つの部分だけです。

CreateMeshContainer 関数内の当該部分
スキニングに関連する部分は

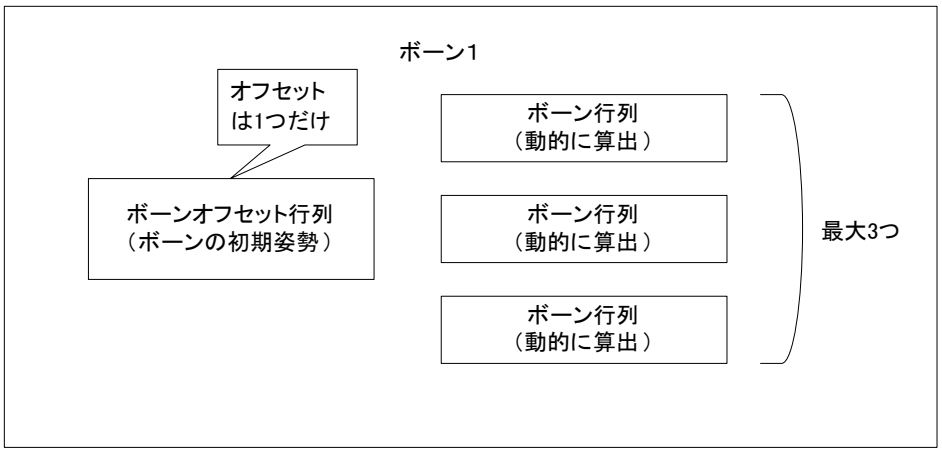
// 当該メッシュがスキン情報を持っている場合（スキンメッシュ固有の処理）
if (pSkinInfo != NULL)
{
}
の if 文ブロックです。

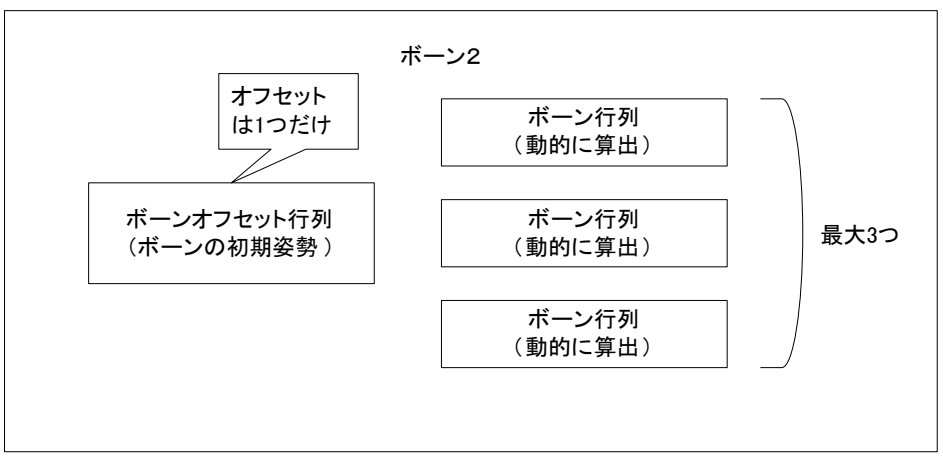
pMeshContainer->pSkinInfo = pSkinInfo;
メッシュコンテナーにスキンインフォインターフェイス ID3DXSkinInfo のポインターをコピーしておきます。
スキンインフォインターフェイスとは、スキニング処理の主にボーン行列の操作に用います。
pSkinInfo->AddRef();
インターフェイスの参照数を +1 します。これを行わないと、メッシュコンテナーを破棄する際にランタイムエラーを起こします。

dwBoneAmt = pSkinInfo->GetNumBones();
ボーンの数を取得します。本サンプルの SkinArm メッシュはボーンを 8 個使用しているので、dwBoneAmt は 8 になります。Amt は筆者の癖で Amount「総量」という意味合いです。

pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX[dwBoneAmt];
ボーンのオフセット行列のメモリを確保します。ここでは、8 個のオフセット行列分のメモリを確保することになります。
オフセット行列とは、そのボーンの初期姿勢行列であり、ローカル座標系のようなものです。オフセット行列から “ ボーンの階層アニメーション ” であるという側面が見えます。レンダリング中は、このオフセット行列に、最大 3 個の合成行列を掛け合わせて、その瞬間のボーンを算出します。

図 18-2

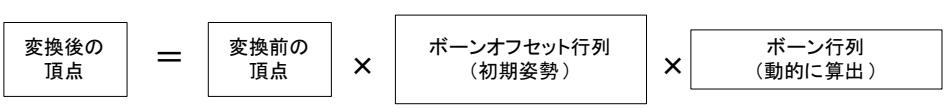




▪
▪
▪

頂点には、ボーンオフセット行列にボーン行列を掛けたものを掛けます。これを頂点ブレンドと言います。

図 18-3



頂点ブレンドは、各ボーン行列による頂点変換を適当な割合で線形に合成（要するに加算）します。
どのボーン行列をどの程度足すかは、その頂点が持つ“重み”によります。
“重み“は、頂点フォーマットで、座標の後で、かつ、法線の前と決められています。

図 18-4

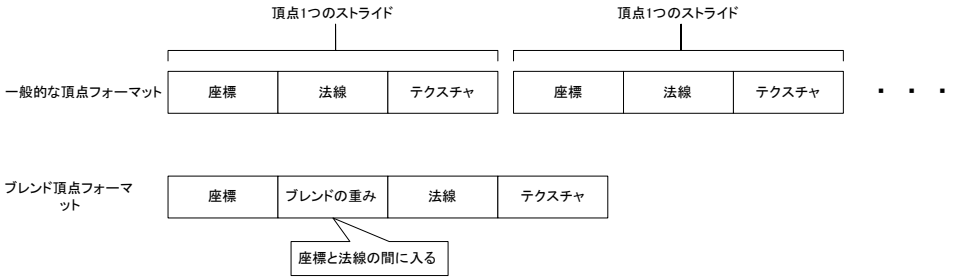
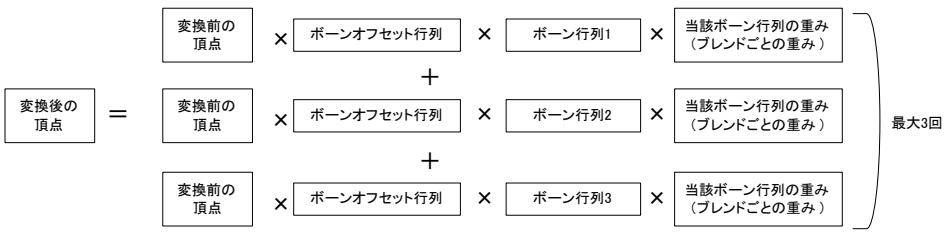


図 18-5




```
for (DWORD i= 0; i < dwBoneAmt; i++)
{
        memcpy(&pMeshContainer->pBoneOffsetMatrices[i],
        pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(i),sizeof(D3DMATRIX));
}
```

ボーンオフセット行列に値を入れます。値とは、そのボーンの絶対座標系での姿勢です。

```
if(FAILED(pMeshContainer->pSkinInfo->ConvertToBlendedMesh(
        pMesh,
        NULL,pMeshContainer->pAdjacency,NULL, NULL, NULL,
        &pMeshContainer->dwWeight,
        &pMeshContainer->dwBoneAmt,
        &pMeshContainer->pBoneBuffer,
        &pMeshContainer->MeshData.pMesh)
        ))
{
        return E_FAIL;
}
```

ID3DXSkinInfo::ConvertToBlendedMesh メソッドにより、スキンメッシュを完成させます。

AllocateAllBoneMatrices 関数と AllocateBoneMatrix 関数
この 2 つの関数は再帰的に 2 つで 1 つの仕事をします。仕事とは、ボーン行列のメモリを割り当てて値を初期化することです。
AllocateAllBoneMatrices 関数は再帰の入り口関数で、アニメーションには付き物といってもいいロジックですので解説は省略します。
AllocateBoneMatrix 関数が実際の仕事をする関数です。ボーンの数だけ、ボーン行列を作成しています（オフセット行列とは別の行列です）。

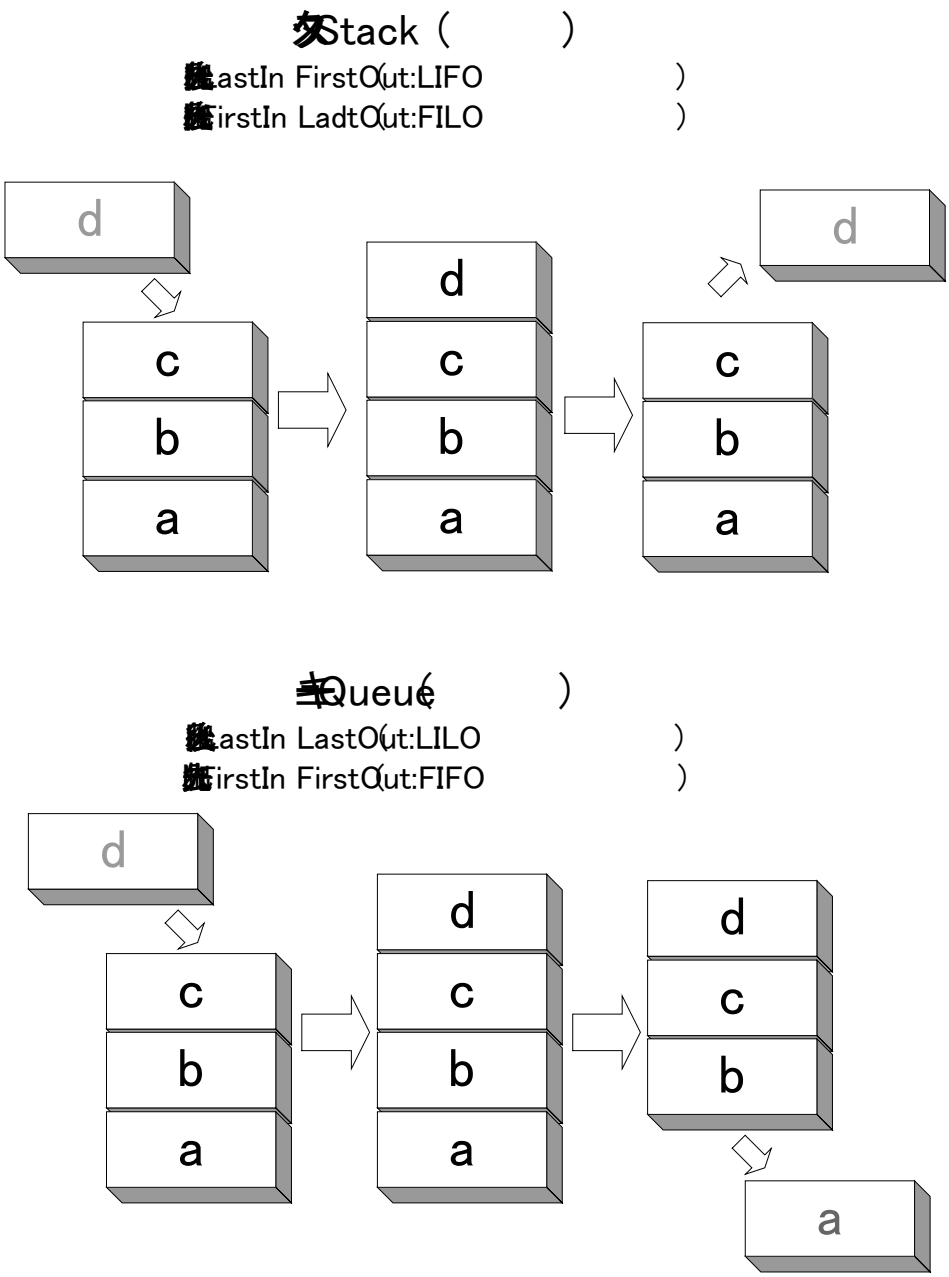
pMeshContainer->ppBoneMatrix[i] = &pFrame->CombinedTransformationMatrix;
ボーン行列は、フレームの合成行列と同義です。このことからも、スキンアニメーション＝ボーンのフレームアニメーションという側面が分かると思います。

レンダリング段階
注目するのは RenderMeshContainer 関数内の次の if 文ブロックだけです。
// スキンメッシュの場合
if(pMeshContainer->pSkinInfo != NULL)
{
}

このブロック内で行っている処理を簡潔に述べると次のようになります。
①　ボーンのオフセット行列と合成行列を掛け合わせて、それを行列スタックに積み上げる。
②　レンダリングステートを行列スタックによる頂点ブレンドモードに設定した後に DrawSubset メソッドによりレンダリングする。DrawSubset は行列スタックによりレンダリングするようになる。
③　①②の処理ををボーンの数だけ繰り返す。
変数名での Wight は重み、Bone はボーンを表しているので、処理が理解できればコードの理解は容易でしょう。

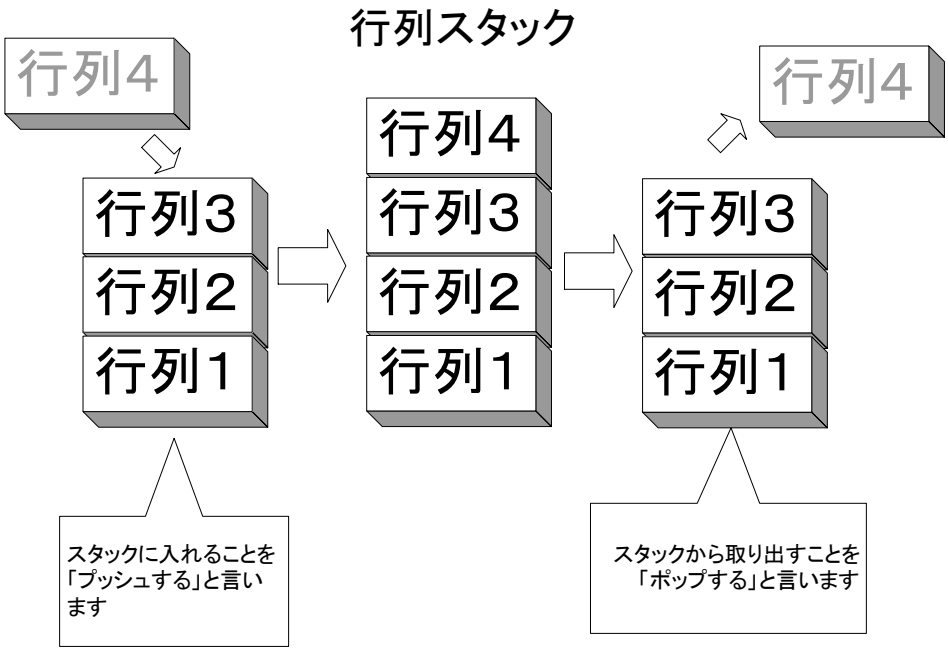
オフセット行列とボーン行列を掛け合わせているのは、
matStack=pMeshContainer->pBoneOffsetMatrices[iMatrixIndex]*(*pMeshContainer->ppBoneMatrix[iMatrixIndex]);
の部分です。
この演算の結果である行列を行列スタックという場所に格納しますが、行列スタックが分からないという人もいることと思います。行列スタックの前に、スタックとはなにかを説明します。
データを保存したり取り出したりする際には、メモリー上になんらかのバッファーを設けるわけですが、バッファーがスタックである場合、一番最後に入れたデータは最初に取り出されます。そして、一番最初に入れたデータは一番最後に取り出されます。
これはちょうど、干し草を積み上げるようなイメージで、それは Stack という言葉の語源でもあります。最初にあった干草は、底のほうで押しつぶされ、上に乗っかっている干草を取り除かなければ、取り出すことができません。
ちなみに、スタックと対比されるデータ構造にキュー（Queue) があります。キューはスタックの逆で、最初に入れたデータが最初に取り出され、最後に入れたデータが最後に取り出されます。キューは、順番待ちに似ていることから “ 待ち行列（人の行列）” とも呼ばれます。Queue という言葉の意味も “（人の）行列 ” という意味です。
少々変かもしれませんが、筆者は次のように覚えています。
データの入出力順序をラーメン店の順番待行列（人の行列です）と考えて、
スタックは不公平。（最後に並んだ人が最初に入ってしまう）
キューは公平。（最後に並んだ人が当然、最後まで待つ）

図 18-6



アセンブラしか無かった時代、コードは CPU ともろに直結したもので、今考えれば恐ろしいくらい融通の利かないコードでした。CPU がスタック構造しか受け付けないので、プログラミングはスタックとの戦い、何をするにもプッシュとポップの連続で、プッシュポップのロジックといってもいいくらいです。底のほうで押し潰されている干草を効率的に取り出すのに苦労したものです。

もうお分かりかと思いますが、行列スタックとは、複数の行列をスタック構造で格納するバッファーのことです。
図 18-7



行列スタックにボーン行列をプッシュしているのは、
pDevice->SetTransform(D3DTS_WORLDMATRIX(k), &matStack);
の部分であり、本サンプルでは k が 2 又は 3 になります。
それらの行列をポップして演算に使用しているのは、DrawSubset メソッド内部になるのでソース上には現われていません。

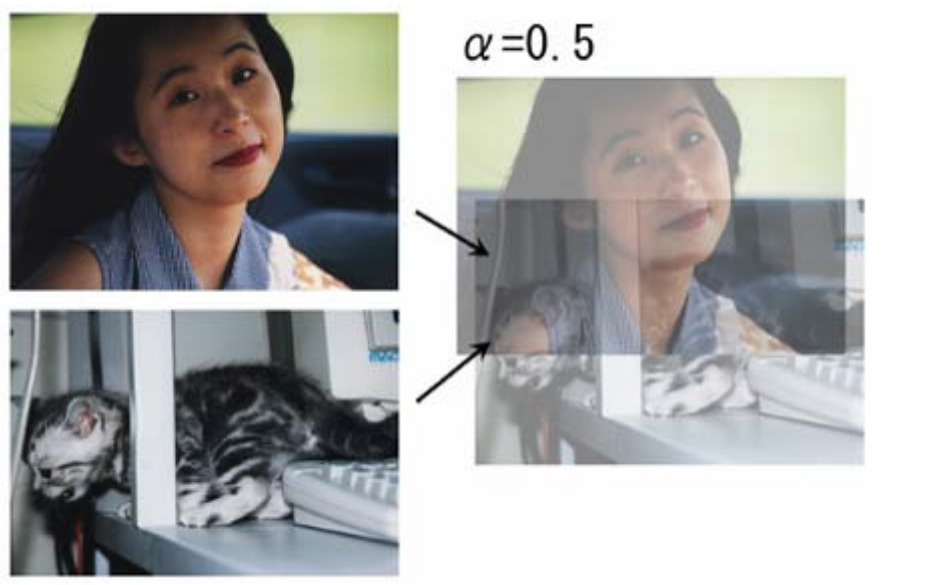
19 章 アルファブレンディング

この本の読者層でアルファブレンドがどのようなものか知らない人は少ないとは思いますが、念のため説明します。

アルファブレンディングとは半透明処理のことです。透明度は 0.0 ～ 1.0 の範囲をとる“アルファ値“というパラメーターで表され、アルファ値が 1.0 の時は完全不透明、アルファ値が 0 で完全な透明、インビジブル（そんなタイトルの映画ありましたね）になります。

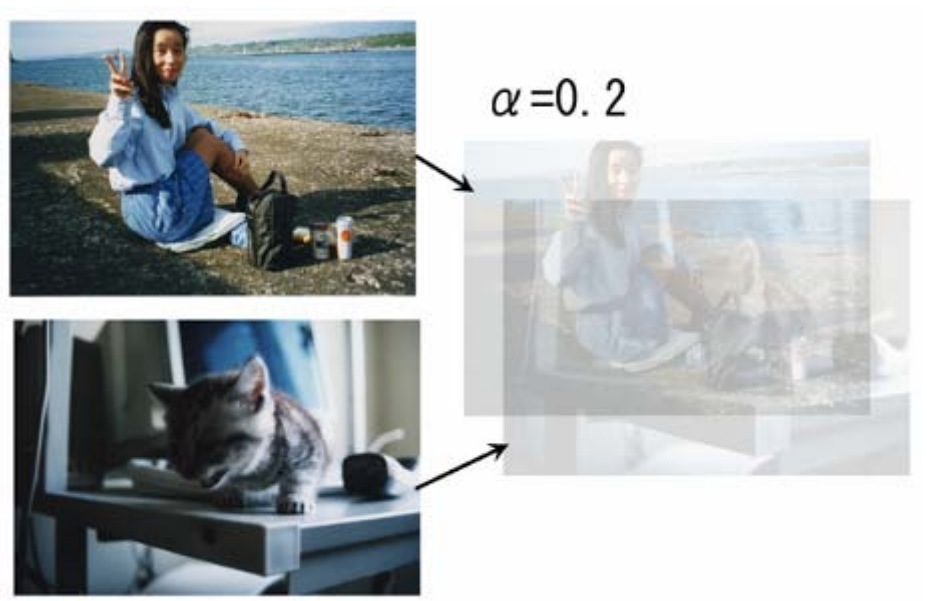
モニター上でのアルファブレンディングは、なんらかの絵（ピクセル）を描画する際、その場所に既に描画されているピクセルと合成することにより、透けて見えるような効果を生みます。その透け具合、透明度はアルファ値で調整します。

図 19-1　文字通り“半”透明



モデル：妻と愛娘　危ない、落ちるよっ

図 19-2　$\alpha = 0.2$ かなり透明度が高い



モデル：妻と愛娘　防波堤っていいね

アルファブレンディングという言葉を聞いたことがあっても、あるいは、実際にアルファブレンディングをコーディングしたことがある人さえも、なにが“αアルファ”なんだろうと思ったことはないでしょうか？
なぜアルファかというと、1970年代に当時NYIT(New York Institute of Technology)のCGラボラトリーに席を置いていたAlvy Ray Smithという人が、RGBチャンネルに追加する4番目の新たなチャンネルをアルファチャンネルと呼んだことが始まりです。（Adobe photoshopやその他画像編集ソフトにおけるアルファチャンネルはこれと同じものです）その後、Thomas PorterとTom Duffという人が1984年7月に「Compositing Digital Images」という論文をACM SIGGRAPHにて発表し、アルファチャンネルを含むアルファブレンディングという方法論が完全に定義されることとなります。
余談ですが、アルファの命名者Alvy Ray Smith氏はニューヨーク大学、UCLAバークレー校の教授職のあとゼロックス・パロアルト研究所閣員教授、Pixerの副社長、ルーカスフィルムのディレクター、及びマイクロソフトのCG部門特別研究員という具合に、肩書きの変わりようが目まぐるしく、まさにアメリカの歩くCG業界みたいな人です。最終的には自身が創設した会社のCEOに納まっていましたが、結局マイクロソフトに買収されたようです。
さて、アルファチャンネルはアルファブレンディングを実現する1つの手段ですが、時には透明以外のマスキング処理目的に使用されることもあります。ここでは「半透明処理」そのものを意味するアルファブレンディングを詳しく見ていくことにしましょう。

Direct3Dには、
フレームバッファ・アルファ
テクスチャ・アルファ
頂点・アルファ
マテリアル・アルファ
レンダリングターゲット・アルファ
というように様々なアルファブレンディングがありますが、アルファブレンドをかける対象が異なるだけでブレンド原理自体は全て同じです。

レンダリング“順序”が全て

アルファブレンディングをかけると透明になるのは、既に描画されているピクセルデータと、そこに新たに描画しようとするピクセルを合成することにより、背後のピクセルがうっすらと見え、“透明感”が出るものです。
たとえば、最も直感的なアルファブレンドとしてアルファ値=0.5とすると次のようになります。
newPixel = (PixelA × 0.5) +(PixelB × 0.5)
となります。お互いの色素を半分ずつ合成することになります。
赤いセロファンと青いセロファンが重なる部分は紫になるでしょう。それと全く同様で、もともと描画されているピクセルが24ビットカラーで0xff0000（赤）、そこに重ねるピクセルが0x0000FF（青）であれば、
newPixel=(0xFF0000 × 0.5) + (0x0000FF × 0.5)
=0x7F0000 + 0x00007F
=0x7F007F
0x7F007Fは紫のカラー値です。
これは、1つのピクセルの計算式ですが、全てのピクセルに同様の計算をすれば、背後の絵がうっす

らと見えるようになり、透明だと感じるのです。背後に絵が無い場合はスクリーンの色と合成され、その場合も透明感が出ることになります。
DirectDraw ベースのアプリケーションでは、DirectXDraw にアルファブレンディング機能がないのでアプリケーション側で自前にアルファブレンドしなくてはなりません。そのアルゴリズムは基本的に上のものです。実際、筆者は次のようにアルファブレンディングをアプリ側でゴリゴリ行っていました。
DirectDraw サーフェイスをロックしてバックバッファーのピクセルにアクセスできるようにしてから、もともとそこにあるピクセルと描画しようとしるピクセルをそれぞれ半分の色成分にしてからブレンド（加算）します。ブレンドした最終的なピクセルをもともとのピクセルと差し替えます。これで 1 ピクセル分の作業が終了です。これを、アルファブレンドするピクセルサイズの縦幅×横幅（正確にはピッチ幅）だけの回数行うというものです。
なお、これは基本的な手順であって、まともにこれをやると致命的な遅さになってしまいますので、なんらかの最適化をします。筆者は、2 ピクセルないし 4 ピクセルごと処理するようにして高速化を図りましたが、やはり、Direct3D のようにハードウェアブレンドではないので（MMX 命令をアセンブラで書くという方法もありますが)、どう頑張っても非常に低速な処理になってしまいます。ただ、自前にコーディングするがゆえにアルファブレンドの仕組みを理解するには好都合でした。
さて、メッシュ A とメッシュ B で考えてみましょう。メッシュであっても、レンダリングパイプラインの最終工程ではピクセルデータにラスタライズされます。もっと言うとメッシュに関わらずスクリーンに映るものは最終的には全てピクセルデータになるわけです。アルファブレンドはピクセルベースで行われます。
メッシュ B が 3D 空間上及び見かけ上でもでメッシュ A のちょうど背後にあったとします。
アルファブレンディングをかけてメッシュ A をレンダリングすると、背後のメッシュ B がうっすら見えなくてはなりません。しかし、アルファブレンディングはレンダリング順番に気を付けてやらないと失敗する場合があります。
例えば、次のようにメッシュ B をレンダリングしてからメッシュ A をレンダリングすると上手くいきます。

```
pMeshB->DrawSubset();
pMeshA->DrawSubset();
```

ところが、次のようにメッシュ A を先にレンダリングすると上手くいきません。

```
pMeshA->DrawSubset();
pMeshB->DrawSubset();
```

メッシュ A をレンダリングした瞬間には、スクリーン上にメッシュ B がないので、メッシュ B との合成ができるわけがありません。メッシュ A をレンダリングした時点で、そこにあるのは、スクリーン色（黒や青などのクリアー色）しかないでしょう。結果として、メッシュ A とメッシュ B が重なっている部分は、メッシュ A とスクリーン色だけのの合成色となり、メッシュ B が、その部分だけ “ くり抜かれた ” ような奇妙な格好になります。
アルファブレンドが成功するためには “ 見かけの位置的 “ に背後にあるものは先にレンダリングしなくてはなりません。少なくとも Direct3D は、位置的に背後にある物のレンダリングを自動的に調整してはくれませんので、これが鉄則です。
例えば、100 個の壁メッシュを並べて、モニター上で全て重なって見えるような視点であったとします。100 個の壁の手前にメッシュ A があったとしても、メッシュ A を一番最初にレンダリングしてしまうと、メッシュ A は X 線も真っ青になるくらい全ての壁を貫通して、最も奥のスクリーン色を浮き

出してしまいます。それはそれである意味面白いですが、メッシュ A は 100 個の壁の最後、101 番目にレンダリングすべきです。

19-1 メッシュ同士のアルファブレンディング

図 19-3　視点によっては、全く透けてくれない

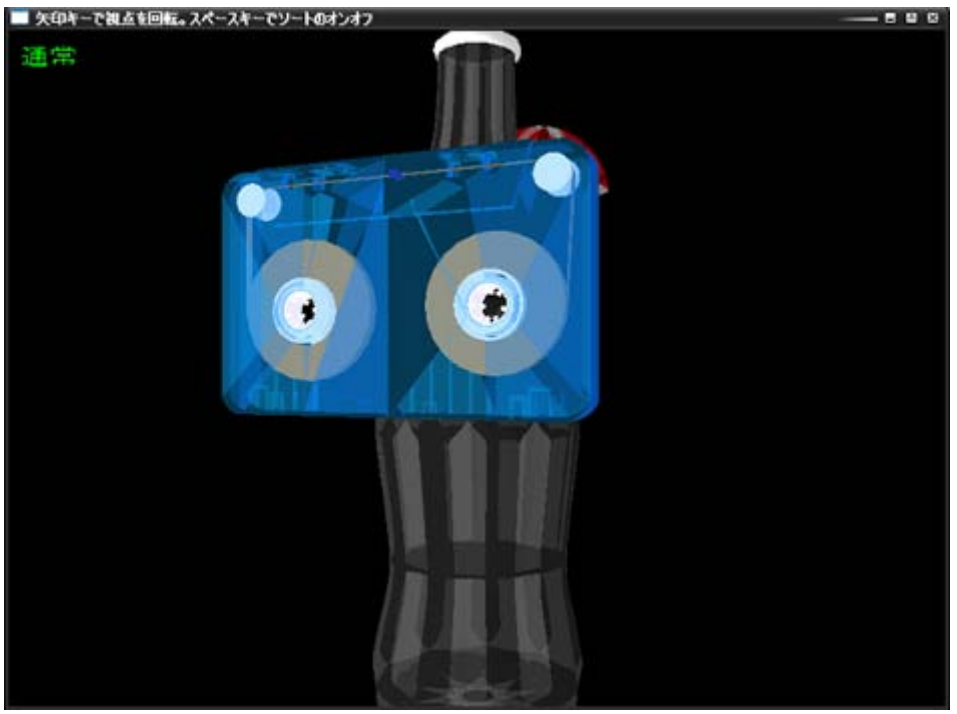
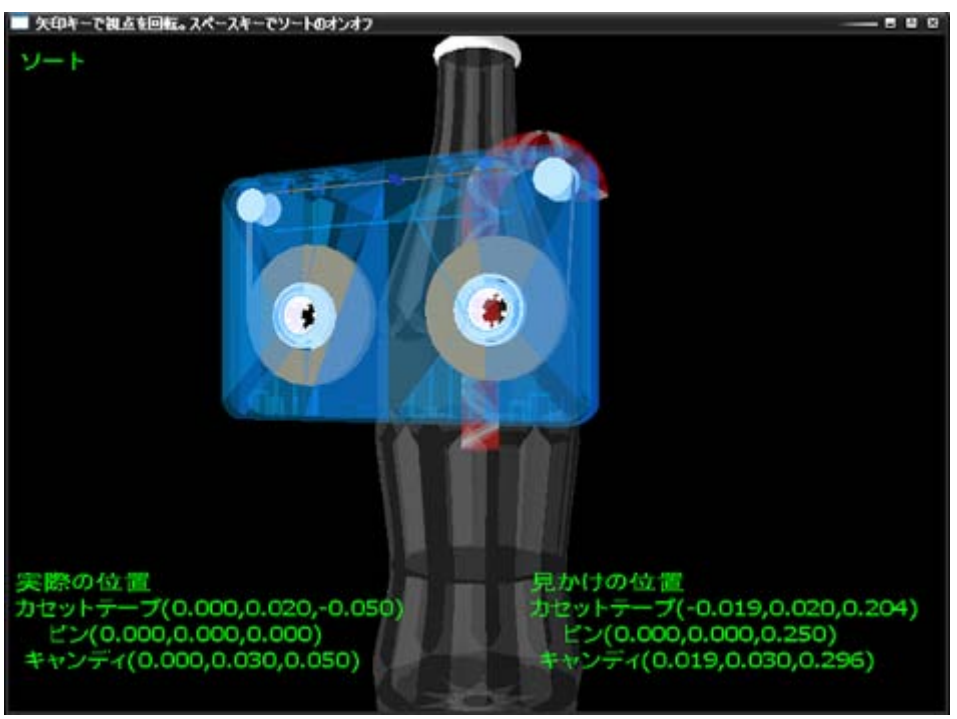


図 19-4　ソートにより、視点に関係なく常に透ける



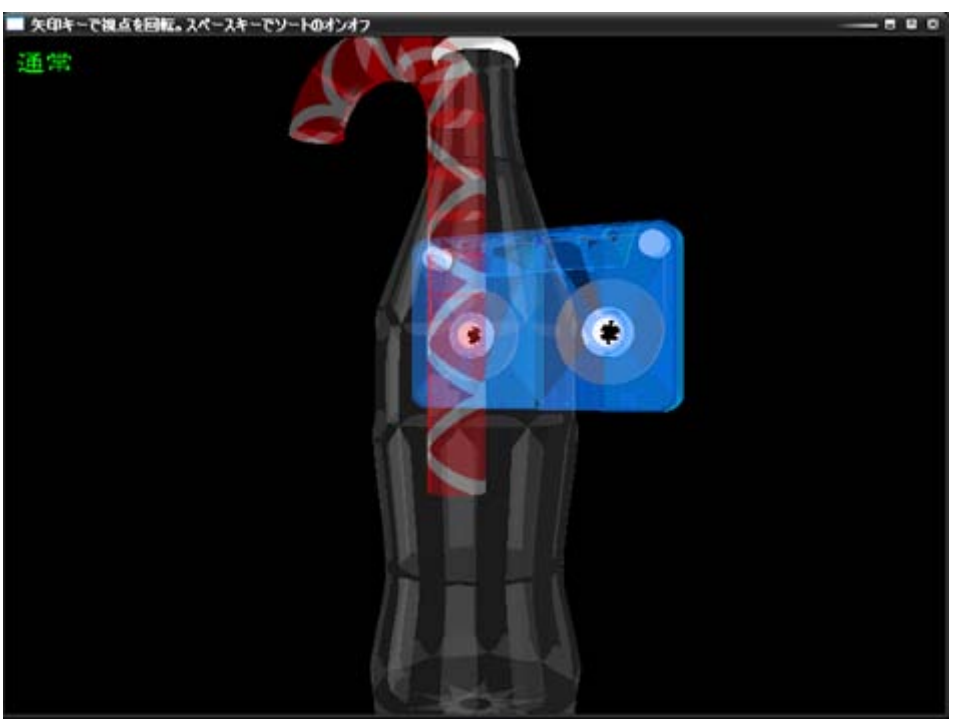
先述の理由で、アルファブレンディングを成功させるには、レンダリング順序を調整しなくてはなり

ません。
最初から最後まで動かないメッシュ同士であれば、レンダリングのタイミング調整は簡単です。先の例のように、ソース上でのレンダリング順番を調整してやればいいのです。
しかし大抵の場合、ましてやゲームの場合は視点やメッシュを移動させるため、前後の順番が入れ替わってしまい、ある瞬間でのみ上手くいくようなコードを書くと別の瞬間では失敗してしまいます。
このことから、常に位置とレンダリング順序を動的にシンクロさせる必要があります。具体的な手法としては、位置情報を基に動的にソートしてレンダリングするしかありません。
本サンプルは、その手法によりアルファブレンディングを成功させています。
何を、どのようにソートするのか？
ソートの判断に使うのは、“ 見かけの位置 ” つまり、ビュー変換後の位置（の Z 成分）です。ビュー変換前のワールド座標でソートすると失敗します。なぜなら、ワールド座標系では奥にあるものでも、カメラを回転させると前面にくることがあるからです。
とにかく、あくまでもモニターに移っている Z オーダー（位置ベクトル Z 成分の大小）でソートしなくては意味がありません。
見かけの Z オーダーを基にソートして、見かけの Z オーダーで奥にあるものから順番にレンダリングします。そして、基本的に全てのメッシュの数だけループさせる必要があります。

ソートしない場合の不具合

たまたまその時のビューが、コード上でのレンダリング順番と上手い具合に一致していれば、ソートをしなくてもアルファブレンドは上手くいきます。次に図はソートしていないときの、ある瞬間です。

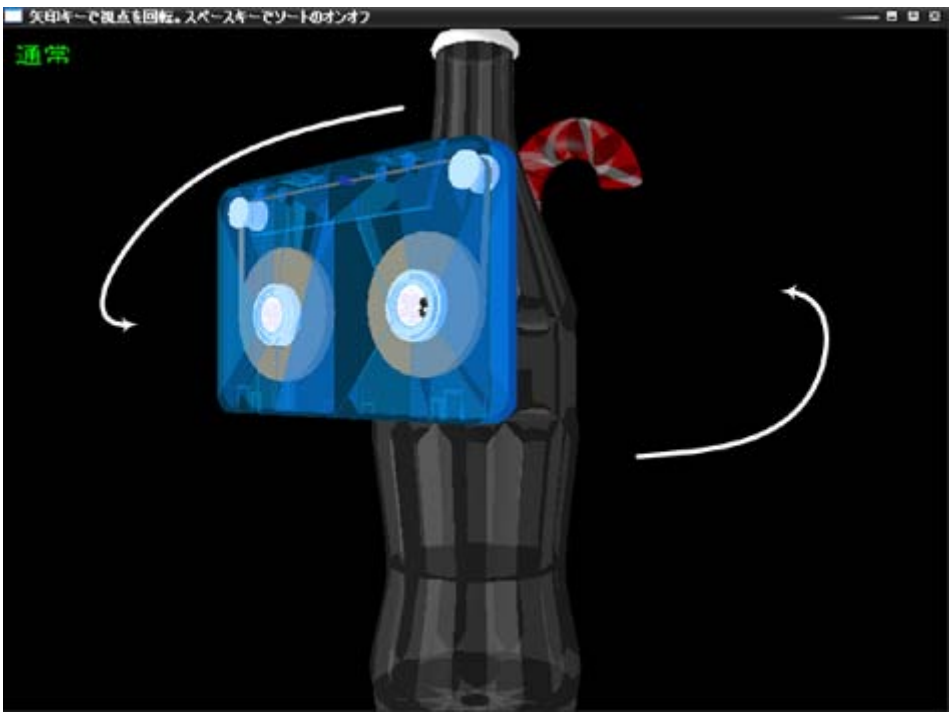
図 19-5　特定の視点では問題ないが…



キャンディメッシュの背後の、ボトルメッシュ、カセットメッシュが綺麗に透けています。
これはコード上でのレンダリング順番が
Thing[0]　カセット
Thing[1]　ボトル
Thing[2]　キャンディ

となっているので、たまたま見かけ奥のものから順番にレンダリングされているからです。
ところが、次の図のように視点を回転させることにより、見かけの Z オーダーが反転してしまうと、全く透けません。

図 19-6　視点を回転すると透けない



サンプルプログラム

プロジェクトフォルダ名「ch19-1 メッシュ同士のアルファブレンディング」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
左右矢印キーで視点を回転させます。
スペースキーでソートのオン・オフ。

コード解説

本サンプルは、ソートした時と、しない時、それぞれの場合におけるレンダリングの違いをトグルできるようにするために、コードが 10 行程度余分になっています。
ソート・レンダリングが選択されている場合は、レンダリングは SortRender 関数により行い、ソートしない場合は、NormalRender 関数により行います。
NormalRender 関数は、本書の他のサンプルでも毎回登場するコード片で、単純なものであるので解説の必要はないでしょう。

```
VOID NormalRender()
{
        for(DWORD i=0;i<THING_AMOUNT;i++)
        {
                RenderThing(&Thing[i]);
```

```
        }
}
```

解説の必要があるのは、ソートしているほうの関数です。
SortRender 関数

```
THING* pThing[THING_AMOUNT+1];
for(DWORD i=0;i<THING_AMOUNT;i++)
{
        pThing[i]=&Thing[i];
}
```

わざわざ、Thing のポインターを作成しているのは、理由があります。
この後で使用する qsort 関数（C ランタイムライブラリ）用のコールバック関数は VOID 型の配列しか受け付けないため、型に依存しないポインターの配列（ポインターのポインター）という形で渡さなければならないからです。

```
qsort(pThing,THING_AMOUNT,sizeof(THING*),Compare);
```

クイックソート qsort を実行します。qsort は C ランタイムライブラリです。第 4 引数の Compare は qsort が内部でソート評価に使うコールバック関数のポインターです。qsort のコールバック関数はアプリケーション側で実装するもので、すぐ後で解説します。

```
for(DWORD i=0;i<THING_AMOUNT;i++)
{
        RenderThing(pThing[i] );
}
```

qsort から処理が帰ってきた段階で、pThing は、ソートが反映され、Z オーダーの順番で並べ替えられています。ですので、そのまま Render 関数に渡せます。
これ以降の行は、モニターに確認用の情報を表示するルーチンなので、重要ではありません。

Compare 関数
先ほど説明したように、これは qsort 関数ライブラリ関数用のコールバック関数の実装です。
処理内容は、比較する 2 つの Thing のポインターのポインターを受け取り、そこから見かけの Z オーダーを算出し、その大小を返すというものです。この関数は、qsort が内部で使用するものです。
クイックソートのアルゴリズムは、途中に 1 回以上の大小比較を伴いますので、このような関数が必要であり、また、比較の方法は、ソートするものによって異なるのでアプリケーション側で実装するような設計になっています。

```
THING** ppThingA=(THING**)a;
THING** ppThingB=(THING**)b;
```

引数が VOID* 型なので、THING** にキャストします。

```
D3DXVec3TransformCoord(&(*ppThingA)->vecLook,&(*ppThingA)->vecPosition,&(*ppThingA)->matWorld);
D3DXVec3TransformCoord(&(*ppThingA)->vecLook,&(*ppThingA)->vecPosition,&matView);
```

Thing の vecLook ベクトルは、見かけの位置ベクトルを意味します。
比較する一方の Thing の vecLook ベクトルに、その Thing の、その時点のワールド変換、ビュー変換を掛けます。この段階で、vecLook ベクトルは見かけの位置を表すベクトルになることになります。

```
D3DXVec3TransformCoord(&(*ppThingB)->vecLook,&(*ppThingB)->vecPosition,&(*ppThingB)->matWorld);
D3DXVec3TransformCoord(&(*ppThingB)->vecLook,&(*ppThingB)->vecPosition,&matView);
```

同様に、他方の Thing の vecLook ベクトルも変換します。

```
if((*ppThingA)->vecLook.z<(*ppThingB)->vecLook.z)
{
	return 1;
}
else if((*ppThingA)->vecLook.z>(*ppThingB)->vecLook.z)
{
	return -1;
}
```

見かけの位置ベクトルのＺ成分は、モニター上での奥行きを意味します。そのＺ成分を比較することは、手前と奥の判断をしていることになります。

最後に一言にまとめると、アルファブレンディングの成功させカギは「カメラ座標のＺ値を頼りにレンダリング順番をソート（並べ替え）する」ことです。将来、ハードウェアレベルで自動的にソートしてくれる日がくるかもしれませんが、現状ではアプリケーション側で調整してやるしかありません。

19-2 同一メッシュ内でのアルファブレンディング

図 19-7　中身が見えないカセット

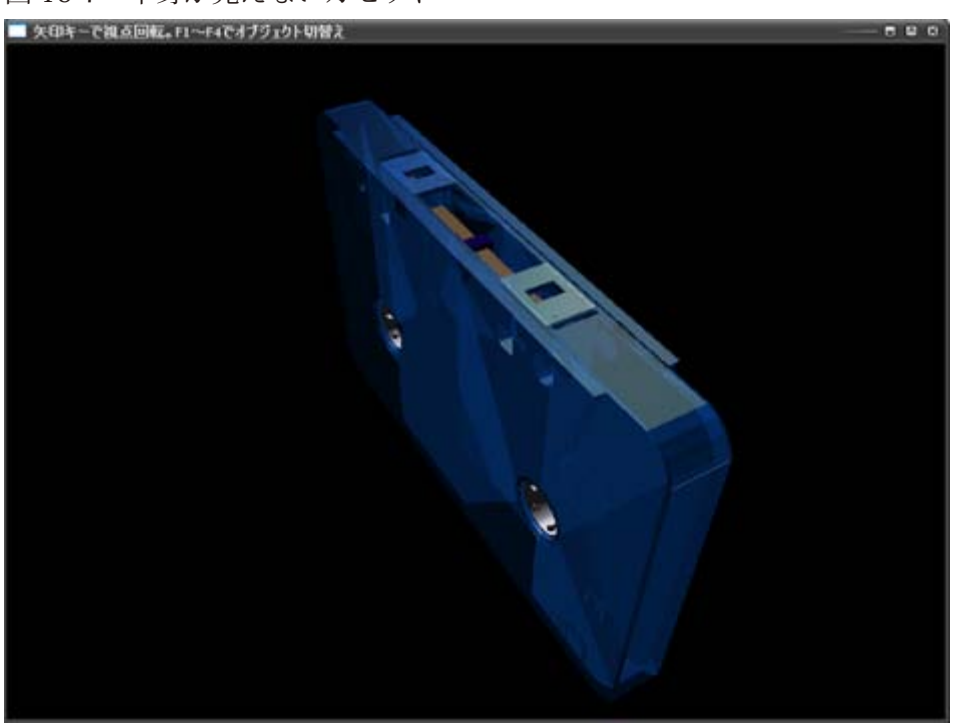


図 19-8　中身が見えるカセット

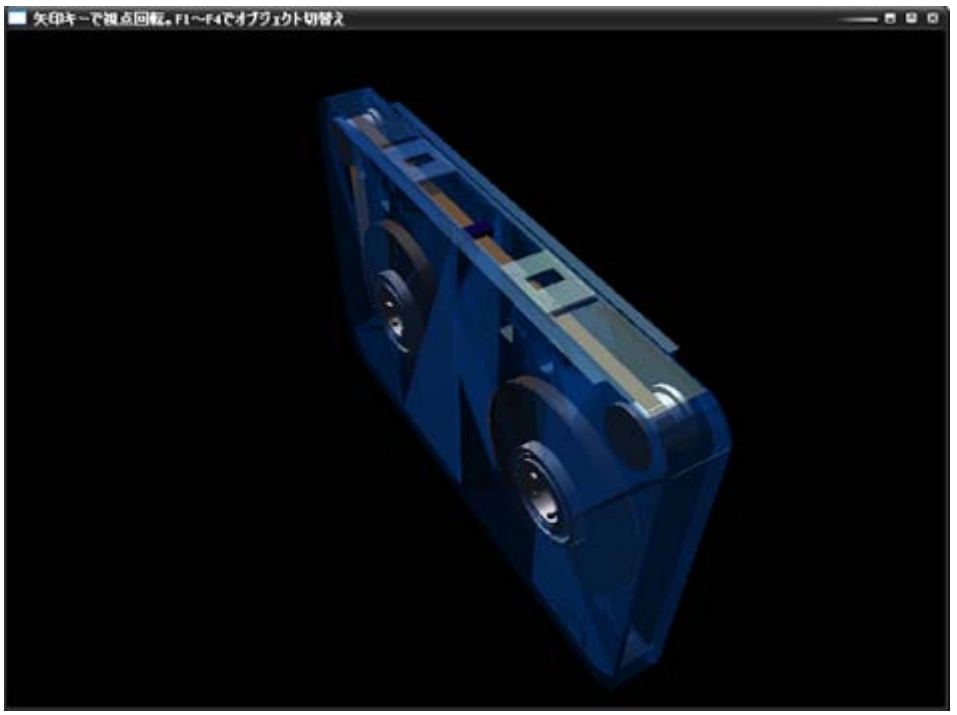


図 19-9　中身が見えないチョコバー

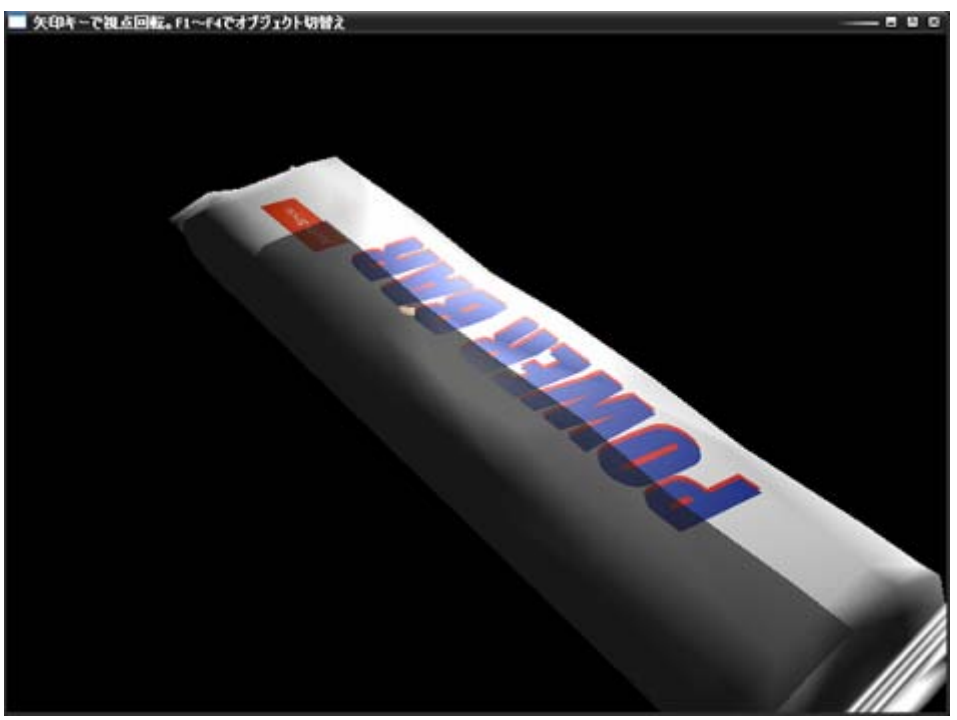


図 19-10　中身が見えるチョコバー

サンプルプログラム
プロジェクトフォルダ名「ch19-2 メッシュ内部のアルファブレンディング」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
F1 キー～ F4 キーで、4 つのメッシュを切り替えます。
左右矢印キーで視点を回転します。

前章ではメッシュ同士のアルファブレンディングを見ました。本章では、1 つのメッシュの中でのアルファブレンディングを考えます。
図 19-123 と図 19-124 を見てください。2 つとも同じカセットテープメッシュですが、図 19-123 のほうは何か違和感がありませんか？このカセットは外枠が透明なんですが、カセットの中身が見えないのです。それに対し、図 19-124 のほうは、中身のテープや部品が見えるので自然です。
同じことが、図 19-125 と図 19-126 にも言えます。図 19-125 のほうは、包装の窓からチョコレートバーが見えません。
カセットメッシュもチョコレートバーメッシュもメッシュとしては 1 つの単体メッシュです。カセットメッシュにおいて、外側ケース、テープ、部品は全て同じ X ファイル内にありメッシュとしては単一です。同様にチョコレートバーも外装と中身のチョコで 1 つの X ファイル、1 つのメッシュです。
ソースを見てもらうと分かると思いますが、自然に見えるほうのメッシュは、X ファイル名に _ Modified が付いています。つまり、自然なほうは不自然なメッシュとは違うファイルです。不自然なほうのメッシュは、意図的に不自然になるような作り方をして、また、自然なほうは、自然な透明感になるように修正しています。

まず不自然なほうのメッシュについて、なぜ、中身が透けないのか？　そこから見ていきましょう。

当然レンダリング順番がカギ

1 つのメッシュであっても、アルファブレンディングの成否を左右するファクターは当然レンダリング順番です、
メッシュが 1 つしかないのに、順番もなにも無いのではないかと思うかもしれません。しかし、実はあるのです。
「メッシュをレンダリングする」ということは、「1 個以上のマテリアルをレンダリングする」ことです。Direct3D におけるマテリアルとは、CG ソフトでいうところのサーフェイスとかポリゴンフェースのことです。CD ソフトにおける " サーフェイス " と同様、マテリアルも「色・質感」という意味と、「同じ色・質感をもつポリゴンのグループ」としての 2 つの意味を持ちます。ここでは、どちらかというと 2 つ目の意味で考える必要があります。ID3DXMESH インターフェイスはレンダリングの際は DrawSubset メソッドによりレンダリングするわけですが、DrawSubset の付近のコードをよく見てください。" マテリアルの数でループ " させているのが分かるかと思います。つまり、マテリアルの数だけ DrawSubset を実行しているということです。

```
for( DWORD i=0; i<pThing->dwNumMaterials; i++ )
{
  pThing->pMesh->DrawSubset( i );
 }
```

DrawSubset は " マテリアル単位 " でレンダリングするメソッドであって、DrawSubset 一回でメッシュ全体をレンダリングしているわけではありません。ただし、メッシュ全体でマテリアルが 1 つしかない場合は DrawSubset は一度しか実行されませんが、そもそもマテリアルが 1 つの場合メッシュ内での透明度に気を使う必要がありません。

次のような図を用意しました。この図は、マテリアルがバックバッファーにレンダリングされる過程を現しています。最終的に全てのマテリアルのレンダリングが完了した時がメッシュとしてレンダリング完了です。
なお、モニター上では、これらの重ねレンダリングは一回の画面更新 (Present メソッド) 時間内ですべて行うのが普通ですから（1 フレーム内で for ループにより全てのマテリアルをレンダリングするわけですから)、最終的な結果しか、つまりメッシュ全体としてのレンダリングしか見ることができません。

図 19-11　マテリアル Cassette_Tape_Gear（滑車部品）がレンダリングされる瞬間

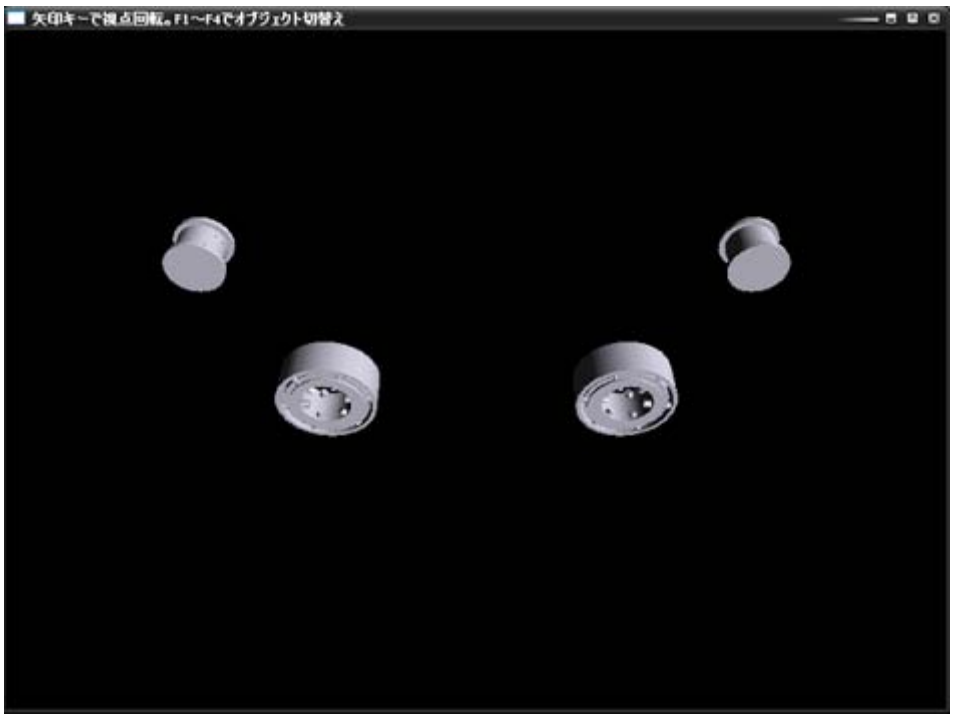


図 19-12　マテリアル Cassette_Tape_sponge（テープを安定させる小さなスポンジ）がレンダリングされる瞬間

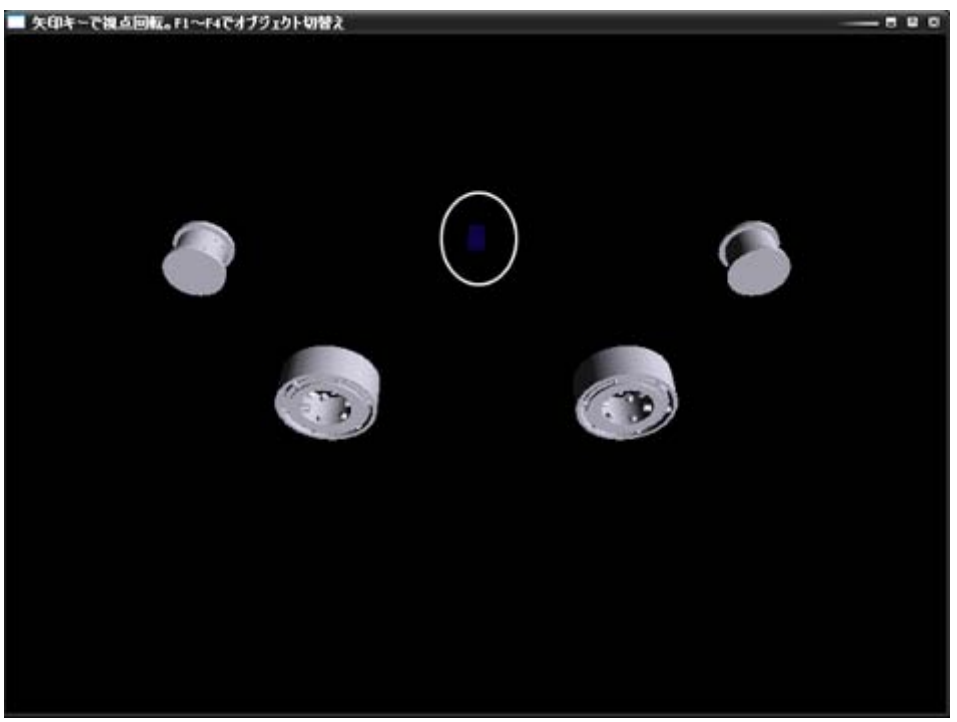


図 19-13　マテリアル Cassette_Tape_tape（磁気テープ）がレンダリングされる瞬間

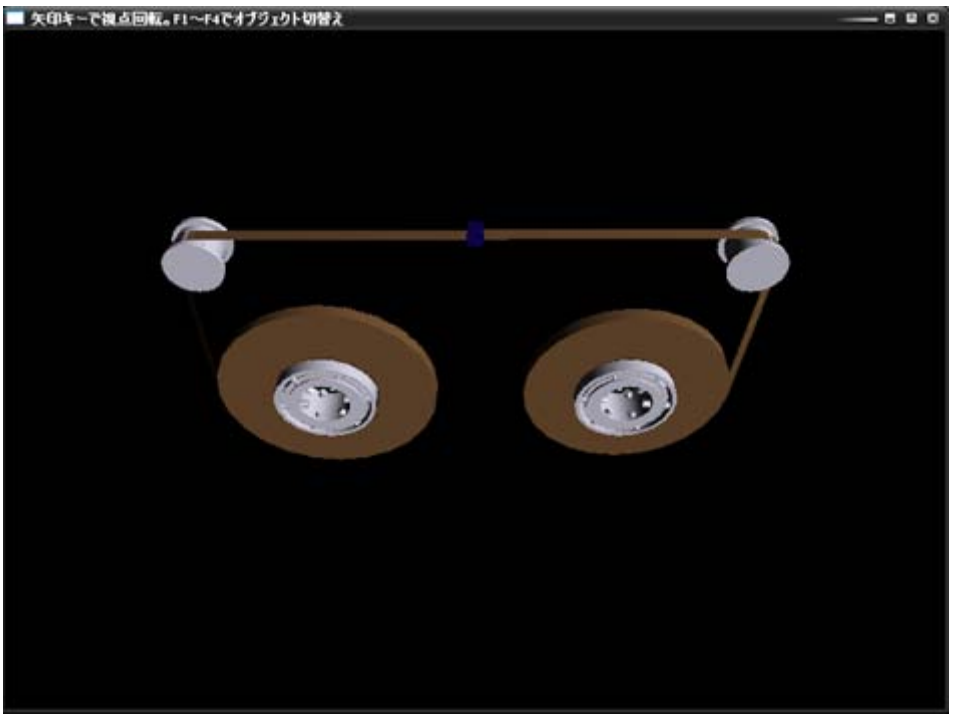
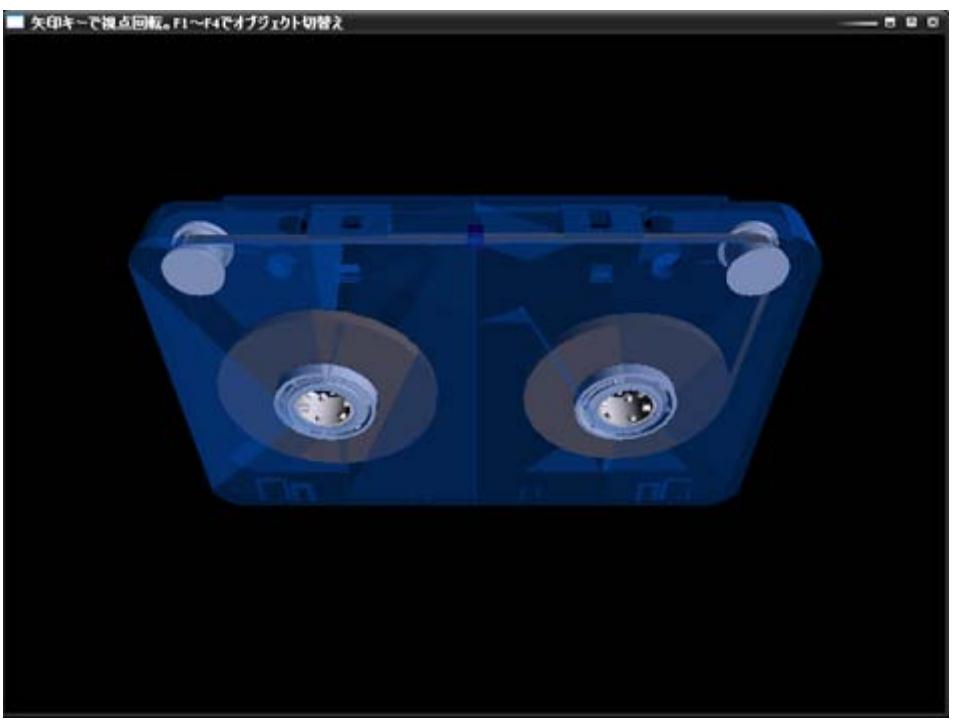


図 19-14　マテリアル Cassette_Tape_Body（外側ケース）がレンダリングされる瞬間



我々は通常、メッシュ内の全てのマテリアルを重ねレンダリングした結果を見ているわけです。
ここまで読んで、勘の良い読者はもう気付いたことでしょう。1 個のメッシュ内でのレンダリング順番とは、“ マテリアルのレンダリング順番 ” です。

ここで、4 つの X ファイルの中身を覗いてみましょう。
X ファイルには、マテリアルそのものの情報とマテリアルとポリゴンを対応させる際に使うインデックス情報が収録されます。そのインデックスは “ マテリアルリスト ” と呼ばれます。各メッシュのマテリアルリストは次のようになっています。

ChocoBar.x ファイル内のマテリアルリスト
```
MeshMaterialList
{
   （途中のデータは省略）

     { ChocoBar_Cover }
   { ChocoBar_Cover_Inside }
   { ChocoBar_Cover_Window }
   { ChocoBar_Ingredient_ }
}
```

ChocoBar_Modified.x ファイル内のマテリアルリスト
```
MeshMaterialList
{
   （途中のデータは省略）

     { ChocoBar_Modified_Choco }
   { ChocoBar_Modified_Cover }
   { ChocoBar_Modified_Cover_Inside }
   { ChocoBar_Modified_Cover_Window }
}
```

Tape.x ファイル内のマテリアルリスト
```
MeshMaterialList
{
   （途中のデータは省略）

     { Cassette_Tape_Body }
   { Cassette_Tape_Gear }
   { Cassette_Tape_sponge }
   { Cassette_Tape_tape }
}
```

Tape_Modified.x ファイル内のマテリアルリスト
```
MeshMaterialList
{
   （途中のデータは省略）

     { Cassette_Tape_Modified_A_Gear }
```

    { Cassette_Tape_Modified_A_Sponge }
    { Cassette_Tape_Modified_A_Tape }
    { Cassette_Tape_Modified_Body }
}

マテリアルリストには、実際はもっと多くのインデックスデータが記載されていますが、今は関係ないので省略しています。マテリアルリストの最後には、各マテリアルの名前が記録されています。この名前は X ファイル作成ソフト側でのサーフェイス名です。
Direct3D は、このリストの順番で、この名前の順番で、マテリアルをレンダリングしていき、最終的にはメッシュとしてレンダリングします。
したがって、たとえば、Tape メッシュの場合、マテリアル名の順序は。Cassette_Tape_Body、Cassette_Tape_Gear、Cassette_Tape_sponge、Cassette_Tape_tape となっていますから、
DrawSubset(0) で　Cassette_Tape_Body マテリアルであるポリゴングループをレンダリング（カセットの外側ケース）
DrawSubset(1) で　Cassette_Tape_Gear マテリアルであるポリゴングループをレンダリング（テープを巻きつける滑車とその他の滑車）
DrawSubset(2) で　Cassette_Tape_sponge マテリアルであるポリゴングループをレンダリング（スポンジ…小さくて分かり辛いですが…）
DrawSubset(3) で　Cassette_Tape_tape マテリアルであるポリゴングループをレンダリング（磁気テープ部分）

することになります。
実際は、DrawSubset はマテリアルの数でループさせて、1 フレームで全てのマテリアルをレンダリングしますから、段階的にレンダリングされる過程は見えません。
さて、この Tape メッシュは一番最初に “ 外側ケース ” をレンダリングしています。一番外側のものを一番最初にレンダリングするので、その後に中身の部品やテープをレンダリングしても、透明感が出ません。ほとんど外側ケースだけしか見えないということになってしまいます。
ChocoBar メッシュも同じで、順番が悪いため、中身の “ チョコ ” が袋の窓から見えません。

これを解決するために、それぞれのマテリアルリストが意図した順番になるように修正したのが、ChocoBar_Modified メッシュと Tape_Modified メッシュです。筆者はライトウェーブを使用しています。ライトウェーブは X ファイル作成時にサーフェイスをマテリアルとして記録するわけですが、その際、記録する順番はアルファベット順になることが分かっているので、サーフェイス名を意図した順番になるようにした後に、X ファイルを出力しました。
Tape_Modified メッシュを見てもらえれば分かりますが、上手い具合にアルファベット順になるような名前が浮かばないときは、頭に A_ をつけて、いささか強引にでも先頭になるようにしています。

なお、この手法では、X ファイルベースでの順番に依存しているので、あるタイミングでは成功しても、別のタイミングでは成功しないことになりますが、前章のようなメッシュ同士の場合ほどは表面化しないでしょう。なぜなら、メッシュ内でのマテリアルの位置はそうそう変わるものではないからです。それでも気になる場合は、前章のように “ マテリアル単位でソート ” すればいいでしょう。

本サンプルは、コードというよりも X ファイル側での調整なので、コード解説する必要が無くなります。なんら特別なトリックを施すことなく単純にメッシュをレンダリングしているだけです。

20 章　マルチビュー・レンダリング

画面を分割し、かつ、分割した領域ごとに独立したビュー変換が可能なレンダリングをマルチビューレンダリングと呼ぶことにします。
マルチビューを使用する局面としては、例えば、メッシュビューアーや自作のモデラーを作成する場合があると思います。LightWave のモデラーのような 4 画面分割画面がそうです。また、車内のバックミラーを実現する際もマルチビューレンダリングが必要になります。
1 つのメッシュ、1 つのシーンを複数の角度から見たときのそれぞれのビューを、異なる画面領域、あるいはウィンドウ自体異なる別ウィンドウに表示したい場合もあります。ここではそれらを実現していきます。

20-1 ひとつのウィンドウ上で領域を分割

図 20-1　シングルビュー・モード

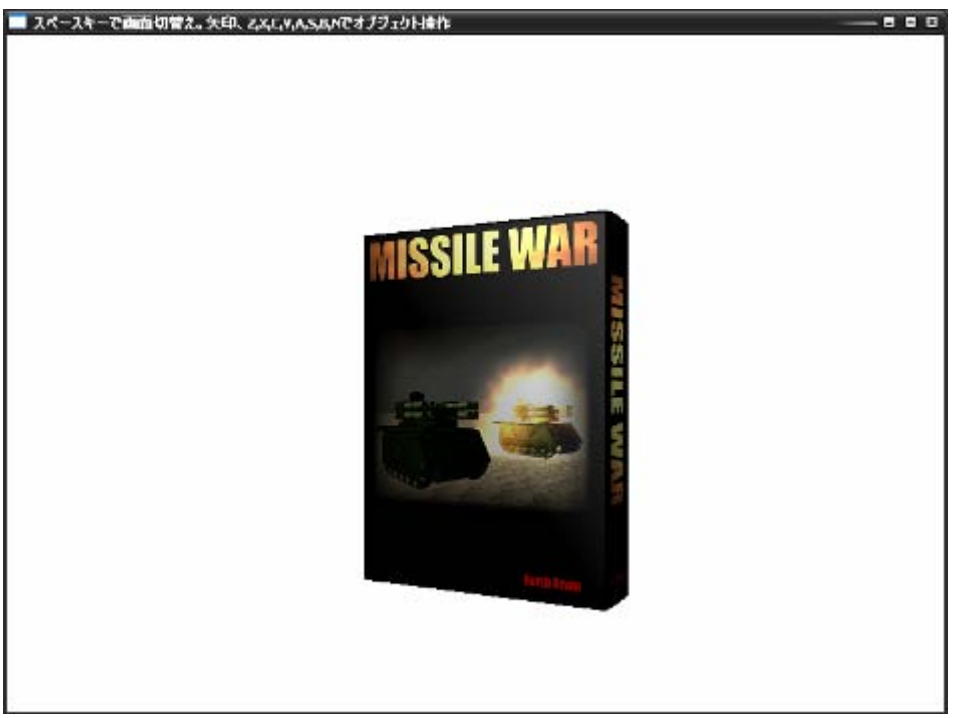
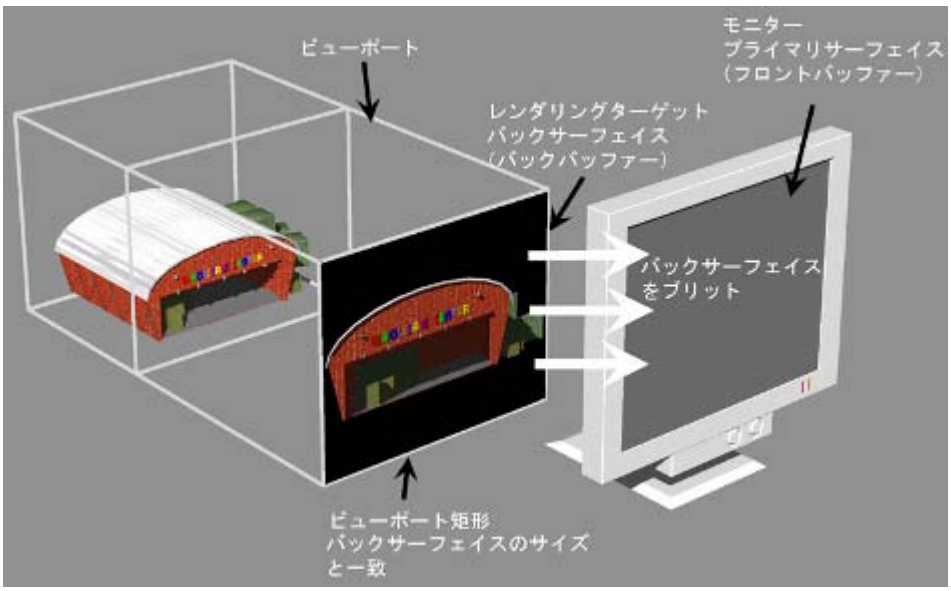


図 20-2　マルチビュー・モード

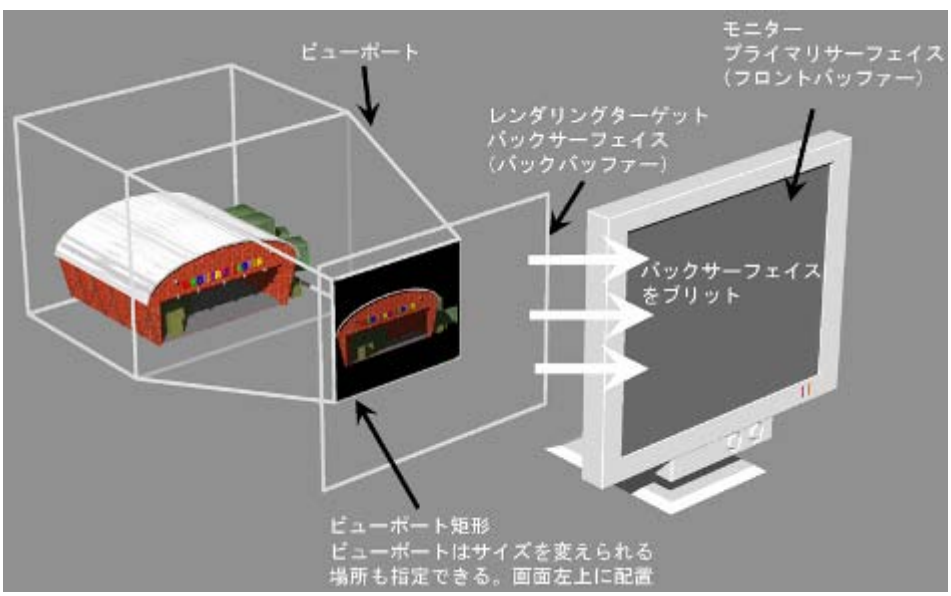
ここでは、1 つのウィンドウを 4 つの領域に分割し、それぞれの領域に異なる視点からのビューをレンダリングするということを行います。
仕組みとしては、ビューポートを調整することによって、1 つのウィンドウのクライアント領域上に複数のビューをレンダリングすることを実現します。
ビューポートによってモニター上に描画される領域をビューポート矩形と言います。
通常、ビューポート矩形サイズはクライアント領域のサイズに一致しています、例えば、ウィンドウのサイズが 800 × 600 ピクセルであれば、ビューポート矩形も 800 × 600 ピクセルとなっています。
ビューポート矩形は、ウィンドウのサイズとは無関係に任意のサイズに設定することができます。したがって、例えば、ウィンドウサイズが 800 × 600 ピクセルであったとしても、ビューポート矩形を 100 × 100 ピクセルにすることもでき、その場合、レンダリングが描画される部分は 100 × 100 ピクセルの領域でしかなく、クライアント領域のほとんどはウィンドウのバックグランドカラーとなることになります。
また、ビューポート矩形は、描画開始位置を任意の位置に設定できます。通常はクライアント座標 (0,0) が描画開始点になりますが、例えば、(400,300) とすれば画面真ん中から描画することができます。
次の図は、ビューポートとビューポート矩形のイメージ図です。

図 20-3



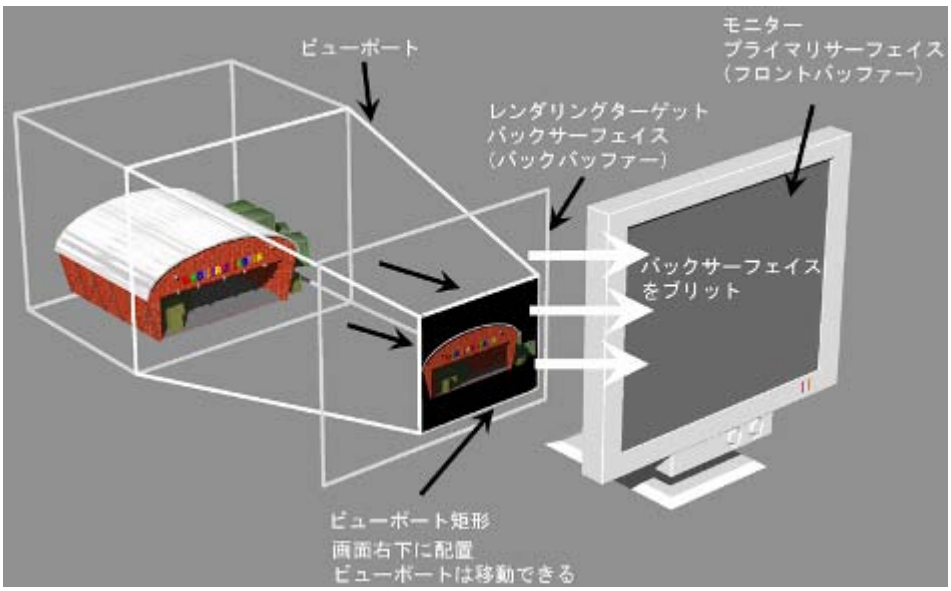
ビューポート矩形に手を加えない通常の場合、矩形はバックバッファーと同じサイズとなるので、矩形とモニターを同一視できます。

図 20-4



矩形のサイズを変えることにより、クライアントの一部分のみにレンダリング結果が描画されます。

図 20-5



そして、描画開始位置を設定することにより、好きな場所から描画を開始できます。つまり、矩形を移動できます。
この機能を応用すれば、4 画面分割ができるのは想像できるでしょう。図では、1 つのビューしかありませんが、1 フレーム中に、矩形のサイズを画面の 4 分の 1 にして、位置を適切に配置し、それを 4 回繰り返せば 4 画面になります。

サンプルプログラム

プロジェクトフォルダ名「ch20-1 ビューポートで画面分割」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
移動：矢印キー
ヨー回転：Z キーと X キー
ピッチ回転：C キーと V キー
ロール回転：B キーと N キー
スケーリング：A キーと S キー
マルチビュー、シングルビューのトグル：スペースキー

コード解説

注目すべきコード部分は、本質的には ChangeViewport 関数だけですが、異なる 4 つのビュー作成に関連している部分 SetViewMatrix 関数、Render 関数も解説しておきます。

ChangeViewport 関数
D3DVIEWPORT9 vp;
ビューポート用の構造体は D3DVIEWPORT9 型として定義されています。そのインスタンスを vp という名前で生成しています。

vp.X=dwX;

vp.Y=dwY;
ビューポート矩形の描画開始座標です。

vp.Width=dwWidth;
vp.Height=dwHeight;
ビューポート矩形のサイズです。

vp.MinZ=0;
vp.MaxZ=1;
これは、レンダリング深度値で、0.0 ～ 1.0 の値をとります。通常はこのように最小に 0 を、最大に 1 を指定します。両方に 0 を指定した時にレンダリングしたものは強制的に前面にレンダリングされ、逆に両方に 1 を指定すると強制的に最深部（もっとも奥）にレンダリングされます。

if(FAILED(pDevice->SetViewport(&vp)))
SetViewport が、本サンプルでの核心部分です。上で設定した矩形をレンダリングに反映させるにはこの関数をコールします。
SetViewMatrix 関数
4 つの領域には、異なるビューをレンダリングします。この関数はビュー変換（カメラ座標変換）を行います。通常は、Render 関数内でワールド変換とプロジェクション変換の間でビュー変換を行っていますが、本サンプルのビュー変換のコードは長く見辛くなるので。関数として独立させました。処理内容自体は全く難しくありません。VIEW 列挙定数によって場合分けして、それぞれのビュー変換を行っているだけです。
ビュー列挙定数は、ファイル上部で次のように定義しています。

```
enum VIEW
{
	NO_SPLIT,//（通常）分割無し
	TOP,//（分割有り）上からの視点
	FRONT,//（分割有り）正面からの視点
	LEFT,//（分割有り）左からの視点
	BIRD//（分割有り）鳥瞰視点　Bird's-Eye View
};
```

それぞれの意味は、コメントの通りです。
vecEyePt は視点ベクトルです。視点をそれぞれのビューに見合った位置に設定している以外は、ビュー行列を作成して、それをレンダリングパイプラインに渡す、といういつもの処理です。

Render 関数
boQuad は、通常の 1 画面か、4 分割画面かのトグルスイッチ（として機能するブール変数）です。因みにクアッドとは 4 分割という意味があります。

if(boQuad) ブロックが長いように見えますが、1 つのビューに係る 3 行のコードが 4 つ分あるだけなので、その代表として TOP ビュー部分のみを説明します。

SetViewMatrix(TOP);
レンダリング時のビュー行列を上方視点にセットします。

ChangeViewport(0,0,WINDOW_WIDTH/2,WINDOW_HEIGHT/2);
ビューポート矩形の位置をスクリーン左上 (0,0) の位置に、そして、縦横の長さをそれぞれ半分に、つまり、4 分の 1 のサイズに設定します。
ViewRender(D3DXCOLOR(1,0,0,1));
後は通常通りレンダリングするだけです。関数名は ViewRnder となっていますが、いままで毎回登場していた RenderThing 関数と同じものです。

20-2 複数のウィンドウに分割する

図 20-6　シングルウィンドウによるシングルビュー

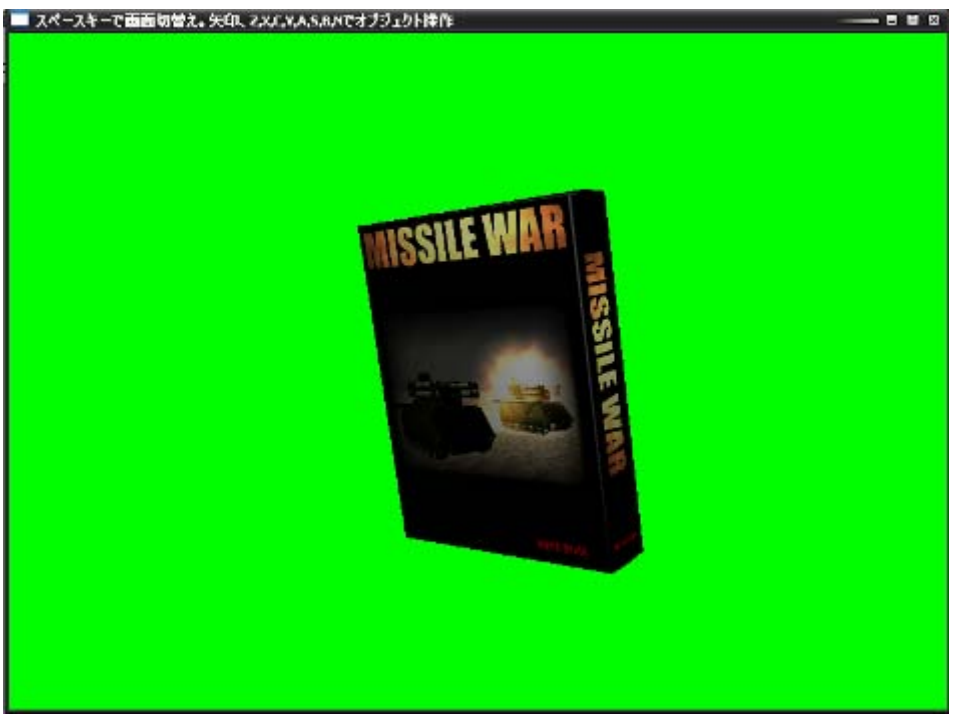


図 20-7　マルチウィンドウによるマルチビュー

ここでは、ウィンドウ自体を複数使って、それぞれに異なるビューをレンダリングする手法を実践します。
サンプルは、4 つの子ウィンドウに、前章と同様の 4 つのビューをレンダリングします。フレームウィンドウ（親ウィンドウ）を含めると、ウィンドウを 5 つ使用していることになります。
この手法は、ウィンドウそのものが複数あることにより、よりツール的な需要に対応するものとなるでしょう。各ビューは " ウィンドウ " なので、位置やサイズを自由に変えることができます。このようなスタイルは、ゲームというよりも、それこそモデラーのようなツールに馴染むと思います。

仕組みとしては、まず、子ウィンドウを 4 つ作成します。（当然ですね）
前章では、1 つのレンダリングターゲットにビューポート矩形を複数使用しましたが、ここでは、レンダリングターゲットを複数使用します。逆にビューポートは、1 つで、サイズや位置はデフォルト、つまり、レンダリングターゲットと一致しています。
各ビューの作成プロセスは前章と全く同じで、ビュー作成の関数も同じものを使っています。（SetViewMatrix 関数）
複数のレンダリングターゲットを作成するための技術的な方法は、" スワップチェーン " を作成することにより実現します。
スワップチェーンとはバックバッファーのことです。Direct3D では (かつては DirectDraw でも) バックバッファーとフレームバッファーをフリップして、テアリングの無い滑らかな画面更新を実現します。ダブルバッファーとかトリプルバッファーとかの言葉を聞いたことがあるでしょう。それぞれバックバッファーを 2 個ないし 3 個使用して、それらを連続的にフリップする手法です。スワップチェーンはメモリ（とビデオカードの性能）の許す範囲の個数で作成できます。

サンプルプログラム

プロジェクトフォルダ名「ch20-2 子ウィンドウで画面分割」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。

使用方法
移動：矢印キー
ヨー回転：Z キーと X キー
ピッチ回転：C キーと V キー
ロール回転：B キーと N キー
スケーリング：A キーと S キー
マルチビュー、シングルビューのトグル：スペースキー

コード解説

24 行目
LPDIRECT3DSWAPCHAIN9 pSwapChain[5+1]={NULL};
フレームウィンドウを含めて必要なスワップチェーンは 5 つです。+1 としているのは筆者の習慣で、配列やその他何らかのメモリ領域を確保するときは必ず 1 つ多めに確保します。これは真似しなくて構いません。結果的にここではスワップチェーンインターフェイスのポインターを 6 つ分宣言していることになります。
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
LRESULT CALLBACK ChildWndProc_0(HWND ,UINT ,WPARAM ,LPARAM);
LRESULT CALLBACK ChildWndProc_1(HWND ,UINT ,WPARAM ,LPARAM);
LRESULT CALLBACK ChildWndProc_2(HWND ,UINT ,WPARAM ,LPARAM);
LRESULT CALLBACK ChildWndProc_3(HWND ,UINT ,WPARAM ,LPARAM);

他のサンプルでは、ウィンドウが一つなので、メッセージプロシージャーも 1 つでした。本サンプルでは 5 つのウィンドウを作成するので、メッセージプロシージャーも 5 つ作成します。一番上がフレームウィンドウ用、その下 4 つが子ウィンドウ用です。

SetViewMatrix 関数
指定された 4 つのビューを作成します。これは、前章と同じものです。

Render 関数
これも、構造は前章と全く同じです。
前章では、ビューポートを変更しましたが、ここではレンダリングターゲットを変更します。（ChangeRenderTarget 関数）

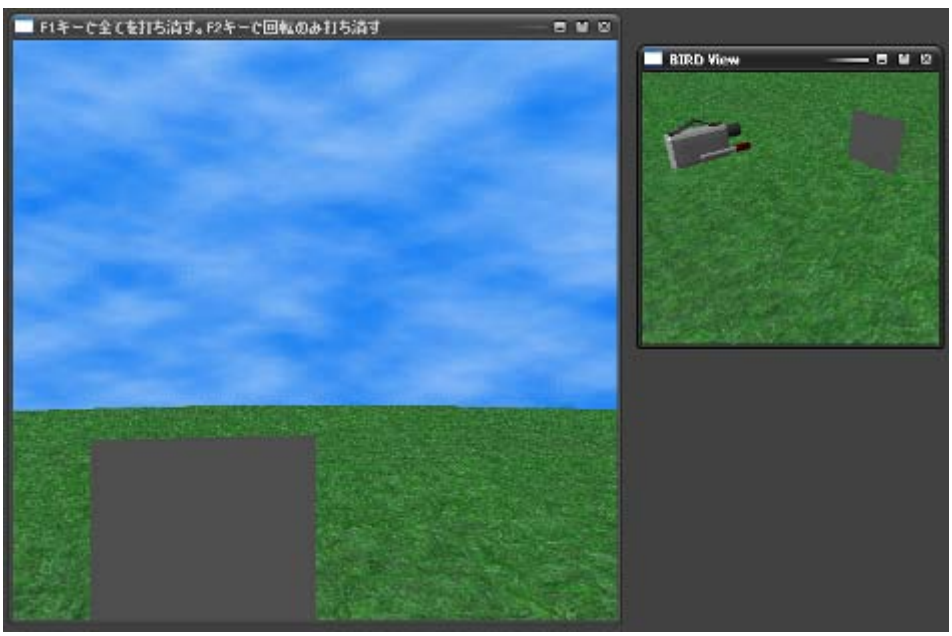
ChangeRenderTarget 関数
LPDIRECT3DSURFACE9 pBackBuffer = NULL;
関数内部でのみ使用する一時的なバックバッファーポインターを宣言します。
pSwapChain[dwIndex]->GetBackBuffer(0,D3DBACKBUFFER_TYPE_MONO, &pBackBuffer);
dwIndex は、どの子ウィンドウかのインデックスです。
その子ウィンドウに対応したスワップチェーンからバックバッファーへのポインターを取得します。
pDevice->SetRenderTarget(0,pBackBuffer);
そのバックバッファーを、その時点のレンダリングターゲットとします。

pSwapChain[dwIndex]->Present(NULL,NULL,hWnd,NULL,NULL);
画面を更新（フリップ）します。

RendeToCurrentWindow 関数
関数名は RendeToCurrentWindow となっていますが、いままで毎回登場していた RenderThing 関数と同じものです。

21 章　ビルボーディング

図 21-1　ミニウィンドウにより、ビルボードの挙動が一目瞭然



ビルボードとは

ビルボードとは、カメラの回転や移動の一部ないしは全部の影響を受けない（正確に言うと影響を打ち消す）ジオメトリを言います。ジオメトリは多くの場合、単純な板ポリゴンです。
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且つ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呼びます。ビルボード（Billboard）とは、「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 “ 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。

ビルボードの目的

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
モニターの固定位置に何らかのインターフェイスとしてレンダリングしたい時

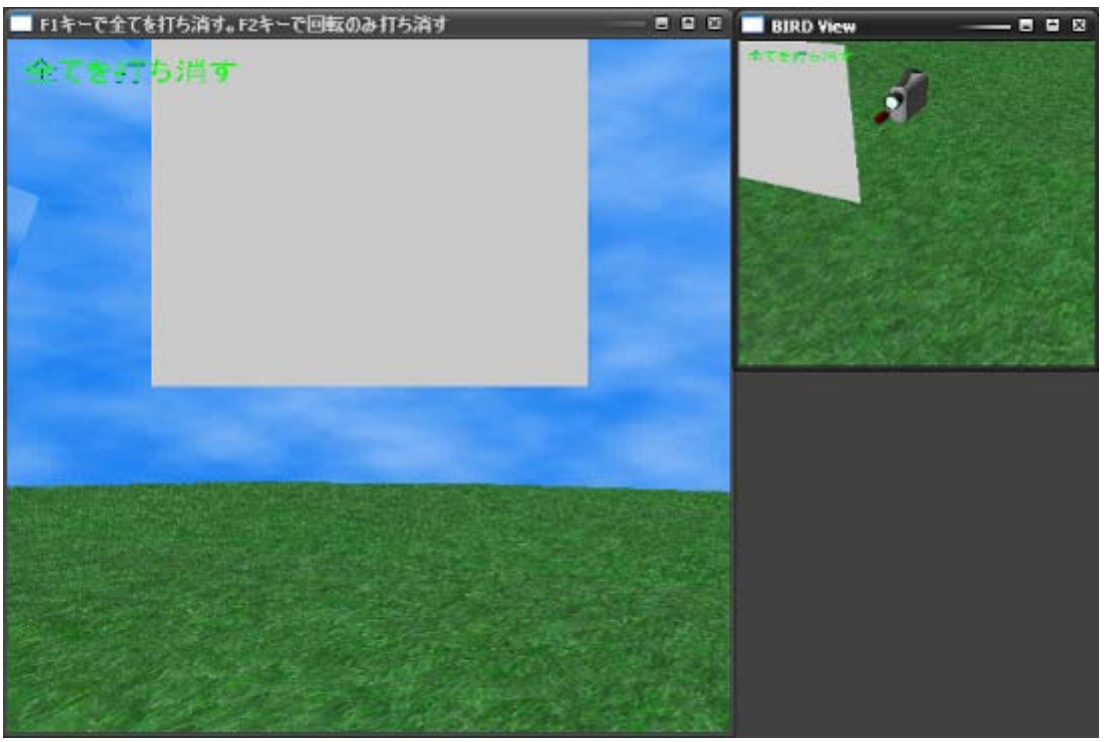
ではどうしてこれらの場合にカメラの姿勢の影響を受けないようにしたいのか、言い換えると、なぜビルボードにしたいのか、その理由は次のとおりです。
雲などの、非常に不規則で粒子の細かい物体をジオメトリ的に表現するとなると、膨大な頂点、ポリゴンを用意しなければなりません。それは、レンダリングコストを考えると非現実的であり、リアルタイムレンダリングである Direct3D で、事実上不可能なのは明らかです。雲などの物体は、もっとも単純な板ポリゴン（3 頂点や 4 頂点）に雲などの絵をテクスチャーとして貼り付けて、“ それらしく ” 見せます。その際、通常のようにカメラの姿勢変化に反応させて板ポリゴンが移動・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（これは有り得ません、ボリュームレンダリングは数分オーダーの時間がかかります）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と同等の結果を高速にレンダリングするトリックです。
ビルボーディングの原理については、「はじめての 3D ゲーム」で解説していますので割愛しますが、要するに、“ カメラ、つまりビュー変換の逆変換をジオメトリかけることにより結果的に影響を受けていないようにする ” ことです。

サンプルプログラム

プロジェクトフォルダ名「ch21 ビルボーディング」
VC6、VC2002、VC2003、3 つのフォルダそれぞれにあります。
余談ですが、ビルボーディングの解説は当初の予定には入っていませんでした。ある日ふと気付き、急遽本章を追加したのですが、好都合なことに子ウィンドウによるマルチビューレンダリングを解説していたので、その仕組みを取り入れ、ビルボードの効果を明確に体感できるようなサンプルを作成することができました。
マルチビューに関わるコード部分は前章と同様なので、もちろん割愛します。

使用方法
カメラメッシュを操作します。メインウィンドウにおいてカメラメッシュはカメラとして機能するので、画面には見えません。それに対しミニウウィンドウではカメラは単なるジオメトリとなります。
移動：矢印キー
ヨー回転：Z キーと X キー
ピッチ回転：C キーと V キー
ロール回転：B キーと N キー
スケーリング：A キーと S キー
回転と移動を打ち消す：F1 キー
回転のみを打ち消す：F2 キー

全てを打ち消した場合は、次の図のようにカメラメッシュの姿勢や位置に関わらず、ビルボード（板ポリゴン）はスクリーン上の一定の場所に常に留まります。
図 21 － 2



回転のみを打ち消した場合、板ポリゴンは常にこちらを向きますが、カメラメッシュの位置が変わるとスクリーン上の位置も変わります。
図 21 － 3

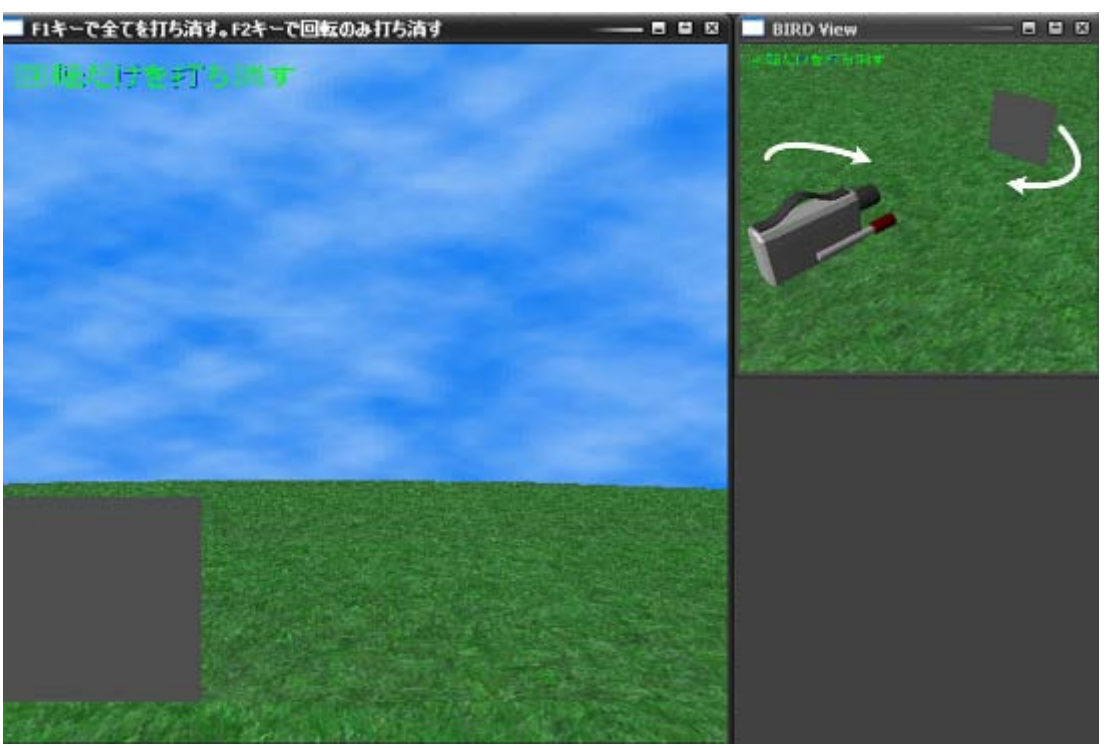
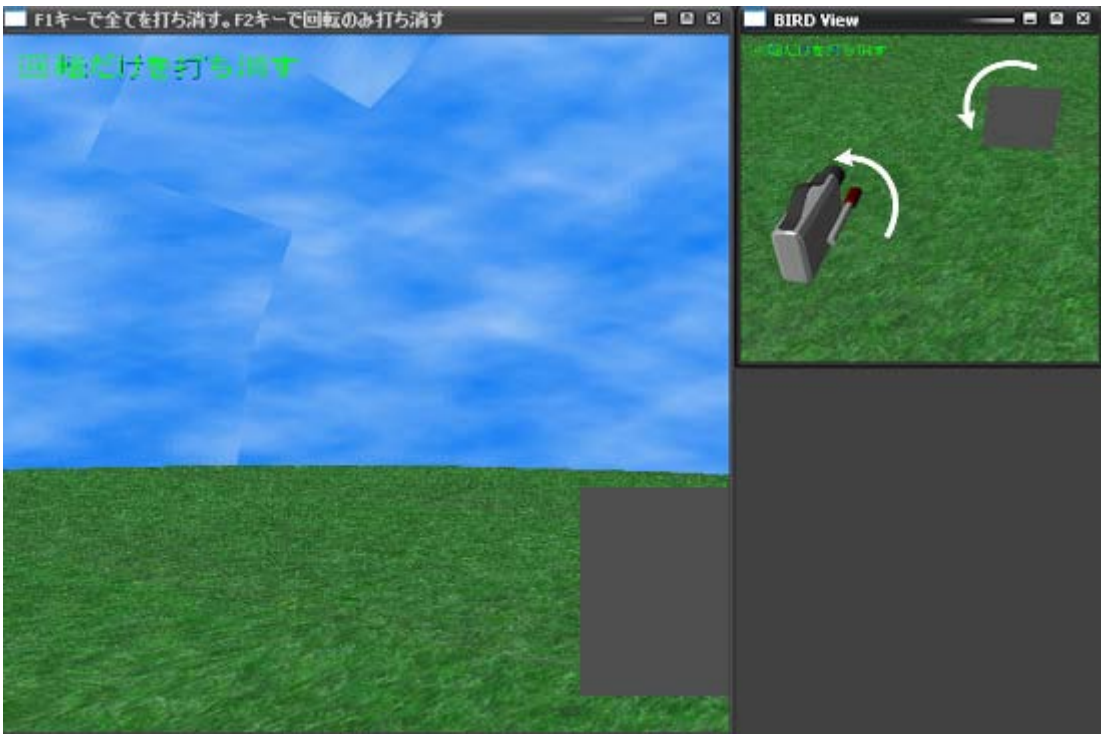


図 21 － 4



コード解説

本サンプルのキーポイント部分は Render 関数及び RenderToCurrentWindow 関数です。ここでの Render 関数及び RenderToCurrentWindow 関数にはビルボーディングのエッセンス全てが凝縮されています。

Render 関数
メインウィンドウ部分　324 行～ 340 行
メインウィンドウには、カメラからの視点による FPV：1 人称ビューをレンダリングします。
カメラメッシュの位置と回転からビュー変換行列を matView として作成し、その行列の逆行列を matBillborard として作成します。
RenderToCurrentWindow 関数により、ドームメッシュ（空メッシュ）とビルボードメッシュ（板メッシュ）をレンダリングします。ビルボードメッシュの場合は、RenderToCurrentWindow 関数の第 3 引数に matBIllboard を渡しています。RenderToCurrentWindow 関数は、第 3 引数に行列が渡されると、その行列でビュー変換を打ち消します。NULL の場合は、ビルボーディングを行いません。
最後に Present して、ひとまずメインウィンドウの画面を更新します。
ミニウィンドウ部分　341 行～ 353 行
ミニウィンドウは、ビルボード及びカメラを TPV：3 人称ビューで見ることにより、効果を視認でき易くするために作成しました。カメラとビルボードジオメトリがどのようにシンクロしているのか、このミニウィンドウからの眺めによってより明らかになるでしょう。
メインウィンドウとほぼ同じ処理です。異なるのは、TPV であるので、視点はカメラメッシュを外部から眺めることができる位置に設定している部分だけです。
最後に、ミニウィンドウの画面も更新します。

RenderToCurrentWindow 関数
ビルボードに関わる部分は次のコード部分です。

```
if(NULL != pmatBillboard)
 {
          D3DXMATRIX matBillboard=*pmatBillboard;
          if(boCancelAll)
          {
                    matWorld=matScale*matRotation*matPosition*matBillboard;
          }
          else if(boCancelRotation)
          {
                    D3DXMATRIX matCancelRotation;
                    matCancelRotation=matBillboard;
                    matCancelRotation._41=0;
                    matCancelRotation._42=0;
                    matCancelRotation._43=0;
                       matWorld=matScale*matRotation*matCancelRotation*matPosition;
          }
 }

if(boCancelAll)
{
          matWorld=matScale*matRotation*matPosition*matBillboard;
}
```
“ 全てを打ち消す “ が選択されている場合は簡単です。ワールド変換行列の最後にビルボード用の行列（ビュー変換行列の逆行列）を掛ければいいだけです。

```
else if(boCancelRotation)
{
          D3DXMATRIX matCancelRotation;
          matCancelRotation=matBillboard;
          matCancelRotation._41=0;
          matCancelRotation._42=0;
          matCancelRotation._43=0;
              matWorld=matScale*matRotation*matCancelRotation*matPosition;
}
```
逆に手間が要るのは、“ 回転のみを打ち消す ” が選択されている場合です。
ビルボード行列の平行移動成分をクリアします。クリアしないと、移動もキャンセル（打ち消し）されてしまい、“ 全てを打ち消す ” と同じになってしまいます。44 行列の 4 行目の横ラインが平行移動の成分であるのは、8 章で解説しています。
そして、ワールド変換行列を作成する際には、“ 回転行列の直後 ” に打ち消す行列を掛けます。これがミソです。
```
matWorld=matScale*matRotation*matCancelRotation*matPosition;
```