HIView Programming Guide

（1）HIViewのコンセプト

このドキュメントでは、HIViewモデルについて解説します。HIViewモデルは、Mac OS XのCarbon Frameworkにおいて、コントロールやメニューなどの新しいユーザーインターフェース描画環境を提供します。HIViewモデルはMac OS X 10.2から導入されており、10.3と10.4でそれぞれ機能が拡張されています。Carbonにおけるコントロールの挙動を詳しく調べたい方は、「Handling Carbon Windows and Controls」ドキュメントを参照してみてください。

・HIViewとは何か？

・HIViewでは 「View」と呼ばれる描画領域が提供されます。
・ViewではCore Graphics（Quartz 2D）がネイティブの描画システムです。
・QuickDrawによる描画も可能ですが10.4以降では推奨されません。
・Viewは浮動小数点で指示する座標系を用います（整数ではない）。
・Viewは階層化を行うことが可能です（別のViewに埋め込み可能）。
・Viewはウィンドウ内に配置可能ですが、そこからの分離もまた可能です。
・サブクラスとしてオリジナルなカスタムViewを作成することが可能です。

Carbon環境において、すべてのユーザインターフェースはView上に描画され、それぞれ階層化されることで管理されています。また、すべてのViewのベースクラスはHIObjectとなります。

・Control ManagerとMenu ManagerとHIViewの関係

HIViewモデルで利用できるAPIは、Control Managerで利用できるAPIをよりモダンにし、拡張したものとなっています。例えば、コントロールのコンポジット（合成）描画などについては、HIViewモデルでしかサポートされていません。加えて、Combo BoxやSegmented Viewのように、HIviewモデルでしか利用できないコントロールも存在しています。ちなみに、Mac OS X 10.3までは、List Box、Scrolling Text Field、Data Browserといったコントロールは、HIViewモデルではなく旧Control Managerモデルとして定義されていますので注意してください。Mac OS X 10.4以降ではこうした制限は

解除されています。

旧Control ManagerのAPIで利用するControlRefは、HIView APIで利用するHIViewRefと同等な意味を持ちます。また旧Control ManagerのAPIのほとんどは、HIView APIで置き換えることが可能です。例えば、HIViewGetValue()はGetControl32BitValue()とまったく同じ働きをします。つまり名称が異なるだけなのですが、今後のMac OS Xの遷移を考慮すると、HIView APIに置き換え可能なものは、そちらに置き換えておいた方が良いでしょう。また、DrawContorols()、Draw1Control()、UpdateControls()などの旧Control Managerの描画APIは、HIViewモデルでは効力を発揮しませんので使用しないでください。旧Control Manager APIについては、ControlDefinitions.hを参照してください。HIView APIについてはHIView.hで参照可能です。

Mac OS X 10.3以降のHIViewモデルでは、メニュータイトルやアイテムの表示領域もHIObjectから継承されたViewとして定義されています。よって、ウィンドウで利用できる旧Control ManagerやHIViewのAPIは、その領域でも同じように利用できることになります。つまり、特別な仕組みを組み込まなくても、メニュータイトルやアイテムに対して好みのカスタム描画が可能なわけです。ただし、MenuRefの意味は、ControlRefやHIViewRefとは異なります。指定されたMenuRefからHIViewRefで参照できる描画用のView（Menu Content ViewやMenu View）を得るには、HIMenuGetContentView()などの適切なAPIを利用します。

・Viewの階層と埋め込み

旧Control Managerでのコントロールの階層描画は、すべてウィンドウ内で処理されていました。一番上位に「Root Control」が存在し、そこから順次コントロールの階層化がなされていきます。例えば、以下のような階層構造となります。

「Root control」->「Radio Group」->「Radio Button」

ここで言うRoot Controlとは抽象的なオブジェクトで、何かしらのコントロールが表示されているわけではありません（インビジブル）。 Root ControlはGetRootControl()に対象ウィンドウのWidnowRefを渡すことで得られました。また、ウィンドウと関連させずに単独でコントロールを確保するようなことはできませんでした。

HIViewモデルにはRoot Controlの代わりに「Root View」が存在します。その下の階層に「Close Button」「Resize Control」「Content View」などが続き、これらがウィンドウの構成要素となります。ウィンドウのRoot Viewを得るには、HIViewGetRoot()を用います。例えば、ドキュメントウィンドウの白紙領域は「Content View」に一致しています。まとめると、以下のような階層構造となります。

「Root View」->「Content View」->「Radio Group」->「Radio Button」

メニューの場合もウィンドウと同じで、最上部には「Root View」が存在します。あるViewの上位階層Viewを「Supreview」と呼び、下位階層Viewを「Subview」と呼びます。Viewベースのコントロールは、Tab Controlとその内部コントロールの関係と同じように、SubViewをSupreviewに埋め込む（Embedding）ことが可能です。また、同じ階層のViewベースのコントロールには順序（Ordering）があり、この情報により、どちらを上に表示するのかが決定されます。

「Root View」->「Content View」->「Content View」

また、Viewベースのコントロールが旧コントロールと大きく違う点は、それがウィンドウと関連付けず単独で定義できることです。当然、この場合に表示はなされていませんが、カット＆ペースト処理などで、単独オブジェクトとして別ウィンドウに移動させるような処理が簡単にできます。

・描画モデル

ウィンドウ上にHIViewベースでのコントロールを描画する場合には、そのウィンドウがコンポジット（合成）モードである必要があります。そうでないと、すべてのコントロールに対して旧Control Managerベースの描画が採用されます。コンポジットウィンドウでのViewの描画順序は、同じ階層での順序情報により決まり、描画時のα値（透明度）も正しく認識されます。その結果として、描画におけるコントロールの重なり具合も美しく再現されます。コンポジットウィンドウにおけるViewの再描画については、Carbon Event Handlerを用意し、イベントクラスがkEventClassControlで、そのイベント種類がkEventControlDrawのCarbon Eventを受け取った時に実行するようにします。

旧WaitNextEvent()ループによるアップデートイベント時の再描画や、Carbon EventのEventWindowUpdateやkEventWindowDrawを受け取ったことを認知しての再描画は行えませんので注意してください。なぜなら、コンポジットウィンドウでは、そうしたイベントは発生しないからです。もし直ちにViewの再描画を行いたい場合には、目的に応じたAPIを用いて「再描画領域」を記録させます。例えば、HIViewSetNeedsDisplay()やHIViewSetNeedsDisplayInRegion()などです。また、Mac OS X 10.4以降ではHIViewSetNeedsDisplayInRect()やHIViewSetNeedsDisplayInShape()なども利用することができます。この時の再描画は、RunApplicationEventLoop()やWaitNextEvent()で実行されているイベントループの処理タイミングに依存します。もし、イベントループに依存せず、直ちにViewの再描画を行いたい場合には、Mac OS X 10.3以降で利用できるHIViewRender()を用います。

・描画のための座標系

HIViewで描画を行う場合の座標系は、左上が原点となります。これは一般ウィンドウやQuickDrawの座標系と同じです。これと比較してCore Graphics（Quartz 2D）による描画の原点は左下となっています。

HIViewモデルでの描画は、ウィンドウが右下へ拡大されることを考慮した座標系を使います。このため、Viewから得られたCGContextRefを対象にしてCore Graphics APIで何らかの描画を行うと、座標原点は自動的に左下から左上に変換されることになります。ただし、この状態でCGContextDrawImage()などのCore Graphics Image描画APIを用いると、表示される画像も上下反転してしまいます。この現象を回避するには、Mac OS X 10.3から利用できるHIViewDrawCGImage()を用います。

HIVIewモデルでは、Viewのオリジナル矩形枠を「Local Bounds」と呼び、そのViewのSuperviewの原点を基準にした矩形枠を「Frame Bounds」と呼びます。こうした座標に関わるパラメータをやり取りする場合には、HIPoint、HISize、HIRectといった構造体が利用されますが、これらはCore Graphics APIで用いられているCGPoint、CGSize、CGRectと同等であり、すべての値は浮動小数点で表記されます。

・HIViewベースの新しいユーザインターフェース

HIViewベースの新しいユーザインターフェースは、現在以下の7種類が用意されています。このうち、Search Field、Segmented View、Text View、Web Viewは、Mac OS X 10.3以降のみ、Movie ViewはMac OS X 10.4以降のみで利用できます。また、Web Viewに関しては、同時にSafari 1.0以上がインストールされていることが必要です。

（1）Combo Box

編集可能テキストフィールドとポップアップメニューを組み合わせたユーザインターフェースです。旧Control Manager APIからもアクセスすることが可能です。

（2）Image View

昔からあるPicture表示用のコントロールと同様に、Core Graphics Imageを表示するためのViewです。

（3）Scroll View

何らかの情報表示の上下左右スクロールをサポートすることができるViewです。通常は、Image ViewやText Viewを埋め込んで（Embeded）利用されます。

（4）Search Field

Finder、Mail、Safari等で利用されている、検索情報を入力するための編集可能テキストフィールドです。検索項目を一覧するためのポップアップメニューも利用できます。

（5）Segmented View

複数ボタンを区切り表示し、グループ内の特定アイテムを選択するためのコントロールです。Finderのウィンドウにおいて、アイコン、リスト、カラムビューを切り替えるために用いられています。

（6）Text View

MLTE （Multilingual text engine）を利用した編集可能なテキスト領域です。Scroll Viewに埋め込むことで、上下左右のスクロールも可能です。

（7）Web View

ウェブ・コンテンツを表示するためのViewです。現状ではInterface Builderから配置することはできません。また、Carbon APIで作成することは可能ですが、内容を制御するにはObjective-CベースのCocoa API（Web Kit）を利用する必要があります。

（7）Movie View

QuickTime Movieの再生と制御を行うためのViewです。詳細についてはQuickTime FrameworkのHIMovieView.hを参照してください。

（2）HIViewのタスク

HIViewモデルの標準タスクには、作成、埋め込み、自動レイアウト、順序変更、描画、Viewベースの自作コントロールの実装などがあります。

・コンポジットウィンドウの作成

ウィンドウ上にHIViewベースでのコントロールを描画する場合には、そのウィンドウがコンポジット（合成）モードである必要があります。ウィンドウをコンポジットモードに切り替えるには、CreateNewWindow()を実行する時点で、そのアトリビュートビットにkWindowCompositingAttributeをセットします。もし、nibファイルベースのウィンドウを用いているなら、Interface Builderのインスペクター（Inspactor）ダイアログで、ウィンドウオブジェクトのアトリビュートである「Compositing」をONにしておきます。ちなみに、ChangeWindowAttributes()を使い、表示されたウィンドウを後からコンポジットモードへ切り替えることは不可能ですので注意してください。

DrawControls()やDraw1Control()といったQuickDraw依存のコントロール再描画用APIは、HIViewSetNeedsDisplay()といったモダンAPIに切り替えてください。また、コンポジットモードでは、逐次コントロールの背景を消すような処理は必要とされません。

・Viewの作成

ソースコードからダイレクトにViewを作成する場合、古いコントロールについては、今まで利用してきた旧Control ManagerのAPIを利用します（CreateCheckBoxControl()など）。この時、引数のWindowRefにNULLを渡すと、どのウィンドウにも属さないコントロールを得ることが出来ます。新しいHIViewベースのコントロールやViewについては、以下のAPIを利用します。

Combo box	HIComboBoxCreate
Image View	HIImageViewCreate
Scroll View	HIScrollViewCreate
Segmented View	HISegmentedViewCreate
Search Field	HISearchFieldCreate
Text View	HITextViewCreate
Web View	HIWebViewCreate
Movie View	HIMovieViewCreate

作成後の表示位置の調整には、 HIViewSetFrame()やHIViewPlaceInSuperviewAt()を用います。また、作成直後のViewは表示不可となっていますので、HIViewSetVisible()かShowControl()を用いて表示可に切り替える必要があります。また、Web View以外は、Interface Builderによりウィンドウに配置してnibファイルへ保存し、アプリケーションから利用することも可能です。メニューに関しては自動的に描画用のViewが作成されますので（Mac OS X 10.3以降）、特別な操作は必要ありません。

・Viewの埋め込み（Embedding）

あるViewをSuperviewへ埋め込む場合には、HIViewAddSubview()を用います。埋め込み後の配置変更には、HIViewSetFrame()やHIViewPlaceInSuperviewAt()を用います。

```
OSStatus HIViewAddSubview (HIViewRef inParent, HIViewRef inChild);
```

Viewの削除には、 HIViewRemoveFromSuperview()を用います。

```
OSStatus HIViewRemoveFromSuperview (HIViewRef inView);
```

あるViewのSupreviewを得るにはHIViewGetSuperview()を用います。

```
HIViewRef HIViewGetSuperview (HIViewRef inView);
```

ウィンドウのRoot Viewを得るにはHIViewGetRoot()を用います。

```
HIViewRef HIViewGetRoot (WindowRef inWindow);
```

・Viewの配置

ViewのLocal Boundsを得るのにはHIViewGetBounds()を用います。設定する方は、HIViewSetFrame()です。

```
OSStatus HIViewGetBounds (HIViewRef inView, HIRect *outRect);
OSStatus HIViewSetFrame (HIViewRef inView, const HIRect *inRect);
```

ViewのFrame Boundsを得るのにはHIViewGetFrame()を用います。これは旧Control ManagerのGetControlBounds()で得られるウィンドウに対するローカル座標と同じ意味を持ちます。

```
OSStatus HIViewGetFrame (HIViewRef inView, HIRect *outRect);
```

Viewの位置を平行移動させるには、HIViewMoveBy()を用います。また、特定座標へ移動させるには、HIViewPlaceInSuperviewAt()を用います。

```
OSStatus HIViewMoveBy (HIViewRef inView, float inDX, float inDY);
OSStatus HIViewPlaceInSuperviewAt (HIViewRef inView,float inX, float inY);
```

・Viewの自動レイアウト

Mac OS X 10.3以降のHIViewモデルから、Viewの自動レイアウト機能を利用できるようになりました。これにより、対象コントロールのSuperviewや隣のコントロールのフレーム枠が変化したとしても、それと相対的な位置を保って自動でコントロールを配置し直すことが可能となりました。自動レイアウトによる位置決めやサイズ決めのために、以下の3つの情報が用意されています。

（a）Bind情報（他のViewとの縁との距離を保つ）
（b）Scale情報（他のViewとの相対サイズを保つ）
（c）Position情報（他のViewとの相対位置を保つ）

自動レイアウト情報はHILayoutInfo構造体に代入し、HIViewGetLayoutInfo()を用いて特定のView（コントロール）に設定します。現状では、構造体にメンバーのversionには、kHILayoutInfoVersionZeroをセットしてください。その設定を有効にするのにはHIViewApplyLayout()を、一時的に無効にするにはHIViewSuspendLayout()を、その状態から再度有効にするにはHIViewResumeLayout()を、現在有効なのか無効なのかを調べるにはHIViewIsLayoutActive()を利用します。また、現在設定されている自動レイアウト情報を参照するには、HIViewGetLayoutInfo()を用います。nibファイルベースのコントロールを用いる場合には、Interface Builderのインスペクター（Inspactor）ダイアログのlayoutタグで、Superviewに対するBind情報を設定しておくことが可能です。

```
struct HILayoutInfo {        // 自動レイアウト情報構造体
    UInt32 version;          // kHILayoutInfoVersionZeroを代入
    HIBinding binding;       // Bind情報報のための構造体
    HIScaling scale;         // Scale情報報のための構造体
    HIPositioning position;  // Position情報のための構造体
    };

OSStatus HIViewSetLayoutInfo (HIViewRef inView,
                    const HILayoutInfo inLayoutInfo);   // レイアウト情報の設定
OSStatus HIViewGetLayoutInfo (HIViewRef inView,
                        HILayoutInfo *outLayoutInfo);   // レイアウト情報を得る

OSStatus HIViewApplyLayout (HIViewRef inView);      // レイアウト情報を有効
OSStatus HIViewSuspendLayout (HIViewRef inView);  // 一時的に無効にする
OSStatus HIViewResumeLayout (HIView inView);        // 無効から復帰させる
Boolean HIViewIsLayoutActive (HIViewRef inView);    // 有効か無効かをチェック
```

HlLayoutInfo構造体のメンバーには、HlBinding、HlScaling、HlPositioningの3つの構造体が含まれており、それぞれBind情報、Scale情報、Position情報を保持しています。

（a）Bind情報

Bind情報はHlBinding構造体に格納されます。HlBinding構造体には、メンバーとして4つのHlSideBinding構造体が含まれており、それぞれが上下左右の縁と別のViewの縁との距離を一定に保つかどうかを指定します。HlSideBinding構造体のtoViewには、関係を保つViewのHlViewRefを設定します。もし縁を合わせる対象がSuperviewであれば、NULLを代入しておきます。またkindには、固定する縁の方向をkHlLayoutBindTop、kHlLayoutBindLeft、kHlLayoutBindBottom、kHlLayoutBindRightから選択して代入します。固定したくない場合には、kHlLayoutBindNoneをセットします。

```
struct HlBinding {
    HlSideBinding top;
    HlSideBinding left;
    HlSideBinding right;
    HlSideBinding bottom;
    }

struct HlSideBinding {
    HlViewRef toView;
    HlBindingKind kind;
    };
```

（b）Scale情報

Scale情報はHlScaling構造体に格納されます。HlScaling構造体には、メンバーとして2つのHlAxisScale構造体が含まれており、それぞれX方向およびY方向に対象となるViewとのサイズ比率を代入できます。HlAxisScale構造体のtoViewの意味はBind情報と同じです、またkindにはkHlLayoutScaleAbsoluteを代入しておきます。ratio値にゼロを代入すれば、スケール情報は無視されます。1.0を代入すれば、関係を保つViewと同じサイズを保つことを指示します。同様に0.5であれば、半分のサイズを保つことになります。

```
struct HlScaling {
    HlAxisScale x;
    HlAxisScale y;
```

```
struct HIAxisScale {
    HIViewRef toView;
    HIScaleKind kind;
    float ratio;
    };
```

（c）Position情報

Position情報はHIPositioning構造体に格納されます。HIPositioning構造体には、メンバーとして2つのHIAxisPosition構造体が含まれ、それぞれX方向およびY方向に対象となるViewとの相対位置を設定できます。HIAxisPosition構造体のtoViewの意味はBind情報と同じです、またkindにはkHILayoutPositionCenter、kHILayoutPositionLeft、kHILayoutPositionRight、kHILayoutPositionTop、kHILayoutPositionBottomを代入します。また、Position情報を無効にするには、kHILayoutPositionNoneをセットします。offset値には、位置合わせした後にズレさせた距離をピクセル数で代入します。ゼロを代入しておけばズレは生じません。例えば、x,yにkHILayoutPositionCenterをセットし、offsetがゼロであれば、必ず関係を保ちたいViewの中心に表示されます。

```
struct HIPositioning {
    HIAxisPosition x;
    HIAxisPosition y;
    }

struct HIAxisPosition {
    HIViewRef toView;
    HIPositionKind kind;
    float offset;
    };
```

・Viewの順序変更

同じ階層に表示されているViewベースのコントロールには順序があり、それが表示順位に反映されています。表示順位は、HIViewSetZOrder()によって変更することが可能です。引数のinOpには、kHIViewZOrderAbove（上位へ移動）かkHIViewZOrderBelow（下位へ移動）を設定できます。inOtherには、順序を変更する時に対象となるViewをセットできますが、ここにNULLを代入しておけば、その階層での一番上位または下位への移動を指示したことになります。Supreviewに含まれるSubviewの中で一番最初（上位）のViewを得るにはHIViewGetFirstSubview()を用い、一番最後（下位）のViewをる

にはHIViewGetLastSubview()を用います。また、あるViewの次と前のViewをるには、HIViewGetNextView()とHIViewGetPreviousView()を用います。

```
OSStatus HIViewSetZOrder (HIViewRef inView, HIViewZOrderOp inOp,
                                        HIViewRef inOther);  // 順序を設定する

HIViewRef HIViewGetFirstSubview (HIViewRef inParentView);  // 最初のViewへ
HIViewRef HIViewGetLastSubview (HIViewRef inParentView);  // 最後のViewへ
HIViewRef HIViewGetNextView (HIViewRef inView);                  // 次のViewは
HIViewRef HIViewGetPreviousView (HIViewRef inView);           // 前のViewは
```

・Viewの表示と非表示

Viewの表示と非表示の切換は、HIViewSetVisible()で行います。また、現在特定のViewが表示されているのかどうかを知るには、HIViewIsVisible()を用います。これらのAPIは旧Control Manager APIのSetControlVisibility()とIsControlVisible()とまったく同じ働きをします。

```
OSStatus HIViewSetVisible (HIViewRef inView, Boolean inVisible); // 表示の指示
Boolean HIViewIsVisible (HIViewRef inView);  // 表示されているかどうかチェック
```

・キーボードフォーカスのチェック

Mac OS X 10.2以降から、すべてのViewはキーボードフォーカスを受け付けることが可能です。それ以前では、テキスト入力やリスト関連のコントロールのみ可能でした。ただし、この機能を利用するには「システム環境設定」の「キーボード」の「フルキーボードアクセス：」を「すべてのコントロール」へ切り替えておく必要があります。通常フォーカス移動はtabキーで機能しますが、強制的に次のViewへとキーボードフォーカスを切り替えるのには、HIViewAdvanceFocus()を用います。

```
OSStatus HIViewAdvanceFocus (HIViewRef inRootForFocus,
                                        EventModifiers inModifiers);  // フォーカスを次へ
```

ウィンドウに並んだViewに対してフォーカス処理を行うには、引数のinRootForFocusにHIViewGetRoot(window)を渡します（windowはWindowRef）。対象となるViewには、クラスがkEventClassControlで、種類がkEventControlSetFocusPartのCarbon Eventが送られてきます。HIViewAdvanceFocus()は独自のアルゴリズムで（左から右、上から下）次にフォーカスするViewを選びます。もし強制的に指定したViewへフォーカ

スを移したい場合には、HIViewSetNextFocus()を用います。

```
OSStatus HIViewSetNextFocus (HIViewRef inView,HIViewRef inNextFocus);
```

・座標変換

同じSuperviewを持つ2つのView間で座標変換を行うことが可能です。座標変換の対象がHIPointであればHIViewConvertPoint()を、HIRectであればHIViewConvertRectを、RgnHandleならばHIViewConvertRegion()を用います。引数のHIViewRefにNULLを代入すれば、それはRoot Viewを指示したのと同じ意味となります。

```
OSStatus HIViewConvertPoint (HIPoint *inPoint, HIViewRef inSourceView,
                                                HIViewRef inDestView);

OSStatus HIViewConvertRect (HIRect *ioRect, HIViewRef inSourceView,
                                                HIViewRef inDestView);

OSStatus HIViewConvertRegion (RgnHandle ioRgn, HIViewRef inSourceView,
                                                HIViewRef inDestView);
```

・Viewへの描画

Viewへの何らかの描画については、すべてのCore Graphics APIを利用することができます。QuickDraw APIも利用できますが、Mac OS X 10.4以降では推奨されていません。Viewにおける描画の座標系はQuickDrawと同じですので、特定のView上にCore Graphics Imageを描画しようとすると、上下反対に表示されてしまいます。そのため、View上に正しくImageを描画するためのHIViewDrawCGImage()が用意されています。

```
 OSStatus HIViewDrawCGImage (CGContextRef inContext,
                                const HIRect *inBounds, CGImageRef inImage);
```

もし、Viewにおける座標系をCore GraphicsのCGContext座標系と一致させたい場合には、以下の処理を実行してやればOKです。

```
HIViewGetBounds (view, &bounds);
CGContextTranslateCTM (theContext, 0, bounds.size.height);
CGContextScaleCTM (theContext, 1.0, -1.0);
```

Viewの再描画は、Carbon EventのkEventControlDraw（Viewの再描画が必要な時に発生する）を受け取ったことを認知した時に実行します。もし、直ちにViewの再描画を行いたい場合には、HIViewSetNeedsDisplay()やHIViewSetNeedsDisplayInRegion()を用います。また、Mac OS X 10.4以降では、HIViewSetNeedsDisplayInRect()やHIViewSetNeedsDisplayInShape()も利用可能です。この時、引数のinNeedsDisplayには決してfalseを代入しないでさい。指定された領域のView描画のアップデートが発生しなくなってしまいます。

```
OSStatus HIViewSetNeedsDisplay (HIViewRef inView, Boolean inNeedsDisplay);

OSStatus HIViewSetNeedsDisplayInRegion (HIViewRef inView,
                                        RgnHandle inRgn, Boolean inNeedsDisplay);

OSStatus HIViewSetNeedsDisplayInRect( HIViewRef inView,
                                      const HIRect *  inRect, Boolean inNeedsDisplay);

OSStatus HIViewSetNeedsDisplayInShape( HIViewRef inView,
                                       HIShapeRef    inArea, Boolean inNeedsDisplay);
```

上記APIによる再描画から特定Viewを除外したい時には、HIViewSetDrawingEnabled()を用います。また、再描画の対象になっているのかどうかをチェックしたい場合には、HIViewIsDrawingEnabled()を用います。

```
OSStatus HIViewSetDrawingEnabled (HIViewRef inView, Boolean inEnabled);

Boolean HIViewIsDrawingEnabled (HIViewRef inView);
```

・Menu Viewの操作

Mac OS X 10.3以降、メニューのアイテム表示領域はViewとして実装されています。つまり、ウィンドウ上のViewに対して実行できるすべての描画は、各メニューのContent View対しても実行できるわけです。以下は、特定のメニューからそのHIViewRefを抽出し、それに対してCarbon Event Handerを実装するソースコードのサンプルです。

```
IBNibRef            nibRef;
MenuRef             myMenuRef;
HIViewRef           theView;
EventTypeSpec   myEvent;
```

```
CreateMenuFromNib (nibRef, CFSTR (“myMenu”),&myMenuRef);
    // nibファイルからメニューを作成する
InsertMenu (myMenuRef, 0);
    // そのメニューをメニューバーに追加する
HIMenuGetContentView (myMenuRef, kThemeMenuTypePullDown, &theView);
    // そのメニューのContent Viewを得る
myEvent.eventClass = kEventClassControl;
myEvent.eventType = kEventControlOwningWindowChanged;
InstallControlEventHandler (theView, myPictureHandler, 2, myEvent,0, NULL);
    // Content Viewに対応するCarbon Event Handlerを登録する
```

メニューはCreateMenuFromNib()を使いnibファイルから取り出します。そのメニューのContent View（HIViewRef）は、HIMenuGetContentView()に対象のMenuRefを渡して得ます。メニュー描画のタイミングは、イベントクラスがkEventClassControl、イベント種類がkEventControlOwningWindowChangedのCarbon Eventを受け取った時となりますので、それを処理するためのCarbon Event Handerを登録しています。以下が、メニューのContent ViewにJPEG画像を描画するために実装されたCarbon Event HanderのmyPictureHandler()ルーチンです。

```
OSStatus PictureHandler( EventHandlerCallRef caller, EventRef event,
                                                        void* refcon )
{
    OSStatus    err = eventNotHandledErr;
    WindowRef   owner;

     // Carbon EventからメニューのtOwningWindowのWindowRefを得る
    GetEventParameter( event, kEventParamControlCurrentOwningWindow,
                    typeWindowRef, NULL, sizeof( owner ), NULL, &owner );
    if ( owner != NULL )
    {
        // メニューのContent Viewを探す
        HIViewRef content;
        HIViewFindByID( HIViewGetRoot( owner ), kHIViewWindowContentID,
                        &content );

        // Image（JPEG画像）のData Providerを作成する（Core Graphics参照）
        CFURLRef url = CFBundleCopyResourceURL( CFBundleGetMainBundle(),
                        CFSTR("GoldenGate"), CFSTR("jpg"), NULL );
        CGDataProviderRef data = CGDataProviderCreateWithURL( url );
```

```
    CFRelease( url );

    // Core Graphics Imageを作成する
    CGImageRef image = CGImageCreateWithJPEGDataProvider( data, NULL,
                            true, kCGRenderingIntentDefault );
    CFRelease( data );

    // Core Graphics Imageを表示するためにImage Viewを作成する
    HIViewRef imageView;
    HIImageViewCreate( image, &imageView );
    HIImageViewSetOpaque( imageView, false );  // 半透明表示を許す
    HIImageViewSetAlpha( imageView, 0.3 );       // α値30%に設定
    CFRelease( image );

    // Image ViewをContent Viewのsubviewとし表示順序を一番下に下げる
    HIViewAddSubview( content, imageView );
    HIViewSetZOrder( imageView, kHIViewZOrderBelow, NULL );
    HIViewSetVisible( imageView, true );  // これを実行しないと表示しない

    // Image Vewの表示領域をContent Viewに一致させる
    HIRect bounds;
    HIViewGetBounds( content, &bounds );
    HIViewSetFrame( imageView, &bounds );
}
```

この例はあくまでもサンプルです。実際には、メニューに何らかの画像を描画するような処理は「Aqua human interface guidelines」で禁じられていますので、行わないでください。

・Scroll Viewの作成

Scroll Viewは、Image Viewを埋め込むことで、画像ウィンドウの上下左右スクロール機能を簡単に実現できます。以下は、その簡単なサンプルソースコードです。Image ViewにJPEG画像を表示し、それをScroll Viewに埋め込むことで上下左右のスクロール機能を実現しています。もし、自作したカスタムViewをScroll Viewに埋め込みたい場合には、Carbon EventのkEventClassScrollableに対応させる必要があります（HIView.hを参照）。

```
WindowRef               scrollWindow;
```

```
Rect                    windowBounds = {100, 100, 500, 550};
HIRect                  myViewRect;
HIViewRef               myImageView, myScrollView, myContentView;
CGImageRef              myImage;
CGDataProviderRef   myProvider;
CFBundleRef             theAppBundle;
CFStringRef             filename;
CFURLRef                theURL;

// ウィンドウをコンポジットモードに設定することを忘れない
CreateNewWindow (kDocumentWindowClass,
kWindowStandardHandlerAttribute ¦
            kWindowCompositingAttribute, &windowBounds,&scrollWindow);

// JPEG画像をファイル（アプリパッケージのResourcesフォルダ内）から得る
theAppBundle = CFBundleGetMainBundle();
filename = CFStringCreateWithCString(NULL, "CowPuppy.jpg",
                                                    kCFStringEncodingASCII);
theURL = CFBundleCopyResourceURL (theAppBundle, filename, NULL, NULL);

// JPEG画像のData Providerを作成すしてCore Graphics Imageを作成する。
myProvider = CGDataProviderCreateWithURL(theURL);
myImage = CGImageCreateWithJPEGDataProvider (myProvider, NULL, false,
                                                kCGRenderingIntentDefault);
CGDataProviderRelease (myProvider);
CFRelease(filename);

// Scroll  Viewを作成してContent Viewへ埋め込むみ表示位置を調整する
HIScrollViewCreate (kHIScrollViewOptionsVertScroll ¦
                        kHIScrollViewOptionsHorizScroll, &myScrollView);
HIViewSetVisible (myScrollView, true);  //表示するためには必ず実行する
HIViewFindByID(HIViewGetRoot(scrollWindow), kHIViewWindowContentID,
                                                    &myContentView);
HIViewAddSubview (myContentView, myScrollView);
myViewRect.origin.x = 50.0;
myViewRect.origin.y = 10.0;
myViewRect.size.width = 300.0;
myViewRect.size.height = 300.0;
HIViewSetFrame (myScrollView, &myViewRect);
```

```
// JPEG画像を表示するためのImage  Viewを作成しScroll Viewへ埋め込む
HIImageViewCreate (myImage, &myImageView);
CGImageRelease (myImage);
HIViewSetVisible (myImageView, true);  //表示するためには必ず実行する
HIViewAddSubview (myScrollView, myImageView);

//ウィンドウを表示する
ShowWindow (scrollWindow);
```

・Combo Boxの作成

Combo BoxはHIComboBoxCreate()で作成します。作成時には、いくつかのアトリビュートを設定することができます。

kHIComboBoxAutoCompletionAttributeをONにすると、キー入力の途中で一致する文字列（メニュー項目）を見つけて自動で補完します。

kHIComboBoxAutoDisclosureAttributeをONにすると、キー入力の途中で一致する文字列（メニュー項目）をリストから見つけてリスト上で反転表示させます。

kHIComboBoxAutoSizeListAttributeをONにすると、文字列を表示しているリスト（メニュー）領域のサイズを自動で計算します。このアトリビュートをOFFにした場合には、SetControlData()を用いて、自分自身でサイズを設定する必要があります。

kHIComboBoxStandardAttributesは、上記3つのアトリビュートを同時にONするために用います。

kHIComboBoxAutoSortAttributeをONにすると、文字列（メニュー項目）をソートしてからリストに表示します。

以下は、Combo Box作成のサンプルソースコードです。

```
WindowRef   myPrefsWindow;
HIViewRef     myCombo, myContentView;
HIRect          hiRect;

// ウィンドウの表示位置とCombo Boxのサイズを設定する
Rect myBounds = {100, 100, 500, 500};
```

```
Rect ComboRect = {0, 0, 20, 100};

// Combo Boxを表示するウィンドウを作成する（コンポジットモード）
CreateNewWindow (kMovableModalWindowClass,
                 kWindowStandardHandlerAttribute¦
                 kWindowCompositingAttribute, &myBounds,&myPrefsWindow);

// Combo Boxを作成してContent Viewに埋め込む
hiRect.origin.x = ComboRect.left;
hiRect.origin.y = ComboRect.right;
hiRect.size.width = ComboRect.right-ComboRect.left;
hiRect.size.height = ComboRect.bottom-ComboRect.top;
HIComboBoxCreate (&hiRect, CFSTR ("Hobbes"), NULL, NULL,
                                    kHIComboBoxStandardAttributes, &myCombo);
HIViewSetVisible (myCombo, true);  //表示するためには必ず実行する
HIViewFindByID (HIViewGetRoot(myPrefsWindow),kHIViewWindowContentID,
                                                          &myContentView);
HIViewAddSubview (myContentView, myCombo);

// Combo Boxの表示位置を平行移動する
HIViewPlaceInSuperviewAt (myCombo, 25.0, 25.0);

// Combo Boxのリスト（メニュー項目）を4つ追加する
HIComboBoxAppendTextItem (myCombo, CFSTR ("Hobbes"), NULL);
HIComboBoxAppendTextItem (myCombo, CFSTR ("Plato"), NULL);
HIComboBoxAppendTextItem (myCombo, CFSTR ("Heidegger"), NULL);
HIComboBoxAppendTextItem (myCombo, CFSTR ("Aristotle"), NULL);

//ウィンドウを表示する
ShowWindow (myPrefsWindow);
```

・カスタムViewの作成

Hiviewモデルの最大の特徴は、ユーザ定義によるカスタムView（自前のウィンドウやコントロール）を作成し、それを、すでに存在しているViewのサブクラスとして定義できることです。 カスタムViewの各メソッド（Method）はCarbon Event Handlerで定義され、 コンストラクタ（Constructor）で確保されたインスタンスデータ（Instance Data）は、Event Handlerルーチン内で利用することができます。

（a）HIObjectサブクラスのレジスト（登録）

自分自身のカスタムウィンドウやコントロール（カスタムView）を作成するには、まずはそのクラスを、HIObjectRegisterSubclass()によりレジストする必要があります。

```
OSStatus HIObjectRegisterSubclass (CFStringRef inClassID,
                                   CFStringRef inBaseClassID,
                                   OptionBits inOptions,
                                   EventHandlerUPP inConstructProc,
                                   UInt32 inNumEvents,
                                   const EventTypeSpec *inEventList,
                                   void* inConstructData,
                                   HIObjectClassRef *outClassRef);
```

実際のHIObjectサブクラスのレジストは以下のような処理となります。

```
#define kMyCustomViewClassID CFSTR(“com.myCorp.myApp.myView”)

OSStatus myCustomViewRegister()
{
    OSStatus                err = noErr;
    static HIObjectClassRef sMyViewClassRef = NULL;

    if ( sMyViewClassRef == NULL )  // アプリ実行後 1 度だけレジストすれば良い
    {
        EventTypeSpec       eventList[] = {                     // 3つはコンストラクタ用
            { kEventClassHIObject, kEventHIObjectConstruct }, // インスタンス作成
            { kEventClassHIObject, kEventHIObjectInitialize },    // インスタンス初期化
            { kEventClassHIObject, kEventHIObjectDestruct },   // インスタンス削除
                                                                  // 4つはメソッド用
            { kEventClassControl, kEventControlInitialize },       // Viewの初期化
            { kEventClassControl, kEventControlDraw },             // Viewの描画
            { kEventClassControl, kEventControlHitTest },          // Viewのヒットテスト
            { kEventClassControl, kEventControlGetPartRegion } };  // 部分領域計算

        err = HIObjectRegisterSubclass(
            kMyCustomViewClassID,  // レジストしたい独自クラスのID （CFString）
            kHIViewClassID,                // ベースクラスのID CFSTR("com.apple.hiview")
```

```
            NULL,                       // オプションビット
            myCustomViewHandler,    // Construct&Method用Event Handlerルーチン
            GetEventTypeCount( eventList ),    // レジストするEventリストの個数
            eventList,                  // レジストするEventリスト
            NULL,                       // Construct用のデータ（今回は無し）
            &sMyViewClassRef );     // 返されたHIObjectClassRef
    }                                   // 返す必要なければNULLを代入してもOK
    return err;
}
```

（b）Viewイベントのハンドリング

次は、HIObjectRegisterSubclass()でレジストしたCarbon Event Handlerルーチンの myCustomViewHandler()の処理内容を見てみます。このHandlerルーチンは、全部で7つのCarbon Eventの種類に対応していますが、そのうち最初の3つがインスタンスデータの確保、初期化、削除を行うコンストラクタ＆デストラクタ処理を、残りの4つがカスタムViewのメソッド処理を実行します。

```
OSStatus myCustomViewHandler( EventHandlerCallRef  inCallRef,
                                        EventRef  inEvent, void*  inUserData )
{
    OSStatus                    err = eventNotHandledErr;
    UInt32                      eventClass = GetEventClass( inEvent );
    UInt32                      eventKind = GetEventKind( inEvent );
    myCustomViewData*       c = (myCustomViewData*) inUserData;

    switch ( eventClass )
    {
        case kEventClassHIObject:  // コンストラクタ用のCarbon Event
        {
            switch ( eventKind )
            {
                case kEventHIObjectConstruct:  // インスタンスデータの確保
                    err = myCustomViewConstruct( inEvent );
                    break;

                case kEventHIObjectInitialize:  // インスタンスデータの初期化
                    err = myCustomViewInitialize( inCallRef, inEvent,  data );
                     break;
```

```
            case kEventHIObjectDestruct:  // インスタンスデータの削除
                err = myCustomViewDestruct( inEvent, data );
                break;
        }
    }
    break;
    case kEventClassControl:  // メソッド用のCarbon Event
    {                                   // Viewはカスタムコントロールとして定義
        switch ( eventKind )
        {
            case kEventControlInitialize:  // コントロールの初期化
                err = noErr;
                break;

            case kEventControlDraw:  // コントロールの描画
                err = myCustomViewDraw( inEvent, data );
                break;

            case kEventControlHitTest:   // コントロールのヒットテスト
                err = myCustomViewHitTest( inEvent, data );
                break;

            case kEventControlGetPartRegion:  // コントロールの部分領域を返す
                err = myCustomViewGetRegion( inEvent, data );
                break;
        }
    }
    break;
  }
  return err;
}
```

以下が、今回のカスタムコントロールで利用するインスタンスデータ用の構造体です。メンバーにひとつのControlRefを確保する簡単なものです。

```
typedef struct
{
  ControlRef control;
```

```
} myCustomViewData;
```

続いてコンストラクタ＆デストラクタ用の3つのルーチンを紹介します。最初は、上記インスタンスデータを確保するためのmyCustomViewConstruct()ルーチンです。

```
OSStatus myCustomViewConstruct (EventRef inEvent)
{
    OSStatus                    err;
    myCustomViewData*       data;

    data = malloc( sizeof( myCustomViewData ) );  // インスタンスデータ用のメモリ
                                                  // 領域の確保
    require_action( data != NULL, CantMalloc, err = memFullErr );

    // 対象ViewのControlRefをインスタンスデータ（構造体メンバ）にセットする
    err = GetEventParameter( inEvent, kEventParamHIObjectInstance,
                        typeHIObjectRef, NULL, sizeof( HIObjectRef ),
                        NULL, (HIObjectRef*) &data->control );

    require_noerr( err, ParameterMissing );

    // userDataにインスタンスデータをセットする。これにより、Carbon Event
    //  Handlerによるメソッドの実行時にインスタンスデータを参照できる。
    err = SetEventParameter( inEvent, kEventParamHIObjectInstance,
                        typeVoidPtr, sizeof( myCustomViewData* ),
                        &data );
ParameterMissing:
    if ( err != noErr )
        free( data );    // エラー発生時は確保したメモリ領域を開放する
CantMalloc:
    return err;
}
```

次は、インスタンスデータを初期化するためのmyCustomViewConstruct()ルーチンです。

```
OSStatus myCustomViewInitialize (EventHandlerCallRefinCallRef,
                                 EventRef inEvent, myCustomViewData*  inData )
{
```

```
    OSStatus            err;
    Rect                bounds;

    // Superclassへ初期化の機会を与えるためにCallNextEventHandler()を実行する
    err = CallNextEventHandler( inCallRef, inEvent );

    require_noerr( err, TroubleInSuperClass );

    // HIObjectCreate() で渡されたViewの表示矩形領域を得る
    err = GetEventParameter( inEvent, 'Boun', typeQDRectangle,
            NULL, sizeof( Rect ), NULL, &bounds );

    require_noerr( err, ParameterMissing );

    // 得られた矩形領域をコントロール表示領域とする。
    SetControlBounds( inData->control, &bounds );

ParameterMissing:
TroubleInSuperClass:
    return err;
}
```

最後は、不必要になったインスタンスデータ用のメモリ領域を解放するための（デストラクタ）myCustomViewDestruct()ルーチンです。

```
OSStatus myCustomViewDestruct (EventRef inEvent,
                                        myCustomViewData* inData)
{
    free (inData);    // インスタンスデータ用のメモリ領域を解放
    return noErr;
}
```

このCarbon Event Handlerルーチンでは、メソッド用として4種類のCarbon Eventに対応しています。どんなメソッドを実装するのかは、作成した自作コントロールがどのような機能に対応しているかに依存します。今回は、kEventControlInitialize（コントロール初期化）については何も処理をしません。また、kEventControlGetPartRegionの対処については、自作コントロールの内容によって大きく変化します。

kEventControlDraw（コントロールの描画）では、kEventParamCGContextRefパラ

メータでCGContextRefを得てから、Core Graphics APIを用いてコントロールの描画を行います。この時のコントロール描画領域（Region）は、kEventParamRgnHandleパラメータで得ることが出来ます。もし、Core Graphicsの代わりにQuickDraw APIを用いたいのなら、kEventParamGrafPortパラメータを用いてCGrafPtrを得る必要があります。以下は簡単なサンプルです。

```
short               err＝eventNotHandledErr;
HIViewRef           view;
CGContextRef   ctx;
HIRect              vrt;

view=data->control; // HIViewRefはインスタンスデータから得る
err=GetEventParameter(inEvent,kEventParamCGContextRef,
                      typeCGContextRef, NULL,sizeof(CGContextRef),NULL,&ctx );
                                    // ViewのCGContextRefを得る
HIViewGetBounds( view,&vrt ); // Viewの矩形枠を得る
if( IsControlHilited( view ) )       // ハイライト表示かどうか？
{
    CGContextSetRGBFillColor( ctx,0.1,0.1,1.0,0.3 ); // ペイント用カラー設定
    CGContextFillRect( ctx,vrt ); // 矩形領域をペイントする
}
CGContextStrokeRect( ctx,vrt ); // 黒色で矩形枠の縁取りする
```

kEventControlHitTest（ヒットテスト）では、kEventParamMouseLocationでマウスクリックされた位置を得て、その座標が自作コントロールのどの部分なのかを示す派＝ー都コードを（例えばkControlButtonPart）kEventParamControlPartパラメータとして、SetEventParameter()を用い設定します。以下は簡単なサンプルです。

```
ControlPartCode part=kControlButtonPart;
short               err＝eventNotHandledErr;
HIViewRef           view;
HIRect              vrt;
HIPoint             pt;

view=data->control;
GetEventParameter( inEvent,kEventParamMouseLocation,typeHIPoint,NULL,
                                                    sizeof(pt),NULL,&pt );
                                    // クリックされた座標を得る
HIViewGetBounds( view,&vrt );       // Viewの矩形枠を得る
```

```
    if( CGRectContainsPoint( vrt,pt ) ) // クリックは矩形枠の中か？
    {
        SetEventParameter(inEvent,kEventParamControlPart,
                                        typeControlPartCode, sizeof(part),&part );
                                        // 適切なパートコードを設定する
        err=noErr;
    }
```

（c）Viewインスタンスの作成

実際に、カスタムコントロール（カスタムView）のインスタンスを作成するには、サブクラスをレジストした後（この作業は一度だけで）HIObjectCreate()を実行します。この時、表示矩形枠など初期化で必要となるパラメータは、kEventHIObjectInitializeイベントを作成し、そのパラメータとしてセットした後に、HIObjectCreate()に引数として渡します。

```
OSStatus CreateMyCustomView ( WindowRef inWindow, const Rect*inBounds,
                                                    ControlRef* outControl )
{
    OSStatus            err;
    ControlRef          root;
    EventRef            event;

    // 先んじてサブクラスをレジストする（処理は一回だけ実行...ソースコード参照）
    err = myCustomViewRegister();
    require_noerr( err, CantRegister );

    // 初期化用データを渡すためのkEventHIObjectInitializeイベントを作成
    err = CreateEvent( NULL, kEventClassHIObject, kEventHIObjectInitialize,
                                            GetCurrentEventTime(), 0, &event );
    require_noerr( err, CantCreateEvent );

    // コントロールの表示矩形枠をイベントのパラメータとしてセットする。
    // これによりmyCustomViewInitialize()内で表示矩形枠を利用可能となる
    if ( inBounds != NULL )
    {
        err = SetEventParameter( event, 'Boun', typeQDRectangle,
                                                sizeof( Rect ), inBounds );
```

```
        require_noerr( err, CantSetParameter );
    }
    // View（カスタムコントロール）インスタンスを作成しControlRefを得る
    err = HIObjectCreate( kMyCustomViewClassID, event, (HIObjectRef*)
                                                        outControl );
    require_noerr( err, CantCreate );

    // もし親ウィンドウが存在していればHIViewAddSubview()でそこに配置する
    if ( inWindow != NULL )
    {
        err = GetRootControl( inWindow, &root );
        require_noerr( err, CantGetRootControl );
        err = HIViewAddSubview( root, *outControl );
    }
CantCreate:
CantGetRootControl:
CantSetParameter:
CantCreateEvent:
    ReleaseEvent( event ); // 作成したEventを削除する
CantRegister:
    return err;
}
```

Interface Builderを使い、ウィンドウにClass IDを定義したHIViewオブジェクトを配置しておけば、nibファイルからウィンドウを作成した時にHIObjectCreate()を呼び出したのと同じ効果を得られます。ただし、nibファイルを利用する前に、そのサブクラスのレジストだけは実行しておく必要がありますので注意してください。この時のコントロール初期化用のパラメータ（現在地や最大値など）もInterface Builderのインスペクターダイアログの「Attribute」タグで先設定しておくことが出来ます。

（d）カスタムメニューの作成

Mac OS X 10.3以降であれば、MDEFを使う代わりに、HIViewのサブクラスとしてカスタムメニューを作成することが出来ます。カスタムメニューは、一般的にHIMenuViewかHIStandardMenuViewクラスのサブクラスViewとして定義します。

HIStandardMenuViewクラスは一般的に表示されているAquaベースのメニューです。HIMenuViewのサブクラスとして定義する場合には、HIViewChangeFeatures()を用いて、定義されたViewのkHIViewDoesNotUseSpecialPartsとkHIViewAllowsSubviews

ビットをセットします。そしてCarbon Event Handlerをインストールし、カスタムメニューの機能に準じて各種イベントを処理します。

カスタムメニューの作成にはCreateCustomMenu()を用います。このAPIはメニューを表示する時に HIObjectCreate()を呼び出します。CreateCustomMenu()に引数として渡すMenuDefSpec構造体は、以下のように定義されています。MDEFベースのカスタムメニューを実装する時とは異なり、defProcを代入する代わりにclassIDとinitEventを代入します。

```
enum {kMenuDefClassID = 1};  // Viewベースではこのタイプを指定する
struct MenuDefSpec
{
    MenuDefType defType;
    union
    {
        MenuDefUPP defProc;  // MDEFベースの場合にセット
        struct
        {
            CFStringRef classID;  // Viewベースの場合にセット
            EventRef initEvent;    // 初期値パラメータなどを受け渡す
            }view;
        }u;
    }

#define kmyCustomMenuViewClassID
                        CFSTR("com.apple.sample.kMyCustomMenuClassID”);
...
    MenuDefSpec       defSpec;
    MenuRef             theMenu;
    HIObjecClassRef  theClassIDRef;
    OSStatus             err;

    // カスタムメニューのサブクラスとCarbon Event Handlerをレジストする
    HIObjectRegisterSubclass( kMyCustomMenuViewClassID,
                        kHIMenuViewClassID,kNilOptions, MyMenuHandler,
                        GetEventTypeCount( kMyMenuEvents ), kMyMenuEvents,
                        NULL, &theClassIDRef );

     // MenuDefSpecをkMenuDefClassIDタイプに指定して
```

```
    // kMyCustomMenuViewClassIDを代入する。
    defSpec.defType = kMenuDefClassID;
    defSpec.u.view.classID = kMyCustomMenuViewClassID;
    defSpec.u.view.initEvent = NULL;

    // カスタムメニューの作成してタイトルを表示する
    err = CreateCustomMenu( &defSpec, 0, 0, &theMenu );
    if ( err == noErr )
        SetMenuTitleWithCFString( theMenu, CFSTR("Button") );
```

以下は、カスタムメニューのインスタンスデータ用のMyMenuData構造体と、Carbon Event Handlerで受け取るべきイベント種類を指示したEventTypeSpec構造体です。

```
typedef struct
{
    HIViewRef           view;
    HIViewRef           button;
}
MyMenuData;

static const EventTypeSpec  kMyMenuEvents[] =
{
    { kEventClassHIObject, kEventHIObjectConstruct },  // コンストラクタ
    { kEventClassHIObject, kEventHIObjectDestruct },   // ディストラクタ
    { kEventClassControl, kEventControlInitialize },         // 初期化
    { kEventClassControl, kEventControlBoundsChanged },  // サイズ変更
    { kEventClassControl, kEventControlGetOptimalBounds }, // サイズ決定
    { kEventClassScrollable, kEventScrollableGetInfo }  // 各種サイズを返す
};
```

最低限実装すべきイベント種類としては、kEventControlGetOptimalBounds、kEventScrollableGetInfo、kEventControlDraw、kEventControlHitTest 、kEventControlGetPartRegionなどがあります。ただし後者の3つは、Menu Content Viewに埋め込んだ個々のView側で処理します。以下のサンプルでは、Menu Content Viewにプッシュボタンコントロールをひとつ埋め込んで表示しています。

また、オプションにより実装するイベントとしては、kEventMenuCreateFrameView、kEventMenuGetFrameBounds、kEventMenuBecomeScrollable、kEventMenuCeaseToBeScrollable、kEventScrollableScrollTo、

kEventControlSetFocusPartなどがあります。

```
static pascal OSStatus MyMenuHandler( EventHandlerCallRef inCaller,
                                      EventRef inEvent, void* inRefcon )
{
    OSStatus        err = eventNotHandledErr;
    MyMenuData* data = (MyMenuData*) inRefcon;

    switch ( GetEventClass( inEvent ) )
    {
        case kEventClassHIObject:  // HIObjectコンストラクタ＆デストラクタ

            switch ( GetEventKind( inEvent ) )
            {
                case kEventHIObjectConstruct:  // コンストラクタ
                {
                    // インスタンスデータのメモリ領域を確保
                    data = (MyMenuData*) calloc( 1, sizeof( MyMenuData ) );
                    require_action( data != NULL, CouldntAllocData,
                                    err = memFullErr );

                    // Menu Content ViewのHIViewRefを得る
                    GetEventParameter(inEvent,kEventParamHIObjectInstance,
                     typeHIObjectRef, NULL, sizeof( data->view ), NULL, &data->view );
                    // 次回からの参照のためuserDataにインスタンスデータをセット
                    SetEventParameter(inEvent,kEventParamHIObjectInstance,
                                                  typeVoidPtr, sizeof( data ), &data );
                    break;
                }
                case kEventHIObjectDestruct:  // ディストラクタ

                    free( (void*) data );  // インスタンスデータのメモリ領域を開放
                    err = noErr;
                    break;
            }
            break;

        case kEventClassControl:  // メニュー表示用のメソッド
```

```
switch ( GetEventKind( inEvent ) )
{
    case kEventControlInitialize:  // メニューの初期化
    {
        Rect        bounds = { 0, 0, 0, 0 };

       // プッシュボタンコントロールを作成しMenu Content Viewに埋め込む
       // プッシュボタンのHIViewRefをインスタンスデータにセットする
        err = CreatePushButtonControl( NULL, &bounds,  CFSTR("Beep!")
                                                              , &data->button );
        require_noerr( err, CouldntCreateButton );
        HIViewAddSubview( data->view, data->button);
        break;
    }
    case kEventControlGetOptimalBounds:  // メニューサイズの決定
    {
        HIRect bounds = { { 0, 0 }, { 90, 100} };

       // HIRectを用いメニューサイズを送る
        SetEventParameter( inEvent, kEventParamControlOptimalBounds,
                                        typeHIRect, sizeof( bounds ), &bounds );
        err = noErr;
        break;
    }
    case kEventControlBoundsChanged:  // プッシュボタンのサイズ決定
    {
        HIRect bounds;

       // プッシュボタンのHIViewRefをインスタンスデータより得る
        HIViewGetBounds( data->view, &bounds );
        HIRect frame;
        frame.origin.x = 10;
        frame.origin.y = bounds.size.height/2 - 10;
        frame.size.height = 20;
        frame.size.width = bounds.size.width -20;
       // プッシュボタンの表示矩形枠をセット
        HIViewSetFrame( data->button, &frame );
        err = noErr;
        break;
```

```
            }
        }
        break;

    case kEventClassScrollable:  // メニュー表示領域のスクロール

        switch ( GetEventKind( inEvent ) )
        {
            case kEventScrollableGetInfo:  // スクロール関連の情報をセットする
            {
                HIRect bounds;
                HIPoint origin = { 0, 0 };

                // メニュー表示領域をすべてスクロールの対象とする
                HIViewGetBounds( data->view, &bounds );
                SetEventParameter( inEvent, kEventParamImageSize, typeHISize,
                                            sizeof( bounds.size ), &bounds.size );
                SetEventParameter( inEvent, kEventParamViewSize, typeHISize,
                                            sizeof( bounds.size ), &bounds.size );
                bounds.size.height = 20;    // 任意
                SetEventParameter( inEvent, kEventParamLineSize, typeHISize,
                                           sizeof( bounds.size ), &bounds.size );
                SetEventParameter( inEvent, kEventParamOrigin, typeHIPoint,
                                                    sizeof( origin ), &origin );
                err = noErr;
                break;
            }
        }
        break;
    }
CouldntCreateButton:
CouldntAllocData:
    return err;
}
```

本ドキュメントの履歴

オリジナル2005年8月11日　要約2005年10月15日 v1.00