.NET でドラッグ&ドロップ

2011-04-17

山本 卓 <sgryjp@gmail.com>

http://sgry.jp/pgarticles/dragdrop.net.html

1. はじめに

拙作のテキストエディタエンジン Azuki で、異なるウィンドウ間でドラッグ＆ドロップ（以後 D&D）を実行可能にするために複数の GUI フレームワークでの動作を調べてみました。調査したフレームワークは、いわゆる WinForms フレームワーク、WPF、GTK#、そして Cocoa です。やれるかどうかはさておき、もし WinForms に依存しすぎた D&D 処理を実装してしまうと WinForms 以外に対応する際に大きな障害となってしまいます。そこで、もし移植すると決まっても Azuki を使った既存のアプリケーションに大きな変更を要求しないで済むように、各フレームワークにおけるイベントの流れやコンセプトを参考としてまとめてみました。なお D&D 処理の良い外部仕様を考える資料を作るという意味に加えて、D&D という概念自体の理解を深める勉強の意味もあります。

説明の中では具体的な例を使う方が分かりやすいので、図形描画アプリケーション「MyDraw」という仮想のアプリケーションを例に取ります。このアプリでは選択した図形を他のウィンドウへ D&D できるものとし、また Shift キーを押しながら D&D した場合は図形の「移動」、Ctrl キーを押しながら D&D した場合は図形の「コピー」と扱われる操作仕様になっているとします。

なお、本記事中の「ウィンドウ」という用語は、Cocoa では「View」、Gtk では「Widget」を指していると解釈してください。

2. 共通概念

2.1. D&D とは

D&D は一種のデータ通信です。プログラム的には、マウスカーソルのドラッグという操作によってデータを送信側から受信側へと転送します。UI 操作と密接に関連するデータ通信方式であるため、送信側と受信側が頻繁に情報をやりとりすることになり、単純なデータ通信とは一風変わった手続きが必要になります。

本記事で取り上げるすべてのフレームワークは D&D 処理のために専用の機構を備えており、この記事でもそれを利用します。この機構を使う場合、まず送信側が D&D を開始したいタイミングでフレームワークに D&D 処理の開始を命令します。するとユーザがマウスを操作しても通常のマウスイベントが発生しなくなり、そのかわりに D&D 処理中にのみ発生する専用イベントが発生するようになります。そしてドロップやキャンセル等によって D&D 処理が終了すると、通常のマウスイベントが発生する元通りの状態に戻ります。この記事では便宜上、フレームワークに対して D&D 処理の開始を命令してから処理が完結するまでの間を「D&D セッション」と呼びます。

> なお専用の機構を使わずに D&D を実装するならば、単純に「マウスボタンが押し下げられた」、「マウスカーソルが移動した」、「マウスボタンの押し下げが解除された」という 3 つのイベントを受信して適切にフラグ管理等を行うことを思い付くかと思います。しかし私が調べた限りでは、このアプローチは送信側ウィンドウと受信側ウィンドウが違うと実装が非常に難しくなると思われます。ましてや他人が作った別プログラムのウ

ィンドウと D&D で通信するとなれば絶望的です。したがって異なるウィンドウ間で通信する可能性が少しでもあるならば、最初から D&D の機構を使うことを検討するべきでしょう。なお、たとえば作画アプリケーションで「ドラッグした軌跡の通りに線を描画する機能」を実装する場合など、本質的にデータ通信ではない操作を実装する場合に D&D 用の機構を使うと逆に不便です（この操作を誰が D&D と呼ぶのかという問題もありますが)。本記事では、あくまでデータ通信としての D&D を扱います。

ありがたいことに、本記事で取り上げる 4 つのフレームワークが用意する D&D の機構は同じ考え方で利用できます。以下に送信側と受信側それぞれの典型的な D&D 処理の流れを記します。

2.2. 送信側における D&D 処理の流れ

送信側の典型的な D&D 処理の流れを次に記します。

A) マウスボタンの押し下げを検出した時、その座標を記録
B) マウスボタンの移動時、マウスボタンが押し下げられており、かつ現在のマウスカーソル座標とマウスボタンが押し下げられた座標との距離が一定値を超える場合、D&D セッションを開始
C) D&D が終了したら、受信側が実施した処理内容に応じて送信後の後処理を実行

送信側で重要なポイントは 3 点あります。

(1) D&D セッションは送信側が明示的に開始する

1 つ目のポイントは、送信側から D&D の処理を明示的に開始する必要がある点です。当然のように聞こえますが「プロパティを設定すれば D&D はお手軽かつ自動的に実現できる」というわけではない、という意味で重要なポイントです。

また、D&D セッションを開始するときには「実施可能なアクション」を指定する点も重要です。一般的に D&D で実行できる操作にはコピー、移動、リンクなどがあります。D&D セッションを開始するに当たって送信側は、D&D で転送するデータに対してどの操作を行えるかを受信側にかならず伝えます。もしこれを指定しなければ、たとえば送信側で削除できないデータを受信側が「移動」できてしまうため、話がおかしくなります。

(2) 基本的に D&D セッション中は送信側で何もしない

2 つ目のポイントは、D&D を開始してから D&D セッションが終了するまでの間、送信側では基本的に何もしないで良いという点です。ただし何もしないで良いのは基本的な D&D 処理に限った話であり、高度な処理を実現したい場合は D&D セッション中に発生するフィードバック用のイベントを受け取って送信側の処理を実行する必要があります。D&D セッション中に発生するフィードバック用のイベントは、残念ながらフレームワークによって考え方のレベルで違っています。そのためフレームワークに依存しない一般的な手法を導き出すのは難しいかもしれません。

高度な D&D 処理の例：
Shift キーを押しながら D&D すると「移動」と扱う描画アプリの場合で、図形の D&D 中で Shift キーが押されている間は「移動」操作になることを強調するためにドラッグ元の画面上にある該当図形を半透明で表示する

(3) D&D セッションの実行結果に応じて後処理を実行する

3 つ目のポイントは、送信側は D&D セッション終了時に受信側から報告される実行結果に応じて後処理を実装する必要があることです。たとえば実行結果が「正しく**移動**した」であった場合、送信元では D&D で受信側に送信したデータを削除します（さもなければ両方にデータが存在する、つまり「コピーした」状態になります)。

WinForms と WPF では、D&D セッションを開始するメソッドを呼び出すと D&D セッションが終了するまで処理が戻って来ず（ブロッキング実行)、メソッドの戻り値が D&D の結果を表しています。そのため、D&D の結果に対する送信側の後処理はセッション開始命令の直後に記す形になります。一方、GTK#と Cocoa では D&D セッションの開始を開始するメソッド・メッセージ呼び出しの戻り値から D&D の結果を知ることができません（そもそも GTK#の場合は非ブロッキング実行)。その代わりに、D&D セッション終了時に専用のイベントが発生しますので、そのイベントハンドラで D&D の結果に対する送信側後処理を実装します。

2.3. 受信側における D&D 処理の流れ

受信側の典型的な D&D 処理の流れを次に記します。

A) あらかじめ、受信側の UI 要素がドロップを受け付けられるように設定しておく
B) ドラッグ中のマウスカーソルが領域内に入ってきたら、ドラッグされているデータの種類やマウスカーソルの位置に応じてドロップ可能かどうかを送信側に応答
C) マウスカーソルが領域を移動中にも、同様にドロップ可否を送信側に応答（位置によってドロップ可能かどうか変化することもあるため）
D) ドロップされたら送信元からデータを受信して受信側の処理を実行し、「何を行ったのか」を送信側へと返答

重要なポイントは 3 点です。

(1)　あらかじめ「D&D を受けられる」と宣言しておく

1 つ目のポイントは、あらかじめ受信側は「D&D を受けられる」という宣言を行っておく必要がある点です。あるウィンドウが「D&D を受けられる」と宣言されていなければ、そのウィンドウの上にいくら D&D 中のカーソルを持って行っても何も起こりません。

なおこの宣言はフレームワークによっては不要です。WinForms、WPF では該当ウィンドウの AllowDrop プロパティで宣言し、GTK#では Gtk.Drag.DestSet メソッドで宣言します。Cocoa では特に何もせずとも D&D を受け付けることができます。

(2)　D&D 中のカーソル位置に応じてイベントを処理

2 つ目のポイントです。D&D を受けられるウィンドウに対しては、D&D 中のカーソルが移動するごとにフレームワークからイベントが送信されます。典型的なイベントは、D&D 中のカーソルが該当ウィンドウの領域内に入ってきたときのイベントや、ウィンドウ領域内で移動しているときのイベントです。このようなイベントが発生するたびに受信側は、カーソルの位置や D&D に使われているマウスボタンの種類、さらには Shift キー等の押し下げ状態などを確認した上で「その時点でドロップできるかどうか」および「その時点でドロップしたらどのような処理が行われるか」をフレームワークに返答します。2 つ起動した MyDraw の間で図形を移動する場合、カーソルが受信側 MyDraw のツールバーなどキャンバス以外の場所にあれば「ドロップ不可」と返答し、キャンバス上にあって Shift キーが押し下げられているならば「ドロップ可能」かつ「ドロップしたら移動になる」と返答します。

(3)　ドロップされたら処理を実行し、結果を返す

3 つ目のポイントです。どのフレームワークでも最終的にユーザが「ドロップ」すると、それを伝える専用のイベントが受信側で発生します。受信側はそこで、どのような処理を実行したかを選んで返答します。仮に MyDraw で Shift を押しながら図形をドロップした場合を考えましょう。この場合は製品の仕様から「図形の移動」を実行することになりますので、受信側はフレームワークごとに用意されているメソッド等を使って、D&D で転送された図形のデータをプログラム的に取得してキャンバスに配置します。そしてフレームワークに対して「移動を実行した」とドロ

ップ結果を返答します。受信側の処理は以上で終了となります。

ここで、「受信側では送信側の図形データを削除していない」ことに注意してください。フレームワークは受信側からドロップ結果の返答を受けると D&D セッションの終了処理に入り、その中で送信側へ「受信側で移動が実行された」という情報を伝えます。その結果、送信側は D&D セッションの後処理として元図形の削除を実行しますので[2.2.(3)参照]、D&D セッションが完了すると図形が移動した状態になる、という仕組みになっています。蛇足ですが、移動ではなくコピーを実行した場合は送信元に「受信側でコピーが実行された」という情報が伝わるため、送信元の図形が削除されません。その結果、図形がコピーされた状態になります。

3. 各フレームワークでの実装サンプル

3.1. WinForms

まず送信側についてです。WinForms フレームワークでは、D&D セッションを Control.DoDragDrop で開始します。このメソッドは静的メソッドではなく、また試した限りどの Control オブジェクトのメソッドを呼び出しても違いが無いようです。したがって、どんな場面であっても D&D セッションを開始するには「this.DoDragDrop」と記せば良いと思われます（もちろん Control 派生クラスのメソッド中で開始する場合）。DoDragDrop メソッドの引数には D&D で送信するデータ、そのデータに対してどのような操作を受信側に許可するかを指定します。なおこのメソッドは D&D セッションが完了するまで終了しません（ブロック実行）。そのため、受信側が行った操作を受けて実行するべき送信側の処理は、DoDragDrop メソッドの呼び出しに続けて記す形になります。

続いて受信側についてです。まず最初に Control.AllowDrop プロパティが true に設定されていないコントロールでは D&D 関連のイベントが一切発生しません。そのため D&D を受けたい UI 要素では同プロパティをまず true に設定しておきます。あとは DragEnter、DragLeave、DragOver、DragDrop の各イベントに対する処理を普通に実装すれば問題ありません。

サンプルとして、テキストデータとしてドラッグ可能なラベル「_DragMeLabel」と、テキストデータをドロップ可能なラベル「_DropToMeLabel」が配置されたウィンドウを持つウィンドウの実装コードを以下に記します。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace DragDropWinForms
{
   public partial class Form1 : Form
   {
       Point _LastMouseDownPos;

       public Form1()
       {
           InitializeComponent();

           _DragMeLabel.MouseDown += _DragMeLabel_MouseDown;
           _DragMeLabel.MouseMove += _DragMeLabel_MouseMove;
           _DragMeLabel.GiveFeedback += _DragMeLabel_GiveFeedback;
           _DragMeLabel.QueryContinueDrag += _DragMeLabel_QueryContinueDrag;
```

```
        _DropToMeLabel.MouseMove += _DropToMeLabel_MouseMove;
        _DropToMeLabel.DragEnter += _DropToMeLabel_DragEnter;
        _DropToMeLabel.DragLeave += _DropToMeLabel_DragLeave;
        _DropToMeLabel.DragOver += _DropToMeLabel_DragOver;
        _DropToMeLabel.DragDrop += _DropToMeLabel_DragDrop;
        _DropToMeLabel.AllowDrop = true;
    }

    #region Dragging source
    void _DragMeLabel_MouseDown( object sender, MouseEventArgs e )
    {
        Console.WriteLine( "(src)MouseDown( {0} )", e.Location );
        _LastMouseDownPos = e.Location;
    }

    void _DragMeLabel_MouseMove( object sender, MouseEventArgs e )
    {
        Console.WriteLine( "(src)MouseMove( {0} )", e.Location );

        // 左クリックで一定距離ドラッグされた場合は D&D を実行
        if( e.Button == MouseButtons.Left )
        {
            if( 3 < Math.Abs(e.Location.X - _LastMouseDownPos.X)
                && 3 < Math.Abs(e.Location.Y - _LastMouseDownPos.Y) )
            {
                // D&D でドロップ先に送信するデータを準備
                IDataObject data = new DataObject( DataFormats.Text, ":)" );

                // D&D セッションを開始。
                // ドロップ先に
                // 「_DragMeLabel から Copy または Move が可能なデータを送る」
                // と通知する。
                DragDropEffects allowedEffects = DragDropEffects.Copy | DragDropEffects.Move;
                DragDropEffects result = this.DoDragDrop( data, allowedEffects );

                // D&D の結果を判定
                if( result == DragDropEffects.None )
                    MessageBox.Show( "D&D failed." );
                else if( result == DragDropEffects.Move )
                    _DragMeLabel.Text = "Moved.";
                else
                    _DragMeLabel.Text = "Copied.";
            }
        }
    }

    void _DragMeLabel_GiveFeedback( object sender, GiveFeedbackEventArgs e )
    {
        //Console.WriteLine( "(src)GiveFeedback" );
    }

    void _DragMeLabel_QueryContinueDrag( object sender, QueryContinueDragEventArgs e )
    {
        //Console.WriteLine( "(src)QueryContinueDrag" );
    }
    #endregion

    #region Dragging destination
    void _DropToMeLabel_MouseMove( object sender, MouseEventArgs e )
```

```
        {
            Console.WriteLine( "(dest)MouseMove( {0} )", e.Location );
        }

        void _DropToMeLabel_DragEnter( object sender, DragEventArgs e )
        {
            e.Effect = DecideDropAction( e );
            Console.WriteLine( "(dest)DragEnter --> {0}", e.Effect );
        }

        void _DropToMeLabel_DragLeave( object sender, EventArgs e )
        {
            Console.WriteLine( "(dest)DragLeave" );
        }

        void _DropToMeLabel_DragOver( object sender, DragEventArgs e )
        {
            e.Effect = DecideDropAction( e );
            Console.WriteLine( "(dest)DragOver --> {0}", e.Effect );
        }

        void _DropToMeLabel_DragDrop( object sender, DragEventArgs e )
        {
            e.Effect = DecideDropAction( e );
            Console.WriteLine( "(dest)DragDrop --> {0}", e.Effect );

            switch( e.Effect )
            {
                case DragDropEffects.Copy:
                    _DropToMeLabel.Text = (string)e.Data.GetData( DataFormats.Text );
                    break;

                case DragDropEffects.Move:
                    _DropToMeLabel.Text = (string)e.Data.GetData( DataFormats.Text );
                    break;

                default:
                    break;
            }
        }

        DragDropEffects DecideDropAction( DragEventArgs e )
        {
            Point pos = new Point( e.X, e.Y );
            pos = _DropToMeLabel.PointToClient( pos );

            // 単純に _DropToMeLabel の左半分ならば Copy、右半分ならば Move とする
            if( pos.X < (_DropToMeLabel.Width / 2) )
            {
                return DragDropEffects.Copy;
            }
            else
            {
                return DragDropEffects.Move;
            }
        }
        #endregion
    }
}
```

3.2. WPF

WPF の D&D 関連処理は、ほとんど WinForms の場合と同じです。

まず送信側についてです。WPF では D&D セッションを DragDrop.DoDragDrop で開始します。このメソッドには送信元（ドラッグされたデータを所有している UI 要素）、D&D で送信するデータ、そのデータに対してどのような操作を受信側に許可するかを指定します。なおこのメソッドは D&D セッションが完了するまで終了しません（ブロック実行）。そのため、受信側が行った操作を受けて実行するべき送信側の処理は、DoDragDrop メソッドの呼び出しに続けて記す形になります。

続いて受信側についてです。まず最初に UIElement.AllowDrop プロパティが true に設定されていない UI 要素では D&D 関連のイベントが一切発生しません。そのため D&D を受けたい UI 要素では同プロパティをまず true に設定しておきます。あとは DragEnter、DragLeave、DragOver、Drop の各イベントに対する処理を普通に実装すれば問題ありません。

サンプルとして、テキストデータとしてドラッグ可能なラベル「_DragMeLabel」と、テキストデータをドロップ可能なラベル「_DropToMeLabel」が配置されたウィンドウを持つウィンドウの実装コードを以下に記します。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace DragDropWpf
{
   public partial class Window1 : Window
   {
       Point _LastMouseDownPos;

       public Window1()
       {
           InitializeComponent();

           _DragMeLabel.MouseDown += _DragMeLabel_MouseDown;
           _DragMeLabel.MouseMove += _DragMeLabel_MouseMove;
           _DragMeLabel.GiveFeedback += _DragMeLabel_GiveFeedback;
           _DragMeLabel.QueryContinueDrag += _DragMeLabel_QueryContinueDrag;

           _DropToMeLabel.MouseMove += _DropToMeLabel_MouseMove;
           _DropToMeLabel.DragEnter += _DropToMeLabel_DragEnter;
           _DropToMeLabel.DragLeave += _DropToMeLabel_DragLeave;
           _DropToMeLabel.DragOver += _DropToMeLabel_DragOver;
           _DropToMeLabel.Drop += _DropToMeLabel_Drop;
           _DropToMeLabel.AllowDrop = true;
       }

       #region Dragging source
```

```
void _DragMeLabel_MouseDown( object sender, MouseButtonEventArgs e )
{
    Console.WriteLine( "(src)MouseDown( {0} )", e.GetPosition(this) );
    _LastMouseDownPos = e.GetPosition( this );
}

void _DragMeLabel_MouseMove( object sender, MouseEventArgs e )
{
    Console.WriteLine( "(src)MouseMove( {0} )", e.GetPosition(this) );

    // 左クリックで一定距離ドラッグされた場合は D&D を実行
    if( e.LeftButton == MouseButtonState.Pressed )
    {
        Point pos = e.GetPosition( this );
        if( 3 < Math.Abs(pos.X - _LastMouseDownPos.X)
            || 3 < Math.Abs(pos.Y - _LastMouseDownPos.Y) )
        {
            // D&D でドロップ先に送信するデータを準備
            IDataObject data = new DataObject( DataFormats.Text, ":)" );

            // D&D セッションを開始。
            // ドロップ先に
            // 「_DragMeLabel から Copy または Move が可能なデータを送る」
            // と通知する。
            DragDropEffects allowedEffects = DragDropEffects.Copy | DragDropEffects.Move;
            DragDropEffects result = DragDrop.DoDragDrop( _DragMeLabel,
                                                          data,
                                                          allowedEffects );

            // D&D の結果を判定
            if( result == DragDropEffects.None )
                MessageBox.Show( "D&D failed." );
            else if( result == DragDropEffects.Move )
                _DragMeLabel.Content = "Moved.";
            else
                _DragMeLabel.Content = "Copied.";
        }
    }
}

void _DragMeLabel_GiveFeedback( object sender, GiveFeedbackEventArgs e )
{
    //Console.WriteLine( "(src)GiveFeedback" );
}

void _DragMeLabel_QueryContinueDrag( object sender, QueryContinueDragEventArgs e )
{
    //Console.WriteLine( "(src)QueryContinueDrag" );
}
#endregion

#region Dragging destination
void _DropToMeLabel_MouseMove( object sender, MouseEventArgs e )
{
    Console.WriteLine( "(dest)MouseMove" );
}

void _DropToMeLabel_DragEnter( object sender, DragEventArgs e )
{
    e.Effects = DecideDragAction( e );
```

```
            Console.WriteLine( "(dest)DragEnter --> {0}", e.Effects );
        }

        void _DropToMeLabel_DragLeave( object sender, DragEventArgs e )
        {
            Console.WriteLine( "(dest)DragLeave" );
        }

        void _DropToMeLabel_DragOver( object sender, DragEventArgs e )
        {
            e.Effects = DecideDragAction( e );
            Console.WriteLine( "(dest)DragOver --> {0}", e.Effects );
            e.Handled = true; // これが無いと標準のイベントハンドラがマウスカーソルを戻してしまう
        }

        void _DropToMeLabel_Drop( object sender, DragEventArgs e )
        {
            e.Effects = DecideDragAction( e );
            Console.WriteLine( "(dest)Drop --> {0}", e.Effects );

            switch( e.Effects )
            {
                case DragDropEffects.Copy:
                    _DropToMeLabel.Content = e.Data.GetData( DataFormats.Text );
                    break;

                case DragDropEffects.Move:
                    _DropToMeLabel.Content = e.Data.GetData( DataFormats.Text );
                    break;

                default:
                    break;
            }
        }

        DragDropEffects DecideDragAction( DragEventArgs e )
        {
            Point pos = e.GetPosition( _DropToMeLabel );

            // 単純に _DropToMeLabel の左半分ならば Copy、右半分ならば Move とする
            if( pos.X < (_DropToMeLabel.ActualWidth / 2) )
            {
                return DragDropEffects.Copy;
            }
            else
            {
                return DragDropEffects.Move;
            }
        }
        #endregion
    }
}
```

3.3. GTK#

GTK#では、GTK が元となっているので当然ですが、WinForms や WPF とは D&D の扱いも少し異なっています。なお私自身が GTK に詳しくないこともあり、GTK#の場合はあまり深く掘り下げていません。

まず送信側についてです。D&D 処理の話から少し外れますが、GTK の世界では「どの種類のイ

ベントを受信したいか」を前もってフレームワークに宣言しておく必要があるようです。少なくともマウス関連イベントを受信しない限り D&D を開始することはできませんので、送信側では Gtk.Window の AddEvent メソッドを使ってマウスイベントを受信するように設定しておきます。D&D セッション開始するには Gtk.Drag.Begin メソッドを使います。このメソッドには送信元、D&D で送信するデータ、そのデータに対してどのような操作を受信側に許可するか等を指定します。なおこのメソッドは呼び出されると D&D セッションを開始し、その完了を待たずにすぐ終了します（ノンブロック実行）。そのため受信側が行った操作を受けて実行する送信側の処理は、Begin メソッドの呼び出しに続けて記すのではなく、D&D セッション完了時に送信側で発生する DragDrop イベント（成功時）または DragFailed イベント（失敗時）に対する処理として実装します。

続いて受信側についてです。まず最初に Gtk.Drag.DestSet メソッドで受信側のウィジェットをドロップ可能なウィジェットとしてフレームワークに宣言します。あとは DragLeave、DragMotion、DragDataReceived の各イベントに対する処理を普通に実装すれば問題ありません。

以下に簡単なサンプルを記します。WinForms や WPF のサンプルと比べて荒っぽい点はご容赦ください　:)

```
using System;
using Gtk;
using Gdk;

public partial class MainWindow : Gtk.Window
{
   bool _IsDragging = false;

   public MainWindow () : base(Gtk.WindowType.Toplevel)
   {
       Build();

       // すべての GUI イベント通知を受け取るようにする
       this.AddEvents( (int)(Gdk.EventMask.AllEventsMask) );

       // このウィンドウを「ドロップ可能」とする
       Gtk.Drag.DestSet( this,
                       DestDefaults.All,
                       new TargetEntry[1] { new TargetEntry("text/plain", TargetFlags.App, 1) },
                       DragAction.Copy | DragAction.Move );

       EnterNotifyEvent += delegate(object o, EnterNotifyEventArgs e) {
           Console.WriteLine("EnterNotifyEvent: ({0}, {1})", e.Event.X, e.Event.Y);
       };

       MotionNotifyEvent += delegate(object o, MotionNotifyEventArgs e) {
           Console.WriteLine("MotionNotifyEvent: ({0}, {1})", e.Event.X, e.Event.Y);
           if( _IsDragging )
           {
               TargetEntry entry = new TargetEntry( "text/plain", TargetFlags.App, 1 );
               Console.WriteLine("----BEGIN----");
               object ret = Gtk.Drag.Begin( this,
                                            new TargetList(new TargetEntry[]{entry}),
                                            DragAction.Copy,
                                            1, e.Event );
               Console.WriteLine("----END----");
           }
       };
```

```
LeaveNotifyEvent += delegate(object o, LeaveNotifyEventArgs e) {
    Console.WriteLine("LeaveNotifyEvent: ({0}, {1})", e.Event.X, e.Event.Y);
};

ButtonPressEvent += delegate(object o, ButtonPressEventArgs e) {
    Console.WriteLine("ButtonPressEvent: ({0}, {1}) {2}", e.Event.X, e.Event.Y, e.Event.Type);
     _IsDragging = true;
};

ButtonReleaseEvent += delegate(object o, ButtonReleaseEventArgs e) {
    Console.WriteLine("ButtonReleaseEvent: ({0}, {1}) {2}", e.Event.X, e.Event.Y, e.Event.Type);
     _IsDragging = false;
};

DragBegin += delegate(object o, DragBeginArgs e) {
    Console.WriteLine("DragBegin:");
};

DragMotion += delegate(object o, DragMotionArgs e) {
    int width, height;

    this.GetSize( out width, out height );

    e.RetVal = ( e.Y < (height/2) );
    if( e.X < (width / 2) )
    {
        Gdk.Drag.Status( e.Context, DragAction.Move, e.Time );
        Console.WriteLine("DragMotion: move ({0})", e.RetVal);
    }
    else
    {
        Gdk.Drag.Status( e.Context, DragAction.Copy, e.Time );
        Console.WriteLine("DragMotion: copy ({0})", e.RetVal);
    }
};

DragLeave += delegate(object o, DragLeaveArgs e) {
    Console.WriteLine("DragLeave:");
};

DragDataReceived += delegate(object o, DragDataReceivedArgs e) {
    Console.WriteLine("DragDataReceived: info:{0}", e.Info);
    string text = e.SelectionData.Text;
    new Gtk.MessageDialog( this,
                           DialogFlags.Modal,
                           MessageType.Info,
                           ButtonsType.Ok,
                           text,
                           null ).Run();
};

DragFailed += delegate(object o, DragFailedArgs e) {
    Console.WriteLine("DragFailed: {0}", e.DragResult);
};

DragDrop += delegate(object o, DragDropArgs e) {
    Console.WriteLine("DragDrop: {0}", e.Context.Action);
    e.RetVal = true;
};
```

```
        DragEnd += delegate(object o, DragEndArgs e) {
            Console.WriteLine("DragEnd: {0}", e.Context.Action);
        };
    }

    protected void OnDeleteEvent (object sender, DeleteEventArgs a)
    {
        Application.Quit ();
        a.RetVal = true;
    }
}
```

3.4. Cocoa

最後に Cocoa、Mac OS X 標準の GUI フレームワークの場合について記します。なお調査していた頃には MonoMac がまだ話題になっていなかった頃なので、ここでの話は Cocoa を Objective C で普通に使う場合の話となります。.NET でも C#言語でもありませんのでご注意ください。

まず送信側についてです。Cocoa では D&D セッションを dragImage: at: offset: event: pasteboard: source: slideBack:メソッドで開始します（Objective C をご存じない方へ補足すると、これは C#風に言えばメソッド名が dragImage で、第二引数以降に at、offset、event、pasteboard、source、そして slideBack を受けるメソッドです)。なおこのメソッドは D&D セッションが完了するまで終了しません（ブロック実行)。しかし、メソッドの戻り値等では受信側が行った操作を知ることができません（私が試した限り…)。そのため受信側の操作を受けて実行するべき送信側の処理は、D&D セッション完了時に呼び出される送信側のメソッド draggedImage: endedAt: operation:で実装します。

続いて受信側についてです。WinForms 等とは異なり、受信側の view をドロップ可能にするためにすることはありません。特に何も設定せずとも draggingEntered:、draggingExited:、draggingUpdated:、performDragOperation:の各メソッドが呼び出されるため、これらのメソッドに処理を普通に実装すれば問題ありません。

サンプルとして、テキストデータとしてドラッグ可能で、テキストデータをドロップ可能な MyDraggableView という view の実装コードを以下に記します。

```
#import "MyDraggableView.h"

@implementation MyDraggableView

- (id)initWithFrame:(NSRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        // Initialization code here.
    }
    return self;
}

- (void)drawRect:(NSRect)dirtyRect {
    // Drawing code here.
}

- (void)mouseDown:(NSEvent *)theEvent {
    NSPoint curPoint = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    printf("mouseDown (%g, %g)\n", curPoint.x, curPoint.y);
    _lastMouseDownPoint = curPoint;
}
```

```
- (void)mouseDragged:(NSEvent *)theEvent
{
    NSPoint curPoint = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    printf("mouseDragged (%g, %g)\n", curPoint.x, curPoint.y);
    if( 3 < abs(curPoint.x - _lastMouseDownPoint.x)
       || 3 < abs(curPoint.y - _lastMouseDownPoint.y) )
    {
        // retrieve an image which is built in framework
        NSImage* image = [NSImage imageNamed:@"NSInfo"];

        // define a dummy value (this is not used 10.4 and later)
        NSSize offset;

        // set NSString object to a paste board
        // (this code uses NSDragPboard but other paste board also can be used)
        NSPasteboard* pboard = [NSPasteboard pasteboardWithName:NSDragPboard];
        [pboard clearContents];
        NSArray* copiedObjects = [NSArray arrayWithObject:@":)"];
        [pboard writeObjects:copiedObjects];

        // start a drag session
        printf("----- drag session START -----\n");
        [self dragImage:image
                at:curPoint
                offset:offset
                event:theEvent
                pasteboard:pboard
                source:self
                slideBack:YES];
        printf("----- drag session END -----\n");
    }
}

- (void)mouseEntered:(NSEvent *)theEvent {
    NSPoint curPoint = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    printf("mouseEntered (%g, %g)\n", curPoint.x, curPoint.y);
}

- (void)mouseExited:(NSEvent *)theEvent {
    NSPoint curPoint = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    printf("mouseExited (%g, %g)\n", curPoint.x, curPoint.y);
}

- (void)mouseMoved:(NSEvent *)theEvent {
    NSPoint curPoint = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    printf("mouseMoved (%g, %g)\n", curPoint.x, curPoint.y);
}

- (void)mouseUp:(NSEvent *)theEvent {
    NSPoint curPoint = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    printf("mouseUp (%g, %g)\n", curPoint.x, curPoint.y);
}

- (void)draggedImage:(NSImage *)anImage beganAt:(NSPoint)aPoint {
    NSPoint curPoint = [self convertPoint:aPoint fromView:nil];
    printf("draggedImage: beganAt: (%g, %g)\n", curPoint.x, curPoint.y);
}

- (void)draggedImage:(NSImage *)anImage endedAt:(NSPoint)aPoint operation:(NSDragOperation)operation {
    printf("draggedImage: endedAt: (%g, %g) operation:%x\n", aPoint.x, aPoint.y, (uint)operation);
```

```
}

- (void)draggedImage:(NSImage *)draggedImage movedTo:(NSPoint)screenPoint {
   printf("draggedImage: movedTo: (%g, %g)\n", screenPoint.x, screenPoint.y);
}

@end
```

4. 付録

4.1. サンプルプロジェクト

- WinForms のサンプル（Visual Studio 2008 用プロジェクトファイル一式）
  http://sgry.jp/pgarticles/DragDropWinForms.zip
- WPF のサンプル（Visual Studio 2008 用プロジェクトファイル一式）
  http://sgry.jp/pgarticles/DragDropWpf.zip
- GTK#のサンプル（MonoDevelop 用プロジェクトファイル一式）
  http://sgry.jp/pgarticles/DragDropGtkSharp.zip
- Cocoa のサンプル（Xcode 3.2.5 用プロジェクトファイル一式）
  http://sgry.jp/pgarticles/DragDropCocoa.zip

4.2. フレームワークごとの各イベント・操作の一覧表

		WinForms	WPF	GTK#	Cocoa
mouse events	カーソル進入	Control .MouseEnter	UIElement .MouseEnter	Widget .EnterNotifyEvent	mouseEntered:
	ボタン押し下げ	Control .MouseDown	UIElement .MouseDown	Widget .ButtonPressEvent	mouseDown:
	カーソル移動	Control .MouseMove	UIElement .MouseMove	Widget .MotionNotifyEvent	mouseDragged: [*]
	ボタンの押し下げ解放	Control .MouseUp	UIElement .MouseUp	Widget .ButtonReleaseEvent	mouseUp:
	カーソル退出	Control .MouseLeave	UIElement .MouseLeave	Widget .LeaveNotifyEvent	mouseExited:
start	ドロップ可能に	Control .AllowDrop	UIElement .AllowDrop	Gtk.Drag.DestSet	
	D&Dセッション開始	Control .DoDragDrop	DragDrop .DoDragDrop	Gtk.Drag.Begin	dragImage: at: offset: event: pasteboard: source: slideBack:
dragging source events	ドラッグ開始			Widget .DragBegin	draggedImage: beganAt:
	ドラッグ移動				draggedImage: movedTo:
	ドロップ終了			Widget .DragEnd	draggedImage: endedAt: operation:
	ドロップ成功			Widget .DragDrop	
	ドロップ失敗			Widget .DragFailed	
	（D&D中定期的に発生）	Control .GiveFeedback	UIElement .GiveFeedback		
	（D&D中定期的に発生）	Control .QueryContinueDrag	UIElement .QueryContinueDrag		
dragging dst. Events	ドラッグ侵入	Control .DragEnter	UIElement .DragEnter		draggingEntered:
	ドラッグ退出	Control .DragLeave	UIElement .DragLeave	Widget .DragLeave	draggingExited:
	ドラッグ移動	Control .DragOver	UIElement .DragOver	Widget .DragMotion	draggingUpdated:
	ドロップ実行	Control .DragDrop	UIElement .Drop	Widget .DragDataReceived	performDragOperation:

[*] ドラッグ&ドロップの話なので mouseDragged だけ記しています

5. 参照

[1] くれゆに, "GTK+で drag and drop," Crazy Unit (web site).
http://www5.airnet.ne.jp/c-unit/prog/dnd.html

[2] Microsoft, "ドラッグ アンド ドロップの概要," 年代不明, MSDN Library (web site).
http://msdn.microsoft.com/ja-jp/library/ms742859(v=VS.80).aspx

[3] Apple Inc., "Drag and Drop Programming Topics for Cocoa," 2006, Mac OS X Reference Library.
http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/DragandDrop/DraganddDrop.pdf