計算機科学'11 ＃2 —
ユーザインタフェース

久野　靖*

2011.5.31

1　ユーザインタフェースとその位置づけ

1.1　ユーザインタフェースとその評価基準

ユーザインタフェース (User Interface) とは、直訳すればシステムや機器とその使い手 (ユーザ) との間の「接点」という意味であり、さらに具体化すれば、その「接点」がどのようにできているかを意味することば、ということになるでしょう。

では、計算機との関係はどうでしょうか? 上の定義では計算機に限定されず、さまざまな機器/システム一般のユーザとの接点ということになり、実際にそのような意味でこの言葉を使う人もいます。しかし実際には、もっと限定して「計算機が実現しているシステムとユーザの接点」という意味でこの言葉を捉えている人の方が多いでしょう。[1]

それはなぜかというと、計算機ではそのソフトウェアが実現する機構が非常に複雑になり得ること、また画面には「どんなものでも」表示で、それを「どのようにでも」動かせる、という柔軟性があり、大きな可能性があるためではないでしょうか。また、物理的な装置なら、「動き」のようなものがあることで、何が起きているかをユーザが察知できるという面がありますが、ソフトウェアの「動き」は見えないため、画面表示のようなものだけが手がかりになる、という面もあるでしょう。ここでも以下ではソフトウェアによって実現されるユーザインタフェースについて学んで行くことにします。

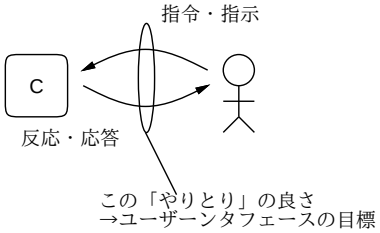

図 1: ユーザインタフェースの概念

次に、「良い」インタフェースとはどのようなものでしょうか。「使いやすいもの」とか言わないように。ここでいう「良い」と「使いやすい」はほぼ同義語です。たとえば「色使いとか見た目はすごくいいけど使いづらい」インタフェースも考えられますが、そのような「使いやすさと無関係な良さ」はここでは扱いません。

*経営システム科学専攻

[1]もっと「計算機と人間のやりとり」であることを明示する用語として、HCI(Human Computer Interaction) という用語が使われることもあります。

それで、インタフェースの「良い (使いやすい)」度合いはどうやって計測したり比較できるでしょうか。たとえば、同じ機能を提供するインタフェースが A、B の 2 種類あるとして、どちらが良いかを決めるにはどうしたらいいか、ということですが。

それには、たとえば次のような方法が考えられます (ほんの一例)。

(a) 利用者に操作してもらってアンケートで尋ねる

(b) 同一の作業を両方のインタフェースで複数回やってみて、所要時間を計測する

(c) 同一の作業を両方のインタフェースで複数回やってみて、疲労度を計測する (フリッカーテストなどの手法があります)

(d) 初めての人にまったく何の説明もせずに使ってもらい、うまくある作業ができるまでの時間を計測する

(e) 初めての人に一定回数 (期間) 繰り返し使ってもらい、所要時間 (等) の習熟曲線を比較する

(f) それぞれのインタフェースを実現するプログラムの行数、所要メモリ量、必要とするハードウェア等を比較する

(g) さらに色々…

最初の (a) は主観的評価ということになります。もっと客観的な評価を求めていたのでは、と思うかも知れませんが、マーケティング的に言えば顧客は「気に入った」ものを購入するでしょうから、用途によっては (コンシューマ向けで好みで買われるようなソフトであれば) これが正解かも知れません。

(b) や (c) などは実験をやった結果ということになり、だいぶ計測らしくなります。しかし時間の方が重要なのでは、と思いますか? 長時間使うインタフェースであれば、1 回の操作が速くても途中で疲れて続けられなくなるのでは意味がないかも知れません。

(d) や (e) はどうでしょう。ごくたまに必要があって使うようなものであれば、使おうとした時には操作方法を忘れているでしょうから、何も覚えていなくても使える、というのが重要かも知れません。一方で、「初めての人でも使える」ということは非常にいいことだと思われていますが、繰り返し使う場合には十分習熟した後での能率の方が重要になります。たとえばコマンド vs メニューの議論がよくあります。コマンドは覚えないと打てないので、初心者には覚えなくても済む (選択肢が向こうから提示される) メニューの方が人気があります。しかし繰り返し使う場合には、いちいちマウスに手を伸ばしてメニューを操作するより、コマンドを打つ方が速くて疲労も小さい場合が多いわけです。

(f) はどうでしょうか。いかによいインタフェースでも、肝心の機器に搭載できなければ訳に立ちません。また、携帯電話のように画面が小さい機器では大きな画面が必要なインタフェースは採用が困難です (すごく小さいメニューにして十分な選択肢が表示できるようにしたとしても、目が悪くなるようでは売れないでしょう)。というわけで、評価基準は「用途によってさまざま」というのがありがちですが結論と言えるでしょう。

1.2 さまざまなユーザインタフェース

今日、計算機システムのユーザインタフェースひとくちに言っても、その環境や制約によって様々なものが考えられます。典型的なものを挙げてみましょう。

- PC(デスクトップ、ノート PC) のユーザインタフェース — 今日の PC は任意のカラー画像を表示可能なディスプレイ、キーボード、ポインティングデバイス (マウスやトラックパッド) を備えていて、その上で **GUI**(Graphical User Interface) を使用するのが普通です。

- コンソールインタフェース — サーバマシンやルータ (ネットワーク機器) はユーザが直接使うためのものではないので、GUI は搭載せず、キーボードと文字画面だけの端末装置を必要時に接続して操作するため、文字インタフェース (**CUI**、Character User Interface[2]) が採用されます。
- PDA のユーザインタフェース — **PDA**(Personal Digital Assistant) とは、ノート PC より小型で持ち運びやすい情報機器で、Zaurus や Palm などが代表的です。これらはキーボードを搭載していないものも多く、搭載していても小型で打ちにくいため、画面をペンで操作することに特化したインタフェースを持つことが多いです。
- 携帯電話のユーザインタフェース — 携帯電話器では PDA よりさらに画面が小さいことと、電話器という由来からテンキーを中心とするキーが必ず備わることから、キーでメニュー項目を選択する形のインタフェースが主流です。
- Web インタフェース — 今日では多くの情報サービスは Web 経由で提供されているため、これらのサービスが提供するインタフェースはブラウザが持つ機能の範囲内で設計され、標準的な Web ブラウザ上で動作するようになっています。これは一般的に言えば GUI の一種ですが、上記の制約から独特の進化をとげています。

以下では、Unix 上の標準的な GUI 環境である X Window を題材に、GUI の土台となるウィンドウシステムの原理と機能、およびその上の GUI 環境について扱い、続いてプラットフォーム独立なインタフェースの具体例として HTML を用いた Web 上のインタフェースについて取り上げます。

2 ウィンドウシステムと X Window

2.1 ウィンドウシステムとその外観

本節では、GUI が使用する機能群を提供する土台である**ウィンドウシステム**と GUI のさまざまな側面を見て行くことにしましょう。ウィンドウシステムとは、GUI を構築するための基本的な機能である「画面にさまざまなものを描く」「画面に対する入力を扱う」という 2 つの機能を提供する部分であり、Windows では OS の一部として最初から組み込まれています。

これに対し、Unix では **X Window**(略して X) は OS とは別個のソフトウェア (**ミドルウェア**) になっています。このことにより、過去においては Unix 上では研究目的でさまざまなウィンドウシステムが作られて来ましたが、その中で X が広く普及し、現在では Unix の標準的なウィンドウシステムとなっています。このような経緯から、X はもともと土台となる OS からは独立したプログラム (群) として動くようにできているため、その構造や動作を調べたりするのには好都合です。以下では X を題材にウィンドウシステムと GUI のさまざまな側面を見て行きますが、ここで取り上げる多くの概念は Windows や Mac OS でも共通しています。

最初に、ウィンドウシステムとはどんなものか改めて見てみましょう。図 2 に、典型的なウィンドウシステムの「道具だて」を示します。

まず、**スクリーン** (画面、利用者がこの上でさまざまな作業をすることから**デスクトップ**と呼ばれることもある) は通常、表示装置の画面全体に相当します。システムによっては、複数のデスクトップを使えるものもあります (作業ごとに別の「机」がある方がよさそうですか?)。

画面の中には複数の**ウィンドウ** (窓) が存在していて、その中で様々な作業を行うことができます。ウィンドウシステムの出現以前は、画面の中では一時に 1 つの作業しかできず、ある作業をやりながら時々別の作業もしたい時は「頭の中で」作業を保留しておいて切り替えるなどの必要がありました。それと比べて、単に作業ごとに別の窓を開いておけばよく、1 つの作業の表示を見ながら別の作業をこなせるウィンドウシステムは大きな進歩だと言えます。

複数の窓の作業のうちで現在どれを取り扱っているか (たとえばキー入力をどのプログラムに渡すか等) は、マウス等のポインティングデバイスによって切り替えるのが普通です。このほか、キーボー

[2]ないし CLI、Command Line Interface

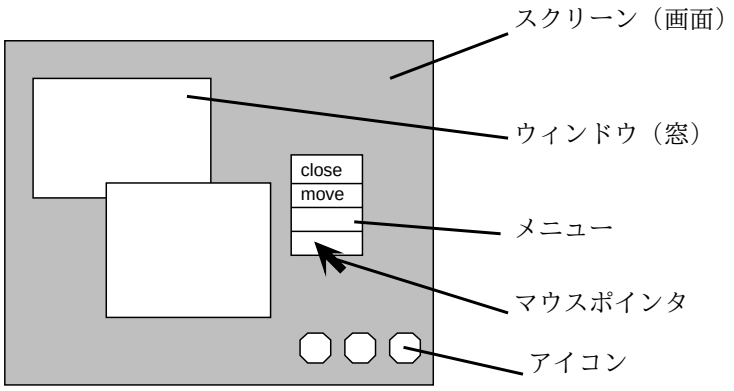

図 2: ウィンドウシステムの道具だて

ドからマウスに手を移さなくても済むように、キー操作だけでも切り替えができるようにしてある場合もあります。

画面上に、マウス等の動きに対応して動く矢印などの目印 (**マウスポインタ**) があり、これで「現在どこを指しているか」が利用者にフィードバックされます。窓を切り替える時の流儀としては、単にポインタがどれかの窓の領域に入っただけで切り替わる流儀と、窓の領域に入った後でマウスボタン等をクリックすると始めて切り替わる流儀とがあります。

画面に表示されるものは作業用の窓以外にも様々なものがあります。代表的なのが**アイコン**で、これは小さな絵などによって「何か」を表したものです。「何か」としても、たとえば次のような様々な流儀があります。

(a) フロッピー、ハードディスク、プリンタなどの装置

(b) ファイルやディレクトリ

(c) 起動できるプログラム

(d) 特定の操作や機能

(e) 窓を一時的に閉じたもの

ある特定のユーザインタフェースで、どの流儀 (複数が混ざっている場合もあります) を採用するかは、そのユーザインタフェースをデザインする人次第で違って来ます。

たとえば Windows や Macintosh ではアイコンはおもに (b) を表しますが、プリンタアイコンなどは (a)、ごみ箱アイコンなどは (d) を表していると言えます。X の場合は、後で説明するようにウィンドウシステム自体は特定のユーザインタフェースを規定しないため一律には言えませんが、もともとの流儀としては (e) に近いと言えます。

なお、ここまでに述べたのはデスクトップ (画面) 全体についての話であり、個々の窓の中がどのようであるかは、その窓に対応するプログラムにすべて任されています。

2.2 ウィンドウシステムの構造

ウィンドウシステムの外観は上記のようなものでしたが、ではそのようなシステムはどのような構造を持っているのでしょう? CPU と計算機の画面は、ハードウェア的には図 3 にあるように、**フレームバッファ**を介してつながっています。

フレームバッファは CPU から見れば普通の主記憶と同様に読み書きできるメモリですが、ただし書き込むとその場所に書き込んだビット列がそのまま対応する画面上の点の赤/緑/青色の輝度の明暗に対応して現れるようになっています。これは、ビデオコントローラがフレームバッファの内容を読

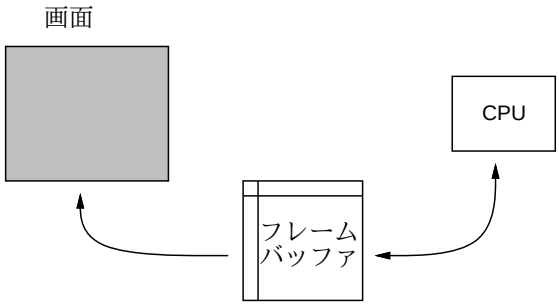

図 3: フレームバッファとビットマップ画面

み出し、それをディスプレイ装置に伝送し、ディスプレイ装置側で縦横に並んだ液晶の点の明暗を変化させることで行なわれます (図 4)。[3]

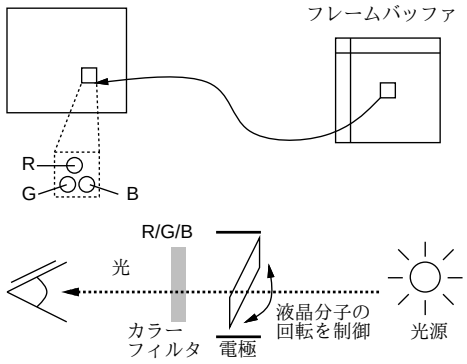

図 4: 液晶ディスプレイの原理

さて、このようなハードウェアがあったとして、その上で先に見たようなウィンドウシステムを作るとしたらどのように設計すればよいでしょう?

一つの方法は、窓を作り出す各プログラムそれぞれが「自分の窓はどこどこにあるから、その場所に窓の内容を描こう」という形で動く方法、つまり各窓のプロセスに任せるやり方です (図 5)。

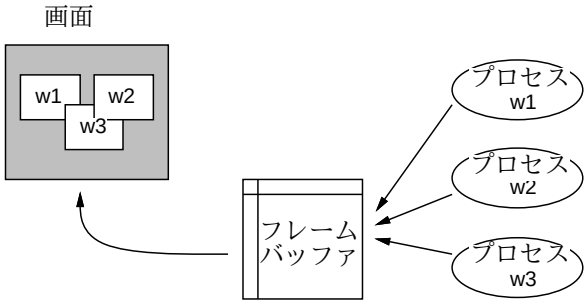

図 5: 直接描画方式のウィンドウシステム

この方式は、各プログラムがフレームバッファに直接書き出すので性能的には有利ですが、各プログラムが描画に必要なグラフィクスのコードを持つ必要があります (実際には決まったライブラリを組み込むだけですが)。また、この方式だと各窓の重なり具合によって隠れているところを互いによけて描く必要がありますし、そうしたとしても実際にはそれぞれが勝手に自分の窓の中身を描くわけ

[3]実際には、液晶は個々の点単位で光の透過率を制御するもので、背後の光源と前面の赤/緑/青のカラーフィルタの間に液晶を入れて制御することで、各色の点の明るさ (暗さ) を制御しているわけです。

にはいかず、どこかに「順番を調整する」部分が必要です (たとえば他のプログラムが描いている最中に自分の窓をその上に移動して隠そうとしたら、描いている方を止める必要があります)。

そこで、**ウィンドウサーバ**と呼ばれるプロセスを 1 つ用意し、このプロセスが窓を作る各プロセスの依頼を受けてフレームバッファへの書き込みを管理する、**サーバ方式のウィンドウシステム**が考案されました (図 6)。X Window はサーバ方式のウィンドウシステムとして広く普及した最初のものです。

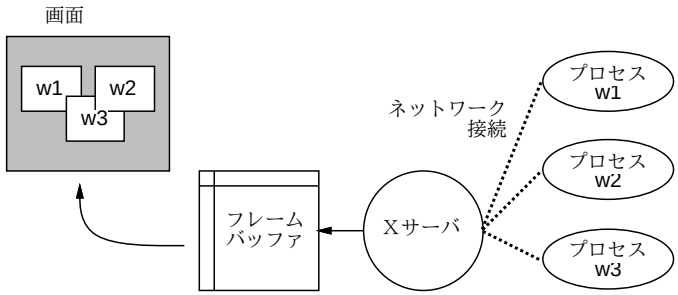

図 6: サーバ方式のウィンドウシステム

この方式では、各種の描画ルーチンはサーバのみが持てばよく、また窓の重なりを考慮した描画もサーバが一括しておこないます。各窓に対応するプロセスはサーバとネットワーク通信機能によってつながり、サーバに対して「窓を作って欲しい」、「窓のどこにどんな図形/文字列を描いて欲しい」などの要求を出します。また、利用者のキー入力やポインタ操作の情報もサーバが受け取って各プログラムに通知します。[4]

サーバ方式のウィンドウシステムでは、サーバと各プロセス (クライアント) が通信できさえすればいいので、**各クライアントはサーバと同じマシンにいなくてもいい**という利点があります。サーバはフレームバッファに書き込むので、必ず画面のある計算機で動かす必要がありますが、その他のプロセスはそれぞれの仕事に都合のよいマシンで動かすことができます (図 7)。

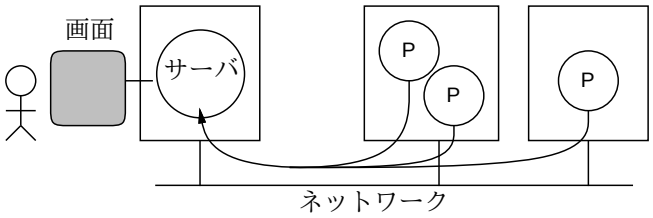

図 7: ネットワーク透過なウィンドウシステム

この時、各プロセスはサーバ以外のマシンで動作していても、利用者にとっては手もとのキーボードやマウスで入力を行ない手もとの画面でその表示を見ますから、プロセスがネットワークの向うにあることは意識されません。このような (ネットワークが間に介在していてもそのことを意識させないという) 性質を**ネットワーク透過性** (ないし分散透過性) と言います。ネットワーク透過であることの利点としては次のものがあります。

- 同じプログラムを、各システムの負荷に応じて一番資源に余裕のあるところで動作させることができる。
- 複数のマシンの資源を 1ヶ所に座ったままで利用できる。

[4]さらに進んだ方法として、サーバ内部にプログラム言語の実行系を用意し、必要に応じて実行時にサーバの機能を拡張していける方式 (プログラマブルサーバ方式) が考案されましたが、通常の (窓を作り出す) プログラムとサーバ内のプログラムの両方を取り扱うという煩雑さのせいか、必ずしも普及しませんでした。

- 特別なマシンでしか動かせないプログラムでも、そのマシンの前に行かずに使うことができる。
- 手もとのマシンがサーバの動作に専念できるので、サーバを効率よく動かすことができる。

なお、X の場合、ユーザの「前で」動くプログラムがサーバであり、ネットワークの「向こうで」動くのがクライアントなので、位置関係が普通のクライアントサーバ型アーキテクチャと「逆」だということに注意してください。

2.3 リクエストとイベント

X サーバとクライアントはネットワーク接続を経由してやりとりする、と説明しましたが、具体的にはどんなやりとりをしているのでしょうか? クライアントから X サーバへの通信は、前にも述べたように「窓を作れ」「どの窓のどの位置に何を描け」といった指示が中心です。これらを X の用語ではサーバへの要求 (**リクエスト**) と呼びます。

では、サーバからクライアントへはどんな内容の通信が行われるのでしょうか? 具体的には次のようなものがあります。

- 入力デバイスの情報 — キーボードのどのキーが押された、どのマウスボタンが押された、マウスカーソルがどの位置にある、など。
- 窓の状況 — 窓ができて内容が見えるようになった、窓を隠していた別の窓が動いて隠されていた部分が見えるようになった、窓の大きさが変化した、など。

これらの情報を X では総称して**イベント**と呼びます。**xev** というプログラムを使ってみると、どのようなイベントがあるかを観察できます。これを動かすと画面上に窓が現れ、その窓に関係するイベントがサーバから送られるとその内容が標準出力に出力されます。

```
% xev
Outer window is 0x3400001, inner window is 0x3400002
(途中略)
Expose event, serial 14, synthetic NO, window 0x3400001,
    (0,0), width 178, height 10, count 3
...
Expose event, serial 14, synthetic NO, window 0x3400001,
    (0,68), width 178, height 110, count 0   ←「見えるようになった」
(途中略)
FocusIn event, serial 15, synthetic NO, window 0x3400001,
    mode NotifyNormal, detail NotifyPointer  ←「カーソルが入って来た」
(途中略)
MotionNotify event, serial 15, synthetic NO, window 0x3400001,
    root 0x25, subw 0x0, time 2423435336, (5,99), root:(507,211),
    state 0x0, is_hint 0, same_screen YES
MotionNotify event, serial 15, synthetic NO, window 0x3400001,
    root 0x25, subw 0x0, time 2423435378, (4,98), root:(506,210),
    state 0x0, is_hint 0, same_screen YES   ←「カーソルが動いている」
...
ButtonPress event, serial 15, synthetic NO, window 0x3400001,
    root 0x25, subw 0x0, time 2423437355, (33,90), root:(535,202),
    state 0x0, button 1, same_screen YES    ←「ボタン押した」
ButtonRelease event, serial 15, synthetic NO, window 0x3400001,
    root 0x25, subw 0x0, time 2423437410, (33,90), root:(535,202),
    state 0x100, button 1, same_screen YES  ←「離した」
KeyPress event, serial 15, synthetic NO, window 0x3400001,
    root 0x25, subw 0x0, time 2423439806, (33,90), root:(535,202),
    state 0x0, keycode 85 (keysym 0x61, a), same_screen YES,
    XLookupString gives 1 characters:  "a"  ←「キー押した」
KeyRelease event, serial 17, synthetic NO, window 0x3400001,
    root 0x25, subw 0x0, time 2423439926, (33,90), root:(535,202),
```

```
    state 0x0, keycode 85 (keysym 0x61, a), same_screen YES,
    XLookupString gives 1 characters:  "a"  ←「離した」
(以下略)
^C
%
```

やってみると、理屈では分かっていたつもりでも、実際の X クライアントでは極めて大量のイベントが次々と X サーバから送られていることに驚かれたのではないでしょうか。

2.4 イベントドリブンプログラム

前項に述べたように、X のクライアントプログラムではすべてのユーザ入力は統一的にイベントとして送られてきます。そこで、クライアントプログラムの流れは一般に次のようになります。

```
初期設定、必要な窓を作る;
while(1) {
  イベントを受け取る;
  イベントの種類毎に対応した処理; ☆
}
```

つまり、普通のプログラムでは処理の必要に応じてあちこちでユーザ入力を受け取りますが、X のプログラムではイベントを読むところは 1 か所だけしかなく、その後の巨大な if 文 (☆のところ) ですべての場合分けと処理を行うわけです。このようなプログラム構造を**イベントドリブン**と呼びます。

お話だけだとつまらないので、簡単なプログラム例を示します。このプログラムは窓を 1 個作り、その中に黒丸を 1 個描きます。そして、マウスポインタが窓の中に入ると、黒丸がマウスポインタにくっついて動きます。キーボードのキーをどれでも押すと終了します。

```
/* xmotion.c --- move a circle according to mouse motion */
#include <X11/Xlib.h>
static int x = 0, y = 0;

main() {
  Display *disp = XOpenDisplay(NULL);             /* 1 */
  Screen *scr = DefaultScreenOfDisplay(disp);
  Window root = DefaultRootWindow(disp);          /* 2 */
  unsigned long black = BlackPixelOfScreen(scr);
  unsigned long white = WhitePixelOfScreen(scr);
  Window mw=XCreateSimpleWindow(disp,root,100,100,800,600,2,black,white);/*3*/
  XSelectInput(disp, mw, PointerMotionMask|KeyPressMask|ExposureMask);   /*4*/
  XMapWindow(disp, mw);         /* 5 */
  while(1) {
    XEvent ev;
    XNextEvent(disp, &ev);      /* 6 */
    if(ev.type == KeyPress)
      exit(0);                  /* 7 */
    else if(ev.type == Expose)
      draw(disp, mw, 100, 100); /* 8 */
    else if(ev.type == MotionNotify)
      draw(disp, mw, ev.xbutton.x, ev.xbutton.y);   /* 9 */
  }
}
draw(Display *disp, Window mw, int x1, int y1) {    /* 10 */
  GC dgc = DefaultGC(disp, 0);
  XClearArea(disp, mw, x, y, 20, 20, 0);            /* 11 */
  x = x1; y = y1;
  XFillArc(disp, mw, dgc, x, y, 20, 20, 0, 360*64); /* 12 */
}
```

図 8: xmotion.c を動かしたところ

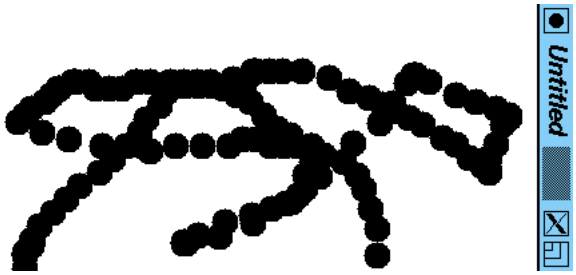

図 9: クリアしないと…

具体的な説明は次の通りです。

1. Display、Screen は画面を現すデータ構造。
2. Window は窓に対応。ここではルートウィンドウとこのプログラムが作り出す窓 (mw) の 2 つを扱う。
3. XCreateSimpleWindow で背景の窓の中に位置 (100,100)、大きさ 800 × 600 の窓を作る。縁の幅は 2 ドット、絵や字は黒、地の色は白。
4. マウスポインタ移動、キー押下げ、窓が見えるようになった、の 3 種類のイベント通知を依頼。
5. 窓を表示させる。
6. 無限ループの中で、まずイベントを受け取り、種類によって分かれる。
7. キー押下げならこのプログラムを終了。
8. 窓が見えるようになったのなら、(100,1000) の位置に黒丸を描く。
9. マウスボタン押下げなら、その時のマウスの位置に黒丸を描く。
10. 以下黒丸を描くサブルーチン。
11. 最後に黒丸を描いた位置から幅 20、高さ 20 の範囲をクリア。
12. 新しい位置を覚え、その位置に幅 20、高さ 20 の黒丸を描く。

これを動かすには、オプションがいくつか必要です (システムによって指定方法やディレクトリは違うかも知れません):

```
% gcc  xmotion.c -I/usr/X11R6/include -L/usr/X11R6/lib -lX11
% a.out
```

上記のようにすると、画面に窓が現われ、その中に黒丸が 1 つ見えます (図 8)。これは、最初に窓が現われた時「見えるようになったよ」と Exposure イベントが送られてきて、プログラムがそれに対して黒丸を 1 個描くためです。そして、この窓の上でマウスを動かすと黒丸がついてきます。

「ついて来る」と書きましたが、実際にはコードにあるように、マウスポインタが移動するごとに「前の位置の黒丸を消し」「新しい黒丸を描く」ことを繰り返しているだけです。つまり、プログラムでどのようにでも画面に描くことができ、人間はそれを見てあたかも「黒丸がついてくる」ような気持ちになる、というわけです。この、見え方次第で人間がさまざまに「騙される」ところがユーザインタフェースの面白いところであり、工夫しどころであるとも言えます。

「騙されている」ことを認識するために、XClearArea の行を一時的にコメントアウトして動かしてみましょう (図 9)。そうすると、プログラムが単に次々と丸を描いているだけだということがよく分かると思います。

また、単純に丸がついてくるのではなく、次のようなさまざまなことを行ってみると、「普通でない」効果が自在に作れることが分かると思います。

- マウスポインタよりも 100 ピクセル左側に丸を位置させる。
- ある箇所より右には丸が行かないようにする。
- 右にいくほどポインタと丸の「ずれ」が大きくなるようにする。
- ある箇所より右に行くと「ずれ」が出来る (ワープする) ようにする。
- マウスポインタを右に動かすと丸が下に動き、下に動かすと丸が右に動くようにしてみる。
- 右にいくほど丸が大きくなるようにする。

たとえばゲームなどではこのようなさまざまな効果を最大限駆使していろいろな工夫をしていますが、通常のアプリケーションでも工夫次第で (単に面白いとかではなく)「使いやすい」インタフェースを作る機会はさまざまにあるはずです。

3 X Window 上の GUI 環境

3.1 X クライアント

X はサーバ方式のウィンドウシステムであり、窓を作るプログラム (**X クライアント**) はどのマシンで動いていても構いません。しかし、ネットワーク中には色々な人のサーバが同時に動いているはずです。「どのサーバに窓を作るか」はどうやって決めるのでしょう? これは次のようになっています。

- プログラムを起動する時、「`-display` ホスト名`:0.0`」というオプションを指定することで明示的に指定する。
- 環境変数 `DISPLAY` に「ホスト名`:0.0`」なる文字列が入っていて、これによって定まる。

オプションの指定があればそれは環境変数に優先します。

ところで、この指定で誰がどの画面にでも窓を作れるのでは安全上問題があります。そこで、通常の状態では画面保護が掛かっていて、その画面で X を起動した人にしかその画面の窓が作れなくしてあります。この保護を外したり元に戻したりするのには **xhost** コマンドを使い、次のようにして制御します:

- `xhost` ホスト名 — 指定したホストから窓が作れるように許可する
- `xhost +` — 任意のホストから窓が作れるように許可する
- `xhost -` — 許可を取り消し、元の保護状態に戻す

ただし、実験用に保護をちょっと外すのは構いませんが、いつも外していると他人に自分の X サーバに接続されて悪さをされる恐れがあります。注意しましょう。

クライアントに対して共通に指定できる`-display` 以外のオプションとして、クライアントの窓の位置や大きさを指定するものがあります。これは

`-geometry` 幅 x 高さ`+X` 座標`+Y` 座標

という形で指定します。座標は画面の左隅/上隅からの距離で指定しますが、座標の前の符合を+の代わりに-にすることで、それぞれ画面の右隅/下隅からの距離でも指定できます。

X 上で動作するクライアントは非常に多種多様ですが、代表的なものを挙げておきます。

- `kterm`、`xterm` — 端末エミュレータ (日本語/英語版)
- `xpaint`、`gimp` — お絵描き/画像加工ソフト
- `kdraw`、`tgif`、`xfig` — 図作成ソフト
- `xv`、`display` — 画像表示ソフト
- `xclock`、`oclock` — 時計
- `xcalc` — 電卓
- `xbiff` — メールが来ているかどうか表示

これらのうち**端末エミュレータ**というのは、先に出て来た画面端末の「まね」(エミュレーション) をするプログラムであり、その中でシェルを動かしてコマンドを打ち込むのが主な用途です (いわゆる「コマンド窓」)。もともと Unix は、シェルにコマンドを実行させることで何でもできるシステムでしたから、X が作られた時もまずはコマンド窓を作ってそこでコマンドを打ち込み作業するようにしたわけです。

なお、端末エミュレータは文字を入力し文字を出力するプログラムなら何を動かすのに使っても構いません。たとえば `kterm` は「`-e` コマンド 引数…」というオプションを指定することで任意のコマンドを指定でき、「`kterm -e tr a b`」を実行すると、窓に打ち込んだものは何でもそのまま打ち返され、ただし「`a`」はすべて「`b`」に置き換えられます (何の役に立つということもないですが)。

ただし、図や画像を扱うのはコマンド窓では済みませんから、そのためのプログラムが次に作られました。また、時計や電卓みたいなアクセサリ的な小物も、X の機能のデモンストレーションを兼ねて作られたわけです。

3.2 リソース

Unix では通常、各プログラムに対する初期設定をホームディレクトリ上のドットファイルに書きます (たとえば`.bashrc` など)。しかし、X Window 関連の場合はこの方法は次の点から好ましくありません。

- 様々な種類の窓を通じて同じフォントや色を指定したいことが多いのに、そのたびに「なんとか rc」というファイルを多数編集するのは面倒
- ネットワーク経由で使うことも多く、その場合はマシンごとにホームディレクトリが違っているかも知れない

では、これらの指定を格納しておくのに適した場所はどこでしょうか? その答は、「X サーバの中」です (どのマシンのどのクライアントも、共通の X サーバにアクセスするわけですから)。そこで、X サーバの中にリソースデータベース (図 10) というものを用意し、オプションの標準値をこの中に格納します。その設定や表示には **xrdb** コマンドを使います:

- `xrdb` [`-m`] [ファイル] — ファイルや標準入力からリソース指定を取り込む。`-m` を指定すると、現在の指定に追加して混ぜる。指定しない場合は現在の指定をクリアして取り換える
- `xrdb -q` — 現在のリソース指定内容を出力

これを使っている例を示します:

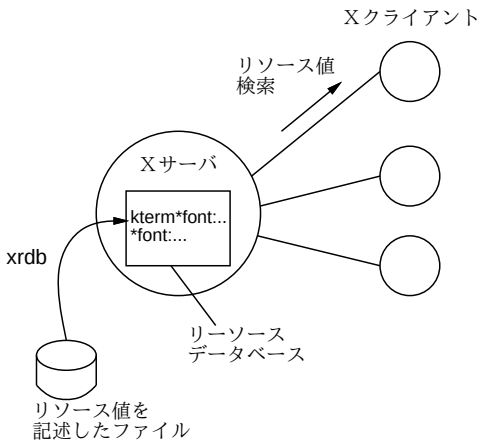

図 10: リソースデータベースの概念

```
% xrdb ~/.x11defaults      : ファイルからデータベース設定
% xrdb -q                  : データベースの内容表示
kterm*font:           a14  : ASCII フォント
kterm*romanKanaFont: r14   : JIS8 フォント
kterm*kanjiFont:      k14  : 漢字フォント
%
```

リソース指定は「クライアント名*リソース名: 指定値」の形をしていますが、クライアント名が空の (指定が*から始まる) 場合は、「すべてのクライアントについて」という意味になります (フォントや色をまとめて指定するのに便利です)。試しに、キーボードから指定を追加してみます (前の指定をクリアしないため-m を指定しています):

```
% xrdb -m
*background: yellow    ←全部のコマンドの背景色は黄色に
*foreground: red       ←全部のコマンドの文字色は赤に
^D                     ← Control-D:終わりの印
% xrdb -q
kterm*font:           a14
kterm*romanKanaFont: r14
kterm*kanjiFont:      k14
*background:          yellow
*foreground:          red
% xclock &
```

このようにした後で xclock を動かすと地の色は黄色、文字盤の色は赤になっています (他のプログラムを起動しても同様)。これはサーバを起動し直せば元に戻りますが、常に黄色や赤にしたければ上記の行を後述するやり方で毎回読み込ませればよいわけです。

なお、色の指定は名前を使う (/usr/X11R6/share/X11/rgb.txt というファイルに使える名前の一覧があります) か、または「#rrggbb」形式 (RGB3 色の明るさを 16 進数 2 桁ずつで指定する形式) を使います。

3.3 サーバの調整

上記のリソースは各種クライアントのオプションをまとめて設定するものでしたが、この他にサーバの状態や背景を変更するには **xset**、**xsetroot** などのコマンドを使用します。

- `xset m` 感度 閾値 — マウスの感度を調整する
- `xset c` ボリューム — 0〜100 の値でキークリック音の大きさを指定
- `xset r on/off` — オートリピートの on/off
- `xset q` — 設定値の現状値を表示
- `xsetroot -solid` 色 — 背景を指定した色にする
- `display -window root` 画像ファイル — 背景を指定した画像ファイルにする

ところでマウスの感度と閾値とは何でしょう? 実はマウスポインタは通常、ゆっくりマウスを動すと 1 ピクセルずつ動きますが、ある程度以上速くマウスを動かすと動きが N 倍に「加速」されるようになっています。たとえば「`xset m 1 1`」とすると加速されなくなるので、この「加速」機能がどれくらい有難いものかよく分かります。逆に「`xset m 100 1`」や「`xset m 100 5`」などとするとポインタの制御がとても困難になります。この値を調整してみると、普段なにげなく使っているユーザインタフェースがどれくらいうまくデザインされているか身にしみて分かるはずです。

3.4 起動時の設定 OPTION

X の起動には **xinit** コマンドが使われます。(システムによっては **startx** というコマンドを使うこともあります。また、常時 X が動いていてログインも X 上で行うような環境もあります。)

- `xinit` — X Window を立ち上げる

このプログラムはまず X サーバを起動し、続いてホームディレクトリにあるファイル `.xinitrc` に書かれている内容を順番に実行します。簡単な `.xinitrc` の内容を次に示します。

```
export PATH=/usr/local/X11R6/bin:/usr/local/bin:$PATH
xsetroot -gray     ←背景を灰色に
xset m 4 2         ←マウス感度を調整
xrdb $HOME/.x11defaults     ←リソースをロード
oclock -transparent -geom 100x100-0+0 &    ←丸い時計
twm &                       ←ウィンドウマネージャ
kterm -geom 80x48+300+100 -T console -n console -C -e bash
```

このように、各種の設定は X の起動後に使うのと同じコマンドで行いますから、これらを変更してやれば毎回好みの設定で作業を始めることができます。たとえば、背景の色を変えたければ `xsetroot` のオプションを変えればいいし、背景に画像を出したければ代りに `display` コマンドを使えばよいわけです。リソース指定もここで読み込ませているので、読み込ませているファイルの内容をいじれば初期状態を変更できます。

また、最初から開いている窓を増やしたければその窓を開くコマンドを追加します。ただしその場合、複数の窓はそれぞれ並行して動く複数のプロセスになるので (上の例の `oclock` などの行のように) 最後に「`&`」をつける必要があります。逆に、一番最後のコマンド (上の例の `kterm`) は、これが終った時に `xinit` が X サーバを停止させるようになっているので、必ず「`&`」なしにします。[5]

3.5 ファイルマネージャ

PC に慣れていて「素の状態の」X Window をはじめて使う人がまずとまどうのは、何かをするにはとにかくコマンド窓を開いてコマンドを打ち込まなければならない、という点にあるようです。特

[5]もし行末に「`&`」をつけると、最後のコマンドが何もしないコマンドで瞬時に終るため、X が立ち上がったと思ったらすぐ終了する、という状態になります。

にファイルやディレクトリについてはいちいち `ls` を使って一覧を表示して、何があるかを確認したり、名前を打ち込んで指定するのが煩わしいという人が多そうです。[6]

もちろん、X 上でも Windows のエクスプローラのように GUI でファイルやディレクトリの構成を表示し操作させてくれるプログラムを作ることは簡単です。そのようなツールは**ファイルマネージャ**と呼ばれることが多いようです。

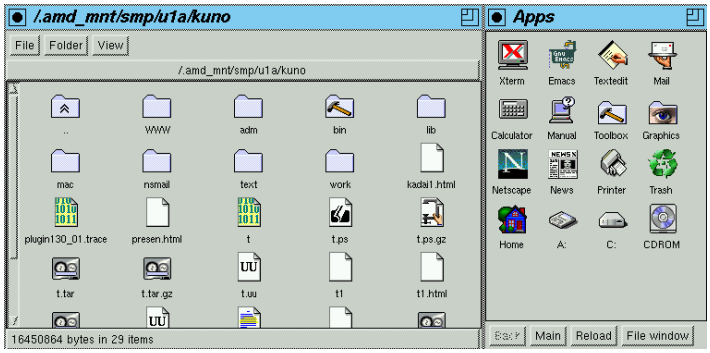

図 11: xfm

ファイルマネージャの例として **xfm** を見てみましょう。このツールを「`xfm &`」により起動すると (第 1 回目だけは設定ファイルを用意するかと聞かれるので「continue」を選ぶ)、図 11 のように 2 つの窓が画面に現われます。ここでタイトルバーにパス名が表示されている窓ではディレクトリやファイルに対応するアイコンが現れ、ディレクトリのアイコンをダブルクリックすることでカレントディレクトリを移動し、そこにあるファイルの一覧を見ることができます。

また、ファイルのアイコンをダブルクリックするとそのファイルに応じたプログラム (`.txt` なら Emacs など) が起動され、そのファイルを表示したり修正したりできます。それとは別のプログラムでファイルを扱いたい場合は、ファイルのアイコンをドラグしてタイトルバーに「Apps」と書かれた窓 (プログラム窓) のアイコンに重ねてやると、そのアイコンに対応したプログラムでファイルを開くことができます。

使い方はさておき、ここで言いたいことはつまり Mac や Windows で「一番の大元」だと思っていた機能は「単なるクライアントの 1 つ」であるということです。ファイルを操作したりプログラムを起動するのは「一番重要な」作業でしょうから、PC を起動すると黙ってその機能が出て来るのは正しいわけですが、ウィンドウシステムの原理という点ではファイルマネージャも他のプログラムと変わらないわけです (他に窓を制御する機能がありますが、これについては次の節で扱います)。

なお、`xfm` がどのファイルをどのアイコンで表示するか、ダブルクリックしたらどのプログラムが動くか、プログラムの窓にどのようなものが現れるか、といったことは初回起動時にホームディレクトリに作らるディレクトリ「`.xfm`」の下の設定ファイル群に書かれています。これらのファイル群を調整すれば、自分の好きなように絵を変更したり新しいプログラムを追加したりできるわけです…が、そこはファイルマネージャなのでいちいちファイルを編集しなくても、ファイルの窓からプログラムの窓にアイコンをドラグする等の操作でも設定を変更できます (つまり GUI から操作すると設定ファイルも対応して書き換えてくれるわけです)。

3.6 ウィンドマネージャ

ここまでで、窓の中をどういうふうにするかはそれぞれのプログラム次第だということは納得できたことと思いますが、では窓の枠の形や操作方法などはどうなのでしょうか? X では窓の管理を行なう特別なクライアントを**ウィンドウマネージャ**と呼んでいます。ウィンドウマネージャの仕事は、

[6] 筆者はよく使うファイルのありかは覚えてしまうし、いちいちマウスなどでファイルを指定するよりキーボードから打ち込んだ方が速くて楽なのですが、そのあたりは個人差や好みの差が大きいでしょう。

窓の位置を変更したり窓をアイコンにしたりといった窓の操作を利用者に行なわせることです。上で「特別」と書いたのは、ウィンドウマネージャは一時には 1 つしか動かせないという制限があるからです (それぞれの窓に同時に 2 種類の窓枠を持たせるというわけには行きませんから)。

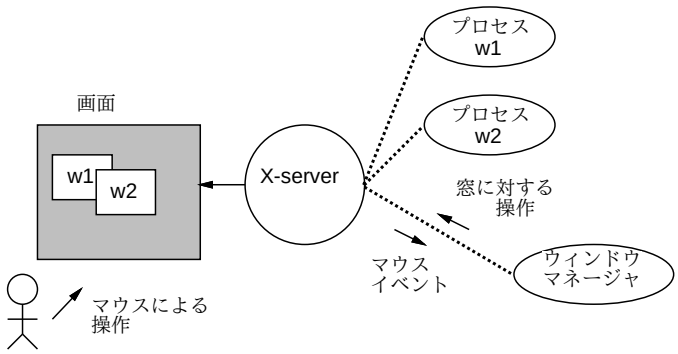

図 12: X におけるウィンドウマネージャの位置付け

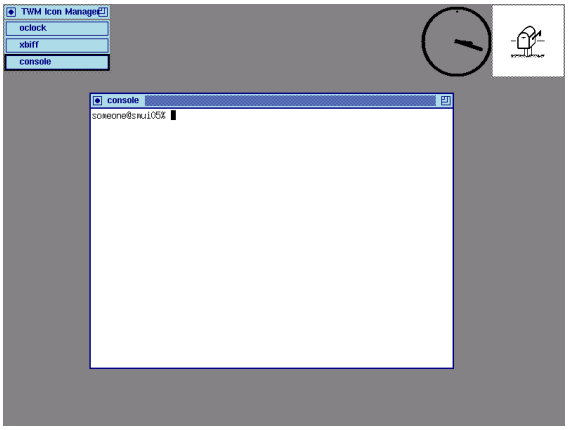

図 13: 簡潔なウィンドウマネージャTwm

たとえば窓にタイトルバー (窓の名前を記した部分) がついているのも、実はウィンドウマネージャの機能の一部です。ウィンドウマネージャは、利用者がマウスで窓を操作したりメニューを出したりといった操作を行なうと、その情報を X サーバから教えてもらい、それに呼応して窓を動かしたりアイコン化します (図 12)。[7]その他、画面の窓以外の背景部分にアイコンを表示させたり、その操作に応答したり、背景上のメニューを表示させたりするのもウィンドウマネージャの仕事です。

実は、このようにウィンドウマネージャが普通のプロセスである、というのは X の特徴の一つです。X 以前のウィンドウシステムでは、窓を動かしたりするのはウィンドウシステムそのものの機能として組み込まれていて、そのやり方を変更するのは不可能でした。これに対し、X ではウィンドウマネージャを取り替えると窓の操作のスタイルが変化します。たとえば、普段の画面は図 13 のような感じですが、`twm` を終了させてみると図 14 のようになります (この状態では窓を動かしたり重なりを変更したりすることは一切できません)。

ここで **blackbox** というウィンドウマネージャを動かしてみましょう。すると、画面の様子が図 15 のように変化します。見た目が違うだけでなく、このウィンドウマネージャは複数の「ワークスペース」(仮想画面) を持つことができます。

- 背景の中ボタンメニューで「New Workspace」を選ぶとワークスペースができる。

[7]より正確に言えば、窓を動かしたりアイコンにしたりするよう X サーバに依頼します。

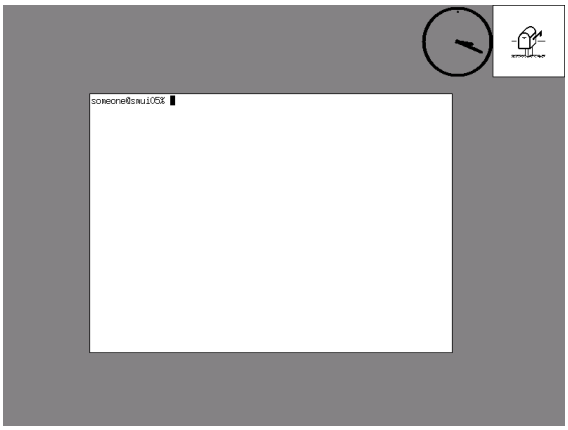

図 14: ウィンドウマネージャをなくすと…

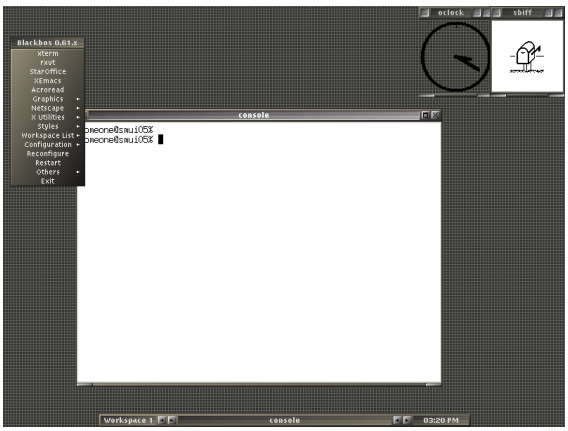

図 15: blackbox ウィンドウマネージャ

- どこかの窓のタイトルで右ボタンメニュの「Send To...」を選ぶと窓を「移送」できる。
- 画面の一番下のバーで左右矢印をつつくとワークスペースを切り替えることができる。

このような仮想画面が沢山あると嬉しいと思う人は、たとえば先の`.xinitrc`の中の「`twm`」を「`blackbox`」に変更すると、最初からこちらが使われるようになわけです。

もっと過激な例として **fvwm95** というウィンドウマネージャを見てみましょう (図 16)。これは、Windows の「そっくりさん」という洒落から始まったけれど、これが使いやすいと思って実用にしている人もそれなりにいるというウィンドウマネージャです。`fvwm95` にもワークスペース (`fvwm95` の用語では「デスクトップ」) の機能がついています。また、予め用意されたアプリケーションがクリック 1 つで起動できるボタンバーという領域が表示させられますが、これは前節で取り上げた `fwm` のアプリケーション窓のようなものがウィンドウマネージャに組み込まれていると考えればよいでしょう。

3.7 デスクトップ環境 OPTION

ウィンドウマネージャやファイルマネージャで窓の管理やファイルの管理がそれぞれ固有のスタイルで行えることはお分かり頂けたと思います。そこで、これらのツールを「まとめて」同じスタイルで用意することで、X の環境全体を統一した形で扱えるようにしよう、というのが**デスクトップ環境**です。その代表的なものとして **GNOME** や **KDE** などがあります。これらを使えばコマンドを打ち込まずに Unix を利用することもできます (筆者の趣味ではありませんが…)。

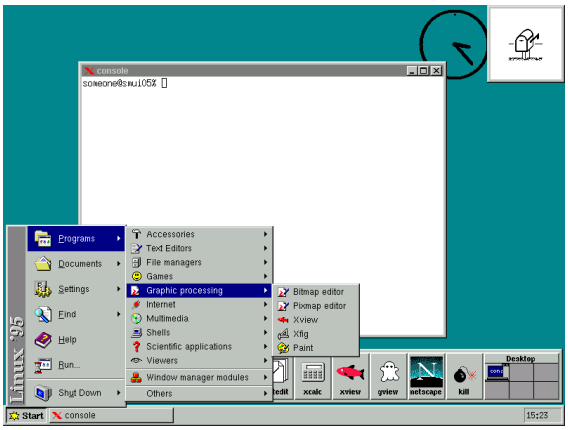

図 16: fvwm95 ウィンドウマネージャ

とりあえず、ここでは小規模な **XFCE** というデスクトップ環境 (図 17) を体験して見ましょう。XFCE のツールは別の場所に入れてあるので、次のように実行パスを追加してから動かします。[8]

```
% export PATH=/usr/local/xfce/bin:$PATH  ← XFCE ツールをパスに
% startxfce4
```

これにより、xfce デスクトップが立ち上がります。Windows とはかなり違いますが、まあ Windows のようにマウス操作だけでほとんどの操作が行わせられます。

ここに入っていないプログラムを起動できるようにしたい場合は、パネルの端で右ボタンクリックして「add new item」を選び、Launcher を選択した状態で Add ボタンを押し、command のところにそのプログラムを起動する Unix コマンド (「`moz` とか「`emacs`」とか「`kterm`」とか) を入れればパネルに入り、以後はそのアイコンを選択することでプログラムが起動できます。

しかしこうしてみると、見た目をかっこよくしたとしても、結局やっていることはコマンドを起動しているだけで、それなら Unix コマンドを覚えた方がずっと応用が効いて便利だと筆者は思うわけです…

3.8 twm の設定ファイル OPTION

また地道な話に戻って、いちばんシンプルな **twm** を題材に、その設定ファイルのようすを見てみましょう。`twm` のふるまいはホームディレクトリにある設定ファイル `.twmrc` によって変更できます。ごく単純な `.twmrc` の例を以下に示します。

```
NoTitle { "xbiff" "xclock" "oclock" "xeyes" }
IconDirectory "/usr/local/X11R6/include/X11/bitmaps"
UnKnownIcon "terminal"
ShowIconManager
IconRegion "400x400-0-0" South East 100 100
RandomPlacement
Color {                          ←ここから各部分の色の設定
  BorderColor "Purple"
  MenuForeground "Purple"
  MenuBackground "PeachPuff"
  TitleForeground "Purple"
  TitleBackground "PeachPuff"
  IconManagerForeground "Purple"
  IconManagerBackground "PeachPuff"
}                                ←ここまで
```

[8] X が起動していない状態でも使えます。その場合は自動的に X を立ち上げてから XFCE デスクトップが動きます。

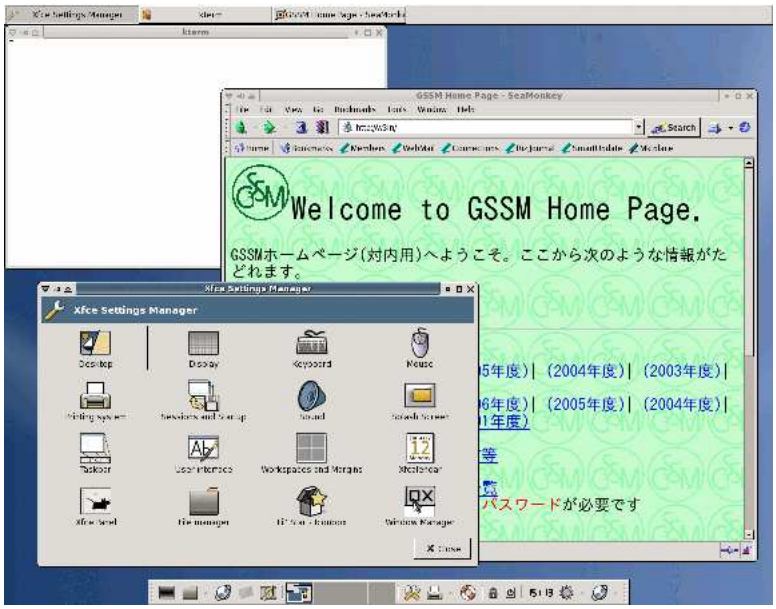

図 17: xfce デスクトップ

```
Button1 =       : title : f.move  ←タイトルで左=窓移動
Button2 =       : title : f.move  ←タイトルで中=窓移動
Button3 =       : title : f.move  ←タイトルで右=窓移動
Button1 =       : icon : f.iconify  ←アイコンで左=アイコン解除
Button2 =       : icon : f.iconify  ←アイコンで左=アイコン解除
Button3 =       : icon : f.iconify  ←アイコンで左=アイコン解除
Button1 =       : root : f.menu "menu-l" ←背景で左=メニュー 1
Button2 =       : root : f.menu "menu-r" ←背景で中=メニュー 2
Button3 =       : root : f.menu "menu-r" ←背景で右=メニュー 3
"r"     = s c   : all : f.raise         ← Shift+Control+R で raise
"l"     = s c   : all : f.lower         ← Shift+Control+L で lower
"i"     = s c   : all : f.iconify       ← Shift+Control+I で icon
"n"     = s c   : all : f.downiconmgr   ← Shift+Control+N で「次」
"p"     = s c   : all : f.upiconmgr     ← Shift+Control+N で「前」
menu "menu-l" {              ←ここからメニュー 1
"Window Control" f.title     ←メニューのタイトル
"Move"           f.move      ←移動
"Resize"         f.resize    ←サイズ変更
"Iconify"        f.iconify   ←アイコンに
"Focus"          f.focus     ←フォーカス
"Unfocus"        f.unfocus   ←フォーカス解除
"Raise"          f.raise     ←前面に
"Lower"          f.lower     ←背面に
"Refresh"        f.refresh   ←画面描き直し
"Delete"         f.delete    ←消す
"Destroy"        f.destroy   ←強制終了
"TWM Control"    f.menu "menu-t"  ←サブメニュー
}                            ←ここまで
menu "menu-r" {              ←ここからメニュー 2
"Create Windows" f.title    ←メニューのタイトル
"Local"   !"kterm -n `hostname` -T `hostname` -e bash &"
"Smp"     !"xon smp PATH=$PATH notty kterm -n smp -T smp -e bash &"
"Smm"     !"xon smm PATH=$PATH notty kterm -n smm -T smm -e bash &"
"Utogw"   !"xon utogw PATH=$PATH notty kterm -n utogw -T utogw -e bash &"
"Emacs"          !"emacs &"       ← Emacs 起動
"Netscape4"      !"ns"            ← Netscape 4 起動
"Mozilla"        !"moz"           ← Mozilla 起動
```

```
"TWM Control"    f.menu "menu-t"  ←サブメニュー
}                                 ←ここまで
menu "menu-t" {              ←ここからサブメニュー
"Twm Control"   f.title      ←メニューのタイトル
"Source .twmrc" f.twmrc      ←.twmrc を読む
"twm Version"   f.version    ←バージョン表示
"Exit twm"      f.quit       ← twm を終わらせる
}                            ←ここまで
```

これを編集して (たとえば色の指定を変えて)、背景メニューの「Restart」を選ぶと、設定ファイルが読み直されて設定が変化します。

一番よくあるのはメニューに各種プログラムを起動するための項目を増やすことかと思いますが、メニューを経ない動作を増やすことも自由です。たとえば、上の例では窓のタイトルバーでどのマウスボタンを押しても窓を移動する操作 (f.move) になっています。ここで

```
Button3 =        : title : f.move
```

のところを

```
Button3 =        : title : f.raise
```

とすると、タイトルバーで右ボタン (ボタン番号 3) を押すと、移動する代わりにその窓が一番前に出て来るようになります。また、現在は背景でシフトキーやコントロールキーを押しながらマウスボタンを押しても何も起きないませんが

```
Button1 = s : root : !"xcalc &"
```

という行を追加しておくと、背景でシフトキーを押しながら左ボタンを押すと電卓が現われるようになります。あまり「おまじない」みたいな設定を増やすと覚えるのが大変ですが、よく使うものはこのような形にしておくと素早く操作できるという利点があります。

XFCE や GNOME などのデスクトップ環境でもこのようなカスタマイズは原理的には可能ですが、ツール群の操作性を統一するために制約を設けている場合もありますし、変更できるとしても多数のツールがあるため GUI 経由でできる以外の変更は複雑でよく分からなかったりします。このような意味でも、X を「Unix らしく」使うにはシンプルなウィンドウマネージャをシンプルに使うのがよいと思うのですがどうでしょうか。

4 Web 上のユーザインタフェース

4.1 Web アプリケーションの原理

WWW の大きな特徴として、Web 文書 (ブラウザの表示内容) の中に「入力機能」を含めることができる点が挙げられます。これにより、さまざまな Web サイトを表示するだけで、その Web サイトが用意したユーザインタフェースを持ち、その Web サイトがサービスする「アプリケーション」(**Web アプリケーション**) を利用できます。

Web 以前には、サービスやそのためのユーザインタフェースをユーザに提供するには、ユーザが使うマシン 1 台ずつに専用のプログラムを設置する必要があった[9]ことを考えれば、これは画期的な進歩だと言えます。ユーザに「この URI を開いてね」と言えばそれだけで済むわけですから…

では、Web アプリケーションはどのような原理で動作しているのでしょう? ここではもっとも基本的なモデルについて説明します。まず、HTML の中にはこのような「アプリケーションによる入力」を可能にするための要素が用意されています。そのような要素は **form 要素**と GUI 部品の要素

[9]文字インタフェースや、単機能の画面端末インタフェースであればこの限りではありませんでしたが、それは GUI の時代には魅力的とは言えません。

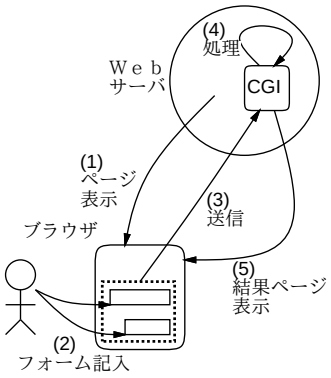

図 18: Web アプリケーションの動作

に分かれています。GUI 部品要素は、入力欄やボタンなど、ユーザがデータを入力したり操作するための機能を提供します。

form 要素の中に GUI 部品要素を入れておき、form の**提出** (submit) 操作を行うことで、form 要素が指定する提出先 URI に、GUI 部品の値が送信されます。そのため、提出先 URI は単なる HTML ではなく、**CGI プログラム**として動作する URI を指定するのが普通です。このプログラムが送られて来たデータを処理し、その結果に応じた出力を行うと、その出力 (通常 HTML) がブラウザに表示されます (図 18)。これを繰り返すことで、図 1 にあるようなユーザとの対話が行えるわけです。

HTML が提供する GUI 部品の種類はごく限られたものですが、その色や大きさや配置などは (CSS やその他の HTML 機能を通じて) かなり自由に調整できますし、部品以外の要素を使って説明や飾りを入れることもできます。これによって、かなり多様なユーザインタフェースを組み立てることができるわけです。

なお、基本的なモデルは上で示した通りですが、HTML と一緒に**クライアント側スクリプト**と呼ばれるプログラム (通常は **JavaScript** 言語がこのために使われます) を含めることで、送信→処理→結果返送を経ない (もっと短時間で応答する) 処理が行えます (もちろんこの場合は、サーバ上のデータを参照するような処理はできません)。

さらに現在では **Ajax**(Asynchronous JavaScript with XML の略とされています) と呼ばれる技術を使うことで、フォームの送信を経ないで JavaScript 内だけでサーバと通信して処理を進める形態の Web アプリケーションも普及してきています。

以下では手軽に体験できるものとして、HTML による GUI 部品と、それを用いた JavaScript プログラムのインタフェースについて取り上げて行きます。

4.2 HTML のフォームと GUI 部品

前節で述べたように、HTML でユーザインタフェースを構築する場合には form 要素と GUI 部品を利用します。まず form 要素から説明しましょう。

- `<form name="`名前`" method="`手法`" action="`*URI*`"> ... </form>` — フォームはサーバにデータを送る「かたまり」を表し、その中に含まれる GUI 部品の値がまとめて送られることになる。`name` はページ内スクリプトからフォームを参照する時に使う。`method` としてはデータが URI にくっついて見える「`"get"`」と別途送られる「`"post"`」のいずれかを指定するが、指定を省略すると「`"get"`」を指定したものとみなす。`action` では、送信先 URI(データを処理するサーバ上の CGI プログラムのありか) を指定する。今回は実際の送信は行わないが、何も指定しないとまずい (HTML として正しくない) ので、「このページと同じ」を表す「`action="#"`」を指定しておく。

form 要素の内側には、さまざまな入力部品 (HTML 用語ではコントロールと言います) を入れます。入力部品は通常のテキストや HTML 要素と混ぜて入れられるので、通常の HTML の機能を使って説明文やラベルをつけたり、見やすく配置することができます。

入力部品は以下で説明するようにさまざまな種別がありますが、すべてに共通するのは name 属性 (名前) と value 属性 (値) です。これらは、サーバ側にフォームが送られる時に対になって送られます。たとえば「name="age" value="30"」という属性だったら、「age=30」という対が送信されるわけです。なお、値は部品によってはユーザが入力した文字列になります。

以下に主要な入力部品とその属性指定を説明します (name と value は上記の通りなので特に注記することがない場合は説明していません。また☆はすぐ後の例題に出て来るものです)。

☆ <button [name="名前"] [value="値"]>...</button> — 送信ボタン。要素の内側部分がボタンの表示内容になる。ボタンが押されるとフォームのデータがサーバに送信される。送信ボタンは複数あってよいが、送信のために押されたボタンの名前と値が (指定されている場合) サーバに送られる。

☆ <input type="text" name="名前" [size="長さ"] [value="値"]> — テキスト入力欄。送信時に欄に入っている文字列が値として送られる。size で欄の幅 (文字数単位) を指定できる。value で最初に入っている文字列を指定可能。

- <input type="password" name="名前" [size="長さ"] [value="値"]> — 上と同じだが、打ち込んだものが見られないように「*」で表示される。ただし、データそのものは暗号化されるわけではないので、本当に盗まれるとまずい情報は暗号化通信を使ったページから送信すること。
- <textarea [rows="行数"] [cols="文字数"]> ... </textarea> — 複数行入力欄。機能は複数行入れられること以外は上と同じ。要素の内側部分が最初の表示内容になる。rows と cols で大きさを指定できる。
- <input type="checkbox" name="名前"> — チェックボックス。画面上でチェックを ON/OFF でき、送信時にチェックされているものだけが「"on"」という文字列を送信します。
- <input type="radio" name="名前" value="値" [checked]> — ラジオボタン。同じ名前のものが複数あってよく、その中でどれか 1 つだけが ON になる。最初に ON であって欲しいものがあれば、それ 1 つだけに checked を指定する。送信時には ON になっているものの値が送信される。
- <input type="hidden" name="名前" value="値"> — 特別な部品で、画面に表示されない (従ってユーザが値を変更することもない)。常に指定された名前と値を送信する。

☆ <select name="名前">...</select> — 選択メニュー。内側には次に示す option 要素を複数入れられ、これらを項目とするメニューができる。

☆ <option [value="値"] [selected]> ... </option> — 選択メニューの 1 項目。この項目が選択されている時は選択メニューの値としてこの項目の value が送られる。ただし value を省略しているときは要素の内側の内容 (項目として画面に表示される) が送られる。最初に選択された状態であってほしい項目には selected を指定する。

では実際に、これらのいくつかを使った HTML の例を見てみましょう (図 19)。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html><head>
<title>sample</title>
<style type="text/css">
form { background: rgb(200,255,235); padding: 1em }
</style>
</head><body>
```

図 19: フォーム部品の入ったページ

```
<h1>フォーム部品</h1>
<div><form name="data">
名字: <input type="text" name="last"><br><br>
性別: <select name="sex">
      <option value="m">♂</option>
      <option value="f" selected>♀</option></select><br><br>
<button name="btn" value="send">送信します</button>
</div></form></body></html>
```

これにより、入力欄、選択メニュー、ボタンを持つ Web ページができます。form で method を指定してないため get が用いられますが、この場合フォームを送信すると同じ HTML が表示されます。ただしそのとき、URI の末尾に名前と値の対が付加されることがブラウザの URI 表示窓で確認できます。

4.3 JavaScript によるクライアント側処理

前述のように、フォーム機能は本来はサーバ側の CGI にデータを渡して処理してもらうために用意されたものですが、JavaScript 言語を用いて手元 (ブラウザ上) だけでの処理も記述できます。

この場合、JavaScript コードは次の 2 通りの方法で指定します。

(1) HTML の head 要素内に (style 要素などと同列に)**script 要素**を「`<script type="text/javascript"> ... </script>`」のように記述し、その内側に JavaScript コードを記述する。この部分では常に実行する初期設定 (変数定義、関数定義) を書くのが普通。変数定義や関数定義は次のような形になる:

```
var 変数名 = 値;  --- 変数定義
funciton 関数名 () {  動作… } --- 関数定義
```

(2) GUI 部品要素やその他の要素にユーザが操作を行った時に実行する操作 (**イベントハンドラ**と呼ぶ) を、**onclick**(クリック時)、**onchange**(変更時)、**onmouseover**(マウスポインタが乗った時)、**onmouseout**(マウスポインタが外れた時)、**onmousedown**(マウスボタンを押し下げた時)、**onmouseup**(マウスボタンを離した時)、**onfocus**(フォーカスが入った時)、**onblur**(フォーカスが外れた時) のいずれかの属性で指定する。たとえば「`<button onclick="func()"> ... </button>`」でボタンクリック時に関数 func() を呼び出せる。イベントハンドラ部分には長い処理を書きづらいのでこのように関数を呼ぶのが普通。

JavaScript コードの「動作」部分は C 言語などと類似した構文だが、「`document.forms.` フォーム名`.elements.` 部品名」でフォーム内の部品が参照でき、さらに「`.values`」をつけることでその部

品の値、また select では「.selectedIndex」をつけることで現在の選択番号が参照できる。[10]

では簡単な例として、摂氏と華氏の温度を相互変換するページを作ってみましょう (図 20)。ここでは、温度を入力する入力欄、結果を表示する入力 (?) 欄、変換の方向 (摂氏→華氏、華氏→摂氏) を選択する選択メニュー、計算ボタンを GUI 部品として用意しています。変換の計算式は次の通です。

$$c = \frac{5}{9}(f - 32), \quad f = \frac{9}{5}c + 32$$

図 20: JavaScript を用いた温度変換

JavaScript コードと GUI 部品定義を含んだ HTML を以下に示します。計算ボタンの onclick ハンドラで JavaScript の関数 calc() を呼び出し、この中で入力欄の値を数値として取り出して計算式に従って計算し、表示欄に書き込んでいます。関数 parseFloat() は文字列を数値に変換する組み込みの関数です。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html><head>
<title>sample</title>
<style type="text/css">
form { background: rgb(200,255,235); padding: 1em }
</style>
<script type="text/javascript">
function calc() {
  var i = parseFloat(document.forms.data.elements.itemp.value);
  if(document.forms.data.elements.dir.selectedIndex == 0) {
    var o = (9/5)*i + 32;
  } else {
    var o = (5/9)*(i - 32);
  }
  document.forms.data.elements.otemp.value = o;
}
</script>
</head><body>
<h1>温度変換</h1>
<div><form name="data" onsubmit="return false">
入力温度: <input type="text" name="itemp" size="5" value="0">
換算温度: <input type="text" name="otemp" size="20"><br><br>
<select name="dir">
<option>摂氏→華氏</option><option>華氏→摂氏</option></select>
<button onclick="calc()">計算</button>
</form></div></body></html>
```

[10] radio の場合は 1 つの名前で複数のボタンが定義されるので「document.forms. フォーム名.elements. 部品名 [番号]」でその何番目かを指定し、さらに「.checked」をつけることで現在 ON か否かを表す論理値 (true/false) を参照できる。

5　まとめと演習

今回はユーザインタフェースについて概観し、ウィンドウシステムの基本機能とインタフェースの面白さ、X を題材にした GUI のさまざまな側面、そして HTML 上の GUI 部品によるインタフェースについて取り上げました。インタフェースを客観視できるようになっておくと、計算機の使われ方についてさまざまな工夫ができるようになると思います。

2-1. `xmotion.c` のプログラムを打ち込み (またはコピーし)、動かして動作を確認しなさい。また、画面クリアを一時的に動かないようにして軌跡が残るようにして同様に確認しなさい。その後、次のような「変わった効果」を 1 つ以上実現してどのように感じるか検討しなさい。

(a) 黒丸の位置がマウスポインタからずれる。ずれがポインタの位置とともに変化するようにするとなおよい。(連続的に変化する、ある場所で突然変化するなどの方法がある。)

(b) マウスポインタの縦横の動きと黒丸の縦横の動きが合わないようにする (右に動かすと左とか下とかに動くなど)。

(c) 黒丸の大きさが位置とともに変化する。(ヒント: 変数 `x`、`y` に加えて円の直径 `w` を用意し、この値に基づいてクリアや円の描画を行う。そして `w` の値を位置に応じて変化させる。)

2-2. `xfm` を動かして、ファイルの操作 (移動、削除など) やプログラム類の起動を試してみなさい。やってみて新たに分かったことを記しなさい。

2-3. `twm` を終らせて別のウィンドウマネージャを動かしてみなさい。または Xfce を動かしてみなさい。それぞれの操作方法の特色や利点/欠点を検討してみなさい。またウィンドウマネージャが取り代わると変化すること、しないことを分類整理してみなさい。

2-4. サンプルの入力部品入りページと同じものを打ち込んで (またはコピーして) 見え方を確認しなさい。さらに、サンプルに入っている以外の部品も同様に入れて見え方を確認してみなさい。

2-5. HTML+CSS で「温度変換のページ」のユーザインタフェースのみ (動かなくてよい) を自分なりに作ってみなさい。例題バージョンとは違う次のような設計はよいか悪いか検討してみなさい。

- 選択メニューの代わりにラジオボタンでどちら方向の変換を行うかを選ぶ。
- 送信ボタンが 2 つあって、そのどちらのボタンを押すかでどちら方向の変換を行うかを選ぶ。
- その他もっと画期的 (?) な工夫。

2-6. HTML+CSS+JavaScript で自分なりの「温度変換のページ」を作ってみなさい。前問のバージョンに動作をつけるのでもよいし、次のような方法も考えられる。

- 「摂氏」「華氏」の 2 つの入力欄があり、どちらかを変更すると他方に対応した値が直ちに入る。[11]

[11] この場合、onchange ハンドラまたは onkeyup ハンドラで計算動作開始を指定する。2 つの入力欄それぞれに別のハンドラを指定することで「一方を変更すると他方が更新される」ようにできる。