計算機科学'13 ＃3 — ユーザインタフェース

久野　靖*

2013.6.4

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、コンピュータとの関係はどうでしょうか? 上の定義ではコンピュータに限定されず、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「コンピュータ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、コンピュータ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」表示でき、それを「どのようにでも」動かせるという、大きな自由度があるためです。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。

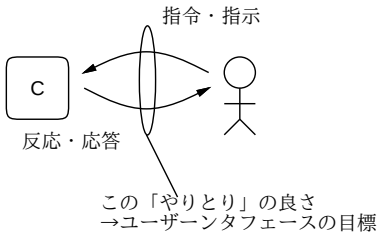

図 1: ユーザインタフェースの概念

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。[2]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)。

*経営システム科学専攻

[1]もっと「コンピュータと人間のやりとり」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。

[2]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。

(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) 初めての人に何の説明もせずに使ってもらい、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) インタフェースを実現するコードの規模、開発労力、資源量、必要なハードウェア等を比較する

(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役に立ちません。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。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、コンピュータ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文字インタフェース (**CUI**、Character User Interface) が採用されます。[3]
- 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。
- スマートフォン、タブレットのユーザインタフェース — これらはこれらはキーボードを搭載していないものが多く、搭載していても小型で打ちにくいため、画面を指や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。
- 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

[3] コンシュマー向けの製品 (ブロードバントルータ等) では、その機器に簡単な Web サーバ機能を組み込んで Web インタフェースにより設定できるようにすることが一般的になっています。

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。

演習 8-1 ユーザインタフェースの使いやすさ/使いにくさに関する以下の課題から 1 つ以上 (できれば全部) を選び、やってみなさい。

a. ソフト/アプリないしコンピュータを使用した機器について、あなたが「使いやすい」と思ったものを 1 つ選び、それがどのようなユーザインタフェースであるかを解説し、「使いやすさ」が何に由来しているのか分析・考察しなさい。
b. ソフト/アプリないしコンピュータを使用した機器について、あなたが「使いにくい」と思ったものを 1 つ選び、それがどのようなユーザインタフェースであるかを解説し、「使いにくさ」が何に由来しているのか分析・考察しなさい。
c. 「『使いにくい』部分を解消すればそれだけで『使いやすい』ものになるわけではない」という主張があります。あなたが知っているソフト/アプリないしコンピュータを使用した機器を具体的な題材として、上記の主張の適否を論じなさい。

「◎」の条件: (1) 小課題が 2 つ以上やってあり、(2) 検討対象について正確に/きちんと説明されており、(3) 分析・考察が (担当から見て) 筋道立っており適切であること。

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す。ウィンドウシステムは、GUI を構築するために必要な「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す。

ウィンドウシステムは、Windows では OS の一部として最初から組み込まれていますが、Unix 系では **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。

画面内には複数の**ウィンドウ** (窓) が存在し、その中で様々な作業を行えきます。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。

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

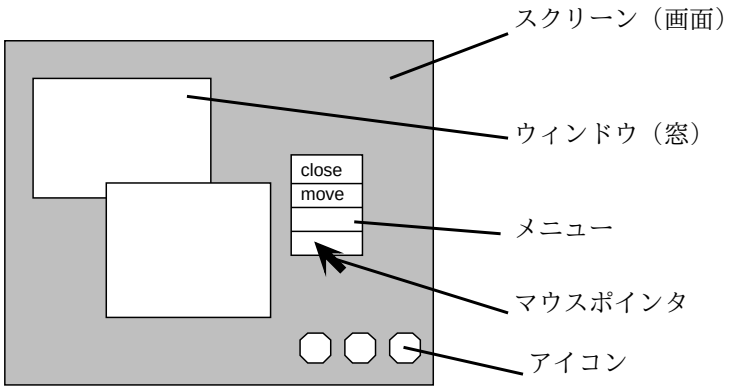

図 2: ウィンドウシステムの道具だて

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窓以外にも色々あります。代表的なのが**アイコン**で、これは小さな絵などで「何か」を表すものです。「何か」には次のような様々な流儀があります。

(a) USB メモリ、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、X では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。ただし Windows や Mac では窓に対する「お約束」が決まっていてそれを守ることがある程度強制されています。

演習 8-2 あなたが普段使っている Window System (GUI) を 1 つ選び、その GUI の全般的な操作方法について「使ったことがない人にも分かるように」説明を記述しなさい。普段必要なプログラムを動かしたり、ファイルを整理するなど「最低限必要な作業」をまずリストアップし、それらが行えるようにきちんと説明すること。

「◎」の条件: 必要な機能の説明が欠落していないこと。未定義な用語や曖昧な記述がなく、説明がきちんと整理されていて、分かりやすいこと。

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通りですが、ではそのシステムはどのような構造を持っているのでしょう? CPU とコンピュータの画面は、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 から見れば主記憶と同様に読み書きできるメモリですが、書き込んだビット列が対応する画面上の点の赤/緑/青色の輝度の明暗に対応しま

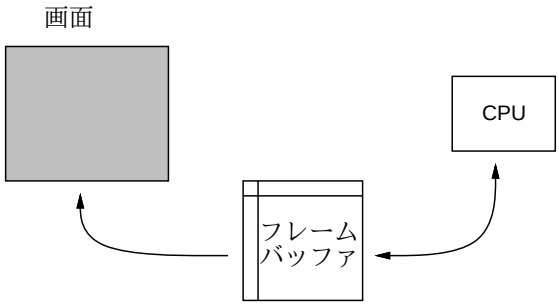

図 3: フレームバッファとビットマップ画面

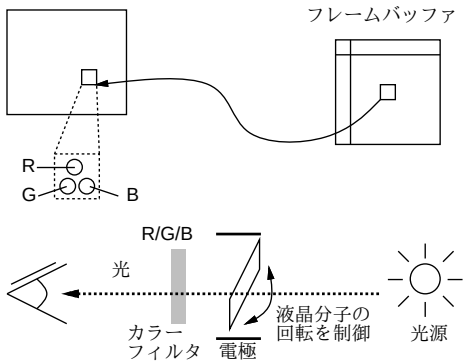

図 4: 液晶ディスプレイの原理

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み出して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点の明暗を変化させることで行なわれます (図 4)。[4]

このようなハードウェアがあったとして、その上で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? 過去においては、個々のプログラムが直接フレームバッファに書き込む方式などもありましたが (図 5)、全てのプログラムに描画機能を組み込む必要があったり、重なり部分の制御などが面倒という問題がありました。

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。この方式では、描画機能はサーバのみが持てばよく、また窓の重なり管理もサーバが一括してお

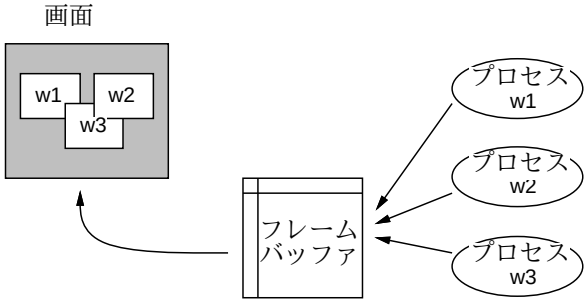

図 5: 直接描画方式のウィンドウシステム

[4]液晶は個々の点単位で光の透過率を制御するので、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。

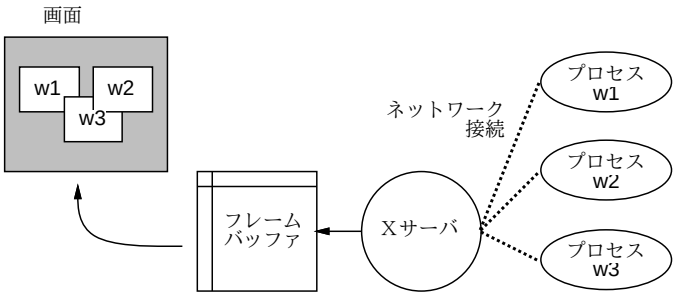

図 6: サーバ方式のウィンドウシステム

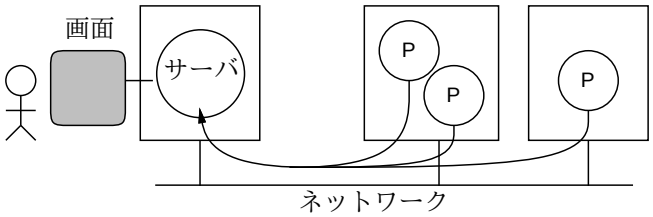

図 7: ネットワーク透過なウィンドウシステム

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。

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、必ず画面のあるコンピュータ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)。

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、負荷に応じて余裕のあるところで動作させられる。
- 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。
- 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」などの指示が中心です。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。

演習 8-3 X 上でプログラム xev を動かし、次の課題から 1 つ以上 (できれば全部) やってみなさい。

a. その窓の上でキーボード (文字キーだけでなく Fn キーや矢印キーなども) を打鍵した時、どのような情報が送られているかを観察してまとめなさい。できれば、Ctrl キーや Shift キーを「押しながら」の場合はどうかについても調べるとなおよい。[5]

b. その窓の上でマウスを動かしたりマウスボタンを操作した時、どのような情報が送られているかを観察してまとめなさい。できれば、ボタンを押した状態で入って行ったり出て行った場合はどうかについても調べるとなおよいでしょう。

[5]たとえば「Shift-Ctrl-A」と「Ctrl-Shift-A」の別ができるかどうかなども調べてみると面白いでしょう。

c. その窓の上に (もしかしたら部分的に) 他の窓を重ねたり、それをどかした時、どのような情報が送られているかを観察してまとめなさい。できれば、その窓を移動したりアイコン化したりアイコンから元に戻す時はどうかも調べてみるとなおよいでしょう。

「◎」の条件: (1) 小問 2 つ以上がやってあり、(2) いずれも「できれば」までやってあって、(3) 観察内容が系統的にまとめてあり、適切な (と担当が考える) 考察がなされていること。

2.4 イベントドリブンプログラム

前述のように、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 */
}
```

具体的な説明は次の通りです。

図 8: xmotion.c を動かしたところ

図 9: クリアしないと…

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、次のようにします (システムによって指定は違うでしょう)。

```
% 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。[6]
- ある箇所より右には丸が行かないようにする。[7]
- 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。[8]
- 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。[9]
- 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。[10]
- 右にいくほど丸が大きくなるようにする。[11]

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。

演習 8-4 `xmotion.c` のプログラムをコピーしてきて動かし、動作を確認しなさい。その後、上に挙げた 6 つの「変わった効果」から 1 つ以上 (できれば 3 つ) 選んで実現し、どのように感じるか検討しなさい。

「◎」の条件: (1) 効果 3 つ以上を実現してあり、(2) その効果がどのように感じられるかについて適切な観察・考察がなされていること。

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。

- プログラム起動時に「`-display` ホスト名`:0.0`」というオプションにより指定する。
- 環境変数 `DISPLAY` に「ホスト名`:0.0`」の形の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

[6] 描くときの X 座標を 100 増やします。
[7] X 座標がたとえば 300 より大きければ 300 にします。
[8] X 座標に X 座標の $\frac{1}{10}$ を足すなど。
[9] X 座標がたとえば 300 より大きい時だけ 100 増やすなど。
[10] 丸を描く時に使う X と Y の値を入れ替えます。
[11] 円の直径 `w` を変数として用意し、この値に基づいてクリアや円の描画を行うようにします。そして `w` の値を位置に応じて変化させればよいでしょう。

- `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

端末エミュレータというのは、画面端末[12]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。

図や画像は端末窓では扱えないので、そのためのプログラムが次々に作られました。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ファイルに書きます。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。

- 様々な種類の窓に同じフォント等を指定したい際、プログラム毎にファイルが違うのは不都合
- ネットワーク経由で多数のマシンを使う場合、ホームディレクトリも多数になってしまう

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。

[12] 画面とキーボードを備えた入出力装置で、コンピュータから文字を受け取って表示し、打ったキーを送ることができるもの。昔はこれをコンピュータに接続して CLI で作業するのが普通だった。

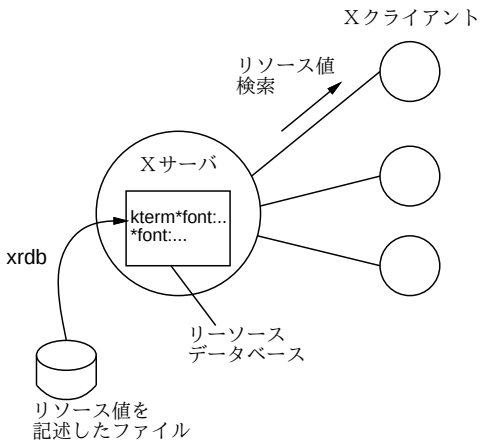

図 10: リソースデータベースの概念

```
% 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。[13]

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。

[13] 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。

- `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

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 (閾値) 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

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。

```
xsetroot -solid tan          ←背景色を設定
xset m 4 2                   ←マウス感度/閾値設定
xrdb $HOME/.x11defaults      ←リソース設定
xbiff -geom 101x101-0+0 &    ←メール通知
oclock -transp -geom 101x101-102+0 &    ←丸い時計
kinput2 &                    ←日本語入力
twm &                        ←ウィンドウマネージャ
if [ X"`tty`" = X/dev/console ]; then  ←コンソールの判定
 opt="-C"
else
 opt=""
fi      ↓コマンド窓の実行 (「&」なし)
exec kterm -geom 80x48+300+100 -T console -n console $opt -e bash
```

このように、各種設定は X の起動後に使うのと同じコマンドで行うので、これらを変更することで好みの設定を常に使用できます。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` コマンドを使います。リソース指定もここで読みませているので、そのファイルを変更すれば初期状態を変更できます。

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。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。[14]

[14]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。

3.5　ファイルマネージャ

「素の状態の」X を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。特にファイルやディレクトリについて、`ls` を使って一覧を表示して何があるか確認したり、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。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。

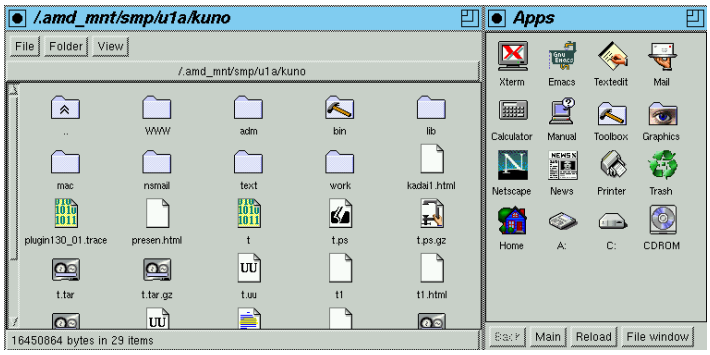

図 11: xfm

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、などの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、

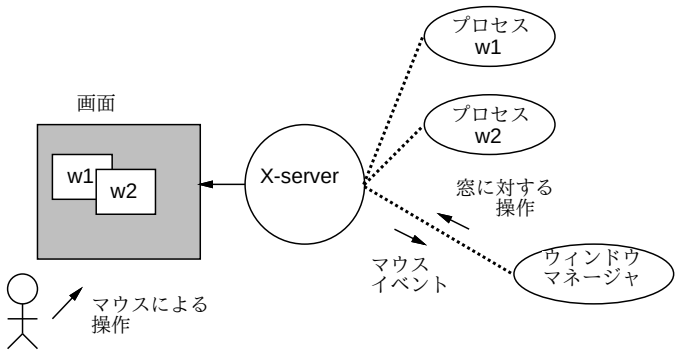

図 12: X におけるウィンドウマネージャの位置付け

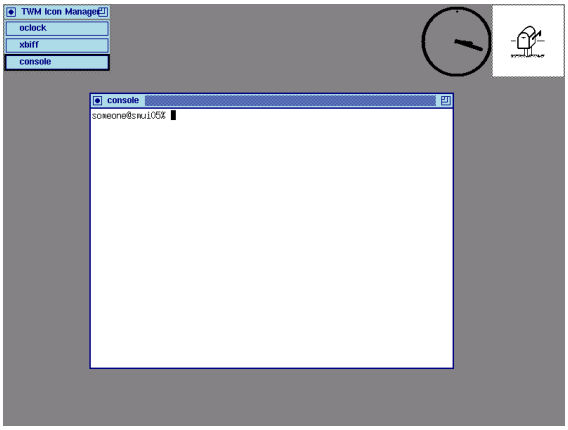

図 13: 簡潔なウィンドウマネージャTwm

窓の位置を変更したり窓をアイコンにしたり等の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)。

たとえば各窓のタイトルバーも、実はウィンドウマネージャが提供しています。ウィンドウマネージャは、利用者が窓を操作したりメニューを出す等の操作を行なうと、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)。[15]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。
- 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。

[15]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。

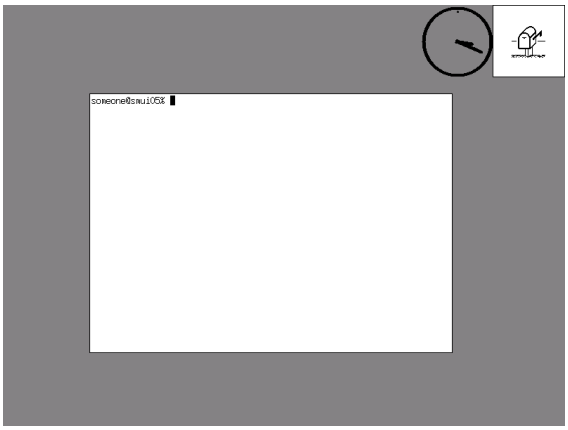

図 14: ウィンドウマネージャをなくすと…

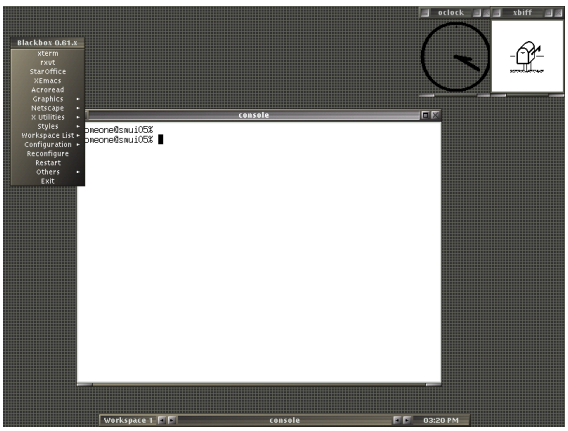

図 15: blackbox ウィンドウマネージャ

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。

Unix 系列のシステム (Linux も含む) では、このように見た目をさまざまな切り替えられるので、ウィンドウマネージャ、ファイルマネージャその他のツールを一式用意することで統一された GUI 環境を作ることができます。これを一般に**デスクトップ環境**と呼び、GNOME、KDE、Xfce など様々なものが作られ提供されています。GSSM では Unix の基本的な機能を体験して欲しいので、デスクトップ環境は使わず、基本的なウィンドウマネージャと端末窓を使っているわけです。

3.7 twm の設定ファイル

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"
```

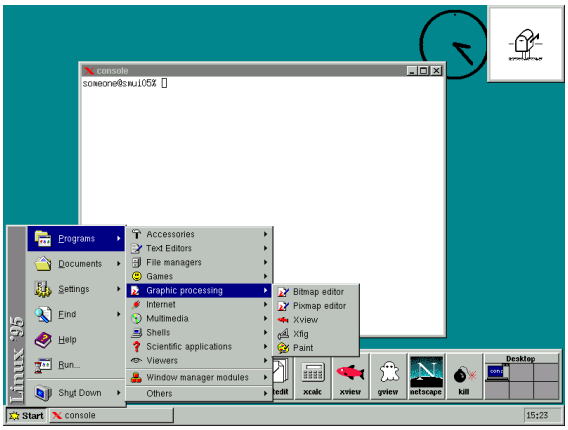

図 16: fvwm95 ウィンドウマネージャ

```
UnKnownIcon "terminal"
ShowIconManager
IconRegion "400x400-0-0" South East 100 100
RandomPlacement
TitleFont "*helv*bold*--18*"
IconManagerFont "*helv*bold*--18*"
Color {                                ←ここから各部の色の設定
  BorderColor "NavyBlue"
  MenuForeground "NavyBlue"
  MenuBackground "AliceBlue"
  TitleForeground "NavyBlue"
  TitleBackground "AliceBlue"
  IconManagerForeground "NavyBlue"
  IconManagerBackground "MAliceBlue"
}                                      ←ここまで
Button1 =       : title : f.move    ←タイトルで左=窓移動
Button2 =       : title : f.move    ←中ボタンも同じ
Button3 =       : title : f.move    ←右ボタンも同じ
Button1 =       : icon : f.iconify ←アイコンで左=アイコン解除
Button2 =       : icon : f.iconify ←中ボタンでも同じ
Button3 =       : icon : f.iconify ←右ボタンでも同じ
Button1 =       : root : f.menu "menu-l"   ←背景で左=メニューL
Button2 =       : root : f.menu "menu-r"   ←背景で中=メニューR
Button3 =       : root : f.menu "menu-r"   ←背景で右=メニューR
"r"     = s c   : all : f.raise        ← SHIFT+CONTROL+R = raise
"l"     = s c   : all : f.lower        ← SHIFT+CONTROL+L = lower
"i"     = s c   : all : f.iconify      ← SHIFT+CONTROL+I = icon
"n"     = s c   : all : f.downiconmgr ← SHIFT+CONTROL+N = 「次」
"p"     = s c   : all : f.upiconmgr    ← SHIFT+CONTROL+P = 「前」
menu "menu-l" {                    ←以下メニューL
"Window Control" f.title           ←メニューのタイトル
"Delete"         f.delete          ←窓を消去
"Move"           f.move            ←窓を移動
"Resize"         f.resize          …
"Iconify"        f.iconify
"Focus"          f.focus
"Unfocus"        f.unfocus
"Raise"          f.raise
"Lower"          f.lower
"Refresh"        f.refresh
"Destroy"        f.destroy
"TWM Control"    f.menu "menu-t"   ←サブメニューTを開く
```

```
}                                   ←ここまでメニューL
menu "menu-r" {                     ←以下メニューR
"Create Windows" f.title
"Local"          !"kterm -n ‘hostname‘ -T ‘hostname‘ -e bash &"
"Sma"            !"xon sma PATH=$PATH notty kterm -n sma -T sma -e bash &"
"Smb"            !"xon smb PATH=$PATH notty kterm -n smb -T smb -e bash &"
"Smc"            !"xon smc PATH=$PATH notty kterm -n smc -T smc -e bash &"
"Utogw"          !"xon utogw PATH=$PATH notty kterm -n utogw -T utogw -e bash &"
"Emacs"          !"emacs &"         ← Emacs
"Mozilla"        !"moz"             ← Mozilla
"Firefox"        !"firefox"
"TWM Control"    f.menu "menu-t"    ←サブメニューTを開く
}                                   ←ここまでメニューR
menu "menu-t" {                     ←サブメニューT
"Twm Control"   f.title
"Source .twmrc" f.twmrc
"twm Version"   f.version
"Exit twm"      f.quit              ← TWMを終了
}                                   ←ここまでサブメニューT
```

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。

演習 8-5 以下の演習課題から 1 つ以上 (できれば 3 つ) やってみてその結果を報告し、また分かったことの考察を記しなさい。

a. リソース設定を変更し、新たにできる窓の背景色や文字色を普通でない値に設定してみる。

b. xset でマウスの感度を調整し、自分の好みの感度を見つけてみる。[16]加えて、xsetroot で画面の背景色をさまざまな色にしてみる。

c. xfm によるファイル操作を体験してみて、コマンドによる操作や普段自分が使っている GUI による操作と比較する。

d. twm 以外のウィンドウマネージャを動かし、窓の操作のしかたやシステムの使い勝手がどう変わるか比較する。

e. twm の設定ファイル (`.twmrc`) を編集して、自分がよく使う動作をす速く起動する設定を試してみる。

[16]「`xset m 1 1`」「`xset m 20 1`」「`xset m 20 5`」など試してみましょう。

f. X の起動時設定ファイル (`.xinitrc`) を変更して、X を動かした時の設定を自分の好きなようにカスタマイズしてみる。

「◎」の条件: (1) 小課題 3 つ以上がやってあり、(2) やってみたことやその結果についてきちんと説明されていて、(3) 分かったことについてきちんと考察されていること。

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[17]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 URL を開いてね」と言えばそれだけで済むわけですから…

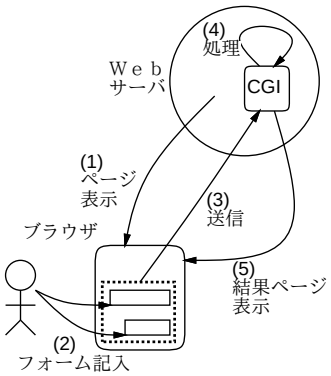

図 17: Web アプリケーションの動作

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 URL に、GUI 部品の値が送信されます。そのため、提出先 URL は単なる HTML ではなく、**CGI プログラム**として動作する URL 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 17)。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。今回はこの基本部分を体験して頂きます。

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

[17] 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。

→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 (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) と呼ばれる技術により、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。

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="`*URL*`"> ... </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。[18]

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。要素の内容がボタン内に表示される。ボタンが押されるとフォーム内容が送信される。

☆ `<input type="text" name="名前" [size="長さ"] [value="値"]>` — テキスト入力欄。送信時にその内容が送られる。`size` で欄幅 (文字数)、`value` で初期値を指定可能。

- `<input type="password" name="名前" [size="長さ"] [value="値"]>` — 上と同じだが、表示は「***…」となる。ただしデータが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で扱うこと。
- `<textarea [rows="行数"] [cols="文字数"]> ... </textarea>` — 複数行入力欄。機能は textarea と同様要素の中身が初期値になる。`rows`、`cols` で大きさを指定。
- `<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、チェックされているものだけが「`"on"`」という文字列を送信。
- `<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 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に `checked` を指定する。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。

[18] `method` としてはデータが URL にくっついて見える「`"get"`」と別途送られる「`"post"`」のいずれかを指定しますが、省略すると「`"get"`」になります。`action` では、送信先 URL(データを処理するサーバ上のプログラムのありか) を指定します。省略すると、本のページと同じ URL が使われます。

☆ `<option [value="値"] [selected]> ... </option>` — 選択メニューの項目で、選択されている時は select の値としてこの項目の `value` 値 (`value` を省略すると要素の内容) 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 `selected` を指定する。

図 18: フォーム部品の入ったページ

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 18)。

```
<!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>
<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、URL の末尾に名前と値の対が付加されることがブラウザの URL 表示窓で確認できます。

演習 8-6 「摂氏」「華氏」の相互変換 (温度換算) を行うページを作りたいものとします。HTML の GUI 部品を使って作ってみなさい (製作の前に紙でデザインをスケッチしてから始めること)。

「◎」の条件: どのような考え方でデザインしたか、どのように工夫したかが書かれていること。実際にブラウザで表示させて操作してみたときにどうだったかの結果や、分かったことの考察が書かれていること。

5 まとめ

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、コンピュータ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。