# 第7章　ウィンドウシステム

現在の計算機システムにおいては、ユーザインタフェースの操作性がとても重要である。その意味で、単なるキーボードと文字表示だけのやりとりから絵やポインティングデバイスを活用したインタフェースへの進化を可能にしたウィンドウシステムは重要な技術である。ここでは Unix の標準のウィンドウシステムである X-Window を中心に、その技術的特徴と活用の可能性を述べる。

## 7.1　ウィンドウシステムの概観

### 7.1.1　ウィンドウシステムの由来と歴史

計算機環境全般において、人間が直接計算機とやりとりする手段ないし媒体はシステムの使われ方に大きな影響を持つ。歴史的に見ると:

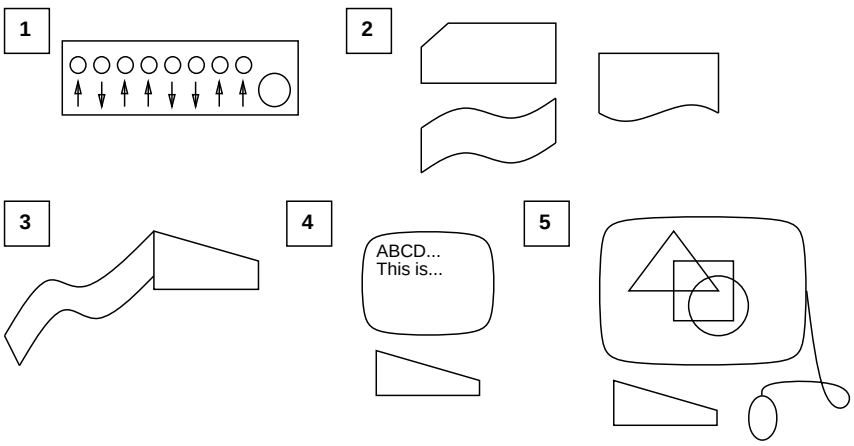


図 7.1: 計算機と人間の対話手段の変遷

- コンソールパネル:押しボタン、スイッチ、ランプなどのあつまり。主記憶とレジスタの内容を読み書きする→ビット列そのものを扱う。

- オフライン媒体。パンチカードとか、紙テープ。字の形で入出力できる、という点では大きな進歩だが、何しろ「予め別の場所で用意する」ので融通が効かない。
- テレタイプ端末。その場で対話的に計算機を使えるのでかなりな進歩だが、入力はともかく出力が遅い。行エディタによる編集。
- CRT 端末。これで画面エディタ、画面モード入出力などぐっと使いやすくなるのだが、しょせん表示できるものは文字のみ。図形端末もなくはないが、高価だし一般的でない。画面もそんなに広くないので、同時に複数の仕事を表示させておくのは現実的でない。
- ウィンドウ。字だけでなく、絵も表示に混ぜられる。字のスタイルなども変化可能。図形が自由に使える。複数の仕事を並行して表示できる。ポインティングデバイス (マウスなど) で「指さす」ことができる。

のような経過をたどってきているわけである。こうして見ると、計算機が贅沢品でなくなるとともに、計算機が利用者のために色々な作業をやって「使いやすく」してくれる、という流れがわかる。その意味ではウィンドウシステムは現在の所一番「使いやすくできる可能性を持つ」計算機との対面方法である、ということになる。

このようなウィンドウ文明を最初に開発したのは Xerox 社で、Alto というシステムが現在のような高性能ワークステーションの「はしり」とされている。その後 Xerox 社ではこれを文書作成用システム (今風にいえば DTP マシン) として、Star という商品名で売り出したがあまり成功しなかった。日本では富士ゼロックスがその日本語版 J-Star を扱っている。一方、Alto に影響を受けた Steve Jobs が Apple 社で開発したのが Lisa とその後継機種 Macintosh で、その隆盛はご存じの通り。また、PC ソフトの「王者」Microsoft が対抗して推進しているのが Windows で、Mac をかなり意識した作りになっている。

一方、Sun を始めとするワークステーションメーカでも次々にこのようなシステムを開発し自社のシステムに搭載するようになった。ただし、これらのウィンドウシステムはそれぞれまったく別個のものであった。しかし、MIT が DEC 等と協力して開発した X-Window というシステムを無料公開し、各社のシステムがこれを搭載することが相次いだ結果 (Sony NEWS もそうだ)、X が少なくとも Unix を搭載するワークステーション文明の中では「事実上の標準」の地位を獲得するに至ったわけである。というわけで、ここで皆様が使われているのも、また以下で説明するのも主として X-Window(version11, Release5 と 6) ということになる。

なお、本章で説明しているような内容を GUI(Graphical User Interface) という言葉であらわすこともある。厳密にいえば、GUI とは「絵やマウスなどを活用したユーザインタフェース」であり、ウィンドウシステムは「GUI を実現できるような機能を提供するシステム」ということになる。

### 7.1.2 ウィンドウシステムの概念

ではここで、ウィンドウシステムとはどんなものか改めて見てみよう。図 7.2 に、典型的なウィンドウシステムの「道具だて」を示す。まず、スクリー

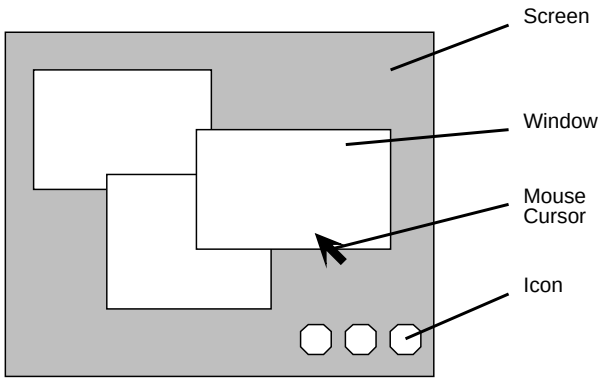


図 7.2: ウィンドウシステムの道具だて

ンというのは要するに物理的な画面全体に相当する。その中に四角い[1] 窓が複数[2] 存在し、それぞれが別々の作業に対応する。[3] そして、マウスの動きに呼応して画面上を動くマウスカーソルがあり、これを使って「どの窓」という切替をやるのが多い。[4][5]

さらに、画面上に窓のほかに小さな絵柄をもった「アイコン」なるものを置けるのが普通である。アイコンが何を表すかはシステムの流儀によってまた大幅に違う。たとえば X を含む Unix のウィンドウシステムでは、アイコンは「窓の閉じたもの」であり、窓がたくさんできた時にごちゃごちゃにならないためにあるという感じが強い。一方、Alto/Star 流ではアイコンはディレクトリやファイル (あちらの用語では文書とフォルダ) とか、プリンタ装置などの「もの」に対応している。Mac もだいたいそうであるが、加えてプログラムもファイルに入っているからアイコンが「動かす前のプログラム」である、という意識が強い。

さて、今度は「窓」の中の「作業」であるが、Unix をベースにしたシステムの場合、これまで前節 4. の CRT 端末を使って仕事をしてきた、という経緯があるので、「CRT 端末の真似をする」窓 (端末エミュレータ) を使ってその上でこれまでと同様に仕事をする、という場合が圧倒的に多い。これだと従

[1] 最近は四角くないのも可能なのが増えているが。
[2] 互いに重なりあっていいものと、重なりを許さないのの 2 通りの流儀があるが。
[3] 場合によっては別の窓の作業のための補助的な窓であることもあるが。
[4] が、窓の切替えのたびにマウスを掴むのは面倒なのでキーボードだけで窓を切替える方がいいといってそういう機能を使う人も多い。
[5] マウスで窓の切替えをやる際、「窓の領域にカーソルが入ったらそれだけでそっちに切り替わる」という流儀と「切替えたい窓にカーソルが行くだけでなく、そこでマウスボタンを押すと始めて切り替わる」という流儀とがある。

来のプログラムがすべてそのまま使えるので便利ではあるが、端末が一杯ある、というだけではあまり面白くないのも確かである。これに対して、 Star や Mac では最初からウィンドウを前提としたシステム作りを行なったので各ソフトとも絵やマウスなどウィンドウの機能を活用しているものが多く、より「使っていて楽しい」といえる。それでも最近はウィンドウを活用したプログラムもいくらかは増えつつある (といっても、Unix でのソフトの蓄積は膨大なので比率からいうと少ないわけである...)。

### 7.1.3　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だが、ではそのようなシステムはどのようにしたら作れるだろうか? CPU と計算機の画面はハードウェア的には CPU と画面は図 7.3 にあるように、フレームバッファを介してつながっている。フレームバッファというのは機能的には普通の主記憶と同様に

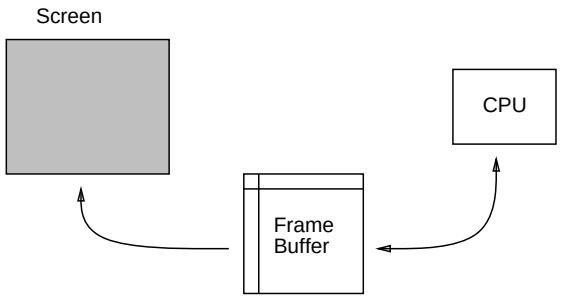


図 7.3: フレームバッファとビットマップ画面

アクセスできるのだが、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輝度の明暗に対応して現れるようになっている。これは、ビデオコントローラが CRT 画面上を操作するのに同期してフレームバッファを 1 列ずつ読みだしながら、その 0/1 に応じて電子ビームの明滅を制御することによって行なわれる (図 7.4)。

さて、こういうハードがあり、ソフトからアクセスできるとして、では前節で見たようなウィンドウシステムを作るにはどうしたらいいと思うか?

一つの方法は、窓を作り出すような各プログラムが「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」という風に各窓のプロセスに任せるやり方である。Macintosh、Windows、SunView などはすべてこの方式である。この様子を図 7.5 に示す。しかし、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るし、これを含め様々な図形出力ルーチンも各プロセスごとに持つ必要があるので大変である。

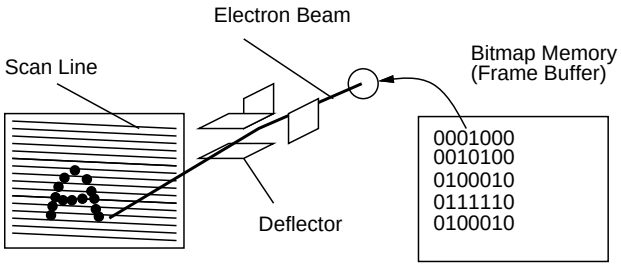


図 7.4: ビットマップディスプレイの原理

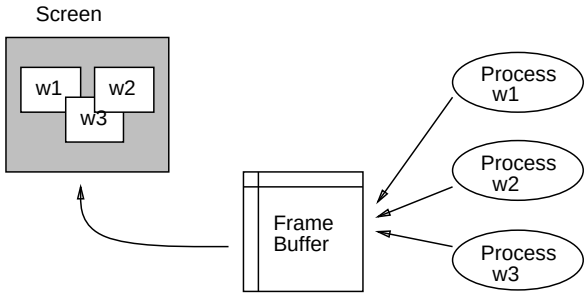


図 7.5: 直接描画方式のウィンドウシステム

もっとも、パソコンなどで共通のサブルーチンは1ヶ所に入れておけるならこのことはそれほど問題ではない。また、割り込み型 (preemptive) マルチタスクを行なわないシステムでは、各プログラムが「都合の悪い」時に別のプログラムに切り替わってしまうことはないので、競合の問題も比較的少ない。[6] ただし、マルチ CPU などのシステムではこのような簡便なやり方では済まないのは明らかである。

これに対し、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」が考案された。X-Window もこの方式を取っている。この様子を図 7.6 に示す。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えばよいので、各窓に対応するプロセスはずっと簡単である。各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」などの要求を出す。

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インタプリタを

[6] Macintosh も Windows も、複数のプログラムが並行して動くように見えるが、その切り替わりは各プログラムが「切り替わってもよいよ」という命令を実行した場所に限られる。言い替えれば、あるプログラムが暴走したらシステム全体が止まってしまうことになる。

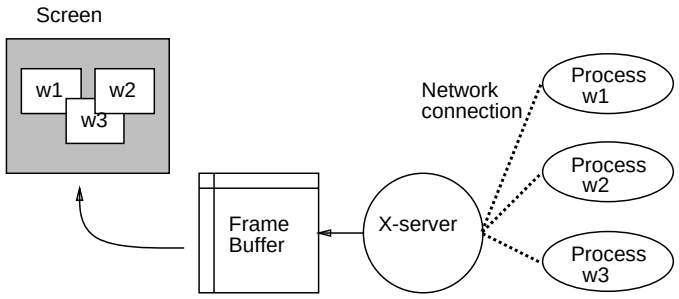


図 7.6: サーバ方式のウィンドウシステム

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) が考案され、Sun NeWS などで実現されている。しかし X-Window が Unix 文明での主流となってしまったため、この方式のウィンドウシステムは今のところマイナーである。

### 7.1.4　ネットワーク透過なウィンドウシステム

ところで、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が通信できさえすればいいので、**各プロセスはサーバと同じマシンにいなくてもいい**ことに注意されたい。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なければならないが、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る。この様子を図 7.7 に示す。

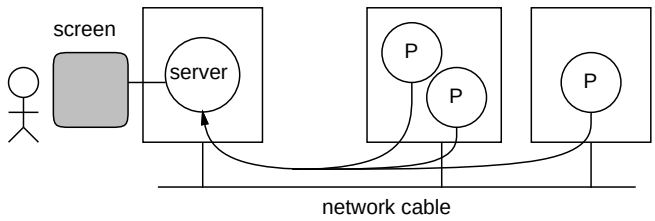


図 7.7: ネットワーク透過性

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るので、プロセスがネットワークの向うにあることは一切意識されない。このような性質をネットワーク透過性という。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挙げられる。

- 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。
- 手もとのマシンでは動かないようなプログラムを居ながらにして使うことができる。
- 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。

最後の点の極端になったものが、X 端末である。X 端末とは、ウィンドウサーバを動かすことのみを目的とした計算機で、そのためディスクなどは持たなくてもよいので安価に作れるし、Unix などを動かすオーバヘッドも避けられる。

例えば、E428 で窓側にあるマシンはすべて X 端末として運用されている。つまり、それらのマシンはすべて X サーバのみを動かすように設定してあり、その上で「窓を開いた」場合、その窓を実現するプロセスは smb/smd/smf/smg など別のマシンで動いているわけである。一方、廊下側のマシンはそうではないので、「Console」の窓はそのマシン自身で動いている。

ウィンドウシステム一般の話からだいぶ X 固有の話に近付いて来たが、次節では皆様が実際に試して見られる、X の諸側面について取り上げる。ただし、例によって「こんなことができるから面白い」というだえけでなく、「内部がどういう構造になっているから、従って外部のこういうふるまいが説明できる」という立場に立って受け止めて欲しい。

## 7.2 X-Window

### 7.2.1 クライアント

X はサーバ方式のウィンドウシステムなので窓を作るようなプログラム (これをサーバと対比してクライアントと呼ぶ) はどのマシンで動いていてもよい。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ある。では、「どのサーバに窓を作るか」はどうやって決めるのだろうか。これには 2 通り方法がある。

1. プログラムを起動する時、「`-disp` 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。
2. 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する。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ちょっと困る。そこで、通常の状態

では (一部ではあるが) 画面保護が掛かっていてその画面で login した人にしかその画面の窓が作れないようにしてある。これを外すには「xhost +」を実行する (「xhost -」で元に戻る)。

その他によく指定するオプションとしては、クライアントの窓の位置や大きさを指定するものがある。これは「-geom 幅 x 高さ+X 座標+Y 座標」なるオプションで指定できる。座標の前の符合を+の代わりに-にすると、「画面の右/下端からの距離」の意味になる。

さて、それでは具体的にはどんなクライアントがあるかも簡単に列挙する。それぞれの詳しい説明は当然マニュアルページを参照されたい。

kterm -- 端末のまねをする (端末エミュレータ)。普段使っているのはこれ。
xterm -- 同上、ただし英語のみ。
bitmap -- アイコンの絵などビットマップの絵を作成する。
xpaint -- カラーのお絵描きツール
xv -- 画像表示ツール
k2d -- MacDraw のそっくりさんのお絵描きツール。
xcalc -- 電卓。
xbiff -- メールが来ているかどうかを知らせる。
xclock, oclock -- 時計。
ico, maze, puzzle, ... -- 様々なお遊び。
xfm -- デスクトップマネージャ
xset -- サーバの様々なオプションを設定する。

これらのクライアントは普通のプロセスだから、終らせるには (終り、というボタンがついているものもあるが) そのプロセスを殺せばよい。

ところで、アイコンや背景などに自分の好みの「絵」を入れたいと思ったらビットマップファイルを作る必要があるので、bitmap だけはもう少し詳しく説明しておこう。起動方法は

```
bitmap -size 幅 x 高さ    ← 32x32 〜64x64 くらいがいい?
```

による (画面、フォント、位置なども指定したければできる)。始まると指定した大きさの「方眼紙」が出てくるので、その上でマウスの左ボタンを押すとそこが黒になり、右ボタンで白くなる。その他丸、四角、線なども描ける。OK と思ったら Write でファイルを書き出し、Quit すると終る。bitmap によって作られたファイルは、アイコンやカーソルなどに使用することができる。試してみるなら

```
xsetroot -bitmap ビットマップファイル名
```

などとやってみよう。

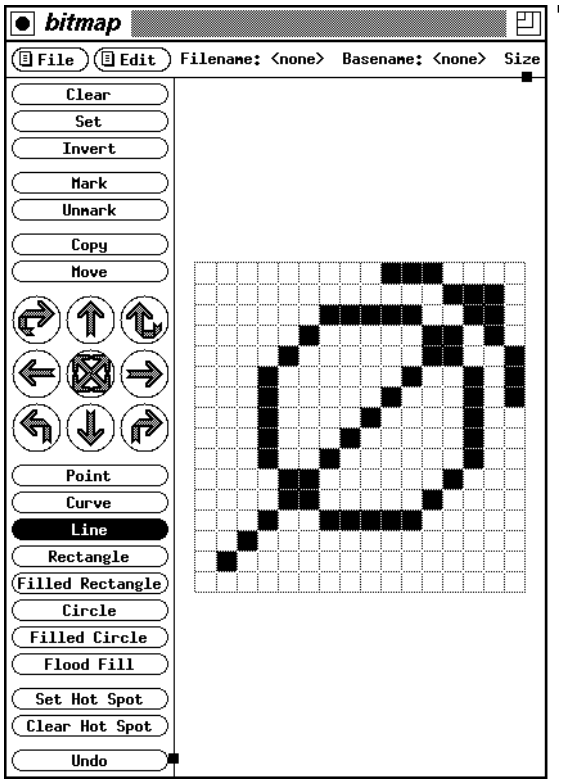


図 7.8: bitmap の窓

### 7.2.2　窓情報の収集と整理

X-Window ではディレクトリの階層構造と同様に、窓のなかに子供の窓、孫の窓、…を置くことができる。実は普段「背景」と思っている部分が (ルートディレクトリと同様) 一番「根本」のルートウィンドウで、普段見ている窓はすべてこれの子孫である。ある窓にどんな子供があるかを見るには「`xwininfo -id 窓 ID -children`」で調べることができる。なお、窓 ID というのは X サーバの中で管理のためにそれぞれの窓につける固有番号 (プロセス ID のようなもの) である。

窓 ID が判らない場合には何も指定しないで `xwininfo` を動かすとカーソルが十字形になって「どこかの窓の上でクリックせよ」と言われるので、調べたい窓の上でクリックするとその窓の情報を教えてくれる。

```
% xwininfo
xwininfo: Please select the window about which you
          would like information by clicking the
          mouse in that window.
(ここで背景のところをクリックした)
xwininfo: Window id: 0x25 (the root window) (has no name)
  Absolute upper-left X:  0
  Absolute upper-left Y:  0
  Relative upper-left X:  0
  Relative upper-left Y:  0
  Width: 1152
  Height: 900
  Depth: 1
  Visual Class: StaticGray
  Border width: 0
  Class: InputOutput
  Colormap: 0x21 (installed)
  Bit Gravity State: ForgetGravity
  Window Gravity State: NorthWestGravity
  Backing Store State: NotUseful
  Save Under State: no
  Map State: IsViewable
  Override Redirect State: no
  Corners:  +0+0  -0+0  -0-0  +0-0
  -geometry 1152x900+0+0
```

すなわち、ルートウィンドウの ID は 0x25 だとわかる。ではルートウィンドウの子供を調べてみよう。(いくつあると思いますか?)

```
% xwininfo: Window id: 0x25 (the root window) (has no name)
  Root window id: 0x25 (the root window) (has no name)
  Parent window id: 0x0 (none)
     12 children:
     0x800021 (has no name): ()  144x217+581+539  +581+539
     0xc00038 (has no name): ()  564x697+300+100  +300+100
     0xc00033 (has no name): ()  100x100+1048+0  +1048+0
     0xc00032 (has no name): ()  78x17+0+0  +0+0
     0xc00031 (has no name): ()  5x5+0+0  +0+0
     0xc00025 (has no name): ()  150x69+0+0  +0+0
     0xc0001d (has no name): ()  99x72+0+0  +0+0
     0xc0001c (has no name): ()  99x72+0+0  +0+0
     0xc0001b (has no name): ()  105x198+0+0  +0+0
     0xc0001a (has no name): ()  105x198+0+0  +0+0
     0xc00019 (has no name): ()  101x144+0+0  +0+0
     0xc00018 (has no name): ()  101x144+0+0  +0+0
%
```

けっこう多数あるでしょう? 現在表示されていないメニューやアイコンに対応する窓も存在はしているので、これだけになるのである。どれがどの窓かを知るのには、窓の内容をデータとして取り込むプログラム `xwd`(X Window Dump) と、そのデータを別の窓で表示するプログラム `xwud`(X Window UnDump) を組み合わせて次のようにすればよい。

```
% xwd -id 0xc00033 | xwud
```

この結果図 7.9 のようなものが見えた。つまり、0xc00033 の窓は時計だったわけである。

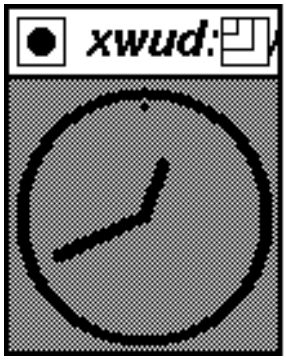


図 7.9: xwd + xwud の表示例

なお、`xwd` も ID を指定しなければ十字カーソルで取り込む窓を指定でき

る。取り込んだものをファイルに保存するには出力リダイレクトすればよい。また XWD 形式を PS(PostScript) に変換するフィルタ「`xpr -dev ps`」を使えば PS プリンタに印刷することもできる。つまり、

```
% xpr -dev ps ｘｗｄファイル名 | lpr -Plw
```

などのようにすればよい。

### 7.2.3　フォント

X に限らず、ウィンドウシステムでは画面上に様々なフォントが表示できる。X では現在どんなフォントが利用可能かを表示したり、あるフォントがどんな形の字かを見てみるのに次の指令が使える。

```
xlsfonts                -- 利用可能なフォントの一覧を表示
xfd -fn フォント指定  -- 指定したフォントの文字を表示して見せる
xfontsel  -- 下記の各パラメタをその場で変更してフォントを選べる
```

例えば、xlsfont の出力の最初の方は次のようになるだろう。

```
% xlsfonts
-adobe-courier-bold-i-normal--0-0-0-0-m-0-iso8859-1
-adobe-courier-bold-o-normal--0-0-100-100-m-0-iso8859-1
-adobe-courier-bold-o-normal--0-0-75-75-m-0-iso8859-1
-adobe-courier-bold-o-normal--10-100-75-75-m-60-iso8859-1
-adobe-courier-bold-o-normal--11-80-100-100-m-60-iso8859-1
-adobe-courier-bold-o-normal--12-120-75-75-m-70-iso8859-1
....
```

一見わけが分からなそうだが…X のフォント名は一般に次の形をしている。

-作者-書体名-太さ-傾き-種別--ドット-ポイント-Xres-Yres-間隔-幅-字種

ただし、

- 作者: メーカ名など
- 書体名: Helvetica、Times、など文字の形を示す
- 太さ: medium(普通)、bold(太い) など
- 傾き: r(立体—普通)、o(oblique、傾いている) など
- 種別: だいたいは normal
- ドット: 画面上の点何個幅か
- ポイント: ポイント数× 10 の値

- Xres, Yres: X 方向、Y 方向の画面解像度
- 間隔: c(constant、各文字の幅が一定)、p(proportional、可変)
- 幅: 1 文字あたりの平均幅
- 字種: iso8859-1(普通のアスキー用)、jisx0208.1983-0(漢字)、jisx0201.1976(JIS8 ビット—いわゆる半角)、gb2312.1980-0(中文)、ksc5601.1987-0(ハングル)、などがある。

`xfontsel` を使えばこれらの各部分をメニュー形式で選択して調べることができる。実際にフォント指定をするときに、この長い文字列を全部書くのでは大変なので、任意の部分を「*」と書いて省略できる。例えば

```
xfd -fn '*-helvetica-bold-r-*--12-*'
```

とすれば、helvetica 書体のボールド立体 12 ドットフォントが指定できる。図 7.10 に xfd で書体を表示させている様子を示す。

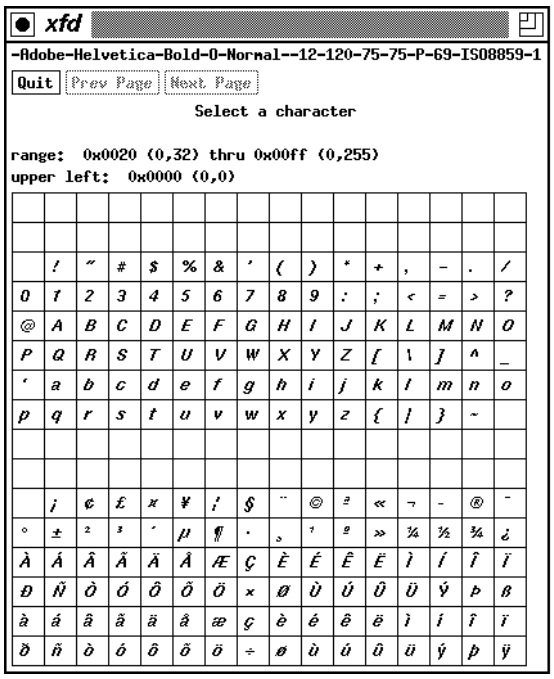


図 7.10: xfd の表示例

一般にどのクライアントでもそれが表示に使用するフォントを「`-fn` フォント指定」オプションで指定することができる。また kterm では合わせて漢字用フォントは「`-fk` フォント指定」、JIS8 用フォントは「`-fr` フォント指定」で指定できる。

### 7.2.4　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の . で始まるファイルに入れておく。たとえば.cshrc などがそうである。しかし、X-Window 関連の場合はこの方法はあまりよくない。というのは、

- 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だろうから、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である。
- さらに、ネットワーク透過なのでホームディレクトリも複数あったりするかも知れない。

などの問題があるからである。

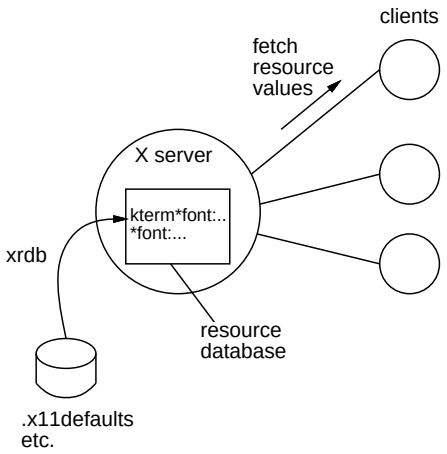


図 7.11: リソースデータベースの概念

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だろう? 答は、「X サーバの中」である (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だから)。そこで、X サーバの中にリソースデータベース (図 7.11 というものを用意し、オプションの標準値をこの中に設定しておく。その内容を見たり設定したりするには xrdb 指令を使う。例えば次の通りである。

```
% 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
%
```

このように、リソース指定は「クライアント名*リソース名: 指定値」の形をしているのが普通であるが、いきなり*から始まってもよい。その場合には、「すべてのクライアントについて」の意味になるので、フォントや色を統一的に指定するのに便利である。

リソースデータベースに値を設定するには「xrdb ファイル」による。ファイルを指定しなければ標準入力から指定を読む。既にある設定に追加する場合にはさらに-m オプションを指定する。例えば次のようにしてキーボードから新しいリソース指定を追加できる。

```
% xrdb -m
xterm*font:  *-helvetica-bold-r-*--25-*
^D
% xrdb -q
kterm*font:          a14
kterm*romanKanaFont: r14
kterm*kanjiFont:     k14
xterm*font:  *-helvetica-bold-r-*--25-*
% xterm &     ← xterm はリソースからフォント指定を読む
```

このようにした後で xterm を動かすと helvatica-bold-25pt フォントで表示する xterm の窓ができる。

### 7.2.5 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だったが、この他にサーバの状態を変更する機能がある。

```
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 -cursor カーサ マスク   -- カーソルの形状を変更
xsetroot -bitmap 背景   -- 背景を指定したビットマップに変更
xsetroot -gray          -- 背景を灰色にする
xsetroot -def           -- 設定を標準値に戻す
```

### 7.2.6 起動

我々のところでは X-Window が最初から動いているマシンと動いていないマシンとがある。動いていないマシンの場合、X を起動するのは xinit コマンドによる。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の内容をシェルスクリプトとして実行する。典型的な.xinitrc の内容を次に示す。

```
% cat .xinitrc      ↓まず PATH の指定
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 -transp -geom 100x100-0+0 &    ←丸い時計
twm &                       ←ウィンドウマネージャ
if [ X"‘tty‘" = X/dev/console ]; then
 opt="-C"
else
 opt=""
fi      ← -C (コンソール) オプションの有無を判定
exec kterm -geom 80x48+300+100 -T console -n console $opt -e bash
%     ↑ コンソールの窓
```

というわけで、X が始まる時に窓を多数準備するなどもっと色々やりたければここを修正すればできるわけである。また、xinit はこのシェルスクリプトが修了すると X サーバを停止して終るので、この例だと最後に書かれている端末窓が修了すると X も終って裸コンソールの状態に戻ることになる。例えば twm が終った時に X も終るようにしたければ twm を最後に&なしで書けばよい。

常に X が動いたままのマシン (特に X 端末など) では、xdm(X display manager) というプログラムが常時動いていて login を扱う。各ユーザが login すると、xdm はそのユーザのホームディレクトリにあるシェルスクリプトファイル.Xsession を実行する。これがない場合には代わりにシステム管理者が用意したものが使われるが、我々のところでは単に.xinitrcを呼び出すようなものを用意している。というわけで、xdm から login しても xinit で起動しても同じ画面から始まることになる。

### 7.2.7　ウィンドマネージャ

ところで、上で出てきた twm というのはウィンドウマネージャと呼ばれる特別なクライアントである。ウィンドウマネージャの仕事は、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ユーザが行なうのを助けることである。例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ある。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

かしたりアイコンにしたりする (正確には窓を動かしたりアイコンにしたりするよう X サーバに依頼する)。この状況を図 7.12 に示す。実は、このよう

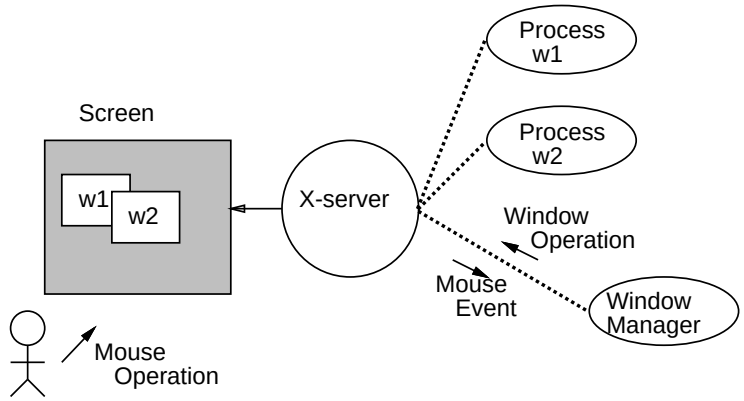


図 7.12: X におけるウィンドウマネージャの位置付け

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 の特徴の一つである。旧来のウィンドウシステム (Star、Macintosh など) 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あった。一方、X ではウィンドウマネージャを取り買えると窓の操作のスタイルが変化する。

また、ウィンドウマネージャの動作自体もその初期設定ファイルを変更するといろいろと変化させられる。ここでは X で一番おおく使われていて、我々のところでも標準として使っている `twm` の場合について見てみる。`twm` のふるまいはホームディレクトリにあるファイル `.twmrc` によって変更できる。典型的な `.twmrc` の内容を以下に示す。

```
NoTitle { "xbiff" "xclock" "oclock" "xeyes" }
          ↑ タイトルバーを出さないクライアントを指定しておく
IconDirectory "/usr/local/X11R6/include/X11/bitmaps"
          ↑ アイコンが指定されてなければこのディレクトリの
UnKnownIcon "terminal"  ← このビットマップファイルを使う
ShowIconManager         ← 窓の一覧を表示
IconRegion "400x400-0-0" South East 100 100 ←アイコンを画面の下左にならべる
RandomPlacement         ← 新しい窓はとりあえずの位置に適当に配置
Button1 =       : title : f.move
Button2 =       : title : f.move
Button3 =       : title : f.move ← どのボタンもタイトルバー
```

では窓の移動

```
    Button1 =       : icon : f.iconify
    Button2 =       : icon : f.iconify
    Button3 =       : icon : f.iconify ← アイコン上でクリックすれば開く
    Button1 =       : root : f.menu "menu-l"
    Button2 =       : root : f.menu "menu-c"
    Button3 =       : root : f.menu "menu-r" ← 背景ではどれかのメニューを出す
    menu "menu-l" {
    "Window Control" f.title
    "Move"           f.move
    "Resize"         f.resize
    "Iconify"        f.iconify
    "Raise"          f.raise
    "Lower"          f.lower
    "Refresh"        f.refresh
    "Unfocus"        f.unfocus
    }                  ← 左ボタンメニューは窓の移動などをおこなう
    menu "menu-c" {
    "Create Windows" f.title
    "Local"          !"kterm -n `hostname` -T `hostname` -e bash &"
    "Mule"           !"mule &"
    "Smb"            !"xon smb PATH=$PATH notty kterm -n smb -T smb -e bash &"
    "Smd"            !"xon smd PATH=$PATH notty kterm -n smd -T smd -e bash &"
    "Smf"            !"xon smf PATH=$PATH notty kterm -n smf -T smf -e bash &"
    "Smg"            !"xon smg PATH=$PATH notty kterm -n smg -T smg -e bash &"
    "Smb Mule"       !"xon smb PATH=$PATH notty mule &"
    "Smd Mule"       !"xon smd PATH=$PATH notty mule &"
    "Smf Mule"       !"xon smf PATH=$PATH notty mule &"
    "Smg Mule"       !"xon smg PATH=$PATH notty mule &"
    }                  ← 中央ボタンメニューは各種の窓を作る
    menu "menu-r" {
    "Twm Control"   f.title
    "Source .twmrc" f.twmrc
    "twm Version"   f.version
    "Exit twm"      f.quit
    }                ← 右ボタンメニューは TWM の制御
```

### 7.2.8 デスクトップマネージャ

さて、ここまで見て来て「しかし、Unix+X-Window は Mac や Windows とはやっぱり違うぞ」と思っている人はいませんか? つまり、Mac などではファイルのコピーやプログラムの実行もマウス操作 1 つでできるじゃないか、というわけである。しかし Unix にできない事などない (?!?)。`xfm` というアプリケーションを見ていただこう。

```
% xfm.install  ←最初に 1 回だけやる
% xfm &
```

こうすると、図 7.13 と図 7.14 の 2 つの窓が画面に現われる。 ここで、図 7.13

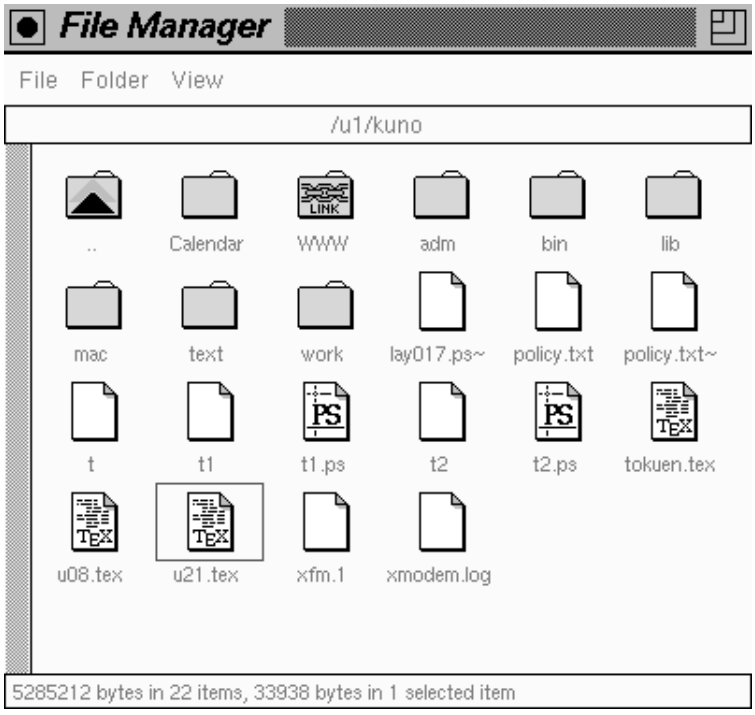


図 7.13: xfm のファイル窓

の窓ではディレクトリ (フォルダ) をダブルクリックして開いたり上に戻ったり、ファイルを移動したりできる。また、ファイル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 (.tex なら mule など) が動き出してファイルを処理できる。さらに、ファイルを「掴んで引っ張り」図 7.14 の窓の適当なプログラムに重ねて離すと、それでもプログラムが起動できる。

使い方はまあどうでもよくて、言いたいことはつまり、Mac や Windows で「一番の大元」だと思っていたプログラムは「単なるクライアントの 1 つ」


図 7.14: xfm のアプリケーション窓

だということである。なお、このようなプログラムのことを「デスクトップマネージャ」などと呼ぶ。納得いきましたか?

ところで、どのファイルはどんな絵で表示されるかとか、どの上でダブルクリックしたらどのプログラムが動くかとか、プログラムの窓にはどんなものが現われているかとかはどうやって決まるのだろう? それは、

```
.xfm/xfmrc
.xfm/xfm-apps
```

という 2 つの設定ファイルにだいたい書いてある。これを調整すれば、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。こういう調整が勝手にできるのは Mac や Windows とは違うでしょう?

## 7.3 ウィンドウソフトウェアの構造

### 7.3.1 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、と説明したが、具体的にはどんなやりとりをしているのだろう? 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ある。これらを X の用語ではサーバへの要求 (リクエスト) と呼ぶ。

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だろう? 具体的には次のようなものがある。

- 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ぶ。どんなイベントがあるかを見てみるには「xev」というプログラムを使ってみるとよい。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を標準出力に表示してくれる。

```
% 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
    %
```

### 7.3.2 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くる。そこで、クライアントプログラムの流れは次のようになる。

```
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; ☆
}
```

すなわち、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るのに対し、X のプログラムではイベントを読むところは 1ヶ所だけしかなく、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しなければならない。このようなプログラム構造を「イベントドリブン」という。

お話だけだとつまらないので、簡単な例を示そう。このプログラムはまず窓を 1 個作り、その窓が表示されると黒丸を 1 個描く。その後、マウスで窓の中をクリックするたびにそこにも黒丸を描き、どれかのキーを押すと終了するというものである。

```
/* t71.c --- very simple X client */

#include <X11/Xlib.h>

main() {
1   Display *disp = XOpenDisplay(NULL);
    Screen *scr = DefaultScreenOfDisplay(disp);
2   Window root = DefaultRootWindow(disp);
    unsigned long black = BlackPixelOfScreen(scr);
    unsigned long white = WhitePixelOfScreen(scr);
3   Window mw = XCreateSimpleWindow(disp,root,100,100,400,200,2,black,white);
4   XSelectInput(disp, mw, ButtonPressMask|KeyPressMask|ExposureMask);
5   XMapWindow(disp, mw);
    while(1) {
      XEvent ev;
6     XNextEvent(disp, &ev);
      if(ev.type == KeyPress)
7       exit(0);
      else if(ev.type == Expose)
8       draw(disp, mw, 20, 20);
      else if(ev.type == ButtonPress)
9       draw(disp, mw, ev.xbutton.x, ev.xbutton.y);
    }
  }
10
```

```
    draw(Display *disp, Window mw, int x, int y) {
      GC dgc = DefaultGC(disp, 0);
 11   XFillArc(disp, mw, dgc, x, y, 40, 40, 0, 360*64);
    }
```

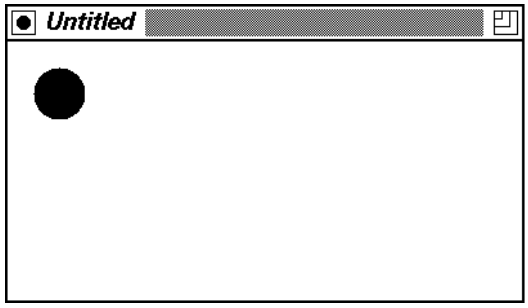


図 7.15: t71.c を動かしたところ

具体的な説明は次の通り。

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 400 × 200 の窓を作る。縁の幅は 2 ドット、絵や字は黒、地の色は白。
4. マウスボタン押下げ、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、(20,20) 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. 内容は、標準の描画方法で、(x,y) を起点に幅高さとも 40 ドットの円を描いて中を塗りつぶす。

これを動かすには、オプションがいくつか必要。

```
% gcc -I/usr/local/X11R6/include t71.c -lX11 -lsocket -lnsl
% a.out                                          ↑ここ以降は smb
```

のみ必要

すると、画面に窓が現われ、その中に黒丸が 1 つ見える (図 7.15)。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 個描くからである。

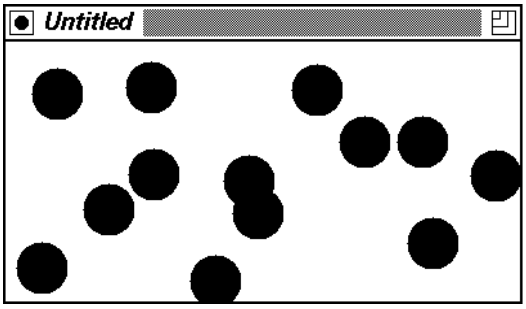


図 7.16: t71.c を動かしたところ (2)

次に、この窓の上でマウスボタンを押すとそのたびにその位置に黒丸が増えていく (図 7.16)。

ところが、別の窓を重ねて…(図 7.17)… どけると、隠されていたところの

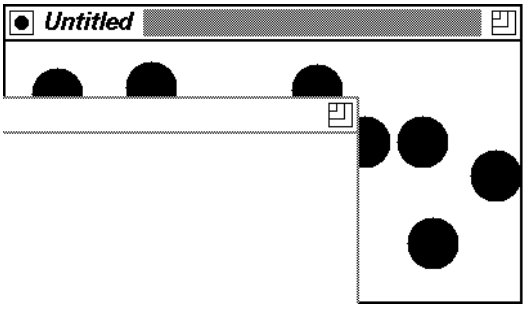


図 7.17: t71.c を動かしたところ (3)

黒丸は消えてしまう (図 7.18)! しかし、最初の黒丸だけはちゃんと復活している。これはつまり、プログラムが窓に描いたものは直接フレームバッファに描かれる他にはどこにも保存されていないので、その上に別の窓が来たりして壊されるとその内容は失われてしまうためである。そこで、邪魔ものがどくと「また見えるようになったよ」と Exposure イベントが送られて来る。このプログラムは Exposure が来た時最初の黒丸しか描かないので、残りの黒丸は消えたままになってしまうわけである。(そんな面倒をなくすために窓の内容をフレームバッファとは別に保存しておいた方がいいと思いますか?)

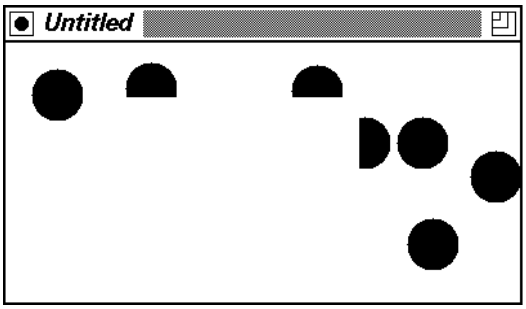


図 7.18: t71.c を動かしたところ (4)

### 7.3.3　オブジェクト指向、ウィジェット

さて、上のような方式で原理的にはあらゆるウィンドウもののプログラムを作ることができる。例えば、まず窓が見えるようになったら各種ボタンや入力欄などをそれらしく描き、マウスボタンが押されたらその位置によって、ボタンの上ならそのボタンが押された時の動作をおこない、入力欄なら入力中のテキストのカーソルを表示し、… のようにすればいい。

しかし、どう考えてもこれはうんざりするほど大変そうである。そもそも、GUI もののプログラムは「ボタン」「スクロールバー」「スライダー」「入力欄」「テキスト表示」などいくつかの決まった「部品」を適切に配置して作られるものなので、それをいちいち裸から作っていたら何人プログラマがいてもお話にならない。

そこで、これらの「部品」をプログラム上でも「部品」ないし「モジュール」として用意し、同じ形態の部品をいくつでも窓の中に配置できるようにする。それには、オブジェクト指向プログラミング技法 (「ものの種類」を「クラス」と呼ばれるモジュールで定義し、具体的な「もの」ないし「インスタンス」を沢山生成できる) を適用するとすっきりまとまる (図 7.19。

GUI プログラミングの世界ではこれを「Window の Object」という意味でウィジェット (Widget) と呼ぶ。各ウィジェットは「自分が画面上のどの範囲を占めている」かをデータの一部として覚えているので、メインプログラムはマウスイベントを受け取ったら画面上にある全インスタンスに対して順に「このマウスイベントはあなたの受け持ちか?」と聞いていけばいいだけである。

実は順番に聞くという動作も決まり切っているのでライブラリとして用意する。そこで、メインプログラムの主な仕事は結局、最初に初期設定をして、必要な部品を順番に配置していくだけ、ということになってくる。

なお、X の場合は C 言語の上で前述のようなオブジェクト指向機能を実現するライブラリ「X ツールキット」(Xt) が標準て提供され、その上で動く様々な部品群が流通している。代表的なものとして、X に標準付属かつ無料

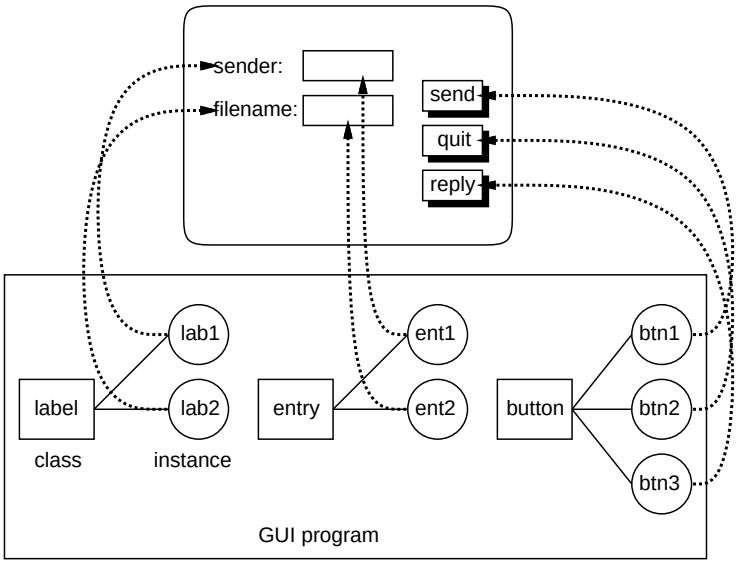


図 7.19: GUI 部品とウィジェット

の Athena Widget Set、有料だが多く使われている Motif Widget Set などがある。

なお、X-Window そのものは特定の操作方法に肩入れしないので、その上ではどのような見え方や操作方法 (Look and Feel) のアプリケーションを作ることもできる。しかし、各開発者がバラバラにそれをデザインすると不統一でえらく使いにくいことになりそうである。たとえば Apple では Macintosh 上のソフトのための Look and Feel 統一指針を昔から公表しており、Mac のソフトはほとんどこれに準拠しているので統一的に操作できる。 X がいつまでも各社バラバラではこれに負けてしまうのは明らかである。

そこで、Unix を使っている各社の組織で話し合いの上、Look and Feel の統一基準を作り、それに従おうという活動を行なった。Motif は OSF(HP、IBM、DEC などが中心となって始めた組織) によって作られた基準であり、現在ではこれが事実上の標準となりつつある。 (少し前まで Sun などは OpenLook という別の標準を提唱していたが、最近敗北を認めて Motif に鞍がえしている。)

### 7.3.4 GUI のための小さな言語

Widget Set を使ったとしても、C 言語による GUI プログラムの開発は大変である。特に、X や Xt、Widget Set に含まれる膨大なライブラリルーチン群とその使い方を熟知するだけでも多くのプログラマには負担である。しかし、上のような事情から考えると、どの部品をどこに配置する、という指定だけならなにも C のような低レベルの言語で書かなくても、たとえばコマ

ンド制御のためのシェルスクリプト、ファイル処理のための awk のように簡単でわかりやすい言語を使ったらよさそうである。

そのような考えのもとに、UCB の John Ousterhout が提唱したのが tcl/tk と呼ばれる言語であり、GUI のための小さな言語としては最も広く使われている。ここではその感じだけでも味わっていただこう。まず、この例の概観は図 7.20 のようなものである。見ればわかるように、送信先、話題、送信す

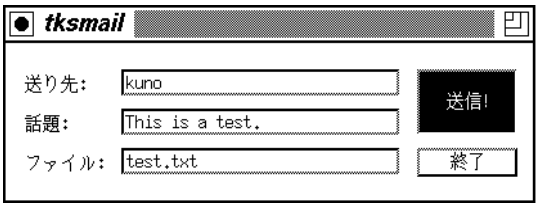


図 7.20: tk で書いたメール送信ツール

べきファイルを入力して「送信」ボタンを押すとメールを送ってくれる。

ではそのリスティングを示そう:

```
 1  #!/usr/local/bin/wish -f
 2  . configure -geom 410x120
 3  label .lab1 -text "送り先:"
 4  place .lab1 -x 10 -y 20
 5  entry .ent1 -relief sunken -width 30
 6  place .ent1 -x 90 -y 20
    label .lab2 -text "話題:"
    place .lab2 -x 10 -y 50
    entry .ent2 -relief sunken -width 30
    place .ent2 -x 90 -y 50
    label .lab3 -text "ファイル:"
    place .lab3 -x 10 -y 80
    entry .ent3 -relief sunken -width 30
    place .ent3 -x 90 -y 80
 7  button .btn1 -relief raised -width 10 -height 3 -text "送信!" -command {
 8    set dest [.ent1 get]
      set subj [.ent2 get]
      set file [.ent3 get]
 9    exec mhmail -s $subj $dest <$file
    }
10  place .btn1 -x 320 -y 20
11  button .btn2 -relief raised -width 10 -height 1 -text "終了" -command {
```

```
    exit 0
}
place .btn2 -x 320 -y 80
```

そのあらましは次の通り。

1. 解釈実行のためのインタプリタ指定を 1 行目に書く。インタプリタは wish というコマンド。
2. まず窓全体の大きさを 410x210 にする。
3. 「送り先」と表示するためのラベルを `.lab1` の名前で作る。
4. そのラベルを (10,20) の位置に置く。
5. 送り先を入力する入力欄を作る。くぼんだ形で、幅 30 文字。
6. 入力欄を (90,20) の位置に置く。(以下しばらく同様)
7. 送信ボタンを作る。出っぱった形で、幅 10 文字高さ 3 行、中に送信と書いてある。そしてボタンが押されたら以下の動作を実行…
8. 変数 (シェル変数のようなものと思えばよい)dest に、.ent1 のに入っている現在の値を取り出したものを入れる。次の 2 行も同様。
9. exec は Unix のコマンドを実行する。もちろん、elm を主題と送り先指定で実行し、指定したファイルを入力リダイレクトで与える。
10. ボタンも place を忘れないように。
11. 終了ボタンも同様だが、動作は単に exit で終わるだけ。

このファイルを例えば tksmail のような名前で用意しておき、実行可能にしておく。これをコマンドとして実行すると、図 7.20 の窓が出て来るわけである。何なら `.xinitrc` に入れておいてもいい。

このように、C よりも高いレベルで操作できる小さな言語を活用することで GUI プログラムが容易に作れるようになる。速度が問題になるような場合は、その部分だけ C で書いて tcl/tk に組み込むこともできるようになっている。

### 7.3.5 GUI ビルダ

GUI プログラムの繁雑さを減らすもう 1 つの方法は、GUI ビルダを使うことである。GUI ビルダとは、要するに各種の Widget の配置や構成を対話的にやってくれるためのツールである。多くの GUI ビルダは Motif ツールキットを呼び出す C 言語などのコードを生成するが、ここでは tcl/tk 用の GUI ビルダ xf を見てみよう。

まず xf を起動すると、図 7.21 のようなパネルと真っ白の窓が出て来る。ここで、パネルを様々に操作して真っ白の窓に部品を組み込み、最終的には動

くアプリケーションにするのが GUI ビルダの仕事である。では先の tksmail と同じものを作るということで、早速やってみよう。

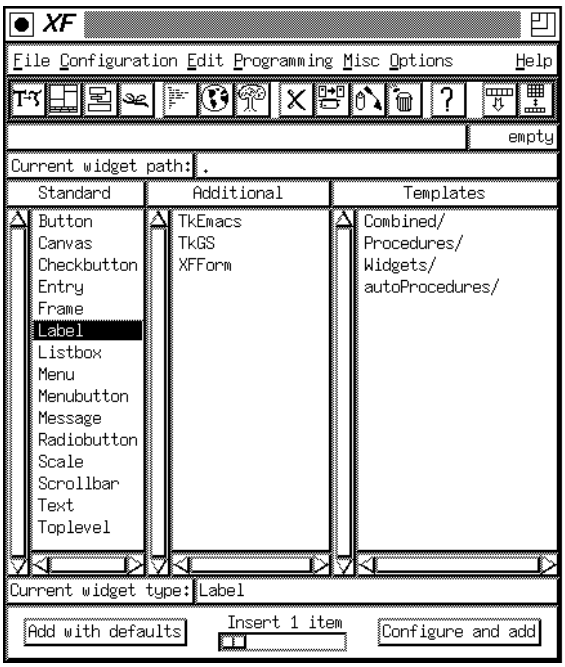


図 7.21: tcl/tk 用の GUI ビルダ xf の主パネル

- current widget path: という欄に「.」が表示されている。これがアプリケーション全体の窓を意味する。
- ここで configuration メニューの parameter(small) を選んでサブパネルを出す。
- その中の set window size のチェックボックスを選んで、その上の width と height のスライダを動かし幅を 410 くらい、高さを 120 くらいにする。できたら OK を選んでサブパネルを閉じる。
- こんどはパネルの 3 列あるリストボックスの左で Label を選んで反転させ、その状態で右下の configure and add を押し別のサブパネルを出す。
- ここで Label: 欄に「Send to:」と入れ (kinput2 を起動してあれば^\で漢字を入れてもよい)、Relief 項目は Flat を選び、OK するとラベルが窓の中央にできる。名前は適当に選ばれる。
- Configureation メニューから Layout サブパネルを出し、User Placer を選ぶ。このパネルはずっと出しておく。
- Meta キーを押しながら send to: のラベルをクリックすると、「最初の

Widget だけど?」というダイアローグボックスが出るが、その選択肢から place and keep parent size を選ぶ。

- 再度 Meta を押しながら send to: のラベルをドラグして、適当な位置に動かす。
- current widget path: の欄をつついて一覧を出し、今作った label を選ぶ。続いてアイコン列の「はさみ」を押して、その 2 つ右の「のり」を 3 回押す。これでラベルが 3 個にコピーできた。
- 3 個のラベルは同じ位置に重なっているので、Meta を押しながらドラグして別の位置に移すことを 2 回やる。
- curerent widghte path: の欄で 2 番目のラベルを選んで、アイコン列の左端を押してサブパネルを出し、Label: 欄を Subject: にする。同様に 3 番目のラベルの Label: 欄は File name: にする。
- current widget path: を一旦「.」に戻してから、こんどはリストボックスの Entry を選び、configure and add を押してパネルを出し、名前は ent1、Relief は sunken、幅は 30 くらいとして OK する。できたら Meta+ドラグで適当な位置に動かす。また、その中をクリックして Del キーで中の文字列をからっぽにする。
- current widget path: で今の entry を選び、はさみ 1 回、のり 3 回でコピーして適切な場所に置く。
- また current widget path: を「.」に戻し、今度はリストボックスから Button を選び、Label: 欄を SEND!にし、Command: 欄に前項で書いたのと同じ set 命令 3 行、exec 命令 1 行を入れる。ただし、.ent1 などの名前は自動的につけたので前と違っているから適宜とりかえること。OK するとボタンができる。
- ボタンの位置変更は前と同様。大きさも Meta をおしながらボタンの右や下の辺をつかんで変更できる。
- また current widget path: を「.」に戻し、今度は終了ボタンを作る。Label: 欄に QUIT、Command: 欄に exit 0 を入れ、OK して大きさと位置を調整。
- これで一応完成したので、そのままの状態で動かしてみることができる。OK だと思ったら File メニューの Save as を選んでファイルに保存すればよい。

というわけで、確かにマウス操作などだけでできるのだが、しかし直接 tck/tk をエディタで編集して作る方がだいぶ速くないだろうか…一般に、GUI ビルダは一見便利なのだが、できたコードが複雑で手を入れにくく、柔軟性に乏しいなどの批判もよく聞く。この辺は、まだどちらがいいという評価は固まっていない分野でもある。

## 7.4　演習

**7-1.** `xlfonts`、`xfd`、`xfontsel` でどんなフォントがあるか調べ、結果をまとめてみよ。

**7-2.** `xrdb -m` で xterm のフォントリソースを指定し、様々なフォントの xterm を動かしてみよ。kterm ではどうか? とくに kterm の場合、漢字と ASCII、JIS8 のフォントの大きさが不釣合だとどうなるか? 気に入ったものがあれば`$HOME/.x11default` を変更して常時そのフォントを使うようにしてもよい。

**7-3.** 適当な窓で「`echo $DISPLAY`」を実行して、現在使っている画面を確認せよ。次に、隣の人と協力して、まずそれぞれ「`xhost +`」を実行して画面保護を外し、それから「`xclock -disp` 相手のディスプレイ」などを実行すると相手の画面で動く自分のクライアントが作れることを確認せよ。

**7-4.** ビットマップエディタ `bitmap` を動かし、簡単な絵を描いてみよ。それをファイルに保存し、`xsetroot -bitmap` ファイル名で背景に設定してみよ。

**7-5.** `xwininfo` でルートウィンドウ、その子供の窓、孫の窓、…を適当なところまで調べ、さらにどの窓 id が画面の上ではどの窓かを分かる範囲で調べよ。窓の中身は「`xwd -id` 窓 ID `|` `xwud`」を実行するとその窓の中身が `xwd` で見える。ただし、問題の窓が画面上に表示されていないと失敗する (が実害は別にない)。

**7-6.** `.xinitrc` を変更して、X 起動直後の画面の様子をカスタマイズしてみよ。例えば端末窓を最初からもっと多く用意するとか、電卓やカレンダーを用意するなど。コマンドの最後の「`&`」を忘れないように。

**7-7.** `.twmrc` を変更してカスタマイズしてみよ。たとえば窓のタイトルでボタンを押した時、全部のボタンで同じ動作にする代わりに、中ボタンならその窓を一番上に出す (f.raise)、右ボタンなら一番下に押し込む (f.lower) などやってみる。また、メニューの項目も増やしてみよ (utogw の窓も開けるようにするなど)。

**7-8.** `.xfm/xfmrc` と `.xfm/xfm-apps` を直して、例えば`.txt` で終わるファイルは`.tex` で終わるファイルと同じように表示されるようにしてみよ。また、電卓アイコンをつつくと Mosaic が現われるようにしてみよ (ヒント: `xcalc` の代わりに `xm` コマンドを実行すればいい)。

**7-9.** t71.c のプログラムを打ち込み、動かせ。また、`XNextEvent` の呼び出しの直後に「`sleep(1);`」(1 秒時間待ち) を入れると挙動はどのように変わるか? まず予想し、次に実行して試せ。

**7-10.** t71.c のプログラムを直して、窓の上に別の窓を重ねてから取り除けても全部の黒丸が復活するように直してみよ。[7]

**7-11.** tksmail を打ち込んで動かせ。さらにこれをちょっと直して、ニュース投稿用のツールに仕立ててみよ。

**7-12.** xf で tksmail のようなもの、または好きなものを組み立ててみて (途中まででもいい)、感想などを述べよ。

今回の提出課題は 7-1〜7-3 から 1 つ、7-4〜7-8 から 1 つ、7-9〜7-12 から 1 つの合計 3 課題とします。

[7] 黒丸の x 座標、y 座標を入れる整数配列と、どこまで入れたかを覚える変数を用意し、マウスクリックするごとに描くとともにここに覚える。Exposure イベントのときは覚えている黒丸を全部 draw する。