アイコン投げシェル[0]

久野　靖[†]、角田博保[‡]、大木敦雄[†]、粕川正充[*]

[†]筑波大学大学院経営システム科学専攻
[‡]電気通信大学情報工学科
[*]お茶の水女子大学情報科学科

概要

近年の計算機システムでは、その操作の多くをポインティングデバイス (マウス、トラックボール、電子ペン等) により行うようになってきているが、その操作方式はおもに初心者が学びやすいことを考慮して設計されており、常時計算機を使う利用者にとっては必ずしも最適でない。筆者らはファイル操作やプログラムの起動を行う部分 (GUI シェル) のユーザインタフェースに着目し、旧来の GUI シェルで多用されているメニューやその他の GUI 部品に替えて、より自由度の高いドラグ&ドロップとそれを高速化した操作である「アイコン投げ」を主に使用したインタフェースを持つ GUI シェルを設計/開発した。現在はまだテスト段階であるが、このようなインタフェースは特にキーボードを持たず液晶タブレットにより表示と入力を行うような計算機システムにおいて有望であるとの感触を得た。

1　はじめに

近年の計算機システムにおけるユーザインタフェースは、その操作のより多くをポインティングデバイス (マウス、トラックボール、電子ペン) によって行うようになって来ている。さらに最近の可搬型計算機システムでは、その可搬性を高めるために小型で (結果として) 打鍵しにくいキーボードを搭載したり、まったくキーボードを持たないものも多く見られるようになっており、そのようなシステムではポインティングデバイスによる操作が中心的な役割りを果たすようになっている。

これらの計算機システムにおいて、ファイル操作やプログラム起動などを通じてシステム全体を制御するグラフィカルユーザインタフェース (以下「GUI シェル」と記す) の代表的なものとして Macintosh Finder や Windows95 Shell などがあるが、これらはいずれもアイコンとメニューを多用している。とくにメニューは文脈に応じてその時点で利用可能な操作が明示でき、初心者にとっても分かり易いため、上記のシステムに限らず GUI 全般において多用されている。

しかしその半面、メニューはその選択に要する操作が複雑であり、操作時間や認知的負荷の観点からは最適ではない。これに対し筆者らは、より単純な操作であるドラグ&ドロップに着目し、その操作時間を改良する方法としてドラグ中のアイコンが行き先アイコンに到達するより前にマウスボタンを放し、あとはアイコンが「慣性で」移動

[0]An “Icon-Throwing” Shell by Yasushi KUNO[†], Atsuo OHKI[†], Hiroyasu KAKUDA[‡], and Masaatsu KASUKAWA[*], [†]Graduate School of Systems Management, The University of Tsukuba, Tokyo, [‡]Department of Computer Science, University of Electro-Communications, [*]Department of Information Sciences, Ochanomizu University.

して行って目的アイコンに到達する方式を提唱し、「アイコン投げ」と名付けている。筆者らの実験 [1] では、4 つの選択肢から 1 つを選ぶという比較的単純な操作について見た場合、「アイコン投げ」はメニュー操作より有意に高速に行えるとの結果が得られている。

筆者らは次の段階として、より実用的な汎用のユーザインタフェースに「アイコン投げ」を採用し、従来の GUI シェルよりも高速に操作でき、なおかつ柔軟性も高いものを作り出すことを試みている。また、その過程において従来の GUI シェルの機能を検討した結果、多くの望ましくない (と筆者らが感じる) 特性を認識することとなり、これらを改善することも併せてめざしている。本稿ではその現状について報告するものである。

以下第 2 節では従来の GUI シェルにおける操作の枠組みについて整理し、汎用のシェルに必要な機能について検討するとともに、Unix Shell との比較に基づき、これまでの GUI シェルに欠けている機能について検討する。続く第 3 節では第 2 節の検討に基づき、「アイコン投げシェル」の設計について述べ、第 4 節ではその開発および試用経験について報告する。最後に第 5 節でまとめと今後の方向について述べる。

2　従来の GUI シェルとその問題点

2.1　従来の代表的な GUI シェル

本節では従来の GUI シェルの代表として Macintosh Finder、Windows 3.1 Program Manager、Windows95 Shell を取り上げ、その枠組みについて検討する。以下では Macintosh の用語にならってディレクトリの意味で「フォルダ」と記す。

Macintosh Finder(図 1) のもっとも基本的な操作はフォルダ (またはディスクボリューム) とその中の要素 (ファイルまたはフォルダ) に対する「開く」「閉じる」「移動」「削除」の各操作である。

フォルダもディレクトリも、通常はアイコンの形で表示することが多いが、数が多い場合には名前の一覧の形で表示する場合もある。各操作は操作対象をマウスクリックにより選択した後、メニューやその他の操作 (ダブルクリック、ドラグ、窓の特定位置のクリック、キーボードショートカットなど) により指定する。

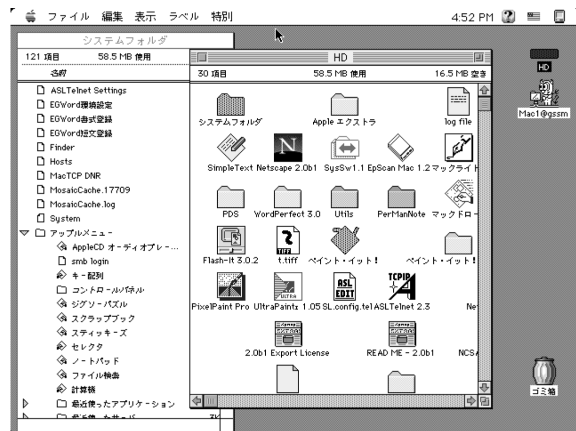

図 1: Macintosh Finder の画面

フォルダは「開く」ことにより、その中の要素にアクセスできるようになる。そして「閉じる」ことにより画面から消えて画面領域を占有しなくなる (特に画面の小さいシステムでは画面領域は貴重な資源である)。

ファイルを「開く」場合は、そのファイルがプログラムである場合にはプログラムを起動する。それ以外のファイルの場合は、そのファイルを扱うプログラムが (利用可能であれば) 起動される。したがって、ファイルを「閉じる」という操作は実際には各プログラムを終了させることに対応している[1]。

「開く」操作はいずれの場合もメニューまたはダブルクリックで指定できる。「閉じる」操作はメニューまたは窓のクローズボックスのクリックで指定できる。

フォルダやディレクトリの「移動」は、移動先が同じボリュームであれば格納位置の移動、違うボリュームであれば指定した位置へのコピー作成を行う。「削除」は指定したフォルダやディレクトリを消去する。「移動」「削除」はともに操作対象を移動先またはごみ箱アイコンへドラグすることで表す。

最後に、比較的最近 (System 7 以降) になって、

[1]ただし、複数のファイルを同時に扱えるプログラムの場合は、ファイルをすべて閉じてもプログラムは終了しないものが多い

ファイルアイコンをプログラムアイコンにドラグ&ドロップすることで指定したプログラムに指定したファイルを開かせることができるようになっている (それ以前は、Finder で直接ファイルを開いた場合、そのファイル自体に指定された特定のプログラムだけが使われていた)。

このように、Macintosh Finder ではその機能のかなりの部分を「ファイルとディレクトリのブラウジングや操作が占めていて、プログラムの起動は「ディレクトリを開く」のと対比させた「ファイルを開く」操作として位置付けられている。このため初心者にとっても操作のモデルを獲得しやすいが、その代償としてファイルが多くなるとプログラムファイルの置き場所を探すのに手間どることが多くなる (そのために alias 機能が用意されてはいる)。

これとは対象的に、Windows 3.1 のプログラムマネージャはプログラムの起動のみを扱うため、各プログラムに対応するアイコンをファイルシステム上の位置とは独立した「グループ」にまとめて整理するようになっている。

ファイルの操作はこれとはまったく分けて、プログラムの 1 つであるファイルマネージャによって扱うが、ファイルマネージャからもプログラムが格納されたファイルを指定してプログラムを起動することができる。プログラムマネージャのグループに登録されていないプログラムについてはこの方法で起動するか、またはコマンド行から起動することになる。また、ファイルマネージャからファイルアイコンをドラグしてプログラムマネージャのプログラムアイコンにドロップすることで Finder と同様のファイルを指定した起動も可能である。

このように、Windows 3.1 ではプログラムに関係する操作とファイルに関係する操作を分けているが、ファイル操作の一環としてプログラムを起動することも可能なので 1 つの事柄を複数の方法で行えるという分かりにくさが生じている。

Windows95 Shell は Finder と Windows 3.1 の中間であり、画面上ではおもにディスクドライブ、ディレクトリ、ファイルをアイコンや窓として表して操作するが、これとは別に「スタートアップ」メニューを通じて Windows 3.1 のプログラムマネージャと同様にプログラムをグループに分類して保持し、それらの起動を行えるようになっている。加えて、Windows 3.1 のファイルマネージャに相当するプログラムも提供されている。

ユーザ人口から見れば上記 3 つの GUI シェルのユーザが圧倒的に多いが、これら以外のシステムでもさまざまな GUI シェルが使われている。たとえば NextStep のワークスペース、SGI の Indigo Magic、X Window 上の X.desktop、xfm などのデスクトップマネージャが代表的であろう。しかし、これらの GUI シェルでもそれらが提供する機能には前述の 3 者と本質的な違いはない。

2.2 GUI シェルの問題点

前節で挙げた GUI シェルの機能について、その主要な問題点は「プログラムの起動について、最低限の機能しかない」ことだと考える。すなわち、基本的に提供されている機能はプログラムの起動「しか」なく、せいぜい最初に読み込むファイルを 1 個ないし数個指定できる程度である。

従って、最初に読み込むファイル以外の情報をプログラムに与えるためには、起動した後のプログラムに対する操作によるしかない。典型的には、「設定」メニューで複数の設定項目から設定内容を選択し、ダイアログボックスを表示させてその中の GUI 部品 (ボタン、メニュー、ドロップダウンリスト、入力欄など) を駆使して設定を行うが、これはかなり複雑で大げさな操作であり、パラメタによって少しだけプログラムの動作を変えるといったことはやりにくい。

言い替えれば、このようなインタフェースでは「複雑で大きく 1 つのプログラムで何でもやる」方向をめざしがちであり、Unix のように簡潔で単機能のプログラムを組み合わせて作業することを自然だと考える筆者らにとっては好ましくないものと感じられる。

また、複雑な操作を通じて動作を指定するのは不便なため、多くのプログラムでは設定した状態をリソースファイルやパラメタファイルに保存するようになっている。しかし、特定の箇所に設定を保存する方式だと複数の設定を使い分けるのは困難である。

複数のパラメタファイルを使い分ける場合にはこの問題はないが、まだ保存していない設定が使いたい場合には一旦そのプログラムを起動して設定ファイルを作成しなければならず、やはり単機能のツールを組み合わせるというのとはほど遠いことになってしまう。また、一時的な設定のつもりで設定後にパラメタファイルを作成しなかったら、後でまた同じ設定を必要として複雑な設定操作を繰り返すはめになったりすることも多い。

もう1つの大きな問題は、コンテキストの欠如である。すなわち、各プログラムはどのように起動しても上述の設定ファイル等で指定した環境のみに依存した状態を持ち、起動時のコンテキストを反映する部分はほとんどない。

たとえば、プログラムによってファイルを書き出そうとすると、多くの場合ファイルシステムの最上位から目的のディレクトリに至る経路を GUI 部品を駆使して指定しなければならないのが普通である (最初に開いたファイルのあるディレクトリをデフォルトとして提供するものはいくらかましだが、常に読み込んだファイルと同じ場所に書き出したいとは限らない)。さらに、複数のプログラムを並行して使いながら作業する場合には、それらのプログラム1つずつについて同じような指定動作を繰り返すことを強いられる。

これは、Unix のカレントディレクトリに慣れ親しんでいて、作業ごとにディレクトリを複数並行して使い分けている筆者らにとっては不便極まりない。プログラムごとに起動後にコンテキストを設定しなければならないことは、上述の「大きいプログラム」を指向させる原因の1つにもなっている。

また、Unix のシェルでは過去に行った入力を再度繰り返すことの無駄はヒストリ機能によって大幅に緩和できるが、これも一種のコンテキスト情報である。これが前述の設定ファイルと違う点は、最初に行った操作は自動的にヒストリリストに保存され、必要になったら検索してきて利用できることである。この点が、明示的に保存しなければならない設定ファイルとは根本的に異なる。

最後に、旧来の GUI シェルに基づくシステムの操作速度についても不満がある。まず、プログラムを起動する操作自体はアイコンのダブルクリックによって行えるが、それ以外の操作は基本的にメニューやその他の GUI 部品にたよることになる。

「はじめに」で述べたように、メニューはその操作系列が複雑であり、慣れてきても十分高速に操作できないためいらいらする。このためにキーボードショートカットが提供されているが、キーのない/打ちにくいシステムではキーボードショートカットは助けとならない。

そして、GUI 部品は全般に多くの画面領域を消費してしまうため、まずダイアログボックスを表示させるといった手順を踏んでから使うことが多く、ますます操作時間を要することになる。このような手間を嫌って Windows アプリケーションでは小さいアイコンを集めたタスクバーが多用されるようになっているが、多数のアイコンを集めたために1つずつのアイコンがごく小さいものとなり、ポインティングに時間が掛かる上、普段使わないアイコンは単なる画面の浪費になっている (このためタスクバーをカスタマイズできるようにしたものもあるが、まずカスタマイズするというのは前述の設定ファイルと同じで、結局設定せずにいつも使いにくいまま使ってしまうことが多い)。

2.3 プロ向けの GUI シェルに望まれるもの

結局、これらの問題点を総合すると「コマンドやオプション等の指定に習熟し、かつ多量のファイルを持ち、複数の作業を並行して進めるような熟練ユーザにとっては Unix シェルのような文字入力インタフェース (CUI) と多数の簡潔なツールの組み合わせが望ましい」という当り前の結論になる。

しかし、最初に述べたように、今後はキーボードが使いにくいようなシステムが増えて行くことが予想される。また、GUI の設計には非常に大きな自由度があるので、現状の (初心者にとっての分かりやすさを設計目標とした)GUI の方式を土台から見直すことにより、CUI に優るような GUI シェルが設計できる可能性もないとは言えない。

文字を1文字ずつ入力させたらキーボードに優るものはないだろうが、ユーザが計算機に与える情報は1文字より大きな「かたまり」から成っているはずで、それをまとめて与えることができれ

ばキーボードより効率良く操作できるかも知れないからである[2]。

そのようなインタフェースが可能だとして、それは次の特性を満たすものであるべきである。

- コンテキスト (カレントディレクトリやコマンドヒストリ) をサポートする — これは多数のファイルを扱ったり、複数の作業を並行して行ったり、小さなプログラム群を組み合わせて作業する上で重要である。
- プログラムのオプション指定や組み合わせをサポートする — 多数の小さなツール群を組み合わせて使用するには不可欠の機能である。
- プログラムへの入力をサポートする — プログラムによっては最初のオプション指定だけが入力である場合もあるが、実行開始後に出力を見ながらさまざまな指示を与える場合もある。その場合、起動後のプログラムには旧来の方法でしか入力できないのでは望ましくない。
- プログラムの出力の還元 — コマンドや入力の内容として、別のコマンドの出力が利用できる場合は多い (たとえばファイル名一覧を表示させて、得たファイル名を次のコマンドのオペランドとして指定するなど)。これを、単なるカット&ペーストでなく、より能率よく活用できることが望ましい (Inter-referential I/O[**?**])。
- 画面の有効利用 — GUI 部品のように画面を消費するものは使わず、画面にはできるだけ利用者にとって有用な情報だけが現れるようにする。これは携帯システムなどの限られた画面では特に必須だが、より大きな画面でも望ましいことに変わりはない。
- 自由度の高さ — GUI 部品のように特定目的の操作だけを受け付けるものは画面の無駄使いなだけでなく、1 つの画面において指定可能な操作の選択肢を狭くしてしまう。効率的な操作のためには、画面を切り替えたりスクロールすることなく、できるだけ多くの操作が指定可能なことが望ましい。
- 自然なカスタマイズ — 限られた画面領域を有効活用するためには、インタフェースのカスタマイズは不可欠である。しかし、そのために専用のダイアログボックスの使用や設定ファイルの編集といった手間を掛けさせるとカスタマイズに対する敷居が高くなり、カスタマイズに費す労力も大きくなる。そうではなく、日常の使用がそのままカスタマイズを兼ねることができれば理想的である。
- メニューの排除 — メニューは (平常時には) 画面領域を消費せず、コンテキストに対応しやすいという利点を持つが、その半面操作が複雑で習熟しても高速な操作が難しく、階層構造という固定された枠組みを持ち、カスタマイズも簡単ではないことから、本稿でめざしているようなインタフェースにはふさわしくないと考える。

これらの選択の代償として、このようなインタフェースはその機能を熟知したユーザのみに有効活用できるものとなりそうであるが、それでよいものと考える。[1][2] にも記したが、筆者らは「初心者に厳しいユーザインタフェースほど、(熟練者にとっては効率よく操作できる) よいインタフェース」ではないか、という予想を持っている。

3 アイコン投げシェル

本節では、前節で挙げた方針に基づき、現在筆者らが Unix + X Window 上で開発している「アイコン投げシェル」(tsh – Throwing SHell) [3]について、その概要を説明する (アイデアのいくつかは、筆者らの過去の研究 [3][4] が土台になっている)。

なお、ここで述べるような設計が前節で挙げた方針の唯一の帰結だというつもりはなく、あくまでも叩き台の 1 つとして見ていただきたい。また、

[2] 筆者ら [5] は、VGA 画面上の単語をペンで拾うことにより、画面上の任意の出力を再利用して入力を効率化する手法を実装してみた。また、増井 [6] は頭文字をペンで選ぶとその頭文字を持ち文脈から使用される可能性の高い単語を提示し、その中から選択させる、という方法でキー入力よりも高速な文章入力が可能であるとの見解を示している。

[3] tcsh と csh の関係とは違って、tsh が sh に基づいているとか上位互換だということは一切ない。

その設計は現在も発展中であるので、ここに示すのは現時点 (1996 年 11 月末) での状態である。

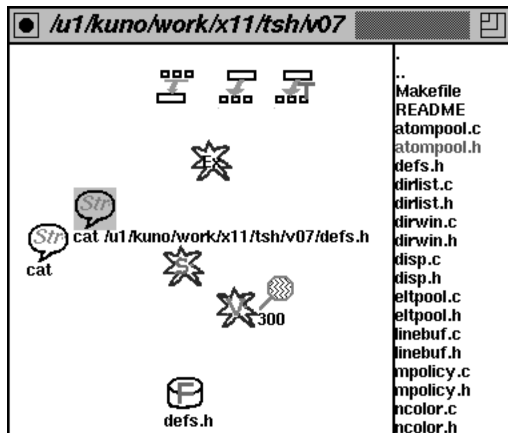

図 2: Tsh の窓

tsh 窓の構造

図 2 に tsh の基本となる窓 (tsh 窓) のようすを示す。前節で述べた方針に従い、tsh ではコンテキストの 1 つとしてカレントディレクトリをユーザに制御させるので、タイトルバーにはカレントディレクトリの絶対パス名が常に表示されている。

タイトルバー自体は現在のところ X のウィンドウマネージャが提供している。タイトルバーにより多くの機能を盛り込むため、ウィンドウマネージャによる制御を行わず自前でタイトルバーと窓制御の機能を提供することも検討したが、他の窓との統一性がなくなるため当面は上述の方針によっている (将来的には再検討する可能性もある)。

窓の本体はアイコン部と ls 部から成っている。ls 部は見ての通り、ディレクトリの一覧が表示されている部分であり、アイコン部は「アイコン投げ」により各種の操作を行う、tsh の中枢部分である。アイコン部については少しあとで説明する。

ls 部の主要な機能は、いちいち ls のようなコマンドを起動しなくてもファイルの一覧が見られるようにすることである (ls は起動回数から見れば Unix で最も多く使われるコマンドの 1 つである)。しかし、場合によっては一覧が画面上で大きな領域を占めることは望ましくない。このため、アイコン部と ls 部が占める大きさをそれぞれ制御できるようにした。

多くの GUI ではこのような場合境界線上にグリップを配してそれをドラグさせるが、大きなグリップは領域の無駄であるし、小さいグリップはポインテイングに時間が掛かる。そこで、tsh ではグリップを使用せず、窓全体の大きさを (ウィンドウマネージャによって) 変更すると、最後にクリックのあった側の幅が伸び縮みし、他の領域は元のままに留まることによって各領域の大きさを制御することとした。これにより、GUI 部品のようなものを使う必要はなくなった。

ls 窓の動作と機能

次に、ls 窓の内容の行数が多い場合には窓の中身を縦方向にスクロールさせる必要がある。これも通常の GUI ではスクロールバーを使用するわけだが、tsh では窓の中の「白い部分」をポインタでたぐることによりスクロールを制御する。

スクロールのために画面をたぐる方法自体は MacPaint 以来の古典であるが、MacPaint ではたぐるモードに切り替えてからたぐる必要があった。しかし、ls 窓を含め、多くの窓において「白い部分」は選択できない (役に立たない) 部分なので、そこでボタンが押された場合をスクロールに利用するのは理にかなっている。

加えて、MacPaint ではたぐる倍率は 1(ポインタの移動量と窓の内容の移動量が等しい) であったが、tsh ではポインタの横位置によってスクロールの倍率を制御するようになっている。

具体的には、ポインタが窓の左端にある時にはポインタの垂直移動量とスクロール量は等しく (くっついて移動)、右端にある時は窓全体の高さぶんスクロールさせると中に入っている行数ぶんのスクロールが行えるように倍率が制御される。左右端の中間では、倍率もその位置に比例して制御される。[4]

ls 窓の「白くない」部分をクリックした場合には、その行が選択される。さらに、ダブルクリックした場合には、その行がディレクトリである場

[4] この方法は暦本純一氏 (現 Sony CSL) のアイデアをもとにしている。

合、tsh プロセスのカレントディレクトリがそのディレクトリに移動する (つまり cd する)。それに伴い、タイトルバーの表示が変化し、ls 窓の内容も新しいディレクトリの内容に変化する。

このように窓自体はそのままでカレントディレクトリが変化するというのは、Unix shell の機能にならったものである。普通の GUI シェルではディレクトリやフォルダを開くと対応して新しい窓が開くものが多いが、これだと (元のリストを見続けることができるという利点はあるが) 画面が窓だらけになり、新しくできた窓の位置や大きさを調整するという余分な作業に手間を取られてしまう。[5]

さらに、ディレクトリを移動した後で前に実行したものと同じコマンドを起動したい (Unix shell でいえば cd の後でヒストリリストにあるコマンドを再実行させることに相当) 場合にもこの方法が自然だと考える。

なお、ユーザが意図して指定したディレクトリを別の tsh 窓として開きたいと思った場合は、後述のアイコン窓の操作によって行うことができる。

アイコン窓の基本

アイコン窓はその名前通り、さまざまなアイコンを中に置くことができる (具体的な種別については順次説明していく)。そして、アイコン窓の基本的な機能は「あるアイコンを別のアイコンに投げつける」ことだけである。その基本的な動作を図 3 に示す。

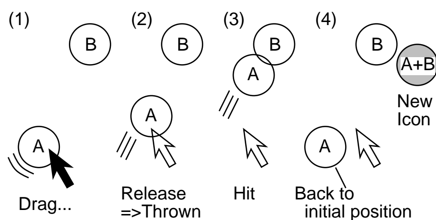

図 3: 「アイコン投げ」の基本動作

すなわち、アイコン投げではあるアイコンをドラグして別のアイコンに向けて移動し (1)、その状態でマウスボタンやペン先を離す (2)。するとそれまでドラグされていたアイコンはそのままの方向/速度で動き続ける。そして、行き先のアイコンに接触すると (3)、両方のアイコンが反応して何らかの動作が起こり (場合によっては新しいアイコンが生成されて行き先アイコンの隣に現れる)、ドラグされていたアイコンは元の位置に戻る (4)。実際に投げつけたときに何が起こるかは、投げつけたアイコンと投げつけられたアイコンの両者の種別によって異なる。

これにより、たとえば窓内にアイコンが N 個あったとすれば、$N \times (N-1)$ 通りの操作を直接指定できる (A を B に投げるのと B を A に投げるのは当然区別する)。この数は、比較的少ない N においても通常のメニューバーなどが提供する選択肢の数より圧倒的に大きく、またアイコンは後述の方法により瞬時に追加/削除できるため、メニューよりはるかに容易にカスタマイズできることになる。

アイコン窓に新しいアイコンが追加される方法には、大きく分けて次の 3 通りがある。

1. アイコンを投げつけた結果として、新しいアイコンが生成される場合 — たとえば、文字列アイコンに別の文字列アイコンを投げつけると、両者が連結された新しい文字列アイコンができる。
2. よそからドラグしてきた場合 — tsh の各種のツール群は、その中身の要素をドラグしてきてアイコン窓に入れることができる (ドラグ状態のポインタがアイコン窓に入った瞬間に新しいアイコンが生成され、それがアイコン窓の中でそのまま継続してドラグされる)。
3. カット&ペースト — X の `PRIMARY` セレクションの内容 (文字列) は、マウス右ボタン (またはペンの胴体スイッチ) を押すことでアイコン窓の中に文字列アイコンとしてペーストすることができる。[6]

アイコンを投げたが、行き先に別のアイコンがない場合には窓のふちまで来ると停止する。

[5] X Window 上のフリーソフトの GUI シェルである xfm も新しい窓ができずに既存の窓の中だけで切り替わる方式を採用している。

[6] ただし、長い (複数行にわたる) セレクションはこの方法に向かないので、tsh 用の「クリップボード」のようなものを用意し、いったんここにペーストしてから利用することも考えている。

ごみ箱の機能

上述の各操作により、アイコン窓の中のアイコンは急速に増えるのが普通である。それを捨てるのに当初は「ごみ箱」アイコンに投げつける方法を採っていたが、あまりにもごみの発生が多いため、とにかく窓の外にドラグすれば捨てられるように変更した (あたり構わずごみを捨てているようで気持はよくないが、この方がはるかに効率的である)。

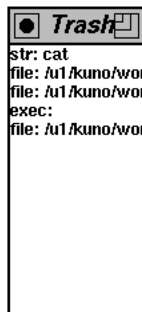

図 4: ごみ箱の窓

捨てたアイコンはすべて、ごみ箱の役割をする窓 (図 4) に順次集積されていく。従って間違って捨ててしまってもすぐに取り返せる。ごみ箱は常に、最後に捨てられたものが一番上にあり (本物のごみ箱と同じ)、内容の肥大化を緩和するため、同じものを繰り返し捨てた場合は古い同一内容のものは消えてなくなる (本物のごみ箱とはだいぶ違う)。また、ごみ箱から取り出してもごみ箱の内容は変化しないので、アイコンをコピーしたければごみ箱に入れてから複数回拾えばよい。

複数の tsh 窓が存在する場合、ある窓のアイコンをドラグして別の窓に移すことができるが (これも上述の 2 の操作の一種)、その場合も元の窓の外に出た時点でごみ箱に捨てられ、そのまま直ちに拾われたものとして扱われる。

したがって、ごみ箱窓は一群の tsh 関係ツールが共通して使うヒストリリストとして働くといえる。これにより、とりあえず使っていないアイコンは安心して捨ててしまい、アイコン窓の中身を適正に保つことができる。

なお、常にごみ箱をあさっているのも面白くないので、内容や順序を利用者が制御し整理できるような「道具箱」の窓も今後用意する予定である。

さまざまなアイコン

アイコン窓が扱うアイコンには、データないし操作される対象を表すデータアイコンと能動的な操作を表す機能アイコンの 2 種類がある。現在実装されているデータアイコンとしては次のものがある。

- 文字列アイコン — 文字列ないし語を表す。コマンド名、パラメタ、ファイル名など何でも表すのに使える。前述のように、文字列アイコンを互いに投げつけることで連結できる (間に空白文字がはさまれる)。
- 指令アイコン — 文字列アイコンとほぼ同様であるが、tsh のツール群に対する指令を表すものとして扱われる。
- ファイルアイコン — ファイルを表す。文字列と違う点は、cd や他の窓への移動によってカレントディレクトリが変化した場合に、文字列アイコンはそのままだが、ファイルアイコンでは元のファイルを正しく指すように調整される点である。[7]
- ディレクトリアイコン — ディレクトリを表すことの他はファイルアイコンと同様。また、実行アイコンに投げつけることで新しい tsh 窓を生成することができる。
- プロセスアイコン — tsh が起動したプロセスを表す。現在のところプロセス ID の情報のみを持っているが、将来は (その種別にもよるが) 入力を与えたりするのに使えるようにしたい。

また、現在実装中の機能アイコンには次のものがある。

- implode アイコン — 文字列の空白を取り除いてくっつける。
- explode アイコン — 文字列を 1 文字ずつばらばらにする。

[7]アイコンの絵柄はファイルの種別 (Unix の file コマンドが返す出力) に応じていく種類かのものが使い分けられるようにする予定である。

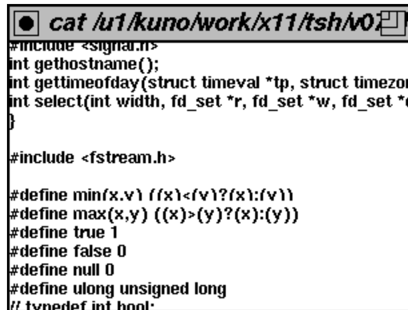

図 5: 出力窓

- tokenize アイコン — 文字列を単語単位でばらばらにする。
- 文字列化アイコン — ファイルやディレクトリのアイコンを絶パス名に変換する。絶対パス名用と相対パス名用がある。
- exec アイコン — コマンドを起動する。
- view アイコン — 出力窓 (図 5) を使用してコマンドを起動する。出力窓の内容は ls 窓と同様、中身をドラグしてきてアイコン窓に文字列アイコンとして取り出すことができる。スクロールの方式も ls 窓と同じである。
- eval アイコン — コマンドを起動して出力を文字列アイコンとして得る。

4 Tsh の開発状況と使用経験

tsh の試作版は C++言語と Xlib API を使用して記述され、現在 FreeBSD 2.1 および Solaris 2.4 上で稼働している。特に Unix の種別に依存する部分はないので、他の Unix システムでもそのまま動くはずである。開発は筆者間の議論に基づいて筆者の一人が行った。本稿の執筆時点において、実装に要した労力はおよそ 2 人月程度、ソースコード行数は 2000 行である。

tsh を稼働させるプラットフォームとしては PC 互換機や SparcStation などのデスクトップシステムでも可能だが、最初に述べたように携帯性の高いシステムでの使用に興味があるため、Wacom PT486 や三菱 Amity などの液晶タブレットを採用したシステム (以下タブレット PC と記す) での評価も行っている。

これらのタブレット PC は、Windows for Pen などのシステムを搭載して手書き文字認識入力機能により使うのが普通であるが、筆者らはこの上に FreeBSD を搭載し、X の上でソフトキーボード、T-Cube[8]、Unistroke[9] などの入力機能を自作して利用している [5](X Window 上のソフトキーボードは xkeycaps というフリーソフトを利用している)。

まだ開発途上なので、対照実験はもちろん使用経験に基づく主観的な評価も始めたばかりであるが、とりあえず感じたことをまとめておく。

- tsh のようなインタフェースは、マウスよりはペンの方がはるかに楽に操作できる。これは特に、マウスでは「ボタンを押しながら移動する」のがやや不自然な操作であるのに対し、ペンでは「ペン先を表示面に接触させながら移動する」のは筆記用具で書くのと同じであり、日常慣れ親しんだ操作だからではないかと考えられる。
- 既に引数が定まった定型コマンドについては、きわめて快適に起動できる。これは、CUI でのヒストリリスト探索よりも、使いたい対象がアイコンの形で画面上に見えている方がアクセスが容易であり、またアイコンの配置も使っている過程で自然に「使いやすいように配置する」ことが起きるためではないかと思われる。
- 一方、新たなコマンドを (引数などをくっつけながら) 組み立てて行くのはまだ煩わしい感じがする。たとえば、文字列アイコンとして存在していないものを使いたければ現状ではキーや Unistroke などで入力してカット&ペーストで文字列アイコンにするか、または既存のアイコンを explode でばらして切り貼りし、implode でくっつける必要がある。この問題に対処するには、使いそうな文字列をまとめて保管し、その中から必要なものを高速に取り出せるような仕組みを用意する必要がある。前者は既述の「道具箱」により実現できるが、後者については今後の検討が必要である。

5　まとめと今後の課題

本稿では、従来の GUI シェルについてその問題点を検討し、より熟練ユーザに適したインタフェースの一例として「アイコン投げシェル」(tsh) の設計・実装・使用経験について述べた。

従来の GUI を見直そうという動きは筆者ら独自のものではなく、たとえば [10] などは多くの有益な示唆を含んでいる。しかし、本研究のように習熟したユーザを対象とした、シェル全体を置き換える具体的なインタフェースの提案はまだほとんどないようである (Magic Cap のインタフェースなどは独自のシェルであるが、初心者を対象としている)。

tsh はまだ開発途上であり、機能的にも十分とは言えないが、少なくともこのような方式のインタフェースを実用に用いることは十分可能だとの見通しは得られたと考える。

今後は、欠けている機能群の実装と、試用からのフィードバックに基づく機能の追加･改良を行っていく予定である。最後に、現時点で実装を予定している機能について挙げておく。

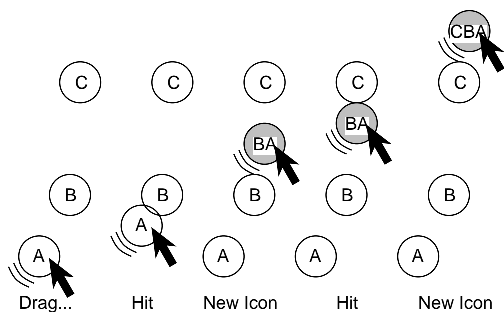

図 6: 「くし刺し」の動作

- 「くし刺し」— 現在の版を使用してみると、多数の文字列アイコンを連結してコマンドを組み立てるのに、1 つずつ投げてくっつけるのがかなり煩わしい。また、中間生成物のアイコンが全部ごみになって残るのもじゃまである。そこで、アイコンをドラグして他のアイコンの上を通過させると、それらが連結されたアイコンが生成され、これまでドラグしていたアイコンと「すり代わる」方式 (図 6) を考案し、これを「くし刺し」(drag through) と名付けた。これにより、コマンドの組み立てが高速になり、余分なごみアイコンの発生も抑制される。
- アイコンダイアログボックス — 従来の GUI ではプログラムが何らかの障害に遭遇するとダイアログボックスが提示され、その中にある選択肢を選ぶようになっている。しかし、コマンド名やオプション指定が違ったような場合には、正しいコマンドやオプションを「その時」選ぶだけでなく、取っておいて次回からは正しく操作できるようにしたい。そこで、可能な選択肢をアイコンとして提示し、その中から利用したいものを抜き出してアイコン窓などに保管したり、正しいコマンド行を組み立てるのに利用することを検討している。
- 他の窓の制御 — 現在はアイコン窓の中でだけアイコンがさまざまな動作を担っているが、アイコンをドラグして他の窓にドロップすることで他の窓に指示が出せるようにしたいと考えている。
- 自己反映操作 — tsh の窓自身に対する操作 (窓の終了やモードの変更など) をアイコン投げで扱いたいので、アイコン窓の中に「自分自身」を表すアイコンを導入したい。
- 文字列加工器 — 前節で挙げたように、すぐに利用可能な文字列がない場合にこれを何らかの方法で組み立ててくるような仕組みが必要である。たとえば、文字列の一部を選択してから別の文字列を投げるとその部分が置き換わる、といった方法が考えられる。

謝辞

ソニーコンピュータサイエンス研究所の増井俊之氏と暦本純一氏には、ユーザインタフェース全般に関して多くの議論をしていただいた。ここに感謝します。

参考文献

[1] 久野, 大木, 角田, 粕川: 「アイコン投げ」ユーザインタフェース, コンピュータソフトウェ

ア, vol. 13, no. 3, pp. 38-48, 1996.

[2] 久野, 角田, 大木, 粕川: アイコンは投げられるか?, 第35回プログラミングシンポジウム報告集, pp. 121-132, 情報処理学会, 1994.

[3] 久野, 角田: 流れて行かない Unix 環境, 情報処理学会論文誌, vol. 29, no. 9, pp. 854-861, 1988.

[4] 佐藤, 久野, 鈴木, 中村, 二瓶, 明石, 関: CLU マシンのユーザインタフェース, 第29回プログラミングシンポジウム報告集, pp. 13-22, 情報処理学会, 1988.

[5] 大木, 久野, 角田, 粕川: ペンコンピュータと UNIX, 個人メモ, 1996.

[6] 増井: ペンを用いた高速文章入力手法, Proc. WISS'96, 日本ソフトウェア科学会, 1996.

[7] Draper, S. W.: Display Managers as the Basis for User-Machine Communication, in Norman, Draper, eds., User Centered System Design, Laurence Elbaum, 1986.

[8] Venolia D., Neiberg, F.: T-Cube: A Fast, Self-Disclosing Pen-Based Alphabet, Proc. CHI'94, pp. 265-270, 1994.

[9] Goldberg, D., Richardson, C.: Touch-typing with a stylus, Proc. INTERCHI'93, pp. 80-87, 1993.

[10] Gentner, G., Nielson, J.: The Anti-Mac Interface, CACM, vol. 39, no. 8, pp. 70-82, 1996.