図形を用いたユーザインタフェース
に関する研究

平成3年12月

松浦 敏雄

内容梗概

本論文は，筆者が大阪大学基礎工学部および同大学院基礎工学研究科に在学・在職中，嵩研究室，藤澤研究室，および，谷口研究室において行なったソフトウェア工学に関する研究のうち，図形を用いたユーザインタフェースに関する研究をまとめたものである．

第1章緒論と2章以降の各章の第1節では，研究の背景，意義，および，本研究の内容について概説している．また，各章の最後の節，および，全体の結論では，本研究で得られた主な結果と今後に残された問題点について述べている．

計算機の発展と普及により，その利用者数が急速に増大し，より分り易いユーザインタフェースが求められるようになってきており，これに伴いユーザインタフェースに関する様々な研究が行なわれている．本論文では，図形を用いたユーザインタフェースに関して，図形の入力・編集方法，プログラムの内部情報の図的表示とその上での入力操作法，および，対話型アニメーションの方法の3つの分野についてそれぞれ新しい方法を提案・実現し，それらの有用性を示している．また，図的ユーザインタフェースの一つの応用例として，計算機シミュレータを提案・実現し，その有用性も明らかしている．

2章では，まず，図形そのものを如何にすれば容易に入力・編集ができるかという問題を取り扱っている．この問題に対するアプローチの方法としては，図形記述言語を用いてテキスト形式で図形を記述する方法と，画面上でのマウス等の直接操作によって図形を入力する方法がある．前者の方法では，同じ図の繰り返し等，柔軟に図形の記述ができる可能性があるが，どんな図が描かれるかが直観的には分りにくい．後者の方法では手軽に図形の入力・編集が可能であるが，繰り返しのある図などの入力には手間がかかる．ここでは，この2つの方法を融合した図形の入力・編集方法として，画面上で描画した図形を直接操作により作図用部品に加えることができる部品定義機能と，図形記述言語を用いて図形部品を定義する機能を合わせ持ち，さらに，図形間の接続・包含関係を自動認識し，図形の移動等の操作においてそれらの関係を保存する機能を提案している．また，上記の機能を含んだ作図ツールを作成し，いくつかの典型的な図に対して，作図の操作回数および所要時間を測定し，これらの機能の有効性を実証するとともに，それらの機能の実現法が妥当であることを明らかにしている．

3章では，応用プログラムで取扱う様々な内部情報を分り易く画面に表示し，その上での入力操作を容易に行なう方法に関して論じている．内部情報をユーザのコマンドによって操作するような応用プログラムは多い．これらのプログラムでは，内

部情報の様子をユーザに分り易く図示し，その表示を用いて簡便にコマンド入力が行なえることが求められている．計算機で扱う内部情報は，一般に各節点に情報を持たせたグラフとして表現すると分り易い．それらを見易い形で画面に表示し，グラフに対する更新操作を画面上で行なえるソフトウェアもいくつか提案・試作されているが，実用規模のデータに対して，全体の構造も把握し易く，任意の部分の詳細も簡便に参照することが出来，かつ，効率良く動作するものは見あたらなかった．ここでは，広く用いられる木構造データを対象とし，数百～数千程度の節点を持つ木構造データに対して，木構造全体の概形，および，その任意の複数の部分の拡大図を表示し，その画面上でのマウスの操作により木構造データに対する操作指示を応用プログラムに伝えることができる機能を持つライブラリを提案し，その実現法を示している．また，作成したライブラリを実際にいくつかのアプリケーションプログラムに対して適用してその有用性を示し，さらに，数千程度の節点を持つ木構造データに対しても，節点の配の計算および画面の更新が効率良く行なえることを実証している．

4章では図形を用いたユーザインタフェースの一つの応用例として，計算機の基本アーキテクチャの理解を助けるための計算機シミュレータを取り上げている．計算機の構造・動作を理解するために，汎用のCADシステムを用いることも可能であるが，一般に，回路シミュレーションを行なう前に，接続関係の情報を抽出するための(コンパイルに相当する)手続きを必要とし，このため思考が中断され，初心者にとっては使いにくい．また，画面上で計算機内部の動作を確認できる計算機シミュレータも試作されているが，固定した計算機を対象としていたり，ベースとなるシステム(Smalltalk-80)に関する深い知識を必要とするなど，計算機の構造・動作を理解するために適当なものはなかった．そこで，これらの点を考慮して，図的ユーザインタフェースを重視した計算機シミュレータを新たに提案し，その実現法を示している．作成したシミュレータでは，画面上でマウスを使って部品を配置，配線することにより計算機回路を入力でき，回路の入力中でもシミューレションが可能で，回路図の入力画面と同一の画面上でその動作を視覚的に観察することが出来る．さらに，計算機構成用の高機能な部品が予め用意されているので，低レベルの回路から作り始めなくてもよく，新たな部品の定義も比較的簡単に行なえる．また，実際に計算機回路の入力例を挙げ，比較的簡単に入力できたこと，部品定義機能が有効であることなどを示している．

5章では，図形を用いたユーザインタフェースをさらに一歩進めて，動きのある図形，すなわち，アニメーションを行なう方法を論じている．アニメーションは，図形やイメージ情報に，時間情報を付加できるので，有力な情報の表現手段である．アニメーションに関する研究は，1フレーム毎の静止画を予め計算し，逐次ビデオに

収録するという方式に関するものが中心であった．しかし，ユーザインタフェースとしてのアニメーションを考えた場合，対話型である (すなわち，ユーザからの入力によってアニメーション実行が変化できる) 必要がある．ここでは，汎用ワークステーション上での対話型のアニメーションに対して, 指定された時間通りの速さでオブジェクトを動作させ, かつ, スムーズな表示になるように, できるだけ指定された時間間隔でフレームの画像を更新するアニメーションの方法を提案している．この方法では, 指定された時間間隔でフレームを更新するために必要な画像情報 (実行時情報と呼ぶ) 等をアニメーションの実行に先立ち自動収集し, 実行時にはそれに基づいて表示を行なう．このとき, (1) 指定された時間通りにオブジェクトを動作させる, (2) 計算機の違いを意識せず, 計算機の能力に適した実行時情報を自動収集する, (3) アニメーションの実行時に利用者からの入力によって動的に表示画像が変化するような場合にもなるべくスムーズな表示を行なう, (4) メモリの使用量をできるだけ少なくする, などの点を考慮している．さらに, この方法を用いたアニメーションシステムを試作し, 実験によって，実行時情報が, 機械の速さに応じて適切に自動生成されたこと, それらによりシナリオの指定通りにスムーズに表示できたことを確かめている．

目 次

目次

1章

緒論

計算機の発展と普及により，その利用者数が急速に増大し，より分り易い計算機のユーザインタフェースが求められるようになってきている．従来の計算機では，文字列によって情報を表示し，また，キーボードなどを通して文字列によって情報を入力していた．これに対して，ビットマップディスプレイやマウスの普及によって，2次元平面を利用した入出力が手軽に利用できるようになり，これに伴い，ユーザインタフェースに関する様々な研究が行なわれている．[1],[2],[3]

本論文では，図形を用いたユーザインタフェースに関して，図形の入力・編集を容易に行なう方法，応用プログラムの扱う内部情報を分り易く表示し，それらの更新操作を容易に行なう方法，および，対話型のアニメーションの方法について，それぞれの問題点を指摘し，新たな機能および方法を提案して，それらに基づくシステムを作成し，実験によりそれらの有用性を示している．また，図的ユーザインタフェースの一つの応用例として，計算機の回路入力および動作確認が容易にできる計算機シミュレータを提案・実現し，その有用性も示している．

図形を用いたユーザインタフェースにおいては，まず，図形そのものを如何にすれば容易に入力・編集ができるかが重要な問題である．この問題に対するアプローチの方法としては，図形記述言語を用いてテキスト形式で図形を記述する方法[13],[11]と，画面上でのマウス等の操作によって図形を入力する方法[4],[6],[7] がある．前者の方法は柔軟な図形の記述ができる可能性があるが，図形の記述を見ただけで全体がどんな図であるかを直観的にイメージするのは，困難である．後者の方法では，全体の図の様子が常に画面に表示されているので，その把握は容易であり，また，手軽に図形の入力・編集が可能であるが，繰り返しのある図等の入力においては，図形記述言語で記述する方が簡単な場合もある．本論文の2章では，この2つの方法を融合した図形の入力・編集方法について論じている[関連発表論文 (1),(2),(6),(8),(10)]．ここでは，まず，図形の入力・編集を容易に行う方法として，画面上で描画した図形を

直接操作により作図用部品に加えることができる機能と，図形を記述するための言語を用いて図形部品を定義する機能を提案している．さらに，編集操作を容易にする方法として，描かれた図形間の接続および包含関係を作図ツール側で自動的に認識し，図形の移動，拡大・縮小等の操作においてこれらの関係を保存する機能を提案している．また，上記の機能を含んだ作図ツールを作成し，実験によりこれらの機能の有効性を実証するとともに，それらの機能の実現法の妥当性を明らかにしている．

2番目に取り上げた問題は，応用プログラムで取扱う様々な内部情報を，どのように分り易く画面に表示し，また，それらに対する更新操作をどのようにして容易に行なうかということに関してである．ユーザの発するコマンドによって，計算機の内部情報を操作するような応用プログラムは多くみられる．これらのプログラム，例えば，代数的仕様における項書換え系では，内部情報の様子，すなわち，対象とする項をユーザに分り易く図示し，その表示を用いて簡便にコマンド入力が行なえることが望まれる．計算機が扱う内部情報は，一般に各節点に情報を持たせたグラフとして表現すると分り易い．それらを見易く画面に表示し，グラフに対する更新操作を画面上で行なえるソフトウェア (グラフエディタ) もいくつか提案・試作されている[25],[26]．しかし，実用規模のデータに対して，全体の構造が把握し易く，任意の部分の詳細も簡便に参照することが出来，かつ，効率良く動作するものは見あたらなかった．3章では，計算機が取り扱うデータ構造として広く用いられている木構造データを取り上げ，数百～数千程度の節点を持つ木構造データに対しても実用的に対処できるグラフエディタを提案し，その実現法を示している[関連発表論文 (5),(14)]．提案するグラフエディタは，木構造データに対して，木構造全体の概形を画面に表示し，同時にその任意の複数の部分を拡大表示できる機能を持っている．また，作成したグラフエディタを実際にいくつかの応用プログラムに対して適用してその有用性を示し，さらに，数千程度の節点を持つ木構造データに対しても，節点の配置の計算および画面の更新が効率良く行なえることを実証している．

図形を用いたユーザインタフェースは様々な応用プログラムで利用可能であるが，4章では一つの応用例として，計算機の構造・動作の理解を容易にするための計算機シミュレータについて述べる．汎用のCADシステムを用いて計算機の構造・動作を理解させることも可能であるが，一般に，シミュレーションを行なうためには，回路図からシミュレーションを行なうために必要なデータファイルを作成する手続き (コンパイルに相当する) を必要とする．このため，回路の修正とシミュレーションを繰返し行なうとき，回路の修正の毎に，この手続きを行なう必要があり，思考が中断され初心者にとっては使いにくい．また，計算機の構造・動作を理解させるために適当なレベルの部品が用意されていないなどの問題がある．この他，画面

上で計算機内部の動作をシミュレート出来るものとしてVEGA[22]，INSIST[20]等がある．しかし，VEGAは固定した計算機を対象としたものであり自由に計算機を構成できず，INSISTはCADシステムのプロトタイプ作成用でありベースとなるSmalltalk-80[17]に関する深い知識を必要とするなど，上記の目的として適当なものは見当たらなかった．そこで，本研究では，これらの点を考慮し，図的ユーザインタフェースを重視した計算機シミュレータを新たに提案し，その機能および実現法について述べている[関連発表論文 (3),(7),(9),(11),(12)]．作成したシミュレータは，以下の特徴を持つ．(1) 画面上でマウスを使って部品を配置，配線することにより計算機回路を入力できる．(2) 計算機回路の入力中でもシミューレションが可能で，構築画面と同一の画面上でその動作を視覚的に観察することが出来る．(3) 計算機を構成するため高機能な部品が予め用意されているので，低レベルの回路から作り始めなくてもよく，また，部品の定義も比較的簡単に行なえる．また，作成したシミュレータを用いて実際に計算機回路の入力例を挙げ，比較的簡単に入力できたこと，部品定義機能が有効であることなどを示している．

静止した図形だけでなく，動きのある図形，すなわち，アニメーションを情報の表現手段として用いることは，静止した図形のみに比べて，より多くの情報を伝えることができる．コンピュータによるアニメーションの研究は，個々のフレームを前もって作成し，それを1フレームずつビデオに収録しておくという方法に関するものが中心であった[47],[52]．しかし，ユーザインタフェースとしてのアニメーションを考えた場合，ユーザからの入力によってアニメーション実行順序を変えることができるような対話性が必要であろう．また，アニメーションに登場する各オブジェクトの動きは，例えば,CAIの教材として物体の運動の様子をアニメーションで表現するような場合には，画面内での物体の速度を指定できる機能が必要である．オブジェクトの大きさや傾きが変化したり多くのオブジェクトが同時に動くような場合，一般にフレームに表示すべき画像の計算や表示に時間がかかるので，決められた時間内 (例えば1/10秒) にフレームの更新を行なうには，そのための画像情報等 (実行時情報と呼ぶ) を予め用意しておかなければならないが，対話型のアニメーションでユーザ入力があったとしてもなるべく利用できるような形で実行時情報を用意しておくことが望ましい．また，実行時情報を作成する際に，計算機の違いを意識せずに済むことや，実行時情報の量が少ないことも望まれる．5章では，汎用ワークステーション上での対話型の簡易アニメーションに対して，これらの要求を満たすような実行時情報の生成法，その実行時情報を用いたアニメーションの実行法などを述べている[関連発表論文 (4),(13)]．アニメーションの実行時のフレームの更新方法は，実行時情報がなくても決められた時間内に表示できる区間は，その場で各オブジェク

トの位置, 大きさ・傾きを計算し, 各オブジェクトを重なりの順に描く. 実行時情報がないと決められた時間内に表示できない区間については, そのときの大きさ, 傾きをもつ各オブジェクトの画像情報を生成・保存しておき, それを用いて重なりの順に描く. それでも間に合わない区間に対しては, フレーム間の画像の差分を生成・保存しておき, それを用いて表示する. それらの実行時情報は, アニメーションの実行に先だって, アニメーションを試行することによって, 計算機の能力に応じて自動的に生成される. また, この方法を用いたアニメーションシステムを試作し, 実験によって, その方法の有効性および限界を確認している.

関連発表論文

(1) Makoto NAKAMURA, Toshio MATSUURA, Hajime NAOTA, Jun SAKAGUCHI, Nobuo YOSHIOKA, and Yuji MATOBA: "An Interactive Drawing Tool on UNIX and its Figure Description Language," *Proc. 1988 Joint Technical Conference on Circuits/Systems, Computers and Communications*, pp.A2-2-1~6, Seoul, Korea (Nov. 1988).

(2) 松浦敏雄, 直田創, 中村眞: "図形の部品化および接続包含関係の保存機能を持つ作図ツール Key3", 電子情報通信学会論文誌 *(D-I)*, Vol.J73-D-I, No.11, pp.864-872 (1990-11).

(3) 吉岡信夫, 松浦敏雄, 的場裕司 : "画面上で計算機の構築が可能な計算機アーキテクチャ教育用シミュレータ GCS", *CAI* 学会論文誌, Vol.8, No.2, pp.80-89 (1991-06).

(4) 松浦敏雄, 梶本雅人, 谷口健一: "ワークステーション上での実時間アニメーションシステムとその評価", 電子情報通信学会論文誌 *(D-I)*, (投稿中).

(5) 松浦敏雄, 中村 亨, 谷口健一, 増田 澄男: "概形表示および部分拡大表示機能を有する木構造グラフエディタの作成とその評価", 電子情報通信学会論文誌 *(D-I)*, (投稿中).

(6) 北村 俊義, 松浦敏雄, 吉岡信夫, 的場裕司 : "図形間の接続・包含関係を考慮した作図システム", 情報処理学会第 *31* 回 *(*昭 *60* 年後期*)* 全国大会, 7G-10, pp.1507-1508 (1985-10).

(7) 杉本直樹, 阪田全弘, 松浦敏雄, 吉岡信夫, 的場裕司 : "画面上で計算機の構築可能な教育用計算機シミュレータ", 情報処理学会第 *33* 回 *(*昭 *61* 年後期*)* 全国大会, 4W-2, pp.2371-2372 (1986-10).

(8) 北村 俊義, 松浦 敏雄 : "インタラクティブな作図ツール key3 の作成", 8th UNIX Symposium Proceedings, pp.29-34 (1986-11).

(9) 杉本直樹, 松浦敏雄, 吉岡信夫, 的場裕司 : "Smalltalk-80 上でのウィンドウ制御クラス —DOWLAND—", 日本ソフトウェア科学会ソフトウェア研究会, SW-87-2-2, pp.9-16 (1987-07).

(10) 中村 眞, 松浦敏雄, 吉岡信夫, 的場裕司 : "図形記述言語 keyfig の機能とその PostScript への変換", 情報処理学会第 *35* 回 *(*昭 *62* 年後期*)* 全国大会, 7Dd-8, pp.2661-2662 (1987-09).

(11) 杉本直樹, 高田司郎, 松浦敏雄, 吉岡信夫, 的場裕司 : "Smalltalk-80 上での高機能なウィンドウ制御クラスの作成", 情報処理学会第 *35* 回 *(*昭 *62* 年後期*)* 全国大会, 1R-5, pp.811-812 (1987-09).

(12) 小池田恒行, 松浦敏雄, 杉本直樹, 吉岡信夫, 的場裕司 : "論理回路から簡単な計算機レベルまで適用可能な教育用シミュレータ", 情報処理学会第 *35* 回 *(*昭 *62* 年後期*)* 全国大会, 3Ee-6, pp.2673-2674 (1987-09).

(14) 梶本 雅人, 松浦 敏雄, 谷口 健一 : "X Window 上での実時間アニメーション法", 情報処理学会第 *42* 回 *(*平 *3* 年前期*)* 全国大会, 7P-2, 分冊 **2**, pp.365-366 (1991-03).

(15) 中村 亨, 松浦 敏雄, 谷口 健一 : "木構造データの可視化ライブラリの作成と使用例", 情報処理学会第 *43* 回 *(*平 *3* 年後期*)* 全国大会, 1P-2, 分冊 **5**, pp.103-104 (1991-10).

2章

図形の入力編集方法

2.1 序言

ビットマップディスプレイやマウスを備えたワークステーションの普及に伴ない，ワークステーション上でインタラクティブに作図できるツールも登場してきた[4],[5],[6],[7]．これらのツールは画面上で比較的簡単な操作で図を描けることから，急速に普及しつつある．

インタラクティブな作図ツールでは，通常，アイコンとして用意された長方形や線分等の図形部品を用いて作図を行なうが，これらの基本図形のみを用いて作図するのは容易ではない．このため従来の作図ツールにも，既に描かれた図形群をひとまとめにして，新たな図形部品として再利用できるものがあった[7]．しかし，画面上で描かれた図形を手軽な操作で部品として定義できる機能に加えて，図形記述言語を用いて柔軟性に富んだ図形部品の定義機能を備え，かつ，いずれの方法で定義した部品も基本図形と同様の簡単な操作方法で利用できるものはなかった．

本論文では，このような部品定義機能を持つ作図ツールを提案している．直接操作と記述言語の両方のアプローチを，文書の作成に対して試みた例としてVoTeX[8]があるが，このような2つのアプローチを作図の面で実践した例はない．

また，図を描く場合，特に技術論文や設計書等の図では，長方形や楕円の中に文字や小さな図形を描き，これらを線図形で結ぶような図を描くことが多い．このような場合，意味的に関連のある図形群はまとめて扱いたいし，線図形で結んだものは，それらの接続関係を移動や拡大縮小に際し保存したいことが多い．従来の作図ツールにも，図形群をまとめて取扱うために，グループ化機能等[4]があったが，操作対象の図形群を明示的に指定しなければならなかった．また，接続関係を保存する機能を持つ汎用の作図ツールはなかった．

本論文では，図形間の接続・包含関係を作図ツール側で自動的に認識し，特別な

操作なしにこれらの関係を保存して移動，拡大・縮小などの操作ができるような仕組みも提案している．

さらに，実際に，上記の部品化機能および接続包含関係の自動認識機能を含む作図ツール Key3 を作成し，実験によりこれらの機能の有用性を実証すると共に，その実現法が妥当であることを明らかにしている．従来，このような作図ツールにおける操作性の向上を目指した研究はあまり行なわれていなかった．

以下，**2.2**では，本研究で作成した作図ツールの概要ならびに従来の作図ツールの課題分析とその解決法を述べ，**2.3, 2.4**では，その実現方法に触れる．さらに，**2.5**では，提案した機能の有用性および実現方法の実行効率等について論じる．

2.2 課題分析とその解決法

2.2.1 作図ツール Key3

まず，言葉の定義を兼ねて，作図ツール Key3 を簡単に説明する．

Key3 は図を描くための CanvasWindow(CW と略す)，アイコンを表示・選択するための IconWindow(IW)，および，CW の状態や図形の属性等を表示・変更するための StatusWindow(SW) から構成される (図 2.1)．

作図方法は，描きたい図形部品に対応するアイコンを IW から選び，必要ならば線の太さや文字フォント等の属性を指定し，CW 上の適当な位置で，マウスをクリックもしくはドラッグする．クリックした時にはディフォールトの大きさの図形が描かれ，ドラッグした時はドラッグによって決まる長方形の幅と高さに合わせて図形が描かれる．あらかじめアイコンとして用意している基本図形としては，線分，折れ線，楕円弧，スプライン曲線 (以上線図形と呼ぶ)，長方形，楕円，多角形，および，文字列がある (以上面図形と呼ぶ)．

2.2.2 図形部品の定義機能

作図用の部品化機能として満たされるべき条件は，

(1) 新たな部品が容易に定義できること，
(2) 種々の部品が定義できること，
(3) 必要な部品集合を取捨選択できること，
(4) 多数の部品を同時に参照できること，

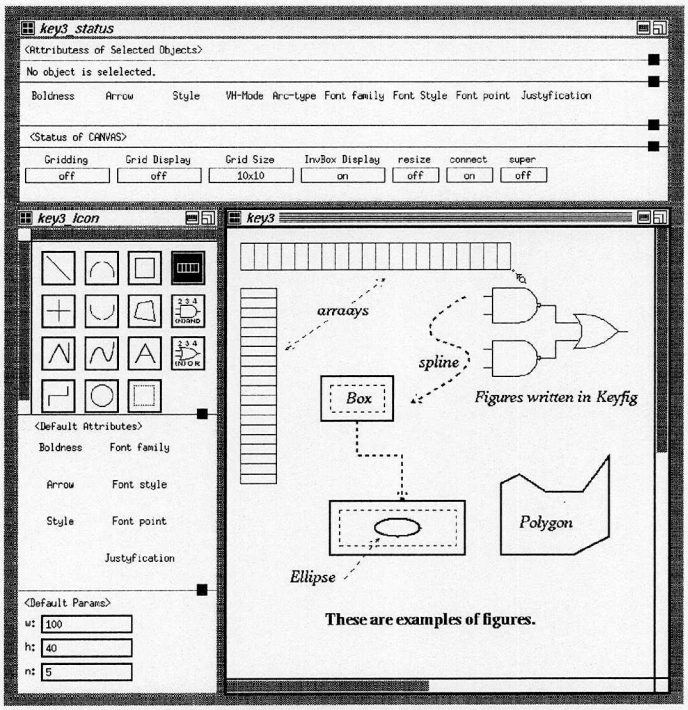


図 2.1: **Key3** の画面表示例

等が重要であろう.

図形部品の作成方法としては，画面上で描いた図形群をそのまま部品にする方法（直接操作による部品化と呼ぶ）と，図形記述言語を用意し，それによって記述された図形を部品にする方法（図形記述言語による部品化と呼ぶ）が考えられる．前者は，条件 (1) を満たし，後者は言語によってパラメータや変数を用いて柔軟に部品を定義できることから条件 (2) を満たす.

Key3 では，条件 (1) および (2) を満たすために，両方の部品化方法を実現している．さらに，Key3 では部品群のセーブ・ロード，消去を可能にすることで，作図したい図の種類に応じて必要な部品集合のみを IW に置けるようにしている (条件 (3))．また，ウィンドウの大きさを可変とすることで，多数のアイコンを表示できるようにしている (条件 (4))．さらに，部品の配置も指定できるようにし，部品の検索も容易にしている.

2.2.3 図形間の接続・包含関係の認識と保存機能

このような図形間の接続関係あるいは包含関係を取り扱うときに満たされるべき条件は,

(1) 図形間の関係を指定する手間が少ないこと,

(2) 望みの図形を容易に選択できること,

等が重要であろう.

その実現方法として,これらの図形間の関係を,(1) 明示的に指定させる方法,および,(2) 自動的に判断する方法が考えられる.方法 (1) では,図形間の関係はユーザの意図通りにできるが,関係を指定するための手間が無視できない.方法 (2) では,関係を指定するための操作を必要としないが,ユーザの意図しない関係を認識してしまう可能性がある.

既存の作図ツールでは,MacDraw のグループ化機能など,方法 (1) によって図形群をひとまとまりとして扱う機能などが提供されている.また,方法 (2) によって図形間の関係を認識するものが CAD や CASE ツールの中にあるが[9],[10],汎用の作図ツールにはなかった.CAD や CASE ツールの場合,一般に,対象となる図形の種類や図形群に対する操作が限定されているので,その実現は比較的容易であるが,汎用の作図ツールの場合は,取り扱う図形の種類も多く,また,上述したユーザの意図しない関係を認識した場合にも対応しなければならないなど,その実現が容易ではなかった.

これに対して,Key3 では,図形の選択方法の工夫などによってこの問題点を解決し (**2.4.2**参照),図形間の接続関係並びに包含関係を自動的に認識し,ユーザの特別な操作なしに,包含関係にある図形を一括して指定でき,図形の移動,拡大・縮小に際して,これらの関係を保存する機能を提供している.

2.3 部品化機能の実現

本節では,まず,図形記述言語による部品化を実現するために導入した新たな図形記述言語 Keyfig について述べる.その後,Keyfig を用いた部品化および直接操作による部品化の実現法について述べる.

2.3.1 図形記述言語 Keyfig と図形部品の記述

図形部品を記述するための言語として,既存の PostScript[11](PS と略す) や,troff[12] のプリプロセッサとしての PIC[13] などを用いることもできる.しかし,PS は逆ポーランド記法を用いなければならないなど,プログラムの読み書きが容易でない.また,PIC は楕円弧を描いたり,点線で曲線を描くなどの troff にない機能は実現されておらず,図形の記述能力が十分でない.そこで,Key3 の図形部品を記述するために,新たな言語 **Keyfig** を設計し,その処理系 Keycom(**2.3.3**参照) を作成した.

Keyfig はC言語とほぼ同じ構文を持っており，C言語では特別な関数 main の本体が実行されるのと同様に，Keyfig では main と呼ばれる図形定義の本体部で記述された図を描画する．Keyfig は Key3 の図形部品を記述するだけでなく，汎用の図形記述言語としても利用できる様に設計しており，Key3 で描いた図全体をファイルに保存する際にも用いている．さらに，Keyfig から PS や PIC への変換フィルタを作成しているので，プリンタへの出力や troff の文書中に図の挿入ができる．

Keyfig での図形部品の定義は，次のようにC言語の関数定義に似ているが，仮引数の省略値を指定できる．

図形名 (仮引数: 省略値, ...) { 図形定義本体部 }

図形定義本体部では，基本図形もしくは他の図形定義を関数呼出しのように参照して図形を記述する．記述には，仮引数，および，繰返し制御や計算結果を保持するための変数などを用いることができる．このように定義された図形部品は Key3 のアイコンにでき，図形定義に対する実引数は，その図形の描画時に与えることができる (**2.3.2**参照)．Keyfig による図形部品 **"array"** の定義例を以下に示す．

```
array(w : 40, h : 20, n : 3){
        /* w : 幅, h : 高さ, n : 箱の数 */
  int i;
  Box (w, h) [BOLD] at (0, 0); /* 全体の長方形 */
  if ( w > h ) {      /* 横長の場合 */
     for (i = 1; i < n; i + +)
        Line from (w/n * i, 0) to (w/n * i, h) ;
  } else {            /* 縦長の場合 */
     for (i = 1; i < n; i + +)
        Line from (0, h/n * i) to (w, h/n * i) ;
  }
}
```

これは長方形をその短い方の辺に平行な線分で，指定した数の長方形に等分割した図形 (**配列図形**と呼ぶ) を定義している (図 2.2)．

2.3.2 部品化の方法と部品の使用法

Key3 において部品化とは，画面上の図形群または Keyfig で記述された図形定義を IW 上のアイコンにすることであり，アイコンになった部品は基本図形のアイコンと同様の操作で利用できる．

図 2.2: "array" 部品の描画例

直接操作により部品化するには，CW で部品化したい図形群を選択し，部品化するためのメニューコマンドを選び，図形部品名を入力すればよい．図形記述言語により部品化するには，IW のメニューコマンド (LoadIcons) を選び，Keyfig で定義した図形の記述ファイルの名前を入力すればよい．

部品化された図形を CW 上で描画するとき，その図形に対する実引数を決定しなければならない．この方法を以下に示す．ユーザがアイコンを選ぶと，IW のサブウィンドウ DefaultParams(DP と略す) に，その図形の持つ全ての引数の名前とそのデフォールト値を表示する．通常，第 1，第 2 引数は図形の幅と高さであり，直接操作によって作られた部品の引数はこの 2 つのみである．ユーザはこれらの値をその場で変更でき，その後，CW 上でマウスをクリックすると，DP で指定された値を実引数として図形を描画し，マウスをドラッグすると，さらに，第 1，第 2 引数をドラッグ操作によって決まる長方形の幅と高さで置換えた図形を描画する．

2.3.3 部品化に対する内部処理

Keyfig による図形記述は制御構造や変数を含んでいるので，それから図形を描くにはコンパイルに相当する処理を要する．そこで，制御構造や変数を含まない，より単純な図形記述形式 Kif を導入し Keyfig から Kif への変換プログラム Keycom を用意している．Kif 形式は，長方形・線分などの基本図形の識別子，図形の位置，大きさ，属性などを表す単純な系列からなるので，Kif から PS，PIC，および，ウィンドウシステムの図形描画関数等への変換は容易である．Keycom はインタプリタのように動作し，Keyfig による図形定義を逐次的に受けとることができ，構文解析等を行ない内部データとして格納する．また，格納した図形定義に対して，実引数を伴った Kif への変換要求を受けると，その図形の Kif 形式の記述を返す．

図形記述言語による部品化の処理では，Key3 は Keyfig による図形定義を Keycom に渡す．さらに Key3 はアイコンの絵を描くために，Keycom に対してアイコンの大きさを実引数として Kif への変換要求を出し，返されてくる Kif の記述を基にア

イコン上に図形を描画する[1].

直接操作による部品化では，Key3 は画面上で選択された図形群に対する Keyfig の記述を生成し，その後は，記述言語による部品化と同様の処理を行なう.

部品化された図形の描画時には，既に Keycom が各部品の情報を内部データとして保持しているので，Key3 は実引数を Keycom に渡し，Keycom から返された Kif 形式の記述を基に画面上に描画する.

2.4 接続包含関係の自動認識および保存機能の実現

2.4.1 接続包含関係の認識

まず，Key3 において，接続関係をどのように認識しているかについて述べる. 線図形 A の端点が図形 B の線上（線図形のとき）または境界線上（面図形のとき）にあるとき，線図形 A は図形 B に接続している．しかし，線図形の端点を目標点に正確に一致させなくても，その近傍で指示するだけで接続しているとみなした方が一般に操作が容易である．Key3 では，この距離 Δ を 3 ドットとしている．(Δ を変化させて目標点の選択に要する時間を，Sun3/75m の 19inch 白黒モニタ上で測定した結果，Δ が 2 ドット以下ではこの時間が急増し，操作しにくいことがわかっている[16].) また，Key3 では，MacDraw などと同様にグリッディング機能（描画点を最も近くの格子点の座標に合わせる機能）を持っており，この格子間隔を 1, 5, 10, 20 ドットの中から選ぶことができ，いつでも変更できる．この機能により，接続対象の図形が格子線分上にある場合（このような図を扱うことが多い），これらの図形との接続操作はさらに楽に行なえる.

次に，包含関係の認識について説明する．図形あるいは点 **f** が面図形 **g** に含まれているとき，もし，**g** に含まれかつ **f** を含むような面図形が存在しなければ，**f** は **g** に**直接含まれる** (**g** は **f** を**直接含む**) ということにする．一般には，**f** を直接含む図形は複数存在するが，技術論文や設計書等の図では，直接含む図形は一つであることが多く，また，たまたま複数個あったとしても論理的には，一つをその "**親**" として固定して扱った方が好都合であることも多い．また，図の編集のし易さを考慮した場合，図形の移動等によって "**親子**" 関係が変わってしまうよりも，むしろそれぞれの時点で**親子**関係が固定されていた方が都合が良いこともある．そこで，Key3 では各図形に対してそれを直接含む図形の内の一つを**親**として扱っている．複数の図形に直接含まれるような場所で新たな図形を描いたとき，もしくはそこに図形を移

[1]アイコンの絵は，定義された図をそのまま小さく描く以外に，予め用意した bitmap を表示させることもできる.

動させたとき (正確にはそれらの選択を解除したとき), その図形は, それを直接含む図形の中で最も新しくその場所に描かれた図形の "子" となるように決めている. 本論文で言うところの包含関係とは, この親子関係のことを指す. 親子関係を入換えたい場合には, 新たに親にしたい図形を選択し, 続いて子の図形を選択するだけで良い (**2.4.3**).

2.4.2 図形の選択と移動, 拡大・縮小

包含関係にある図形の一括選択機能の実現に際して, 接続包含関係にある図形の選択が容易であることに加えて, 個々の図形を独立して扱うことも容易にでなければならない. このために, Key3 では, 以下のような図形の選択方法, ならびに図形の移動, 拡大・縮小の方法を提供している.

Key3 では, 基本的には, 線図形の線上または面図形の境界線上でクリックすると, その図形だけを選択する. 図形の内部でクリックすると, その点を直接含む一つの面図形と, その面図形のすべての子の図形が再帰的に (すなわちすべての子孫が) 選択される.

マウスをクリックした位置が複数の図形の線上や複数の面図形の内部の場合などでは, 選択されるべき可能性のある図形もしくは図形群が複数存在することがある. このような場合, 選択の候補となる図形 (群) を選択候補リストにつないでおき, その第 1 候補の図形 (群) を選択する. 同じ点でマウスをクリックすることで候補の図形 (群) の中で選択を切替えることができるようにしている. 候補図形 (群) の中では, 線図形の選択を優先し, 面図形同士では, 単独の選択を優先している.

選択した図形群に対して, それらを移動させたい位置までマウスをドラッグすると, マウスの移動に応じて選択された図形群が移動する. このとき, 通常これらに接続している線分はその接続関係が保存されるが, Disconnect キー (通常はシフトキー) を押しながらマウスをドラッグすると, 接続関係は保存されない. また, 図形の選択後, Resize キー (通常はコントロールキー) を押しながらマウスをドラッグすると, マウスの移動量に応じて図形の大きさが変化する. このときも Disconnect キーを併用することが出来る. なお, 図形の移動, 拡大・縮小のためのドラッグ操作では, マウスの押下げ時に図形の選択も行なうので予め図形を選択しておく必要はない.

2.4.3 図形間の関係を表すデータ構造と処理方法

図形間の接続・包含関係を実現するためのデータ構造, および, それを用いた接続包含関係の自動認識および保存機能の実現方法について述べる.

前節で述べたように，Key3 では，どの図形の親も一つに限定しているので，親子関係を有向グラフで表現すると木構造 (**図形木**と呼ぶ) になる (図 2.3)．図形木を更新するのは，図形が選択された時と，その選択が解除された時のみである．

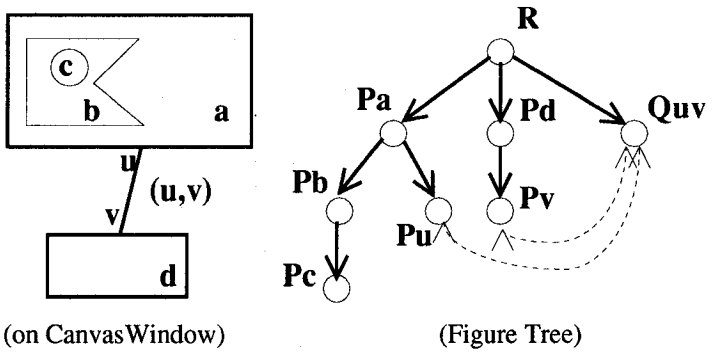


図 2.3: Key3 のデータ構造

包含関係にある図形群が選択されたとき，それらに対応する部分木を図形木から切り離す．選択された図形群の移動や拡大縮小が行われると，対応する部分木の各節点の位置や大きさの情報のみを変更し，それらの操作が終了しても選択を解除しない．他の図形が選択されたとき，もしくは，新たな図形が描かれたとき，元の選択を解除する．このとき図形木の根から幅優先順に (同じ深さの節点に対しては，後から描かれた図形を先に)，各節点の表す図形が (選択が解除された) 部分木の根の表す図形を含むかどうかを調べ，直接含む場合には，その節点の子節点として部分木を付加する．どれにも含まれない場合には，根の子節点として付加する．

接続関係の保存も，図形木を用いて実現できる．線分 **(u, v)** の両端点 **u, v** に対し，包含関係に従って図形木の節点 **Pu, Pv** を構成する．線分 **(u, v)** を表す節点 **Quv** のデータは，**Pu, Pv** へのポインタを含む．**Pu, Pv** では，自分自身の座標データと **Quv** へのポインタを保持している．図 2.3において，図形 **a** を移動させその選択を解除したとき，**Pu** は **Pa** の子節点であるので，**Pa** の移動にともなって **Pu** も移動する．このとき，**Pu** から **Quv** を見つけだし，移動に伴うデータの更新を行い，画面上の線分 **(u, v)** の絵を描き直すことで接続関係の保存を実現している．

2.5 評価および考察

本論文で提案した機能およびその実現方法の評価実験およびその結果等について述べる．特に断らない限り，以下の評価実験は，Sun3/80(メモリ 8MB) 上で X Window[14]のサーバを走らせ，クライアント (Key3) を別の Sun3/60(メモリ 8MB) 上で稼働させた．

2.5.1 新たな機能の有効性の評価実験

提案した個々の機能の有効性を定量的に一般に議論するのは難しいので，ここでは，それぞれの機能が利用できる場合とそうでない場合に，実際に作図し，操作回数および所要時間を調べた．また，参考としてMacDraw(MacII上で稼働)でも実験を行なった．なお，被験者が操作ミスをした場合，時計を止めてやり直しているので，操作ミスによる時間のロスはない．また，描画方法は幾通りもあるが，最適と思われる方法を用いている．

部品化機能の有効性

図形記述言語を用いた部品化機能の有効性を示すために，一例として図2.4(a)の描画に要する操作回数と所要時間を，Key3で基本図形のアイコンのみを用いて描いた場合，図形部品arrayを用いた場合，および，MacDrawの場合について調べた(表2.1)．

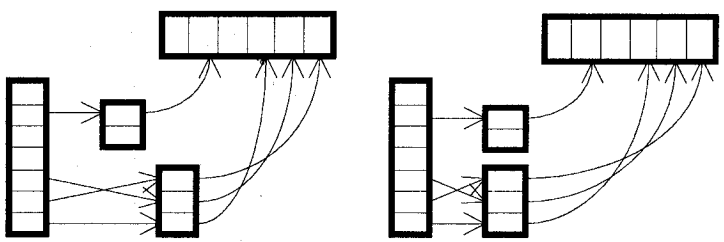


図 2.4: 作図例1("array" を用いた図)

図2.4(a)中には4つの配列図形(図2.2)が描かれているが，array部品を用いずにこれらを描くには，長方形とそれを等分割するための線分を正確に描かなければならず，簡単には作図できない．そこで，簡単のためにこの実験では，長方形の長辺を格子間隔の整数倍の長さに合わせて，配列図形を描いている．4つの配列図形の描画は，分割数がこの程度(2から7)の場合でも，測定データより，array部品を用いた方が半分ぐらいの時間で描画できる．配列図形の分割数が更に大きくなれば，当然この差は開く．MacDrawの場合の操作回数は，array部品を用いない場合のkey3とほぼ同じである．

表 2.1: 図 2.4(a) の作図時間

	Key3				Macdraw	
	array 部品使用		基本図形のみ			
	操作回数	時間	操作回数	時間	操作回数	時間
4 個の配列図形の描画	(1, 4,4,0)	25 秒	(3,17,0,0)	42 秒	(2,18,0,1)	61 秒
8 本の矢線の描画	(2, 8,0,0)	28 秒	(2, 8,0,0)	28 秒	(2, 8,0,2)	32 秒
図 2.4(a) 全体の描画	(3,12,4,0)	53 秒	(5,25,0,0)	70 秒	(4,26,0,3)	93 秒

ただし，操作回数欄の 4 つ組 (C, D, K, M) は，左から順に，マウスのクリックの回数，ドラッグの回数，キー入力の回数，メニューコマンドの実行回数である．表中の時間は，被験者 5 名の平均である．表 2.2，表 2.3でも同様．

接続包含関係の自動認識・保存機能の有効性

接続関係の自動認識・保存機能の有効性を確かめるため，Key3 でその機能を用いる場合とそうでない場合，および MacDraw の場合について，図 2.4(a) から同図 (b) にレイアウトの変更を行い[2]，その操作回数と所要時間を測定した (表 2.2).

表 2.2: レイアウト変更時間 (図 2.4(a) から同図 (b))

	Key3				Macdraw	
	接続関係保存		接続関係保存なし			
	操作回数	時間	操作回数	時間	操作回数	時間
図 2.4(a) → (b)	(0,1,0,0)	1.6 秒	(0,9,0,0)	15 秒	(9,9,0,0)	23 秒

Macdraw の場合，線分の変形操作を行うには，一旦線分をクリックした後で，線分の端点を摘んでドラッグする必要があるが，クリックとドラッグの始点を同じ点 (線分の端点) にすれば，一回のドラッグと同程度の手間で操作できる．従って，ここではこれを一回のドラッグとみなしている．

接続関係の保存機能を用いた場合，この変更は 1 回のドラッグ操作でできるのに対して，保存機能を用いない場合の Key3，および，MacDraw の場合には，それぞれ 9 回のドラッグ操作が必要である．このことから，図 2.4のレイアウトの修正において接続関係の自動認識・保存機能が有効に働くことがわかる．

[2]中央下にある長方形とそれに接続された線分を左方向に移動させる．

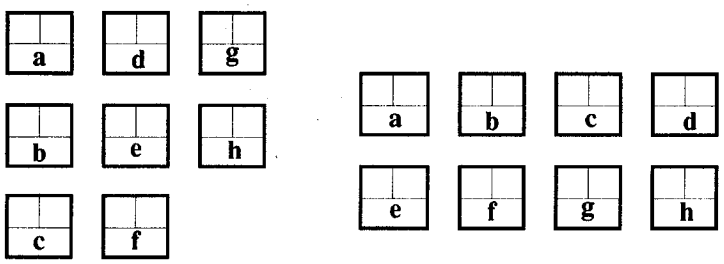


図 2.5: 作図例 2(レイアウトの変更)

表 2.3: レイアウト変更時間 (図 2.5(a) から同図 (b))

	Key3				Macdraw	
	接続関係保存		接続関係保存なし			
	操作回数	時間	操作回数	時間	操作回数	時間
図 2.5(a) → (b)	(0,8,0,0)	19 秒	(0,16,0,0)	36 秒	(0,16,0,0)	39 秒

次に，包含関係の自動認識の有効性を調べるため，図 2.5(a) から同図 (b) へのレイアウトの変更に要する時間を測定した (表 2.3)．包含関係の保存機能を用いる場合 8 回のドラッグ操作で変更できる．一方，包含関係の保存機能を用いない場合は，一括選択機能を用いて移動対象の図形群を選択した後に移動させるのが最善の方法であり，この場合でも合計 16 回のドラッグ操作が必要となり，より時間がかかっている．この時間差は包含関係が複雑に入り組んでいるような場合には，一般にさらに大きくなる．

なお，MacDraw の実験では，グループ化の機能を用いず一括選択機能を用いて図形を選択した後に移動させた．グループ化機能を用いる方法もあるが，その場合はグループ化するための操作が必要となり，この例のように 1 回きりの移動の場合には，グループ化しない方が速く操作できる．

2.5.2 新たな機能の処理時間

ここでは，新たな機能の内部での処理時間についての実験結果を示す．なお，この実験も **2.5.1**と同じ計算機を用いているが，**2.5.1**の測定結果は，ユーザの操作時間が主なものであり，CPU の速さにはほとんど影響されないのに対して，本節で得られたデータは，CPU のスピードにほぼ比例する．

部品化の処理時間

直接操作による部品化の処理時間を調べるため，線分，長方形，楕円について，それらを複数個並べた図形群をアイコンにするのに要した時間 (**Ticonify**) [3]を測定した．また，図形記述言語による部品化についても，同様の図形群の記述ファイルをアイコンにする時間 (**Tloadicon**)[4]を測定した (図 2.6)．この時間は，いずれもアイコン名，もしくは記述ファイル名を入力してから，部品が作られ，そのアイコンが表示されるまでの時間である．ここで，線分の長さは 120(単位はドット数．以下同様)，長方形の大きさは 40x20，楕円は長軸 40 短軸 20 の図を描いているが，部品化やその部品の描画時間は図形の大きさにはあまり依存しない．

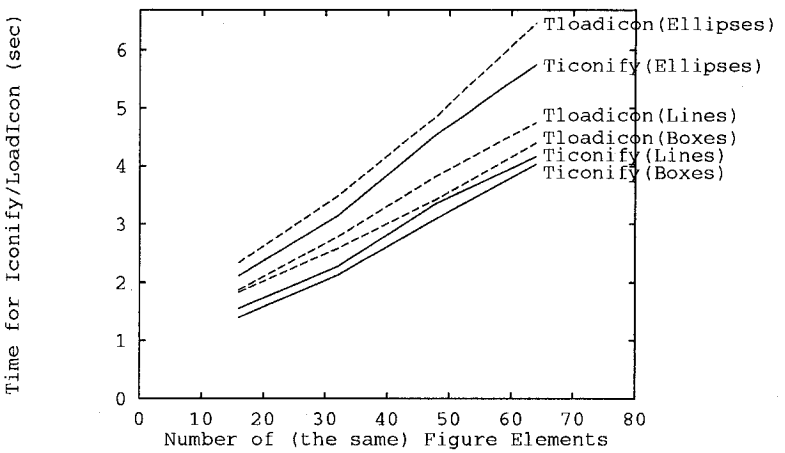


図 2.6: 直接操作による部品化に要する時間

いづれの部品化に要する時間も，その図形に含まれる図形部品の数に比例するが (図 2.6)，今までの使用経験から，通常，実際に図形部品に含まれる基本図形の数は 20〜30 程度以下であることが多く，このような範囲では，約 3 秒以下で部品化できることがわかる．

アイコンになった図形部品を CW 上で描画するのに要する時間 (CW 上でマウスをクリックしてから実際に絵が現われるまでの時間) を図形種類毎に調べた結果を図 2.7の **Tdraw** で示す．図 2.7から，図形部品の描画時間 Tdraw は，図形部品の数が 20〜30 程度の場合は 2 秒以下であることがわかる．Tdraw は，図形記述の実引数を keycom に与えてから keycom が Kif による記述を返すまでの時間 (**Tkif**) と

[3]選択された図形の Keyfig による記述の生成時間, Keyfig から Kif への変換時間, および, Kif を基にアイコンの絵を描く時間を含む.

[4]Keyfig から Kif への変換時間, および, Kif を基にアイコンの絵を描く時間を含む.

Kif から実際に画面上に図を描くまでに要する時間 (**Tdisp**) の和である．Tdisp の測定結果も図 2.7に示す．図 2.7で図形の種類毎の時間差 Tdraw-Tdisp (この時間差は Tkif に等しい) を調べると，どの場合も 1 秒以内である．

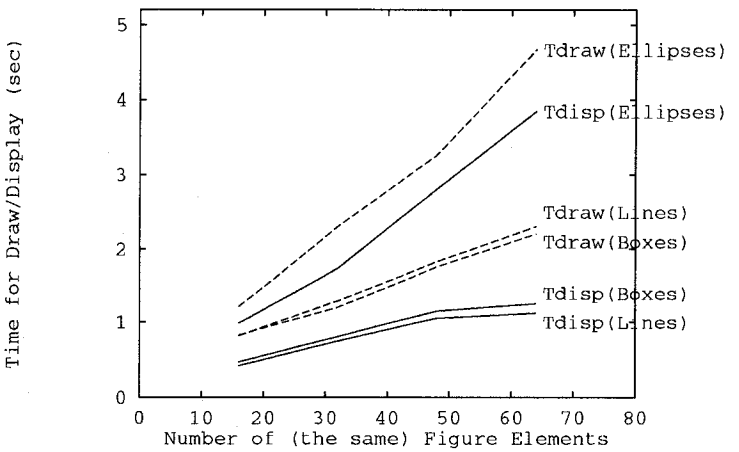


図 2.7: 図形部品の描画時間

接続包含関係の自動認識・保存機能の処理時間

接続包含関係にある図形を移動させた場合，その時間は，移動前の図形の消去時間，図形の移動時間，および移動後の図形の描画時間の和である．図形の消去と図形の描画の内部処理は同じであるので，その処理時間はほぼ等しい．図形の移動中はマウスの動きに追従した描画ができるように，簡略化した図表示にしているので，移動時間は，ユーザのマウスの操作時間によって決まる．従って，マウスの操作時間を別にすると，全体の移動時間は，図形の描画時間 (Tdraw:図 2.7) の 2 倍になる．例えば楕円を 50 個含むような図形部品を移動させるのに要する時間は，図 2.7から (マウスの移動時間+6 秒) となることが分かる．

2.5.3 新たな機能を実現したための影響

ここでは，新たな機能を実現したために，従来の機能に対して与えた時間的な影響について述べる．

接続包含関係を自動的に認識するために，一般にデータ構造が複雑になり，図形の選択や移動などに伴うデータ構造の更新時間が余分にかかる可能性がある．Key3 では，図形木が更新されるのは図形の選択または解除が起きた場合だけである．図形木の更新に多くの時間を要する場合は，頂点数の多い多角形が多数，入れ子状に

存在しているところに，頂点数の多い多角形の選択を解除したときである．実際に，50角形を20段入れ子状に配置し，最も時間のかかる一番内側の図形の選択の解除に要した時間は2.7秒であった．なお，この時間は入れ子の段数に比例し多角形の頂点数の2乗に比例する．

2.6 結言

作図ツールの作図効率を向上させるための機能を提案し，それらの機能を含む作図ツールKey3を作成した．さらに，ここで提案した機能の有効性について，代表的な例について，操作回数，および，それに要する処理時間を実測した．それらの結果は，いずれも，機能の有効性および処理方法の妥当性を示している．

Key3はC言語で記述し，その大きさは約15000行である．またKeycomは，yacc，lex，Cを用いて記述し約3500行である．Key3の主な機能は，X Windowシステムにおけるツールキット(Widget)として実現しており，他のアプリケーションプログラムから図形の入出力用部品としても利用できる．

ここで提案した機能は，例えば，論理回路図や流れ図などを描く場合に特に有効である．これらの図では，それぞれ特有の図形部品（論理回路部品や長方形，菱形など）が繰り返し利用されることから，部品化機能が有効であり，また，それぞれの図形部品が線分，折れ線などで接続されているため，接続包含関係の自動認識機能が有効に機能する．また，本論中では触れなかったが，線分には垂直・水平方向のみにしか描けないという制限を持たせることができ，接続関係を保存した移動の際にもこの制限は保存される．従って回路図や流れ図のようにほとんど水平・垂直線分だけからなる図の場合の修正作業も容易である．しかし，これらの機能が作図作業全体から見た場合にどの程度有効に働くかを定量的に計ることによって，総合的な評価を行うのは今後の課題である．

3 章

木構造データの図的表示とその上での入力法

3.1 序言

グラフは様々なアプリケーションプログラムで用いられるデータ構造である．グラフを理解するには，一般に，図示した方が分かりやすい．ワークステーション技術の発展によって，グラフを画面上に描写したり，画面上での直接操作によるグラフに対する操作が可能になってきた．このような機能を備えたソフトウェアはグラフエディタと呼ばれている[25][26]．

グラフ構造の中でも，有向順序木(以下，単に木と呼ぶ)は最も基本的であり，かつ，よく用いられるデータ構造である．木構造データを扱うアプリケーションには，比較的大きな木（節点数，数百～数千程度）を扱う必要があるものも少なくない(例えば項書換えを利用する検証支援系[33])．しかし，木構造を対象としたグラフエディタでは，画面の大きさの制限と節点の配置の計算時間などの点から，多くの節点を持つ木にどのように対処するかが問題であった[26]．

この問題に対して，従来提案されているグラフエディタでは，“節点の抽象化”と称して，部分木を1つの節点で表現して表示すべき節点数を減らした描写などが考えられていた[25]．しかし，この方法では，木全体の構造が把握しにくく，抽象化された部分木内の節点を参照したい場合には，一旦，その部分木を元に戻して，代わりに他の部分木を抽象化して全体の節点数を増やさないようにして，観察する必要があった．

本研究では，比較的大きな規模の木構造データに対応できる木構造グラフエディタを実現するライブラリプログラム **VTM** を作成した．VTM では，木構造の全体の概形を画面上に表示し，同時に任意に指定した複数の部分を拡大表示できる機能

を提供している．また，木の更新操作に伴う配置変更を効率良く行なえる木構造の配置アルゴリズムも新たに開発した．

本論文では，VTMの機能及び実現方法を述べ，さらにアプリケーションプログラムからの利用法を例を用いて説明する．さらに，木構造データの大きさに対する，新しい配置アルゴリズムによる配置計算時間及び画面更新時間等の測定データを与え，本研究でのライブラリの実現法等の実用上の妥当性を示す．

3.2 要求分析と対策

3.2.1 木構造エディタの基本機能

一般に，木構造エディタとしては，以下の基本機能が要求される．

(1) 木構造データの保持・更新機能 —— 木構造のデータを保持し，アプリケーションプログラムからの節点及び部分木の付加・削除等の更新要求に従って，木構造データを更新する機能．
(2) 木構造の画面表示機能 —— 木構造データを画面上に美的に表示する機能．
(3) 画面からの入力機能 —— 画面上でのマウスによる直接操作によって，任意の節点の選択，及び，節点および部分木の追加・削除の指示等をアプリケーションプログラムに伝達する機能．[1]

3.2.2 実用規模の木構造を扱う上での問題点と対策

画面の表示面積の限界

ワークステーションの画面の大きさの制限のため，配置対象となる木の節点数が多くなった場合，木全体を分かり易く表示できない場合が生じる．そのような場合，一部を拡大して表示する拡大表示機能が有効になるが，この機能だけでは，木全体の概形が把握しにくい．また，拡大表示している部分が木全体のどの部分であるかが分かりにくく，さらに，注目したい節点が常に拡大表示されている領域にあるとは限らないなどの問題がある．

そこで，大きな木構造に対応したグラフエディタの表示方法に関して，満たすべき条件としては，

(1) 木の全体構造が把握しやすいこと，

[1]実際にその節点または部分木を追加・削除するかどうかは，アプリケーションプログラムが決定し，実行するなら，改めて木構造エディタに更新要求を出す．

(2) 木の任意の部分を拡大表示できること,
(3) 木の複数の部分を同時に観察できること,
(4) 望みの部分の表示が容易に得られること,

などが重要である. これらを考慮して, VTM では, 木全体の概形を画面上に表示し, かつ, 同時にその中の任意の (複数の) 領域を, (どの部分であるかを明示しつつ) 任意の大きさに拡大表示する機能を提供している.

木構造のどの部分を拡大表示するかについては, アプリケーションプログラムは何も指示する必要はなく, VTM の機能によってユーザが画面上でマウスを用いて自由にコントロールできるようにしている.

一般に, アプリケーションプログラムからの指示によって木構造が変化した場合, 注目している節点 (current node) が拡大領域に入らなくなってしまう可能性がある. このようなことが起こらないように, 注目している節点を常に拡大表示領域内に入れておくことができる機能も提供している.

画面の更新時間

木構造の節点や部分木が追加・削除された場合, (1) データ構造を更新し, (2) 各節点の位置を配置アルゴリズムによって計算し直し, その後, (3) 画面上の各節点・枝を描き直す. 木構造が大きい場合これらの合計時間 (Ts) が大きくなって, 実用に耐えない可能性がある. Ts を小さくするには,

(a) 配置アルゴリズムの計算時間を短縮する.
(b) 表示画面の更新時間を短縮する.
(c) 配置アルゴリズムの適用回数及び表示画面の更新回数を減らす.

などの方法が考えられる. **3.6**で述べるように, データ構造の更新時間は僅かである.

(a) については, 効率的な配置アルゴリズム (**3.4.3**節) を提案し, それを用いることで実現した. (b) に対しては, 画面の表示を出来るだけ高速に行なう実現法を採用した (**3.6.1**節). (c) については, 一般にデータ構造とその画面表示とは, 同期して更新される方が望ましいが, アプリケーションプログラムによっては, 連続して節点を追加・削除するような場合, すべての修正が終ってから一括して配置を更新するだけで充分な場合もある. そこで, VTM ではアプリケーションプログラムからの要求に応じて, 画面の更新を制御出来る機能も実現している.

3.3 実現方法

3.3.1では，VTM の構成の概略を示し，**3.3.2**以降で **3.2**で述べた問題の解決法の実現について述べる．

3.3.1 VTM の構成

VTM の構成を図 3.1に示す．VTM は，その機能より，データ管理部 (Data Manager)，画面管理部 (Display Manager)，及び，実行管理部 (Execution Manager) からなる．

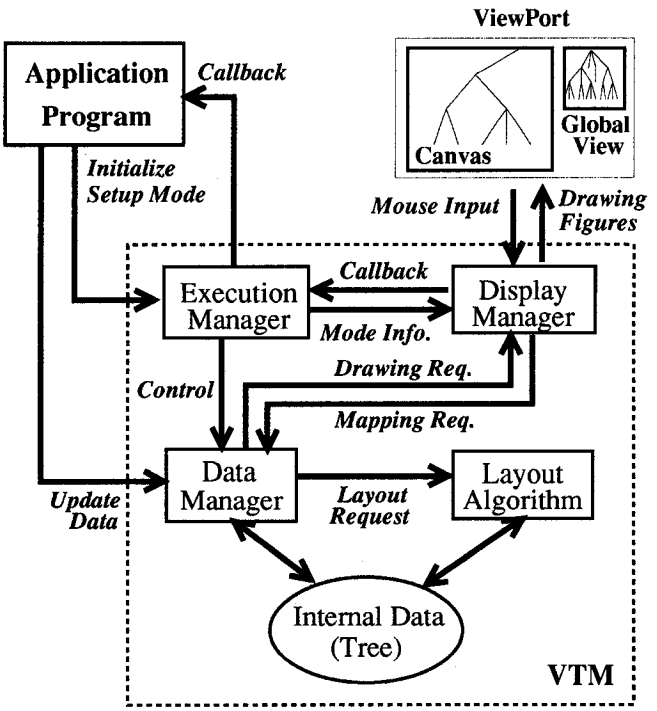


図 3.1: VTM の構成

データ管理部は，木構造データそのものに対する処理を行なう部分であり，アプリケーションプログラムからの要求 (関数呼出し) に従って必要ならば木構造データを更新する．さらに，必要な時点で，配置アルゴリズム (Layout Algorithm) を呼出し，仮想平面上に節点の配置を行なう．画面管理部は，仮想平面上に描かれた木構造全体を画面上の全形表示部 (GlobalView) にマップし，さらに，木構造の指定された長方形領域 (表示領域) を画面上の拡大表示領域 (Canvas) にマッピングする (詳細は **3.3.2**節)．実行管理部は，木構造の更新モードを保持し，更新モードに従って，各部への実行の指示を行なう．

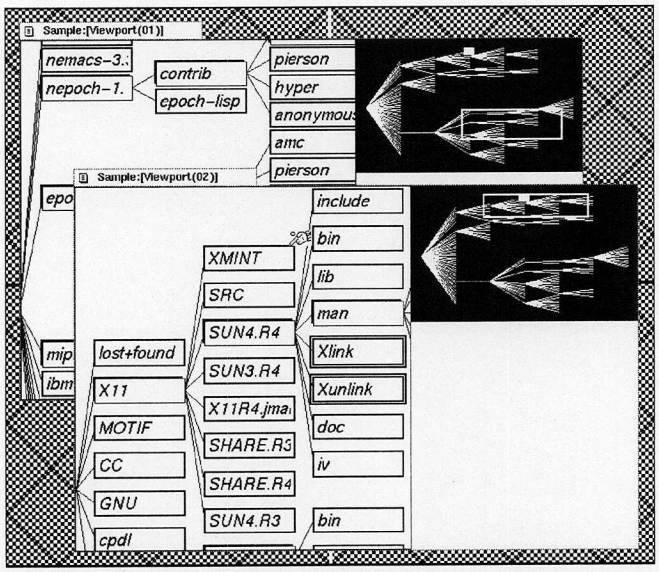


図 3.2: VTM の画面表示例

VTM の画面表示例を図 3.2に示す．各ウィンドウの左の領域が Canvas，右上が GlobalView である．Canvas と GlobalView を合わせて ViewPort と呼ぶ．

なお，木構造の表示は，図 3.2のような根を左端に置く横向き表示以外に，根を上端に置く縦向き表示もできる (図 3.5)．

3.3.2 VTM の画面表示

VTM では，仮想平面 (Virtual Plane) 上の木構造全体とその部分の拡大を同時に観察するため，GlobalView と Canvas を用いる (図 3.3)．GlobalView では，仮想平面上の木構造全体と，Canvas に表示する領域 (表示領域と呼ぶ) を示す長方形 (AreaMark と呼ぶ) を表示する．Canvas では, 仮想平面上の表示領域内の木構造を拡大表示する．ただし，GlobalView 上では，節点を小さな点で表し，節点に付随するラベル (節点に付加されたデータ) は表示しない．Canvas 上では，各節点を長方形で表し，その中にラベルを表示する．複数の部分を拡大表示するために, 複数の ViewPort を開くことできる．

ユーザは，GlobalView 上の AreaMark 内よりマウスの中ボタンを押しながらマウスをドラッグすることで，対応する仮想平面上の表示領域を移動させることができる．また，マウスの左ボタン押しながらマウスをドラッグすることで，表示領域の大きさを拡大 (縮小) できる．これらの操作によって，仮想平面上の任意の長方形領域を Canvas 上に表示できる．

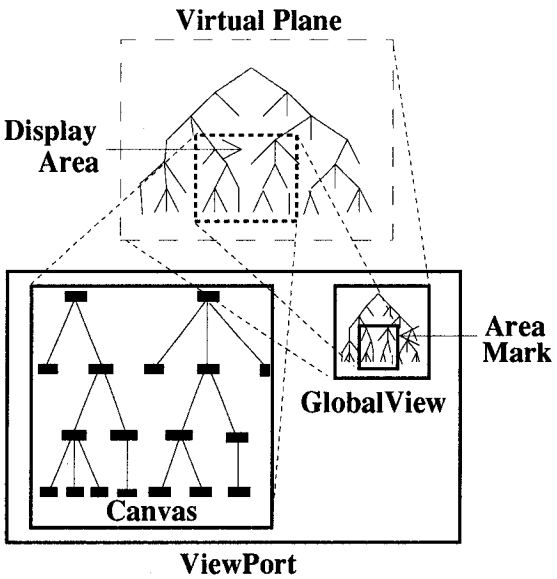


図 3.3: Canvas と GlovalView

3.3.3 画面の更新時間

VTM では，以下のような更新モードを指定することで，データ構造の更新に伴う画面の再表示の方法をアプリケーションプログラムから制御できる．

(1) **自動更新モード** —— 部分木の追加・削除が行なわれる度に仮想平面の配置を更新し，同時に，Canvas 及び GlobalView も更新する．
(2) **部分更新モード** —— 部分木の追加・削除に対して，仮想平面の配置を変えない．追加の場合，追加した部分木のみについて配置アルゴリズムを適用し，Canvas の該当部分の表示を更新する．削除の場合，画面上から該当部分を削除するが，その他の部分の配置は更新しない．追加中の部分木の節点と元の木の節点が重なって表示される可能性がある．
(3) **要求更新モード** —— 部分木の追加削除に対して，仮想平面の配置を変更せず，画面表示も変更しない．

(2),(3) の場合，アプリケーションプログラムから画面の**更新要求**を受けるとその時点での木構造を配置し直し，仮想平面及び画面表示を更新する．このとき，GlobalView と Canvas の一方もしくは両方の更新を指定できる．

自動更新モードは，画面上の表示と内部のデータ構造が常に一致しており最も分かり易いが，節点数が多くなって表示に時間がかかるようになると煩わしくなることがある．節点数が多くなった場合，(2) または (3) のモードが有効になる．要求更

新モードはデータ構造の変更を連続して行ない，最後にまとめて画面を更新することで，画面の更新回数を減らすことが可能となる．部分更新モードでは，要求更新モードと似ているが，更新中の節点を仮表示する．

3.3.4 VTMの実現

VTMの最初の版は，XToolkit[35]を用いて実現した(Xtk版)[2]．しかし，節点数が多い場合，表示に多くの時間を要していたので(**3.6.1**参照)，Canvas上の各節点をXlibの関数で直接描画するように変更した(Xlib版)．

Xtk版の場合，マウスのクリックに対してどの節点が選択されたかは，サーバ側で自動的に判断されるのに対し，Xlib版ではクリックされた座標からプログラムでどの節点が選択されたかを識別する必要がある．

3.4 配置アルゴリズム

3.4.1 従来の研究

2進木を平面上に美しく表示するアルゴリズムが，いくつか提案されている[29][30]が，これらのアルゴリズムは，以下のようないくつかの“美的制約”を満たす(ただし，σ, δはともに正のある定数である)[3]．

(C1) 枝の描写(節点を結ぶ線分)が交差しない．
(C2) レベルがiの節点のy座標は$\sigma \cdot i$とする．
(C3) 親節点のx座標は子節点の中央に位置する．
(C4) 同形の部分木は，同じ形状で配置する．
(C5) レベルの等しいどの2つの節点間の距離もδ以上である．

Reingold[29]らは，C1～C4の制約を満たす配置[4]を線形時間で求めるアルゴリズムを示している．また，Supowit[30]らは，上記のすべての制約を満たす配置のうちで，幅最小の配置を求めるアルゴリズムを提案している．しかし，このアルゴリズムは線形計画法を用いるものであり，あまり効率的でない．これらの既存のアルゴリズムは，いずれも，すべての節点が与えられたときに動作するものであり，節点が逐次的に追加・削除される場合の配置の更新アルゴリズムについては報告されていなかった．

[2]Canvas上の各節点には，Athena Widget SetのLabel Widgetを用いた．
[3]以下では、根を上端に置く縦方向の木について説明する．
[4]データ構造の各要素をある2次元平面に配置すること．

文献 [29] では主に 2 進木の配置について記述されているが，**3.4.2**では，それを拡張した一般の有向木に対するアルゴリズムを紹介する．さらに，**3.4.3**では，節点や部分木が逐次的に追加・削除される場合の木構造の配置更新アルゴリズムを示す．

3.4.2 配置アルゴリズム

$T=(V,E)$ を配置すべき木とし，R_Tをその根とする．本アルゴリズムは，各節点 $v(\neq R_T)$ について，$\pi_x(v) \triangleq$ (vの x 座標) − (vの親の x 座標) なる値を決定する．各節点の y座標は制約 C2 により決まるので，根 R_Tの座標を決めれば，すべての節点の座標が一意に決定されることになる．Tの各節点 vについて，それを根とする部分木を $SBT(v)$ と表す．各節点に対するπ_xの値は，根 R_Tに対して再帰的な手続き $assign$ を実行することによって求める．$assign(v)$ は，部分木 $SBT(v)$ の配置を求めるためのものであり[5]，節点 vが葉であれば何もせずに終了する．vが葉でないものとし，その子を左から順に $v_1, v_2, \cdots, v_k$とする．もし，$k=1$ であれば，$\pi_x(v_1) \leftarrow 0$ として終了する．$k \geq 2$ であれば，まず $i=1,2,\cdots,k$について，$assign(v_i)$ を実行することにより，$SBT(v_i)$ の配置を求める．次に，$i=2,3,\cdots,k$の順で，v_iと v_{i-1}との距離 $d(i)$ を定める．但し，このとき，制約 C1, C5 を満たす範囲で $v_1, v_2, \cdots, v_k$ができるだけ接近して配置されるようにする[6]．そして最後に，$\pi_x(v_1) \leftarrow -\frac{1}{2}\sum_{i=2}^{k} d(i)$ とし，$d(2),\cdots,d(k)$ の値から，以下に示すように，$\pi_x(v_2),\cdots,\pi_x(v_k)$ の値を計算する．

この手続きにおいて，$d(2),\cdots,d(k)$ の値を定めるために，文献 [29] のアルゴリズムと同様に，部分木 $SBT(v_1),\cdots,SBT(v_k)$ の**輪郭**を用いる．ここで部分木 T'の輪郭とは，次のような 2 つの節点列のことである．T'の節点のうち，各レベルで最も左に置かれるべきものを求め，それらをレベルの値の昇順に並べたものを T'の左輪郭と呼ぶ．T'の右輪郭も同様に定義する．例えば図 3.4において，$SBT(w_1)$ の左輪郭は $[w_1, w_2, w_3, w_4]$ であり，右輪郭は $[w_1, w_9, w_{11}, w_6]$ である．部分木 T'の輪郭の管理のために，その左輪郭上の各節点 wに対して，次の節点を指すポインタ $l_{ptr}(w)$ を保持させておく．但し，最後の節点のポインタの値は'$null$'としておく．同様に，右輪郭上の各節点 x にポインタ $r_{ptr}(x)$ を与えておく．左右輪郭の最後の節点のポインタの値は，T'を真に含む部分の輪郭を求める際に更新されることがある．例えば，図 3.4において，$r_{ptr}(w_{11})$ の値は初め ($SBT(w_{11})$ の輪郭を求めた際に) '$null$'とされるが，後で $SBT(w_1)$ の輪郭を求めるときに $r_{ptr}(w_{11}) = w_6$となる．

---

[5] $SBT(T)$ 内の節点間の相対的な位置を決定するもので，v自身や他の節点の座標を決定するものではない．

[6] 左端から順に決定していき，右側で広くとって調整するので，左右対称 (ここでの制約にはない) にはならない．

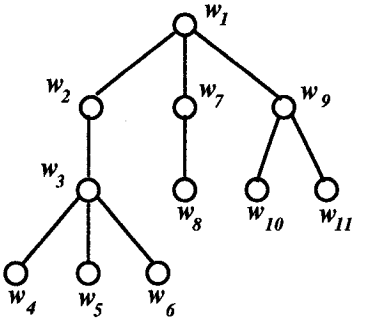


図 3.4: 木構造データ (例)

Tのある節点vについて，その子を左から$v_1, v_2, \cdots, v_k$とする．$i = 1, 2, \cdots, k$について，$SBT(v_i) = (V_i, E_i)$であるものとし，グラフ$SMT(v, i)$を$(\bigcup_{j=1}^{i} V_i, \bigcup_{j=1}^{i} E_i)$と定義する．さらに，各$SMT(v, i)$の輪郭を，部分木と同様に定義する．例えば，図3.4において，$SMT(w_1, 2)$は7つの節点$w_2, \cdots, w_8$を持ち，その左輪郭は$[w_2, w_3, w_4]$，右輪郭は$[w_7, w_8, w_6]$である．

手続き$assign(v)$では，$i = 2, 3, \cdots, k$について，$SMT(v, i-1)$と$SBT(v_i)$の輪郭を用いて，$d(i)$の値と$SMT(v, i)$の輪郭とを求める．以下，$d(i)$の決定方法についてのみ述べる．$SMT(v, i-1)$の右輪郭及び$SBT(v_i)$の左輪郭を，それぞれ$[w_1(= v_{i-1}), w_2, \cdots, w_n]$, $[x_1(= v_i), x_2, \cdots, x_m]$とする．まず，初期値として$d(i) \leftarrow \delta$とし，その後$j = 1, 2, \cdots, Min\{n, m\}$について，$w_j$と$x_j$との(現在の)距離が$\delta$以上あるか否かを調べる($SMT(v, i-1)$と$SBT(v_i)$の配置は既に決定されているので，$d(i)$の現在の値を使って計算できる)．もし，この距離が$\delta$より小さければ，$d(i)$の値を必要なだけ大きくする．

以上，部分木の輪郭を用いて手続き$assign(v)$を実行する方法を述べた．この手続きにより定まる配置が制約C1～C5を満たすことは容易に証明できる．輪郭の計算法など詳細は省略したが，T全体の配置を得るのに要する時間は$O(|V|)$である．

3.4.3 部分木の付加・削除アルゴリズム

前節のアルゴリズムにより木Tの配置を求めた後，ある部分木を削除する，もしくは，別の木を部分木としてTに付加するものとする．本節では，このような場合に，新しく得られる木の配置を求める方法を示す．

まず部分木の付加について述べる．2つの木T, T'について，前節のアルゴリズムにより配置が求められているとする．T, T'の根をそれぞれR_T, rとする．Tのある節点をv，その子を左から$v_1, v_2, \cdots, v_k$とし，rがvのi番目の子となるように，

T'をTに付加するものとする．このとき，明らかに各部分木 $SBT(v_j)(1 \leq j \leq k)$ 及び $SBT(r)(= T')$ の配置を変える必要はない．$SMT(v, i-1)$ の配置についても同様である．従って新しい $SBT(v)$ の配置を求めるためには，$SMT(v, i-1)$,$SBT(r)$,$SBT(v_i)$,$SBT(v_{i+1})$,$\cdots$,$SBT(v_k)$ の輪郭を用いて，前節で述べた方法により，$v_{i-1}, r, v_i, v_{i+1}, \cdots, v_k$ の間の距離を決めればよい．

$SMT(v, i-1)$ 及び各部分木 $SBT(v_j)(i \leq j \leq k)$ の輪郭はポインタ l_{ptr}及び r_{ptr} の最終値 (元の木Tの配置全体を求めた時点での値) から，容易に復元できる．

$SBT(v)$ の新しい配置が求まった後，vの真の先祖を順にたどっていき，途中の各節点wについて $SBT(w)$ の配置の更新を行う (この際にも (上述の) 輪郭の復元処理が必要となる)．全体の配置の更新に要する時間は，T及びT'の形状，T'の付加の位置などに大きく依存し，最悪の場合には全体の節点数に比例したものとなる．

次に，既に配置の求まっている木Tから，ある部分木 $SBT(v)$ 全体を削除するものとする．この場合の更新の方法も，部分木の付加の場合と同様，vの真の先祖を順にたどっていき，途中の各節点wについて，(輪郭の復元処理を行いながら) その子の間の距離を決めていくというものである．部分木の削除に対する配置の計算及び更新時間も，T及び $SBT(v)$ の状況に大きく依存し，最悪の場合にはTの総節点数に比例したものとなる．

3.5 VTM の使用例

3.5.1 ディレクトリブラウザ

UNIX のディレクトリは階層構造 (木構造) をしている．ディレクトリブラウザは，UNIX のディレクトリ階層の一部を木として画面に表示し，ユーザからのマウス入力によって，以下のようにその表示を動的に変化させる．ディレクトリ に対して，その中に直接含まれるすべてのファイルが木構造の節点として表示されているとき，そのディレクトリは，“展開されている” といい，どの子節点も表示されていないとき “閉じている” という．初期画面では，root ディレクトリ (/) から，カレントディレクトリに至るパス上のディレクトリのみが “展開されている”．“展開されている” ディレクトリを表す節点をクリックすると，それを “閉じ”，“閉じている” ディレクトリを表す節点をクリックするとそれを “展開する”．実行可能ファイルの節点をクリックすると，それを実行し，テキストファイルの節点をクリックすると，それを引数としてエディタを起動する．

VTM を用いて新たにアプリケーションプログラムを作成する場合，木構造データをアプリケーションプログラム側で用意する必要はなく，VTM の持つデータ構

造内に必要な情報を記憶させることが出来る．ディレクトリブラウザの場合も，このため比較的容易にプログラミングできた (図 3.2) [7]．

3.5.2 ASL システムの項書換えの可視化

本節では，既存のプログラムに対する VTM の実装例として，ASL システム[33]の項書換え操作[8]の可視化について述べる．

従来の ASL システムでは，項 (表現式) は文字列で出力されているためその構造がとらえにくい．さらに，部分表現式 (木構造内の節点) の指定は，根からのパスを指定するため，長さに応じた手間がかかる．それ故，構文木を常に画面上に図示する機能と，画面上でマウスによって直接節点を指定できる機能を実現すれば，表現式の構文の理解が容易になり，かつ操作効率が向上することが期待できる．

一般に，既存のプログラムに対して，そのデータ構造を VTM を用いて可視化する場合，既存のプログラム内のデータ構造と VTM のそれと 2 重に保持することになる[9]．このような場合，2 つのデータ構造の整合性を保証しなければならない．その方法としては，

(1) 既存のプログラムを直接修正して，内部データ構造を更新する部分に VTM の木構造への更新操作関数を埋込む方法，及び，
(2) 既存のプログラムに出来るだけ手を加えずに，外部に別のプログラムを用意し (Agent と呼ぶ)，既存のプログラムに対して Agent からコマンドを与え，その出力を解析することで，内部の木構造を同定し[10]，VTM の木構造への更新操作関数を埋込む方法

などが考えられる．

一般に，既存のプログラムの内部データを直接更新する操作が 1 箇所 (または数箇所以下) にまとめられている場合には，前者の方法による実現が容易である．また，既存のプログラムの出力の解析が容易である場合には，後者の方法が比較的容易に実現できる．

---

[7] ソースプログラム[36]は 92 行，プログラム作成に要した期間は，設計・デバッグ期間も含めて約 8 時間であった．

[8] ASL システムは，筆者の所属する研究室で開発した，代数的手法によるプログラム開発のためのシステムであり，代数的言語 ASL のための記述・検証支援機能，及び実行機能を持つ．ASL 検証支援機能における主要な操作の一つに，項 (表現式) に対する対話的な書換え操作がある．この操作は，構文木の節点を指定し，さらに，それを根とする部分木に対して適用すべき書換え規則を指定して，それに従って部分木の置換操作を行なうものである．

[9] 既存のプログラムのデータ構造を捨て去って VTM のものを利用することも可能であるが，一般にそのためのプログラム修正は容易でないことが多い．

[10] 当然，このための情報を得るコマンドが既存のプログラムに用意されていなければならない．用意されていない場合は，既存のプログラムにその機能を追加する必要がある．

今回の実現 (図 3.5) では，既存のプログラムのデータ構造を更新する部分が多くの箇所に存在していたことなどから，後者の方法を採用した．

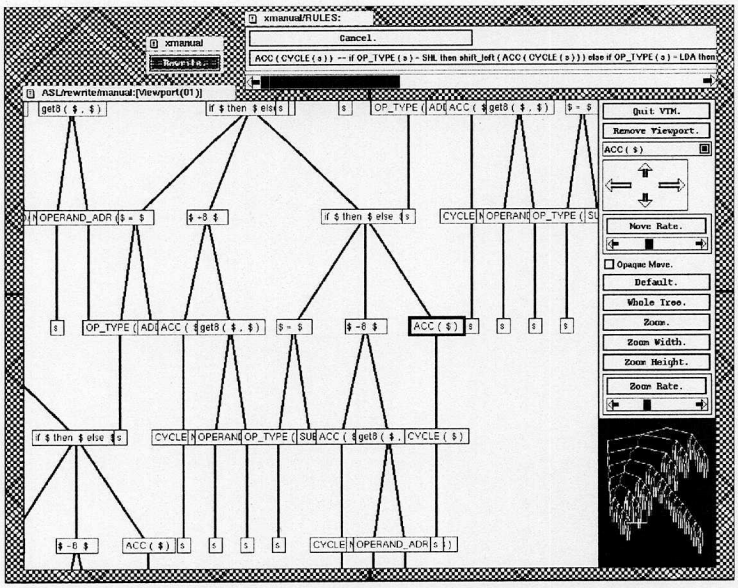


図 3.5: ASL システムの画面表示例

実装方法の検討 (従来のプログラムの一部の解読を含む) に 2 日ほどかけ，方法を決定してから実際に組み込みに要した時間は，デバッグも含めて約 8 時間であった．Agent のソースプログラムの大きさは，約 200 行であった．

3.5.3 LOTOS シミュレータにおけるプログラム実行の可視化

我々のグループが作成した LOTOS シミュレータ[34]は，通信系に対する仕様記述言語 LOTOS で記述された各局のプロトコル仕様 (LOTOS プログラム) 群の動きを観察するための LOTOS の実行系であり，プロトコル仕様間の通信機能も実現している．VTM を用いることにより，LOTOS プログラムの構文木を図示し，次に実行可能な “イベント” を表す節点や “プロセス” を表す節点を明示する．次に実行可能な “イベント” を表す節点をクリックするとそれを実行させる．その結果，“今後実行すべきプログラム” が書き換えられ，その構文木も変化する．“プロセス” の節点をクリックするとその内容を “展開” する．

本シミュレータへの VTM の組込みは，LOTOS シミュレータの内部を熟知しているものが行なったので，**3.5.2**節の (1) の方法を用いた．VTM の組み込みに要した作業時間は 3 時間であった．

3.6 実行時間の測定

木構造データの更新操作に対して，データ構造の更新時間 (Tup) 描写アルゴリズムの計算時間 (Tal)，及び，Canvas 及び GlobalView の画面の更新時間 (それぞれ$Tdr.c$ 及び$Tdr.g$) を調べるため次のような実験を行なった．なお，クライアントプログラム (VTM) 及びサーバ (画面表示) を，別々の DECstation3100 上で実行させた．

(**実験 I**) 高さ 12 の完全 2 進木 (節点数 4095) から，高さ 5 の完全 2 進部分木 (節点数 31) の削除を繰り返し，各回の Tup, Tal, $Tdr.c$ 及び $Tdr.g$を測定した[11]．

実験の範囲では Tup は 0.2msec 以下であった．

3.6.1 画面の更新時間

Xtk 版と Xlib 版に対して，実験 I を行なったときの，部分木削除後の木の節点数と $Tdr.c$ の関係を図 3.6に示す．図中の (1a)～(1c) は Xtk 版についてのもので，(1a) は Canvas 上に木全体を表示したとき，(1b) は Canvas 上に削除すべき部分木を含む領域 (約 120 節点を含む) を表示したとき，(1c) は，削除すべき部分木 (31 節点) のみを含む領域を表示したときの $Tdr.c$ である[12]．同様に (2a)～(2c) は，Xlib 版について測定したものである[13]．Canvas 上に多くの節点を表示した場合，ラベル等の判別が難しくなるので，**通常の使用状況**においては，Canvas 上に表示される節点数は高々100 程度であることが多い．従って，多くの場合 Xtk 版のときは (1b)(Xlib 版のときは (2b)) 以下の時間で Canvas を表示できる．グラフからいずれの場合も，Xlib での実現の方が $Tdr.c$ の時間が短いことがわかる．

図 3.7は，Xlib 版における $Tdr.c$, $Tdr.g$, 及び Talを示している．図中の (2a)～(2c) は図 3.6と同じである．$Tal.O$, $Tal.N$はそれぞれ **3.4.2**及び **3.4.3**で述べたアルゴリズムの計算時間である．

上で述べた**通常の使用状況**では，Ts は $Tup(\approx 0) + Tal.N + Tdr.g + Tdr.c(2b)$ 程度であり，節点数 4000 程度でも 0.4 秒程度ですべての処理が完了する．

---

[11]削除可能な部分木が複数存在するとき，根からの距離の大きい方の部分木を優先して削除し，削除すべき部分木がなくなったとき実験を終了する．実験は自動更新モードで行なった．1 回の部分木削除操作に対して，画面の更新が完了するまでの時間 Ts は，これらの時間の和である．

[12](1a)～(1c) のいずれも，各節点が Canvas 内に表示されるかどうかにかかわらずサーバに対し描画要求を出しており，Canvas に表示するかどうかはサーバが判断している．このため，(1a)～(1c) の差は，実際に画面に描くための操作時間の差だけである．

[13]Xlib 版については図 3.7の方がわかりやすい．

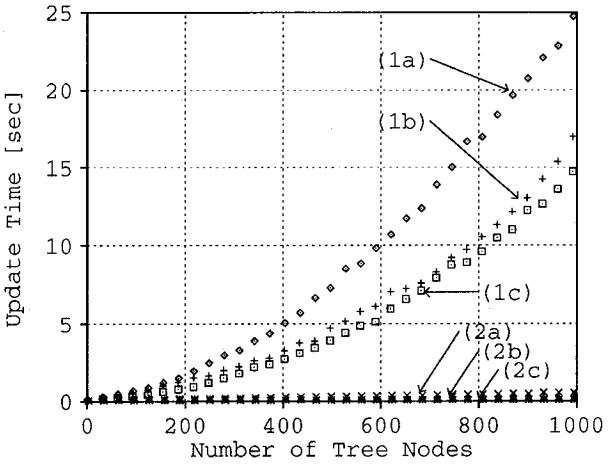


図 3.6: Canvas の更新時間 (Tdr.c)—Xtk 版と Xlib 版

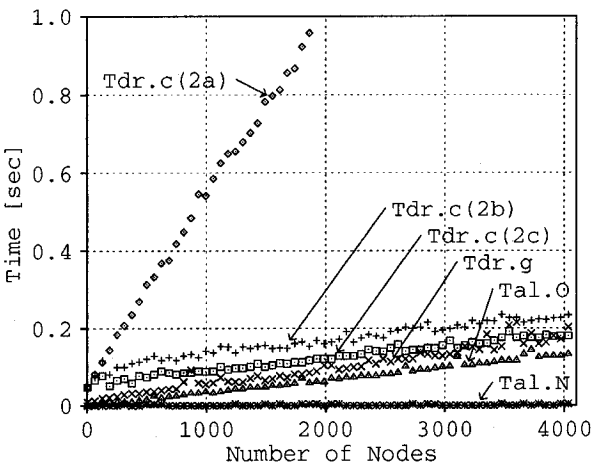


図 3.7: ViewPort の更新時間と描写アルゴリズムの計算時間

3.6.2 描写アルゴリズムの計算時間

3.4.3で示したアルゴリズムを用いて，実験Iを行なった結果，節点数が4095以下の場合，アルゴリズムの計算時間はいずれも $1.0msec$ 以下であった．このアルゴリズムは，木の高さが低いほど効率が良く，完全2進木 (節点数 $|V|$) の場合，その計算時間は $O((log|V|)^2)$ である[32]．

図3.7より，**3.4.3**のアルゴリズムを用いることにより，$Tal \approx 0msec$ となって，Ts が約25%短縮できたことがわかる．

3.7 結言

木構造データを画面上に美的に表示し，画面上でのマウス操作によって節点・枝の指定等の入力を可能にするライブラリプログラムを作成し，適用例によりこのライブラリの有効性を示した．また，多くの節点を持つ木に対しても有効であることを実験で示した．ASL システム及び LOTOS シミュレータへの適用例からも分かるように，既存のプログラムに対する実装も比較的容易であると思われる．VTM のプログラムサイズは，C 言語で約 7000 行である．

今後の課題としては，平面グラフ，有向アサイクリックグラフ (DAG)，階層的グラフなど木以外のデータ構造にも対応することが挙げられる．VTM は，一般のグラフを扱えるデータ構造を有しているので，それぞれのグラフの描写アルゴリズムを用意するだけで，これらのグラフを取り扱うことができる．

4章

図的ユーザインタフェースの計算機シミュレータへの応用

4.1 序言

計算機の基本アーキテクチャの教育においては，CPU，メモリ，入出力インタフェースの個々の機能と共にそれらを総合した計算機の構成と動作を充分に理解させることが必要である．さらに，与えられたものを越えて自分で計算機を構成できるようになることも望ましい．

この教育は，講義，実物のハードウェア，あるいは，CADツールなどによって行うことが出来るが，それぞれ以下のような得失がある．

まず，講義による教育では，設備を必要とせず，技術の変化に柔軟に対応でき，各種の方式の比較などを幅広く行なえる．しかし，計算機中のデータの移動や制御の流れなどのダイナミックな特性を表現するのが容易ではなく，計算機の構成に不完全なところがあっても機械的にチェックすることは出来ず厳密さに欠けるところがあり，さらに計算機の規模が少し複雑になると取り扱いにくいなどの欠点がある．

次に，実物のハードウェアを用いる方法はもっとも深く理解させることが出来る．しかし，多数の学生に対応しようとすると多くの時間と設備を必要とし，また，既成の部品として教育的に適当なレベルのものが少ない．ある程度以上の複雑さのものはLSI化されてしまっていて内部の状態を見ることが出来ない．また，実物のICを使うには電圧，電流，タイミング等の使用条件に関する詳細な知識を必要とするので，基本アーキテクチャの教育には不向きである．

汎用のCADツール[18],[21]は，画面上での回路作成機能や部品の定義機能等を有するものもあり，これをそのまま教育用として用いることも可能である．しかし，汎用のCADツールでは，回路図を基に直接シミュレーションするのではなく，回

路図からシミュレーションを行なうために必要なデータファイルを作成する手続き(これは，プログラムのコンパイルに相当し，シミュレーションを高速に行なうための手続きである．)を必要とする．このため，回路の修正とシミュレーションを繰り返し行なおうとするとき，回路修正のたびにこの手続きを行なう必要があり，このため，思考が中断され，初心者にとっては使いにくいものとなる．また，機能が豊富すぎて使いこなすのが難しいという問題もある．

この他，画面上で計算機の内部の動作をシミュレート出来るものとしてVEGA[22]，INSIST[20]がある．しかし，VEGAは固定した計算機を対象としたものであり自由に計算機を構築できない．また，INSISTはCADシステムのプロトタイプ作成用でありSmalltalk-80[17]のシステムが直接に見えてそれに関する深い知識を必要とする．

本論文では上記のような問題点を考慮し，以下の特徴を持つ教育用の計算機シミュレータ(GCS)を提案し，その機能および実現法について述べる．

(1) 画面上でマウスを使って部品を配置，配線することにより計算機を構築することが出来る．
(2) 計算機の構築途中でも同一の画面上で直ちにその動作のシミュレーションができ，視覚的に観察出来る．
(3) 計算機構築用の高機能な部品が用意されているので低レベルの回路から作り始めなくてもよい．また，新しい部品の定義なども簡単である．

4.2.では，教育用計算機シミュレータに要求される機能について分析し，**4.3.**では，GCSの概要について操作法を中心に説明する．**4.4.**では，**4.2.**で述べた機能の実現法を示す．**4.5.**では，GCSの使用例，および，実現法についての考察を与えている．

4.2 シミュレータの機能分析

4.2.1で，シミュレータを利用した計算機アーキテクチャの教育の形態を分析し，それに基づいて，シミュレータに必要な機能を**4.2.2**以降で述べる．

4.2.1 シミュレータを用いた教育形態

計算機シミュレータを用いた計算機のアーキテクチャの教育の形態を示し，それぞれについて，その教育効果について述べる．

(形態1) 教師が回路を作成し，教師が表示する．

教師が授業などで計算機の回路を示し，それを動作させて各部分間の信号伝達

や信号の値を示すことで，動作を理解させることができる．

(形態 2) 教師が回路を作成し，学生が動作させる．
教師があらかじめ簡単な計算機の回路を作っておく．それを学生が動作させて信号の流れや内部状態を自分で観察するものである．実物のハードウェアを用いるよりも視覚的に観察できるので学生は理解しやすい．さらに，興味ある学生はその場で回路を書き換えて動作させてみることも出来る．

(形態 3) 教師が部品を用意し，学生が回路を作る．
必要な部品を教師が用意し，学生は論理回路や簡単な計算機を作成してシミュレートする．このとき，教師が複雑な部分（例えば計算機の命令デコーダなど）をあらかじめ部品として定義しておくことにより学生は効率的に学習を進めることが出来る．

(形態 4) 学生が部品及び回路を作る．
さらに進んだ段階として学生が自ら部品を作成し，回路を構成することにより，より理解を深めることができると期待できる．

4.2.2 シミュレーションの実行と観察機能

シミュレーションの実行機能としては，実行を任意の時点で中断でき，その時点での計算機の状態が観察できることが必要である．これは，前節で述べたどの教育形態においても，重要な点であるが，特に，(形態 3) および (形態 4) においては，さらに，計算機の全ての回路が完成していなくてもシミュレーションが可能であることが望まれる．このような機能があると，回路の修正を行いその場でシミュレーションをしながら回路を構築していくことも出来る．観察機能としては，以下に示すような，計算機の動作状況を判りやすく提示するための３つの機能が重要である．

(1) 部品の状態観察機能
GCS では，計算機を構成する各部品の状態を観察するための機能として，内部状態をシミュレーションの実行中にも常に表示させてモニタする機能 (Monitor 機能) と，その内部状態が特定の状態になったときにシミュレーションを一時中断させる機能 (Breakpoint 機能)，および配線 (ワイヤ) を流れる信号を常に表示させる機能 (Probe 機能) を提供している．

(2) タイミングチャート機能
複数の箇所での配線上の信号の変化を時間の流れとともに確認したいことも多い．従って，回路上の任意の複数の点における信号を，いわゆるタイミングチャートとして表示する機能を提供している．

(3) アニメーション機能
(1), (2) の機能ではモニタしている部分の情報は得られるが全体の動きが把握しにくい．そこで，各時点でどのワイヤもしくは部品に信号が流れているかを順次，アニメーションで示すことで，計算機全体の動きをより分かりやすくできる．この機能は，特に，(形態 1) で有効であろう．

4.2.3 回路の作成機能

教育形態の (形態 3) や (形態 4) では，特に，計算機の回路をできるだけ簡単に構成できることが重要である．このため GCS では以下のような部品数を減らす工夫をしている．また，マウス操作のみで回路の作成および修正が簡単に出来るようにしている (4.3参照)．

AND や OR 等の低レベルの部品を用いて計算機回路を構成することも出来るが，このような部品のみを用いていたのでは回路が複雑になり，本質的に重要な部分が分かりにくくなる．そこで，部品の抽象化を行い，ALU，ゲートコントローラ等の高機能な部品を予め用意することで少ない部品数で計算機を構築できるようにしている．

また，回路を作成する場合にはレジスタ間の配線などにおいて，データのビット数に等しい本数の配線が必要である．これでは配線の手間が面倒であり，回路図自体も見にくくなる．そこで，GCS では配線 (ワイヤ) の抽象化も行っており，1 ビットに対応するワイヤの他にデータのビット数分の本数に対応するワイヤも取り扱えるようにしている．

4.2.4 部品定義機能

4.2.3で述べたように，高機能な部品を予め用意しておくと，それらのみを用いて作成できる計算機の自由度が少なくなり，(形態 4) で支障をきたす．これを避けるために部品を簡単に定義して使用出来る機能が必要になる．GCS では，信号伝達の方法を規定しているだけであり，どのような部品でも，その入出力と内部状態の変化を与えるだけで自由に定義できる．

4.3 シミュレータの概要

ここでは，シミュレータの画面表示および操作方法の概要を示す．回路の作成と実行の指示は，すべて，画面上でのマウス操作のみで行なえる．

GCS の画面は図 4.1に示すように，シミュレーションウィンドウ，部品ウィンドウ，タイミングチャートウィンドウ，および，Counter や GateControl などの部品毎に設定されたモニタウィンドウからなる．操作対象となる部品の選択は，マウスの左ボタンをクリックすることで行なう．中ボタンをクリックすると，その場所に応じて適当なメニューが現れる．

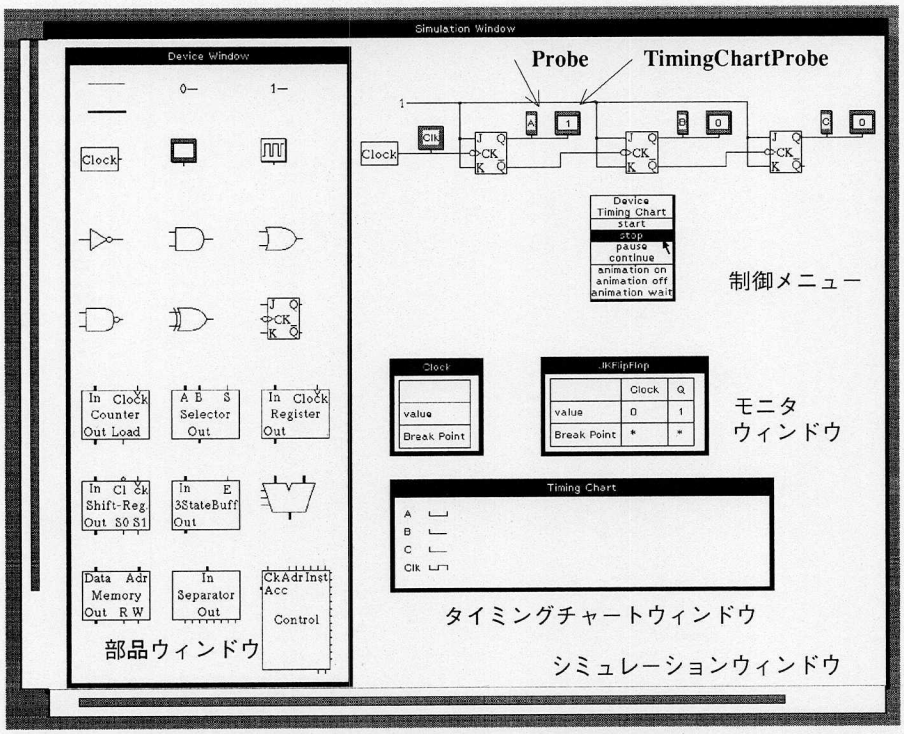


図 4.1: GCS の画面表示

4.3.1 実行・観察

シミュレーションの実行の制御は，制御メニューを用いておこなう．シミュレーションウィンドウ上でどの部品も選択されていないとき，中ボタンをクリックすると制御メニューが現れる．制御メニューでは，シミュレーションの実行制御 $(start, stop, pause, continue)$，および，部品ウィンドウおよびタイミングチャートウィンドウの表示制御 (on, off) が可能である．

観察機能のうち，個々の部品の Monitor 機能は，部品メニューを用いて行なう．部品上でマウスの中ボタンをクリックすることで，部品メニューが現れる．部品メニューでは，部品の削除およびモニタウィンドウの表示制御 (on, off) が可能である．モニタウィンドウでは，その部品の持つ状態変数の値を常に表示し，Breakpoint の設定や解除が可能である．

ワイヤをながれる信号をモニタするためのProbe機能を利用するには，部品ウィンドウにあらかじめ用意しているProbe部品をモニタしたいワイヤに接続するだけで良い．Timing Chartに表示したい点を示すためのTiming Chart Probeも Probe部品と同様の方法で利用できる．アニメーション表示の制御 (on, off) は，制御メニューから行なう．

4.3.2 回路の作成

部品を配置するには，AND， ORなどの部品ウィンドウ上の部品 (図4.2) を示すアイコンの一つを選択しシミュレーションウィンドウ上の適当な位置でクリックすれば良い．これによって，その位置にアイコンに対応する部品が配置される．

部品間の配線を行なうには，部品ウィンドウ上のワイヤを示すアイコンを選択し，接続したい部品の端子どうしを折れ線で接続すれば良い．この接続は，必ずしも部品間で行なう必要はなく，既に配線されているワイヤと接続しても良い．

部品を移動させるには，その部品をマウスでドラッグするだけで良い．部品の移動に際して，部品に接続されているワイヤは，接続関係を保存するように自動的に描き直される．

4.3.3 部品の定義

新たな部品を定義するには，現在のところ，マウス操作だけではできず，4.4.4節で述べるように，動作定義表を文字列として記述しなければならない．このほか，画面上での部品の形状を与える必要があるが，これは，既存のツールを利用している．

4.4 シミュレータの実現

まず，4.4.1でシミュレータの構成の概略を示し，4.4.2以降に，4.2で述べた機能の実現方法とそこでの問題点等について述べる．

4.4.1 シミュレータの構成

GCSでは，動作のシミュレーションを行なうために，オブジェクト指向の動作モデルを採用している．これは，個々の部品をオブジェクトと捉え，信号の流れをメッセージの流れと考えると，自然に動作を記述できるからである．このため，GCSは，Smalltalk-80上に実現している (図4.3参照)．

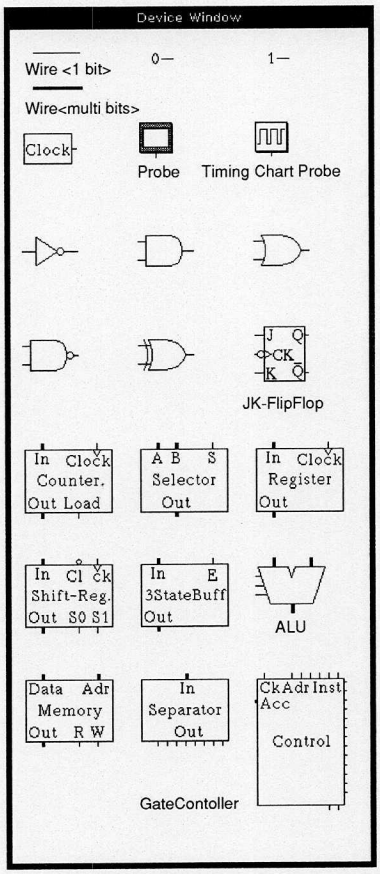


図 4.2: 部品ウィンドウ

まず，*Object*(すべてのオブジェクトの抽象クラス) のサブクラスとして，すべての部品の基となる抽象クラス *AbstractParts* を定義し，そこで，どの部品にも共通して必要な部品の入出力の機能，モニタ機能などを実現している．また，そのサブクラスとして *Parts* および *Wire* の 2 つのクラスを定義している．*Parts* では部品の動作定義表に基づく実行の方法を定義しており，*Wire* では，ワイヤの統合機能 (**4.4.3**節参照) などを定義している．*And*, *Or*, *Register*などの個々の部品は，*Parts* のサブクラスとして定義している．Control などの，部品定義機能によって定義した部品も，*Parts* のサブクラスとして定義される．

さらに，シミュレーションの実行制御を行なうためのクラスとして *EventHandler* を定義している (**4.4.2**節参照)．

4.4.2 実行・観察機能の実現

実行機能の実現においては，部品間の信号伝達の仕組み，および，シミュレーションの時間の制御方法が問題となる．

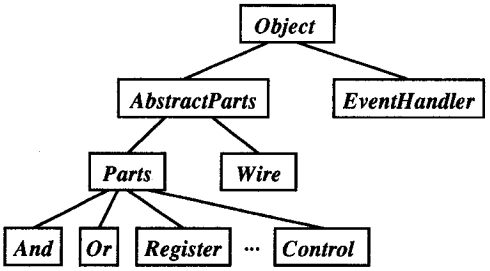


図 4.3: シミュレータのクラス階層

部品間の信号の流れをシミュレートする方法としては，ワイヤをオブジェクトと見なさない場合とワイヤをオブジェクトとして扱う場合の 2 通りの実現法が考えられる．部品間がワイヤで結ばれている場合，前者では，部品から部品に直接メッセージを送ることになり，後者では，部品からワイヤにメッセージを送り，さらに，ワイヤから部品にメッセージを送ることになる．

前者の場合，モデルとしては単純であり実行効率も良いが，配線情報を各部品内に分散して保持しなければならない．従って，例えば，新たな部品が既存のワイヤに接続された場合などの配線情報の更新が容易ではない．

従って，GCS では，後者の方法を採用している．この方法では，部品から部品へ信号を伝達する際にワイヤを経由する必要があるが，配線情報がワイヤ側に保持されているので，上記のような配線の変更は容易である．

GCS では，シミュレーションの実行の中断や再開の時間制御を行なうために，特別なオブジェクト EventHandler を用意している．

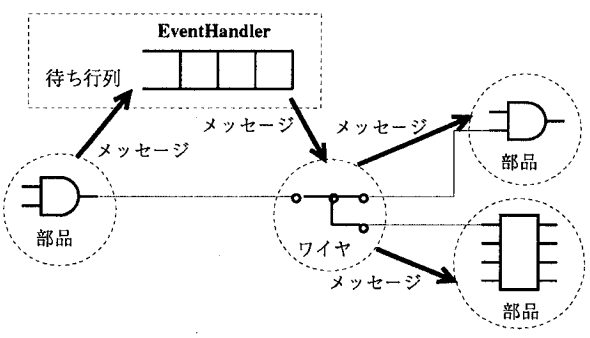


図 4.4: メッセージ伝達の方法

部品からワイヤへのメッセージは，直接ワイヤに送られず，一旦，EventHandler オブジェクトに送られる．EventHandler では，そのメッセージを時間順にソートさ

れたイベント待ち行列につなぐ (図4.4). EventHandlerは，待ち行列の先頭から，順にメッセージを取り出し，送り先のワイヤにメッセージを送る．EventHandlerでは，ユーザからの指示により，任意の時点で，シミュレーションを中断させたり，再開させたりできる．ただし，同じ時刻に実行すべきメッセージが複数，待ち行列につながれている時には，それらのイベントをすべて処理してから実行を中断する．メッセージを受けたワイヤは，配線情報からそれに接続されている部品を調べ，それらに対して，メッセージを伝達する．

観察機能は，システム全体として用意するのではなく，できるだけ個々の部品の機能として実現している．**4.2.2**で述べたMonitor機能は，各部品の付加機能として実現している．Probe部品や，Timing Chart Probe部品などの観察用部品はAND, ORなどの他の部品と同様の扱いである．Timing Chartの表示機能だけは，複数のTiming Chart Probeに関係するので，特別な扱いをしている．

4.4.3 回路作成機能の実現

GCSでは，部品の種類毎にクラスが存在し，画面上で部品が配置されるたびに，そのインスタンスを生成している．ただし，ワイヤは例外である．前節で述べたようにワイヤも，オブジェクトとして扱っているが，どこまでを一つのオブジェクトと見なすかが問題になる．ワイヤの配置では，他の部品と同じように，1回の描画操作で配置したワイヤを一つのワイヤオブジェクトと見なすとすると，既に配置されているワイヤに新たなワイヤを付加した場合，それを別のオブジェクトと見なすことになり，オブジェクトの数が多くなって，部品間の信号伝達のために多くのメッセージを必要とする．そこで，GCSでは，ワイヤ以外の他の部品を介さずに，互いに直接接続されたワイヤの集合を一つのオブジェクトにしている．このため，配線によって2つ以上のワイヤどうしが接続された時に，ワイヤオブジェクトの統合操作（一つのオブジェクトする操作）が必要になるが，シミュレーションの実行は単純になった．**4.2.3**で述べた部品の抽象化の実現方法については，次節で示す．ワイヤの抽象化は，ワイヤと部品間のメッセージの引数を2値データに限るものと，任意の整数値を許すものを用意することで実現している．

4.4.4 部品定義機能の実現

部品の定義を行なうためには，部品の動作定義と画面上でのイメージ (絵) を与えなければならない．後者は，既に述べたように，既存のツールを利用する．部品の動作は，入力信号と部品の内部状態の組に対して，各出力の遅延時間と出力の値，および，新たな内部状態を決めることで定義できる．GCSではこれを以下のような

行を持つ動作定義表として与えることで定義している.

(((' 入力端子名' 値)...) ((' 内部状態名' 値) ...)
遅延時間
((' 出力端子名' 値)...) ((' 内部状態名' 値) ...))

表中の値として，変数や演算子（加減乗除，論理演算子など）を用いることができる．さらに，特別な記号として“*”を指定できる．入力もしくは，内部状態に“*”を指定することは，それがdon't careであることを意味する．出力および状態遷移後の状態変数の値に“*”を指定すると，その状態遷移に対して，その値が変化しないことを意味する．JKフリップフロップの動作定義表を図4.5に示す．このような記述を許すことで，表の大きさを小さくすることが出来，記述を簡単にしている．

```
( ( (('J' * )('K' * )('Clock' 1 ))    (('Q'  * )('Clock' * ))
    4    (('Q'  * ) ('Q*'   * ))    (('Q'  *  )('Clock' 1 ))   )
  ( (('J' 0 )('K' 0 )('Clock' 0 ))    (('Q' 'a')('Clock' 1 ))
    4  (('Q' 'a') ('Q*' 'a~'))    (('Q'  *  )('Clock' 0 ))   )
  ( (('J' 1 )('K' 0 )('Clock' 0 ))    (('Q'  * )('Clock' 1 ))
    4  (('Q'  1 ) ('Q*'   0 ))    (('Q'  1  )('Clock' 0 ))   )
  ( (('J' 0 )('K' 1 )('Clock' 0 ))    (('Q'  * )('Clock' 1 ))
    4  (('Q'  0 ) ('Q*'   1 ))    (('Q'  0  )('Clock' 0 ))   )
  ( (('J' 1 )('K' 1 )('Clock' 0 ))    (('Q' 'a')('Clock' 1 ))
    4  (('Q' 'a~')('Q*' 'a' ))    (('Q' 'a~')('Clock' 0 ))   )
)
```

図 4.5: J-K フリップフロップの動作定義表

部品に対する入力があった場合，この動作定義表を先頭から順に調べて，入力端子の値及び内部状態の値が一致するものを見つけ，そこで定義される遅延時間の後に，動作定義表で定義される“値”を出力端子に出力し，その表に従って内部状態を変化させる．

GCSでは，このような動作定義表を与えるだけで，部品定義でき，どの部品も同じように扱える利点を持つ．既存の部品も同じ方法で定義している．

4.5 考察

この章では，簡単な計算機を作成することによって，実際にどの程度の時間で，操作もしくはシミュレート出来るかを示す．また，実現法についても考察する．

4.5.1 計算機の構成例について

図 4.6 に本シミュレータで簡単な計算機を構築し，シミュレートした例を示す．ここで，作成した計算機は，以下のような特性を持つ．

1命令の長さ	6bit
命令の種類	12種類 (Load, Store, 加減・論理演算, 分岐など)
アドレス指定	直接，間接，即値
1命令	20 クロック
部品数	16

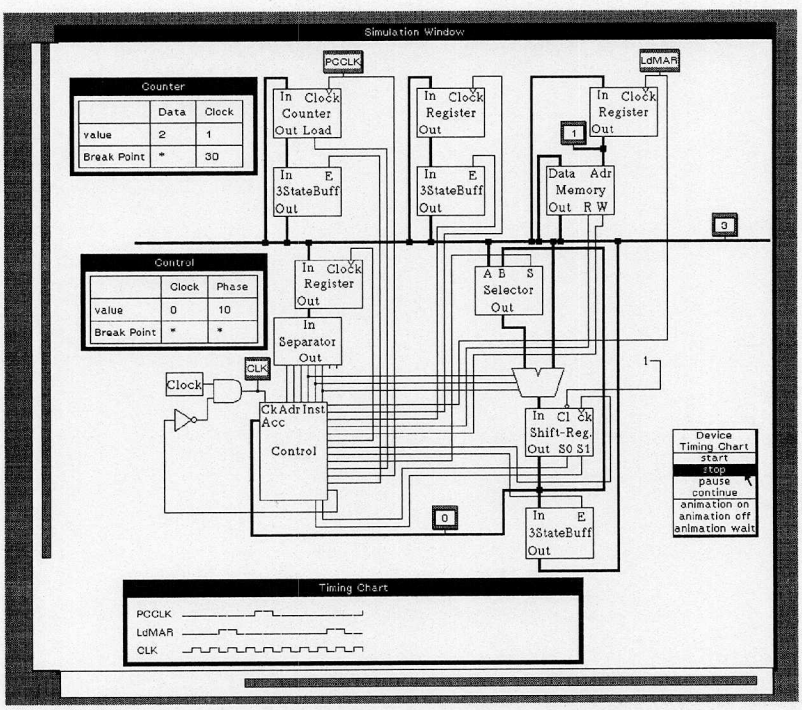


図 4.6: 計算機構成例

この程度の計算機を，AND，OR，レジスタ，ALU などの部品のみを用いて構成すると，多くの部品とかなり複雑な配線を必要とする．そこで，図4.6に示す計算機では，高機能な Control 部品を定義し，それを用いて回路を構成することで，全体の部品数を減らしている．Control 部品は，データライン (アドレス情報も含む)，命令レジスタ，および，クロックを入力とし，命令解読機能を実現しており，ALU，レジスタやその他のゲート等の部品を制御している．Control 部品の動作定義表の大きさは 39 行である．Control, ALU 以外の部品は，ほぼ，現実の IC と対応してお

り，その間の配線は，現実の計算機とほぼ同じである．ただし，抽象化されたワイヤについては，実際には8本接続しなければならない．

シミュレーションの時間の細かさは，マシンクロックの単位であり，パルスの立ち上がり，立ち下がり，未確定値の処理などは，シミュレーションの対象としていない．

この計算機を画面上で構築するのに要した時間は，約1時間で，Sun3/75上のParcPlace Smalltalk-80で動作させたときは1命令のシミュレートには約1秒を要した．これは，計算機のアーキテクチャを観察する際には，十分な速度であろう．むしろ，アニメーションなどを行なう時には，遅延時間をいれて表示している．機械語レベルでの動作を見るには，1命令に1秒要していたのでは，やや遅いが，命令数が少ない状況では問題ない．

以下に，3名の情報工学科の学生(4年)に対して(形態3)によりGCSを試用した結果を示す．

まず，学生には，(1)ブロック図レベルの回路図(図4.6の回路図から制御線を除いたものに対応)，および，(2)Controlを含むすべての部品の動作の仕様書を提示する．さらに，各部品の動作の説明と，図4.6の計算機のアーキテクチャの概要，および命令セット説明する(約2時間)．説明終了後，3名の学生が，正しく動作する計算機を作成するまでの時間は，それぞれ，約3時間, 4時間, 7時間であった．

この試行は，計算機アーキテクチャをある程度理解している学生に対して行なったので，教育効果を論ずることは難しいが,「計算機を構成途中でも逐次シミュレーションができたので，比較的容易に計算機を構築できた」との感想を得た．

また，この試行より，次のようなGCSの問題点が指摘された．(1)部品をうまく配置していないと配線が入り組んで画面が見にくくなる．(2)未配線端子，出力信号どうしの短絡などの回路の誤り検出機能がない．(1)については，部品を少し動かす程度ならば，配線の接続関係を保存して，配線の描画を自動的に修正する機能が役立つ．(2)については，回路の観察機能によりある程度発見することが出来る．しかし，いずれの場合もこれらの機能だけでは十分でない．これについては，**4.6**で改良の方向を示す．

4.5.2 実現法について

既に述べたように，GCSはSmalltalk-80を用いて実現したが，新たに定義したクラスは，全部で25個であった．メソッドの数は，全部で約250であり，プログラムの行数は，約2,000行であった．

このように，比較的コンパクトに作成できたのは，Smalltalk-80のインヘリタンス

の仕組みが有効であったことと，我々が開発したウィンドウ制御クラスDOWLAND[24]を用いたことが大きな要因である．DOWLANDを用いたことにより，ウィンドウのサブクラスを作成する必要がなくなり，ユーザインタフェース部の記述量が少なくて済んだ．さらに，ウィンドウのスクロール機能により，ウィンドウのサイズよりも大きな回路を描くことも可能になった．

4.6 結言

計算機アーキテクチャ教育のための計算機シミュレータGCSの機能およびその実現方法について述べた．GCSは画面上で計算機の構築，および，シミュレーションが可能な教育用ツールとして，講義よりも実物感が強く，実物のハードウェアより操作が簡単であるので，計算機アーキテクチャ教育に有用なシステムである．

現在のシミュレータの問題点と改良方向を以下にあげる．

(1) 部品の配置，配線が入り組んで複雑になる場合がある．これに対して，配線等を階層化して不要な表示を行わないようにしたい．
(2) 未配線端子，出力信号どうしの短絡などの回路の誤り検出機能がない．未配線端子に対しては，シミュレーションを始める際に検出し，警告を発することができるようにしたい．また，出力信号どうしの短絡に対しては，配線時に直ちに検出できるようにしたい．
(3) 部品の定義の際に定義表をリスプ風に文字列として用意しなければならないが，これを画面上で動作定義表を入力し，その場で，矛盾のチェックなどが行なえるインタラクティブなツールにしたい．

今後の課題としては，いくつかの 回路を画面上で組み合わせて一つのブロックとして定義できるようにすること，逆にそのブロックを元の回路に展開して表示することなどがあげられる．

5 章

対話型アニメーションの方法

5.1 序言

近年, ビットマップディスプレイを備えた汎用のワークステーションが急速に普及してきており, これらを用いたCAI[37],[45] やプレゼンテーションシステム[43],[46],[50] として, 計算機の画面上での簡易アニメーションが注目され始めている. このようなシステムでは, 通常 2 次元のアニメーションを対象としており, 手軽にアニメーションが作成できることが要求されている. また, これらのアニメーションでは, アニメーションの実行時に, マウスやキーボードなどの, アニメーションの観察者 (以下, ユーザと呼ぶ) からの入力を受け付けることができ, それによって, 画面の表示内容を変更できるような**対話的**な使い方も要求される.

アニメーションに登場する各オブジェクト (以後, *Cast* と呼ぶ) の動きは, 例えばアルゴリズムアニメーション[37] などでは, 単に時間的な前後関係を指定できるだけで十分な場合もある. しかし, 例えば, CAI の教材として物体の運動の様子をアニメーションで表現するような場合には, 画面内での物体の速度を指定できる機能が必要である (**実時間性**). しかも, アニメーションを行う計算機の CPU や画面の速さにかかわらず, アニメーション自身の記述として, シナリオで画面内での絶対的な移動速度を指定できる方が望ましい. また, マルチメディアシステム[43],[46] などへの応用を考慮すると, 音声や映像などの他のメディアとの同期の必要性からも実時間性が望まれる[1].

従来, コンピュータによるアニメーションの研究は, 個々のフレームを前もって作成し, それを 1 フレームずつビデオに収録しておくという方法に関するものが中心であった[47]. 汎用のワークステーション上でのアニメーションを目指したものと

[1]実際, 本システムはマルチメディアプレゼンテーションシステム Harmony[46] でのアニメーション表示に利用している.

しては, X Window のサーバの拡張機能として, multi-buffering の方法[38] があるが, これは同時に 1 つの *Cast* しか動かせず, 時間の制御も行なえない. また, Macintosh 上では Director[43] などの対話的にアニメーションを作成できるものがあるが, 実時間性は保証されていない. 上述したような, 汎用ワークステーション上での対話型かつ実時間の簡易アニメーションを目指した研究は見当たらない.

アニメーションは, 通常, 短い時間間隔で次々と静止画像 (フレームと呼ぶ) を更新していくことで, *Cast* の動きを表現する. フレームの表示間隔が長くなる (例えば 0.3 秒) と動きがスムーズでなくなる. *Cast* の大きさや傾きが変化したり多くの *Cast* が動いているような場合, 一般にフレームに表示すべき画像の計算や表示に時間がかかるので, 決められた時間内 (例えば 0.1 秒) にフレームの更新を行なうには, フレームを素早く更新するのに必要な画像情報等 (実行時情報と呼ぶ) を予め用意しておかなければならない[2]. しかし, 既に述べたように, 対話型のアニメーションでは, 実行時のユーザからの入力に依存してその表示内容を変化させることができなければならないので, アニメーションに先立ってすべての表示画像を予め用意することは不可能である.

そこで, 本研究では, 汎用ワークステーション上での対話型のアニメーションに対して, 実時間性を損なわず, かつ, できるだけスムーズなアニメーションを行うために, どのような実行時情報をどのようにして作成し, その情報を実行時にどのようにして利用するかの方法を提案している. さらに, それを用いたアニメーションシステム (**RAS**) を X Window[39] 上で試作し[40],[42], 実験により, その方法の有効性を確認した.

本論文で提案する方法は, アニメーションの作者およびユーザが計算機 (CPU およびディスプレイ) の速さの違いを意識する必要はなく, 計算機の能力に適した実行時情報を自動収集すること, アニメーションの実行時に利用者からの入力によって動的に表示画像が変化するような場合にもなるべくスムーズな表示を行なうこと, および, メモリの使用量が比較的少ないなどの特徴を持っている.

5.2では, 本研究で対象とするアニメーションを明示し, アニメーションの記述法, および, 実行時の表示方針を述べ, さらに, **RAS** の構成を概観する. **5.3**では, 提案するアニメーションの方法を説明する. さらに, **5.4**では, 実験により本方式による表示法の評価検討を与えている.

[2]CPU の高速化や表示装置の高速化によって対処する方法もあるが, これらの方法は, 一般に高価なハードウェアを必要とする. 通常の汎用ワークステーションの場合, CPU は比較的高速になってきているものの, 高速の表示装置などは利用できないのが普通であるので, ここでは, ハードウェアによる対策は考えない.

5.2 アニメーションの基本方針

5.2.1 対象とするアニメーション

対象とするアニメーションは2次元であり, アニメーションに登場する個々の *Cast* の形状は, 平面画像 (*image*) によって表す. ただし, *Cast* 同士の遠近関係が存在し, 重なりあった *Cast* に対して, 遠方にある *Cast* は近くの *Cast* によってその一部もしくは全部が隠される.

個々の *Cast* の動きは, **シナリオ**に記述するものとする. *Cast* の動きとしては, 指定した方向もしくは位置への平行移動 (*Cast* 自身の *image* が変化しない) の他, 拡大・縮小や回転を伴うような *Cast* 自身の *image* が変化するものがある.

アニメーションの作者は, 次節で述べるシナリオ記述言語 **SDL** によりシナリオを記述し, **RAS** を用いて, アニメーションを試行し, 実行時情報を収集する. ユーザは,**RAS** を用いて実行のみを行なう. フレームの表示間隔 (例えば 100msec) は, アニメーションの作者が任意に指定できる.

5.2.2 シナリオの記述

シナリオでは, *Cast* の形状が定義できること, それぞれの *Cast* の動作を記述できること, および, アニメーションの実行時のマウスやキーボード入力に対する処理が記述できる必要がある. 以下では, これらの記述を行なうためのシナリオ記述用言語 **SDL** について述べる.

Cast の定義

Cast の形状は, その形状を表す画像データ (**Cast-Image**) とそれを画面上に描くときのマスクパタン (*mask*と呼ぶ) の組として定義する[3]. Cast-Image および *mask* は, 長方形領域の bitmap データであり, 画面上に描かれる形状は, Cast-Image と *mask*のビット毎の論理和をとったものである. これによって, 穴のあいた形状の *Cast* に対して奥にある Cast-Image を見通すことが可能になる. Cast-Image および *mask*は, 別ファイルとして予め作成しておき, シナリオには *Cast* 名とそのファイル名の対応を指定する.

[3] *Cast* の形状は, 1つの固定した Cast-Image の他に, 時間とともに変化する形状 (例えば, 人が歩いている様子など) を表現するために複数の Cast-Image によって定義することもできる. また, 現在のところ RAS では, 各 *Cast* を単一色でしか扱えない.

動作の記述

個々の *Cast* の動きとしては, 画面上での平行移動, 拡大縮小, および, 回転などの基本動作による指定ができる. 指定した時間内での *Cast* 毎の動作を記述したものを *Action* と呼ぶ. 1 つの *Action* において各 *Cast* の動きは, 基本動作で記述しなければならない. *Action* の記述例を以下に示す[4].

```
in 5 sec   # 5 秒間に以下の動作を行なう
   mov CastA to (100, 200)
          # CastA を (100,200) に等速直線移動
   res CastB to (200, 300) x (1.2, 1.2)
          # CastB を (200,300) に等速直線移動
          # しながら同時に x 軸および y 軸方向
          # に 1.2 倍に拡大する
end
```

変数の値や *Cast* の座標値などを用いた論理式の真偽により *Action* の系列の実行順序を変えるための条件分岐や繰り返しなどの制御構造を用意している. この他, アニメーションの実行を制御するために, 画面上の表示を一旦停止する **pause** 命令や, マウスなどの割込み制御を行なう命令などがある.

イベント処理の記述

実行時のマウスやキーボードなどのユーザからの入力に対する処理を記述できなければならない. **SDL** では, 実行時のマウスやキーボードなどの入力をイベントとみなす. イベントは *action* の実行中でも発生し, 実行中の *action* を中断して, シナリオ中の**イベント処理記述部**に制御を移す. イベント処理記述部では, 変数の設定などの処理を行ない, その後, 中断中の *action* を強制終了させて次の *action* の実行を始めるか, その *action* を再開するかを選択できる. シナリオの**動作記述部**では, イベント処理記述部で設定した値を条件文等で参照し, 実行順序や *Cast* の配置等を変えることができる.

[4] *Action* の記述には, その *action* の開始時の *Cast* の位置や大きさ, 傾きなどは含まれないことに注意. 新たな *Cast* を指定した場所に登場させるには, put コマンドを用いる.
(例) **put** *CastA* **at** (100, 200).

5.2.3 アニメーション実行時の表示方針

実行時情報としては, アニメーションに登場する各 *Cast* の様々な大きさ・傾きの画像情報 (**Cast-Image 情報**と呼ぶ), または, 一つのフレームに対して次のフレームとの画像の差分 (**差分情報**と呼ぶ) を用いる.

アニメーション実行のフレームの更新方法は以下の基本方針に基づく.

(1) 実行時情報がなくてもスムーズに表示できる区間は, (メモリ使用量を少なくするため実行時情報を用意せずに) その場で各 *Cast* の位置, 大きさ・傾きを計算し, それに基づいて必要ならば *Cast* の画像を計算し, 各 *Cast* を重なりの順に描く.

(2) 実行時情報がないとスムーズに表示できない区間は, Cast-Image 情報, もしくは, 近似の Cast-Image 情報を用いて, (1) と同様に各 *Cast* を重なりの順に描く方法でフレームの更新が間に合うならば, その方法により表示する.

(3) (2) の方法でフレームの更新が間に合わない区間に対しては, 差分情報を用いて表示する.

差分情報よりも Cast-Image 情報を優先して用いる理由は, 一般に前者の方が多くのメモリを必要とし, 後者は再利用できる可能性が高いからである[5].

5.2.4 システムの構成

RAS のシステム構成を図 5.1 に示す. データ管理部 (DM) は, シナリオや実行時情報のセーブ／ロードを行ない, それらを内部のデータとして格納管理し, 描画部からの要求にしたがって, シナリオを逐次解析し描画に必要な情報を返す. 描画部 (DC) は, 主に, タイマ割り込みによって起動し, 試行時か実行時かによってその処理は異なるが, 基本的には, DM からの情報をもとに, フレームの更新を行なう[6](詳細は次章). ユーザインタフェース部 (UI) は, アニメーションの作者やユーザの指示を受付ける部分で, 実行時情報のセーブ／ロードおよび実行の制御などの指令を受付け, DC にその実行を要求する.

---

[5]実行時の入力が試行時と異なった場合でも, それによって *Cast* の移動方向が変わる程度の場合, Cast-Image 情報を利用できる. また, 違う場面でも同じ大きさ・傾きの Cast-Image が必要なときも再利用できる. 後述のように近似の Cast-Image を利用することで, さらに, 再利用できる可能性が高まる.

[6]画面への描画そのものは, **X Window** の基本描画機能 **Xlib** [39] を用いた手続きからなる. この方法以外に, X Window では, それぞれの *Cast* を Window として作成し, Shape Extension 機能[44] を用いて, *Cast* の Window を移動, 拡大縮小することで, アニメーションを実現する方法もある. しかし, この方法は, *Cast* の表示時間がその Cast-Image の形状に大きく依存し, 長方形などの単純な形状でない場合, その表示に多くの時間を要し実時間アニメーションには向かない.

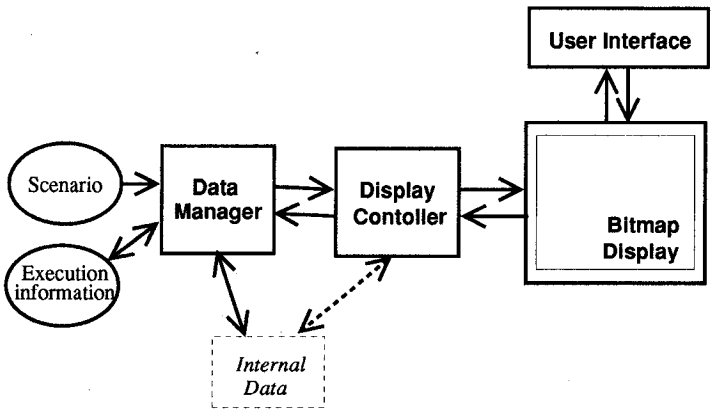


図 5.1: **RAS** の構成図

5.3 RAS でのアニメーションの方法

5.3.1 実行時情報の収集

計算機に応じた実行時情報を得るために, アニメーションの実行に先立ちアニメーションを実行する計算機・ディスプレイを用いて, アニメーションを**試行**する. そして, どの区間はどの表示法で間に合うかを自動測定し, 適切な実行時情報を自動生成し保存する. この方法により, 作者は, 計算機毎の速さの違いを意識しながら, いちいち計算して実行時情報を求める必要はない.

アニメーションの試行は, フレームの表示間隔 T とシナリオを指定して, 以下の**試行手続き**を実行する. なお, 実行時のマウスやキーボードなどの入力を必要とするシナリオに対しては, この手続きのフェーズ 1 においてユーザに変わってアニメーションの作者が入力を与える. フェーズ 1 では, ユーザ入力を受付けてシナリオの実行の流れを一つ固定し, その流れに沿って実行時情報を収集する.

- **フェーズ 1: フレーム更新時間の測定と Cast-Image 情報の計算・保存**

タイマの割込み間隔を T とし, タイマ割込み毎に以下の手続きを実行する.

(1) シナリオ内の現在実行中の *Action* に記述されている各 *Cast* に対して, 割込み発生時刻における位置, 大きさ・傾きを計算し, いずれかに変化のあった *Cast* を画面から消去する (*Cast* の *mask* を用いて背景色で塗りつぶす. この *Cast* と重なり部分を持つ *Cast* の画像も消されるが, この部分は (3) で修復する.).

(2) 前フレームから大きさ・傾きが変化した *Cast* についてはそれらの Cast-Image を計算で求め保存する.

(3) 消去した *Cast* の新たな Cast-Image を新たな位置に重なりの順に描き, 次のフレームを作成する (この描画法を**重描き法**と呼ぶ). このとき, 前のフレームで消去した *Cast* と重なり合っていた *Cast*, および, 次のフレームで描き直した *Cast* の上に重なる *Cast* も含めて重なりの順に描き直す[7].

上記の手続き中は割込み禁止であり, フレーム更新が T 以内に終らなかった場合は, フレームの表示が終了した後, 直ちに割込みがかかる. 計算・表示に 2T 以上の時間を要した場合は, 途中の割込みは無視され, 割込みは一度だけしか起こらない (図 5.2).

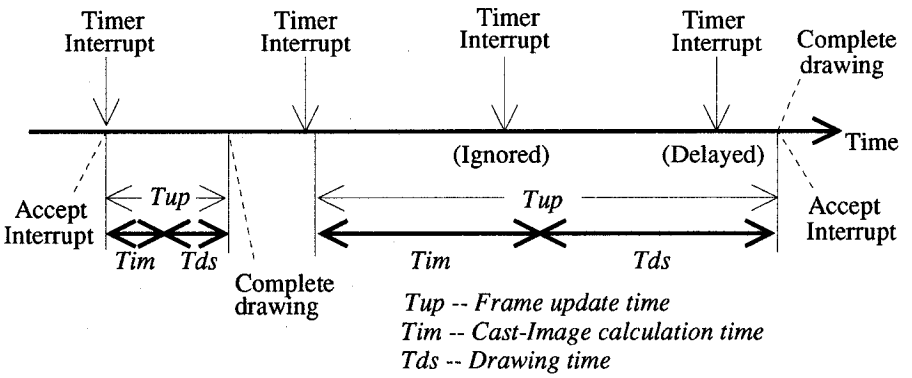


図 5.2: タイマ割り込みと計算・表示時間

上記の方法で次々とフレームを描画しながら, 各フレームの更新時間を測定する. このとき, 時間 T 内に表示できなかったフレーム, および, 割り込みが受付けられずに表示できなかったフレームの集合を **NF1** とする. さらに, **NF1** のうち, 割り込みが受付けられずに計算・表示を試みなかったフレームに対して, 大きさ・傾きが変化した *Cast* の Cast-Image を計算し保存する.

- **フェーズ 2: Cast-Image 情報を用いた描画**

フェーズ 1 で計算した Cast-Image 情報を用いて, 重描き法によりフレームの更新を試みる. この時, T 以内に更新できないフレームの集合を **NF2** とする.

- **フェーズ 3: 差分情報の計算・保存**

NF2 の各フレームについて, 前のフレームとの差分情報 (図 5.3) を計算し *action* 内の時刻と共に保存する.

[7]前者は, *Cast* の消去によって, それと重なりあっている *Cast* もダメージを受けているので, その image を回復するためであり, 後者は, 次のフレームで移動, 変形した *Cast* の上に重なる *Cast* を正しく表示するためである.

- **フェーズ4: 差分情報を用いた描画**

NF2 のフレームに対して, 差分情報を用いて描画しながら (**差分表示法**と呼ぶ), フレームの更新時間を測定する[8]. 時間 T で描画できない区間については, 描画可能な時間間隔を測定し, それを記憶する.

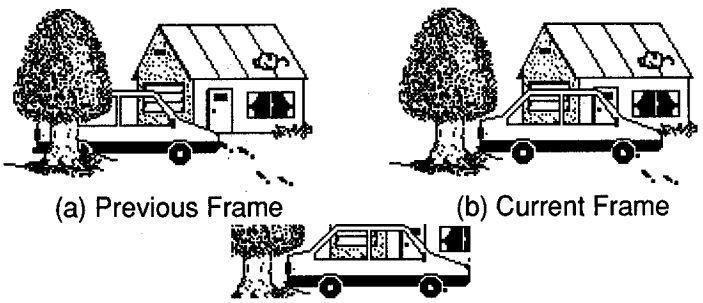


図 5.3: 差分情報

5.3.2 実行時情報の保存法

Cast-Image 情報は, 同じ *Cast* が同じ大きさ・傾きで別の場面に登場したときにも再利用できる. この再利用を容易にするため, Cast-Image 情報については, 各 *Cast* ごとにまとめ, 大きさの順にソートし, さらに, それぞれの大きさ毎に傾きの順にソートして保存している.

ある時刻における, 画面上の各 *Cast* の位置, 大きさ・傾き, 現在実行中の *Action* で参照している各変数の値, および, フレームの表示間隔の組を, その時刻における**アニメーションの状態**と呼ぶ. 実行時の入力等が試行時と異なる場合などでは, 試行時と同じ *Action* の記述を実行する場合であっても, **アニメーションの状態**が異なる場合がある. このような場合, 一般に差分情報を利用できない. 実行時に差分情報を利用できるのは, 試行時と実行時の**アニメーションの状態**が一致したときに限る.

差分情報の保存に際して, 個々のフレームの差分情報毎に, アニメーションの状態を保存するのは, 多くのメモリ領域を (ディスクに保存する場合にはその領域も) 必要とする. 実行時において, ある *action* に対してその開始時の**アニメーションの状態**が試行時と同じであるならば, その *action* の途中でマウスなどの入力がない場

[8]実際には, 差分情報として長方形の集合を用いた表示の他, この長方形集合全体を覆う 1 つの長方形領域を用いた表示も試み, いずれも T 内に表示できたならば, メモリの占有量を減らすため, 面積の和の小さい方を差分情報として採用している.

合, *action* の途中におけるアニメーションの状態は, 互いに常に等しい. そこで, 図 5.4 に示すように, 差分情報を *action* 毎にまとめ, 各 *action* に対して, 試行時におけるその *action* の開始時のみのアニメーションの状態を保存し, その状態に対して, その *action* が開始してから差分情報が適応できる各時刻と, その時刻での差分情報の組を保存している. 従って, 利用できる差分情報があるかどうかは, *action* の開始時に判断するだけで良い.

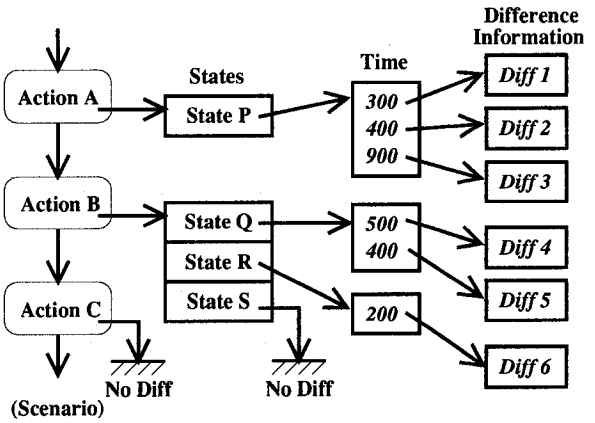


図 5.4: 差分の保存

また, while 文などによって 1 つの *action* の記述が, 繰返し実行される場合や, 同じシナリオに対して何度も実行時情報の収集を行なった場合など, 同一の *action* に対して, 複数のアニメーションの状態から開始したときの差分情報を収集しておく必要があるので, これも可能にしている.

5.3.3 実行時の表示法

実行時には, できるだけ収集した実行時情報を用いて表示する. ただし, 実行時のユーザからの入力が試行時と異なる場合などでは, 用意した実行時情報が利用できず, スムーズなアニメーション表示ができない可能性がある[9]. このような場合もなるべくスムーズな表示が得られるようにしている. 実行時の表示方法を以下に述べる.

[9]この他, 他のプロセスの負荷などの影響によって, 十分な CPU 資源が割り付けられなかった場合にもフレームの更新に時間がかかることがある. この問題に対する本質的な解決は, リアルタイム OS を用いるなどにより対処すべきであり, 本稿では議論しない. しかし, 通常の使用状況において, 同じ計算機上で他のアプリケーションプログラムを走らせないようにすれば (daemon プロセスは動いていても良い), このような影響はほとんど無視できる.

まず, 各 *action* の開始時にアニメーションの状態をチェックし, 同じ状態からの差分情報が保存されている場合は, 保存されている区間については, 差分表示法により表示する. ただし, その *action* の実行途中でマウスなどの入力があった場合, それ以降は差分表示法を用いない. そのような場合, および, もともと差分情報が保存されていない区間については, 描くべき大きさ・傾きの *Cast* の Cast-Image が保存されているならば, それを用いて *Cast* の重なりの順に重描き法により表示する. ちょうど同じ大きさ・傾きの Cast-Image がなくても, 近似の大きさ・傾きの Cast-Image 情報が存在するかを調べ[10], あればそれで代用する. 近似のものも保存されていない場合にのみ, その場でその Cast-Image を計算しそれを用いて表示する. 実行時情報の生成法より, ユーザからの入力がない場合には, その場で計算してもスムーズな表示が可能である.

ユーザ入力によって新しい場面展開が必要となり, その場で Cast-Image を計算しなければならないとき, その計算に時間がかかりコマ落としせざるを得ない状況になることもあり, そのときはスムーズさに欠ける. これに対しては, 試行時に, 幾通りかの入力を与え実行時情報の収集を繰返し行なうことで, 実行時に実行時情報の利用できる確率を高くすることができる[11].

また, 差分情報を用いる場合であっても他のプロセスの負荷などによって, フレームの表示間隔内に表示が完了しない場合が生じ得る. このとき, 実時間性を優先するため, 途中のフレームの表示を飛ばして, その間の差分情報をメモリ上で連続して適応し, 次に表示すべきフレームを生成し表示する.

5.4 評価と検討

5.4.1 実行例に基づく評価検討

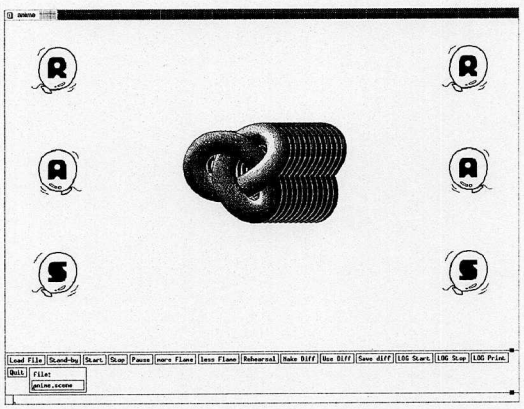
本節では, アニメーションの実行例に基づいて, 評価・検討を与える. アニメーションのスムーズさは, 1 秒間に表示するフレーム数 (FPS) に依存するが, 一般に FPS が 10 以上になると, ほとんどの人がスムーズに感じる. そこで, 以下の議論では, 1 秒間に 10 フレーム表示することが可能な時, スムーズなアニメーションであるとみなす.

[10] 最も近い大きさ・傾きの Cast-Image で, かつ, 大きさ・傾きの差がともに 10%以内のものを近似図形と見なしている.

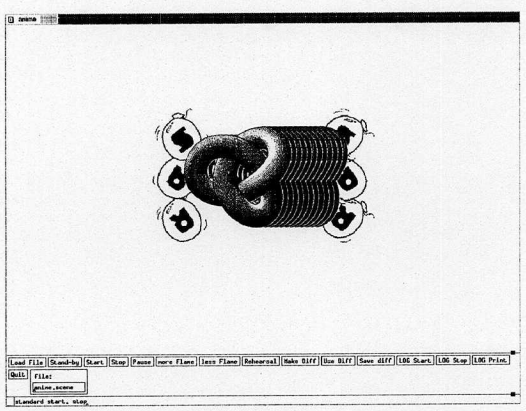
[11] ただし, その場合一般にメモリの使用量は増大する.

ユーザからの入力がない場合の実行例

まず, ユーザからの入力がない場合についての実験結果を示す. 以下のシナリオ A に対し,X Window のサーバを DECstation3100(メモリ 12MB), クライアント (**RAS**) を Sun4/370(メモリ 32MB) として実行時情報の生成およびアニメーションの実行を行なった.



(a) $T = 0\ msec$

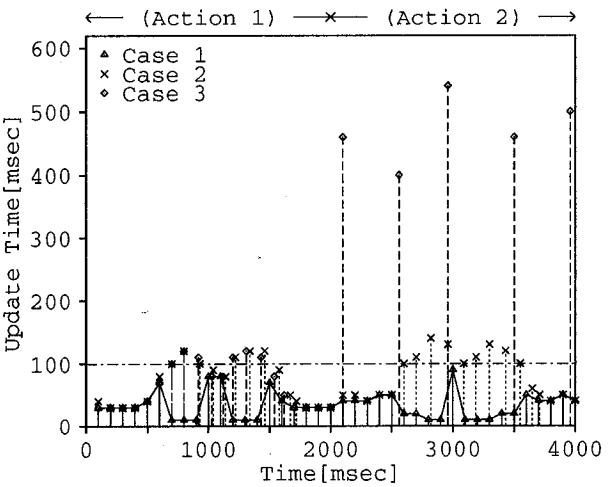


(b) $T = 3600\ msec$

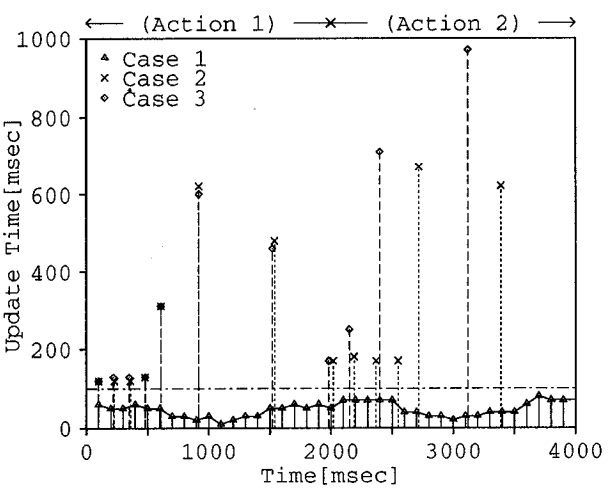
図 5.5: シナリオ A の表示画面

(シナリオ A) 画面の左右に, 3 つずつ 100x100 の大きさの *Cast* を配置し, 画面中央に 150x150 の大きさの *Cast* を 12 個重ねて配置する (図 5.5(a)). 最初の 2 秒間で, 左右に置いた 6 つの *Cast* を画面の中心を通るように直線的に画面の端まで平行移動させる (Action-1). このとき, 左右の 3 つずつの *Cast* は, 上部のを奥にして, 中央のすべての *Cast* の向う側を通す. 次の 2 秒間でこの 6 つの *Cast* をそれぞれ 360 度/秒の速度で回転させながら (図 5.5(b)), 元の位置に直線的に移動させる (Action-2).

RAS は, このシナリオに対して, 試行時には, 700～900msec, 1200～1400msec, 2600～2900msec, 3100～3500msec の区間では, (画面中央の *Cast* の重ね描きに時間を要するために) 差分情報を収集した. また, 2100～2500msec, 3000msec, 3600～4000msec の区間では, (動いている *Cast* が回転しており, それぞれの *Cast* の image を計算するのに時間がかかるので) Cast-Image 情報を収集した. その他の区間については, 実行時情報を収集していない. アニメーションの試行に要した時間は, 実行時情報を生成する時間を含めて 34 秒であった.



(a) Server: DECstation3100



(b) Server: Sun3/80

図 5.6: フレームの更新時間 (シナリオ A)

実行時のアニメーションの経過時間と各フレームの更新時間との関係を図5.6(a)のCase1に示す．各点のX座標はフレームの計算を始めた時刻を表し，Y座標はそのフレームの更新時間(Tup)を表す．判り易くするために，各点を折れ線で結んでいる．実行時のフレームの更新時間が，全区間で100msec以内に収まっていることがわかる．この内，0～600msec，1000～1100msec，1500～2000msecの間は実行時情報を使わずに表示している．その他の区間は，試行時に収集した実行時情報を用いてフレームの更新を行なっている．なお，実行時情報のメモリの使用量は，Cast-Image情報が$226KB$，差分情報が$172KB$であった．

参考として，もし，Cast-Image情報のみを用いて表示した場合の結果をCase2，実行時情報を全く用いない場合の結果をCase3のグラフとして示す．Action2で*Cast*の回転動作を伴うために，Case3では，スムーズな表示が行なわれていない．DECstationは比較的画面表示が速いために，Cast-Image情報だけでもわずかな遅延だけで，ほぼスムーズな表示が得られている(Case2)．

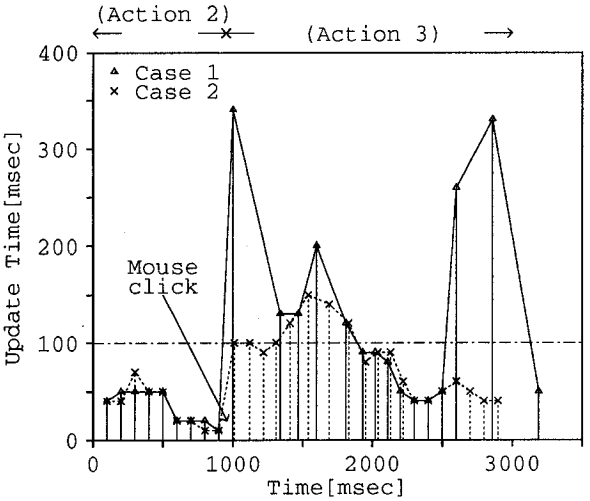
同様の実験をX Windowのサーバを Sun3/80(白黒ディスプレイ)，クライアント(**RAS**)をSun4/470として行なった結果を図5.6(b)のCase1に示す．Sun3/80の場合も，実行時のフレームの更新時間が全区間で100msec以内に収まっていることがわかる．Sun3/80では画面の表示速度がやや遅いため，すべて差分情報を用いている．実行時のメモリの使用量は，差分情報が604KBであった．参考として，Cast-Image情報だけを用いた場合の結果をCase2，実行時情報を全く用いない場合の結果をCase3のグラフとして示す．Sun3では，Cast-Image情報では，十分な更新速度が得られず，差分情報が有効に働くことが分かる．

同じシナリオに対して，それぞれの計算機の能力に適した実行時情報が自動的に収集でき，いずれも，スムーズな画面表示が得られた．

ユーザからの入力がある場合

アニメーションの途中でユーザからの入力を受付けるようなシナリオの実行例を示す．以下のシナリオBに対し，X Windowのサーバを DECstation3100(メモリ12MB)，クライアント(**RAS**)をSun4/370(メモリ32MB)として実行時情報の生成およびアニメーションの実行を行なった．

(シナリオB) ユーザからの入力がない限り，シナリオAのAction-2と同様の表示を行なう．ただし，マウス入力を許す．アニメーションの開始t秒後$(0 \leq t \leq 2)$に，画面上の任意の位置でマウスをクリックしたとき，移動中の各*Cast*のその後の動作は，移動方向を変えず，回転の向きのみを逆にして(回転速度は，$180t$度/秒)，2秒間で目的地に到達する(Action-3)．



(a) Server: DECstation3100

図 5.7: フレームの更新時間 (シナリオ B)

実行時情報生成のためのアニメーションの試行を 1 回だけ行ない, その際マウス入力は与えていない. このときの実行時情報をもとに, 実行した時のフレームの更新時間を図 5.7の Case1 に示す. なお, 実行時には, t=0.9〜1.0 秒の間にマウスをクリックした.

シナリオ B では, 任意の時点でマウスをクリックでき, その瞬間から移動中の各 *Cast* の回転方向が変わる. 従って, マウスのクリック後は, 差分情報は利用できなくなる. しかし, 個々の *Cast* の刻々の形状については, Cast-Image 情報が利用できる場合もあるので (この実験の例では, 3 つのフレーム (T=1000, 2600, 2800msec) を除いて Cast-Image 情報が利用できている), 表示間隔は大きくなるが, Cast-Image 情報を全く利用しないよりは滑らかな表示が得られる. この時のメモリ使用量は, Cast-Image 情報が 226KB, 差分情報が 114KB であった.

さらに, 2 度目のアニメーション試行を行ない, その際に途中 (t=1.7〜1.8 秒の時点) でマウスのクリックを行って実行時情報を追加収集した (93MB 分の Cast-Image の増加). その後もう一度実行させたときのフレームの更新時間を図 5.7の Case2 に示す. なお, このときも実行時には, t=0.9〜1.0 秒の間にマウスをクリックした. 後者の場合, 2 度の試行により Cast-Image 情報が増えており, すべてのフレームで Cast-Image 情報が利用でき, Case1 より滑らかな表示が得られている.

この例では試行を繰返すことで, Cast-Image 情報が増え, それが実行時の表示に役立ったが, シナリオによっては, 試行を繰り返してもなかなか実行時に有効な Cast-Image が得られないこともある.

5.4.2 実行時の各処理に要する時間

まず, 各 *action* の開始時に行なう, 差分情報が利用できるかどうかのチェックに要する時間を測定した. 同時に動く *Cast* の数を 20 個とし, 1 つの *Action* に対して, 異なるアニメーションの状態からの差分情報が 100 個存在する場合の平均チェック時間を測定したところ 21.7msec であった (クライアントは Sun4/370). このことから, 実用的な範囲では, このチェック時間はほとんど無視できると思われる.

次に, 各フレームの計算におけるそれぞれの時間を調べた. まず, 各 *Cast* の位置, 大きさ・傾きは, シナリオから容易に求めることができ, これに要する時間は, 他の時間に比べて十分小さい. Cast-Image の計算は, 求める Cast-Image の各 pixel 毎にその値を計算するので, 計算時間はその面積に比例する. 100msec 以内に計算できる大きさは, Sun4 で 130x130dots,DECstation で 120x120dots, Sun3 で 50x50dots 程度である.

差分情報による表示の限界については, 例えばサーバが DECstation(モノクロ) の場合, 100*msec* 以内に画面全体 (1024x864) を書き直せるので, メモリの使用量を考慮しないならば, どのようなアニメーションも可能である. 長い時間のアニメーションを行なうには, メモリの使用量を押える必要があるが, このためには FPSの値を小さく指定することで, (アニメーションのスムーズさが犠牲になるが,) ある程度対応できる.

サーバがSun3/80 の場合には, 100*msec* で更新できる領域の大きさは730x730dots であった. Sun3 の画面全体(1152x900) を覆うような差分情報を用いる場合には, FPS を 5 程度にしなければならない.

5.5 結言

本研究では, ワークステーションの画面上での, 対話型の簡易アニメーションに対して, 実時間で, かつ, 出来るだけスムーズなアニメーションを行なうための, アニメーションの実行時用情報の生成, および, それを用いた実行方法を提案し, その方法を用いたアニメーションシステム **RAS** を作成した. **RAS** は, X Window(X11R4) の Toolkit ライブラリを用いて記述した. そのプログラムサイズは C 言語で約 12,000 行である.

さらに, 実行時情報としての, Cast-Image 情報, 差分情報が, 機械の速さに応じて適切に自動生成されたこと, それらによりシナリオの指定通りにスムーズに表示できたことを実験により確かめた.

回転や拡大縮小の動作を伴う *Cast* が多い場合, 一般に実行時情報のためのメモ

リの使用量は多くなるが, これに対しては, 画像圧縮の技法[41]を用いることで, メモリの使用量を減らせることが期待される.

今後の課題として, アニメーションの作者がディスプレイ上で直接シナリオを作成できる機能や, **RAS** を他のアプリケーションプログラムのユーザインタフェイスとして利用するために, 他のアプリケーションプログラムからの指示によって各 *Cast* を動作させる機能等を検討することが挙げられる.

6 章

結論

本研究により得られた成果および今後の課題をまとめる．

2 章では，図形の入力および編集を容易に行なう方法について考察し，作図効率を向上させるための機能として，図形記述言語による図形の部品化機能と直接操作による図形の部品化機能，および，図形の接続包含関係の自動認識機能を提案し，それらの機能を含む作図ツール Key3 を作成した．さらに，ここで提案した機能の有効性について，代表的な例について，操作回数，および，それに要する処理時間を実測した．それらの結果は，いずれも，機能の有効性および処理方法の妥当性を示していた．作成した Key3 では，面の塗りつぶしや色付けをすることは出来ないが，これらの機能は，図形の持つ属性を追加することで比較的容易に実現することができる．論文中では触れなかったが，Key3 の図形入力および表示機能は，他の応用プログラムからも利用できるようにしている．しかし，これらの機能は，汎用の作図ツールとして基本的と思われる機能レベルのものであり，それぞれの応用プログラムでは，一般に，その分野に適した (より高い抽象レベルの) 図形の入出力機能が望まれる．そこで，分野毎に必要な図形の入出力機能を分析し，それらの機能を実現するための方法を検討することは興味深い．

3 章では，プログラムが扱う内部情報を画面上に図示し，それらに対する操作を容易に行なう方法に関して述べた．本論文では，対象とするデータ構造を計算機で広く利用されている木構造に絞り，それを画面上に美的に表示し，画面上でのマウス操作によって節点・枝の指定等の入力を可能にするライブラリプログラムを作成し，ASL システム及び LOTOS シミュレータに対して適用することによりこのライブラリの有効性を示した．また，多くの節点を持つ木に対しても有効であることを実験で示した．本システムを平面グラフや一般のグラフなどに適応することを考えた場合，現在知られている自動配置アルゴリズムでは，ユーザが満足する配置を得ることは難しい．そこで，一部の節点の配置などのいくつかの配置の制約条件を与

えて，配置アルゴリズムがその他の節点の配置を決定するなど，ユーザとの対話を行ないながら，望ましい配置を求める方法も興味深い．

4章では，図形を用いたユーザインタフェースの一つの応用例として，計算機の構造・動作の理解を助けるための計算機シミュレータとして，図的ユーザインタフェースを重視した，実行・観察機能，回路の作成機能，および，部品定義機能を有する計算機シミュレータ GCS を作成した．実際に計算機回路の入力例を挙げ，比較的簡単に入力ができること，部品定義機能が有効であることなどを示した．今後の課題としては，本シミュレータでは画面上の直接操作のみで計算機の構築を行っているが，2章で示したようなアプローチ，すなわち，ハードウェア記述言語による言語的構成方法と自由に混合して用いることが出来るようにすることも興味深い．さらに，図的ユーザインタフェースが文字列ベースのユーザインタフェースに比べて，どの程度使いやすさが改善されるのかを定量的に調べるのも興味深い．

5章では，図形を用いたユーザインタフェースをさらに一歩進めて，動きのある図形，すなわち，アニメーションを取り扱う方法について述べた．ここでは，ワークステーションの画面上での，対話型の簡易アニメーションを行なうための，アニメーションの実行時用情報の生成法，および，それを用いた実行方法を提案し，その方法を用いたアニメーションシステム RAS を作成した．さらに，実行時情報としての，Cast-Image 情報，差分情報が，機械の速さに応じて適切に自動生成されたこと，それらによりシナリオの指定通りに決められた時間内にに表示できたことを実験により確かめた．今回提案したシナリオ記述言語 SDL は，対話型のアニメーションの記述に必要な最小限の機能は備えているが，基本的には，時間順にしたがって，その時間におけるすべてのオブジェクトの動きを一緒に記述しなければならない．オブジェクト指向の考え方を取り入れ，オブジェクト毎に独立して動きを記述することもできるような，より高級なシナリオ記述言語の検討などが，今後の課題として残されている．さらに，オブジェクト同士の衝突など，オブジェクト間の関係を記述する方法，および，重力場などの全体のオブジェクトの動きに対する制約の記述方法などを検討することも今後に残された課題である．RAS は2次元のアニメーションを対象としているが，計算機の処理速度がさらに向上してきた場合，3次元のアニメーションに対しても，対話型のアニメーションがどの程度まで実現可能であるか調べることも興味深い．

謝辞

本研究に関して，ご理解あるご指導を賜わり，常に励まして頂いた谷口健一教授に心から深謝致します．

本研究をまとめるにあたり，有益な御助言をくださった情報工学科の嵩 忠雄 教授，都倉信樹教授，鳥居宏次教授に深謝致します．

筆者が大阪大学基礎工学部情報工学科および同大学院に在籍中に，御指導，御教授くださった本学情報工学科の故藤澤俊男教授，柏原敏伸教授，菊野 亨 教授，首藤 勝 教授，西谷紘一教授，橋本昭洋教授，藤井 護 教授，宮原秀夫教授，谷内田正彦教授，脇田 壽 教授，本学産業科学研究所の北橋忠宏教授，豊田順一教授，溝口理一郎教授，本学医学部バイオメディカル教育研究センターの田村進一教授に深謝致します．

研究を進める上で，貴重な御助言，御指導を頂いた甲南大学理学部経営理学科の的場裕司教授，大阪工業大学工学部電子工学科の吉岡信夫助教授に感謝致します．

図形入力の研究に関しまして，プログラムの作成等に御協力頂きました北村俊義氏 (現，野村総合研究所)，直田創氏 (現，富士通株式会社)，阪口純氏 (現，松下電器産業株式会社)，中村眞氏 (現，シャープ株式会社) に感謝致します．

木構造エディタの研究に関しまして，描画アルゴリズムに関して御指導頂いた神戸大学工学部電子工学科の増田澄男助教授に感謝します．また，木構造エディタのプログラムの作成等に御協力頂いた本学大学院生の中村亨氏に感謝致します．

計算機シミュレータに関する研究に対し，プログラムの作成等に御協力頂きました杉本直樹氏 (現，株式会社 CSK)，阪田全弘氏 (現，日本電気株式会社)，小池田恒行氏 (現，株式会社ヤマハ) に感謝致します．

アニメーションの研究に関しまして，プログラムの作成等に御協力頂きました梶本雅人氏 (現，ソニー株式会社)，日野正文氏 (現，NTT ソフトウェア株式会社) に感謝いたします．

さらに，著者の在学，在職中，積極的に御討論頂いた，嵩研究室，旧藤澤研究室，柏原研究室，並びに谷口研究室の方々に心から感謝致します．

参考文献

[1] Lewis, T.G., Handloser, F., Bose, S., and Yang, S. : "Prototypes from Standard User Interface Management Systems", *IEEE Trans. Comput.*, **22**, 5, pp.51-60 (May. 1989).

[2] Myers, B.A. and Buxton, W. : "Creating Highly-Interactive and Graphical User Interface by Demonstration", SIGGRAPH '86, **20**, 4, pp.249-258 (Aug. 1986).

[3] Sibert, J.L., Hurley, W.D., and Bleser, T.W. : "An Object-Oriented User Interface Management System", SIGGRAPH '86, **20**, 4, pp.259-268 (Aug. 1986).

[4] "MacDraw", Apple Computer, Inc. (1984).

[5] "Claris Macdraw II", Claris Corporation (1988).

[6] "Interleaf — Workstation Publishing Software User's Guide", Interleaf (1986).

[7] 国友正彦: "花子入門", アスキー出版 (1987).

[8] P. Chen, and M.A. Harrison: "Multiple representation document development", *IEEE Trans. Comput.*, **21**, 1, pp.15-31 (Jan. 1988).

[9] "Design Works — Version 1.1 User's Guide", Capilano Computing Systems (Jul. 1988).

[10] A.I. Wasserman, P.A. Pircher, D.T. Shewmake, and M.L. Kersten: "Developing Interactive Information Systems with the User Software Engineering Methodology", *IEEE Trans. Software Eng.*, **12**, 2, pp.326-345 (Feb. 1986).

[11] "PostScript — Language Tutorial and Cookbook", Adobe Systems (1985).

[12] J.F. Ossanna: "NROFF/TROFF User's Manual", *UNIX Programmer's Manual*, **2**, 7th Edition, Bell Laboratories (1979).

[13] B.W. Kernighan: "PIC — A Language for Typesetting Graphics", *Software Practice & Experience*, **12**, 1, pp.1-21 (1982).

[14] R.W. Scheifler and J. Gettys: "The X window system", *ACM Trans. Graphics*, **5**, 2, pp.79-109 (Apr. 1986).

[15] M. Nakamura, T. Matsuura, H. Naota, J. Sakaguchi, N. Yoshioka, and Y. Matoba: "An Interactive Drawing Tool on UNIX and its Figure Description Language", Proc. 1988 Joint Tech. Conference on Circuits/Systems, Computers and Communications, pp.A2-2-1~6, Seoul, Korea (Nov. 1988).

[16] 直田 創: "作図ツール key3 の評価と X Window への移植", 大阪大学基礎工学部 特別研究報告 (1988-03).

[17] Goldberg, A. and Robson, D. : "Smalltalk-80 — The Language and its Implementation," *Addison-Wesley* (1983).

[18] "IDEA Series Simulation and Modeling Reference Booklet, V7.0", *Mentor Graphics Inc.* (1990).

[19] 小池田恒行, 松浦敏雄, 杉本直樹, 吉岡信夫, 的場裕司 : "論理回路から簡単な計算機レベルまで適用可能な教育用シミュレータ", 情報処理学会第 *35* 回 *(*昭 *62* 年後期*)* 全国大会, 3Ee-6, pp.2673-2674 (1987-09).

[20] Meulen,P.S. van der : "INSIST — Interactive simulation in Smalltalk", *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp.366-376 (1987).

[21] "PC-LOGS ユーザーズ・ガイド", 日本アイ・ビー・エム, 5600-048 (1986).

[22] 杉本明, 他 : "マイクロプログラミングのためのオブジェクト指向によるハードウェアのモデル化", 電子通信学会技術研究報告, EC85-4, **85**, 19, pp.31-40 (1985).

[23] 杉本直樹, 阪田全弘, 松浦敏雄, 吉岡信夫, 的場裕司 : "画面上で計算機の構築可能な教育用計算機シミュレータ", 情報処理学会第 *33* 回 *(*昭 *61* 年後期*)* 全国大会, 4W-2, pp.2371-2372 (1986-10).

[24] 杉本直樹, 松浦敏雄, 吉岡信夫, 的場裕司 : "Smalltalk-80 上でのウィンドウ制御クラス–DOWLAND–", 日本ソフトウェア科学会ソフトウェア研究会 (関西), SW-87-2-2, pp.9-16 (1987-07).

[25] Paulish, F. N. and Tichy W. F.: "EDGE : An Extendible Graph Editor", *Software-Practice and Experience*, **20**(SI), pp. 63-88 (Jun. 1990).

[26] Karrer, A. and Scacchi, W. : "Requirements for an Extensible Object-Oriented Tree/Graph Editor", Proc. of the ACM SIGGRAPH, *Symposium on User Interface Software and Technology(UIST)*, Snowbird, pp. 84-91 (Oct. 1990).

[27] Tamassia, R., Battista, G. D., and Batini, C. : "Automatic graph drawing and readability of diagrams", *IEEE Trans. Systems, Man, and Cybernetics*, **18**, 1, pp. 61-79 (Jan. 1988).

[28] Wetherell, C. and Shannon, A.: "Tidy Drawing of Trees", *IEEE Trans. Software Eng.*, **5**, 5, pp.514-520 (Sep. 1979).

[29] Reingold, E.M. and Tilford, J.S.: "Tidier Drawing of Trees", *IEEE Trans. Software Eng.*, **7**, 2, pp.223-228 (Mar. 1980).

[30] Supowit, K.J. and Reingold, E.M.: "The Complexity of Drawing Trees Nicely", *Acta Infomatica*, **18**, pp.377-392 (1983).

[31] 西野 哲朗: "木構造プログラム図式の美的配置問題の計算量について", 情報処理学会第 *35* 回 *(*昭 *62* 年後期*)* 全国大会, 4B-8, pp.61-62 (1987-09).

[32] 藤岡 正憲 : "有向木の美的描写及びその更新について", 大阪大学基礎工学部特別研究報告 (1988-03).

[33] 東野 輝夫, 関 浩之, 谷口 健一 : "代数的仕様から関数型プログラムの導出とその実行", 情報処理, **29**, 8, pp.881-896 (1988-08).

[34] 安本 慶一, 東野 輝夫, 谷口 健一 : "LOTOS で書かれたプロトコル仕様群の実行", 信学技報, 情報ネットワーク研究会, IN91-119, pp.51-56 (1991-09).

[35] McCormach, J., Asente, P., and Swick, R.: "X Toolkit Intrinsics - C Language Interface", *MIT* and *DEC*, Massachusetts (1990).

[36] 中村 亨, 松浦 敏雄, 谷口 健一 : "木構造データの可視化ライブラリの作成と使用例", 情報処理学会第 *43* 回 *(*平 *3* 年後期*)* 全国大会, 1P-2, 分冊 **5**, pp.103-104 (1991-10).

[37] Brown, M.H., Sedgewick, R.: "Techniques for Algorithm Animation", *IEEE Software*, **2**, 1, pp.28-39. (Jan. 1985).

[38] Friedberg, J., Seiler, L., and Vroom, J.: "Extending X for Double-Buffering, Multi-Buffering, and Stereo", *MIT* and *DEC*, Massachusetts (1989).

[39] Gettys, J., Scheifler, R., and Newman, R.: "Xlib - C Language X Interface", *MIT* and *DEC*, Massachusetts (1987).

[40] 日野 正文 : "実時間アニメーションシステム RAS のデータ管理部の作成", 大阪大学基礎工学部特別研究報告 (1991-02).

[41] Hunter, R., Robinson, A.H.: "International digital facsimile coding standards", *Proc. IEEE*, **68**, 7, pp.854-867 (Jul. 1980).

[42] 梶本 雅人 : "リハーサルを用いた実時間アニメーションシステム", 大阪大学大学院基礎工学研究科修士学位論文 (1991-02).

[43] "MacroMind Director — Overview Manual", MacroMind Inc., San Francisco (Mar. 1989).

[44] Packard, K.: "X11 Nonrectangular Window Shape Extension", *MIT* and *DEC*, Massachusetts (1989).

[45] Sherwood, B.A. and Sherwood, J.N.: "CMU Tutor: An integrated programming environment for advanced-fuction workstations", *Proc. of the IBM Academic Information System University AEP Conference*, San Diego (Apr. 1986).

[46] Shimojo, S., Fujikawa, T., Matsuura, T., Nishio, S., and Miyahara, H.: "A New Hyperobject System Harmony: Its Design and Implementation", *International Conference on Multimedia Information Systems*, McGraw-Hill, pp.243-257 (Jan. 1991).

[47] Thalmann, N.M, and Thalmann, D.: "Three-Dimensional Computer Animation: More an Evolution Than a Motion Problem", *IEEE Computer Graphics and Applications*, **5**, 10, pp.47-57 (Oct. 1985).

[48] Yankelovich, N., Haan, B. J., Meyrowitz, N. K., and Drucker, S. M.: "*Intermedia*: The Concept and the Construction of Seamless Information Environment", *IEEE Trans. Comput.*, **21**, 1, pp. 81-96 (Jan. 1988).

[49] Fujikawa, T. Shimojo, S., Matsuura, T., Nishio, S., and Miyahara, H.: "Multimedia Presentation System *Harmony* with Temporal and Active Media", *USENIX Summer 1991 Technical Conference*, Nashville TN, (Jun. 1991).

[50] Meyrowitz, N. K. : "*Intermedia*: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," *Proc. of OOPSLA '86*, Portland, Oregon, pp.186-201 (Sep. 1986).

[51] McCormach, J., Asente, P., and Swick, R.: "X Toolkit Intrinsics - C Language Interface", *MIT* and *DEC*, Massachusetts (1990).

[52] Zeltzer, D.: "Towards an Integrated View of 3-D Computer Animation", *Proc. Graphics Interface '85*, pp.105-115 (May 1985).