3次元視覚化表示のソフトウェア工学への応用

小池英樹

3次元視覚化表示のソフトウェア工学への応用

小池英樹

目次

第 1 章

序論

1.1 本研究の背景と目的

現代社会において計算機は既にインフラストラクチャとしての地位を確固たるものとしている。更にハードウェア技術の急速な発達により、計算機の高機能化・低価格化・小型化が進み、金融、証券、エネルギー、交通、製造といった各産業においてシステムの大規模分散並列化をもたらしつつある。一方で、これらを制御するためのソフトウェア開発・保守は、従来にも増して複雑かつ困難なものとなっている。こうしたソフトウェアの低い生産性、保守に要するコスト等が比較的早い時期から問題とされていた。これらを解決すべくソフトウェア開発方法論に関する研究はソフトウェア工学と呼ばれ、(1) プログラムの作成法 (トップダウン設計と段階的詳細化 [118]、構造化プログラミング [27]、モジュール化 [86]、ソフトウェア部品 [59])、(2) プログラミング言語 (modula[119])、(3) プログラマチームの組織と管理 (チーフプログラマ [12], 構造化ウォークスルー [120], 動的設計レビュー [87])、(4) プログラムの作成、保守の支援 (統合化ソフトウェア開発環境 [116]、Interlisp[109], Ceder[110])、(5)CASE (Computer Aided Software Engineering) (6) 知識工学の利用 ([13, 15])、という 6 つのアプローチ [11] にわたって地道だが着実な進歩を遂げてきている。

ハードウェア、ソフトウェア両面における計算機技術の発達は、人間・計算機間の情報伝達形態をより人間にとって自然で、理解しやすい方向へと変化させつつあり、このいわゆるユーザ・インタフェースに関する研究は、人間工学・心理学・認知科学といった他分野の研究とも密接に関連し、とくに認知科学において提唱されたメンタル・モデル [53] を考慮したインタフェース設計の重要性が叫ばれている。このメンタ

ル・モデルという視点に立つと、ソフトウェア開発の複雑さは物理形状を持たないソフトウェアに対するメンタル・モデル形成の困難さに起因し、ソフトウェアに対するメンタル・モデルの迅速な形成を支援することがソフトウェアの効率的開発の鍵であると考えられる。

複雑なソフトウェアに対するメンタル・モデル形成を支援する手段として、従来からフローチャート、データフロー・ダイアグラム等の図が利用されてきた。以前は紙上に記されていたこれらの図は、ビットマップ・ディスプレイ等の開発によって、現在では計算機ディスプレイ上のマルチウィンドウ環境内で使用することが可能となり、ソフトウェア開発を支援するものとして注目されている。この視覚化アプローチはソフトウェア工学の7番目のアプローチとして位置づけることができる[11]。しかし残念ながら、現在までに1部の例外を除いて実際のソフトウェア開発に利用された視覚化ツールはない。大部分はプログラミング初心者教育用としての位置付け、あるいは非常に限定された領域での試用が行なわれているだけである。

著者は、実際のソフトウェア開発における視覚化ツールの利用形態を考えた時、視覚化ツールの実用化に向けて解決すべき問題点は、図形数の問題と図形提示法の問題の2点であると考えている。前者は、第2章で述べる図形言語が宿命的に持つ特性に起因する。実際のソフトウェアを視覚化する場合、視覚化対象の増加は避けられないが、その時必要な情報への迅速なアクセスを行なうために図形数制御が不可欠である。後者は、紙上に描画された図形では支援できない視点を開発者に与えることができて、初めて計算機上で図形を扱う意義がでてくるという点である。

以上のような背景において、本研究の目的は計算機ユーザ・インタフェースの基礎技術の1つとして、

3次元視覚化表示のソフトウェア工学への応用を示すこと

である。従来2次元平面上に拘束されていたソフトウェア視覚化表現を3次元空間に拡張することで、2次元的図形では対応しきれなかったソフトウェア開発時の多様な視点に対する支援が可能となることを実証的に示すのが本論文の主題である。

具体的には、

1. ソフトウェアの3次元視覚化の提案とその利用法の実例を示すこと
2. 汎用の図形数制御手法を開発すること

3. これらの枠組に基づいた試作システムをインプリメントすること

4. 幾つかの実際のソフトウェアに適用し、その有用性を検証すること

を行なう。

試作システムは VOGUE(Visualization Oriented Generic User-interface Environment) と呼ばれ、実際のソフトウェア視覚化に適用される。ソフトウェア視覚化を3次元に拡張しようというアイデアは決してユニークなものではなく、幾つかの提案が行なわれている [39]。しかし、これらはいずれも発想の段階であり、特に実際のソフトウェアに対して適用した時の有用性及び問題点に関して言及したものは見あたらない。

実用システムの実現には入力デバイス等、解決すべき幾つかの技術的な問題が残されている。また、アイコンの形状、色、レイアウト等、図の表現方法は問題領域に特化した表現手法が採られてしかるべきであるが、将来的技術の発達を見据えた上で、より抽象度の高いレベルにおいて視覚化システムの問題点を論じるとともに、汎用性を持った枠組を提案することもまた重要なことであると考える。以上のような理由から、本研究のような基礎的研究は意義のあるものだと思われる。

1.2 本研究の立場

視覚化に対する本研究の立場を説明するにあたり、以下のような2元連立2次方程式の実数解を求めるという問題を考えてみる。

$$x^2 + y^2 = 1 \tag{1.1}$$

$$x + y = 2 \tag{1.2}$$

この問題は幾何学的には Figure 1.1のような円と直線の交点の座標を求める問題と同値である。上式を代数的に解くことによって、実数解が存在しないことを知るのは容易であるが、図から実数解が存在しないことは直感的に把握できる。

次に上式の 1.2が $x + y = 1/2$ という形の場合、図によって実数解が2個存在することは即座にわかるが、正確な値を得ることは困難であり、一般的には代数的手法に

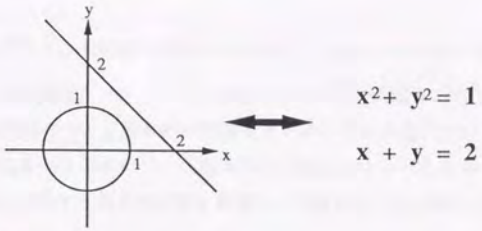


Figure 1.1: 代数と幾何

よって求められることになる。しかし、この場合に図があることによって、交点のおおよその位置は把握できるため、代数的に得られた解の正当性のチェックに役立っている。

つまり、これらの問題は代数的手法だけで解決することは十分可能であるが、図の補助があることによって、問題の把握を早め、かつ解の妥当性に対する保証を与えることになる。

従来のプログラミング言語によるデータ構造やアルゴリズムの論理的記述は、数学における代数的手法に相当し、問題解決の手段として十分な表現力を有し、かつ問題解決の正確さにおいて他に比するものが存在しない。しかし、図による支援を行なうことで、複雑な問題の正確な把握を促進し、上で得られた解の正当性をチェックするのに役立つものであると考えられる。

こうした観点から従来の視覚化研究を見ると、アイコン言語等は図を問題解決の主たる手段としているため、初心者へのプログラム説明、画像処理等、特定領域にしか適用できない。つまり、本研究においては視覚化をそれによって問題を解決する手段として捉えるのではなく、従来のソフトウェア開発手法を支援する 1 手段として捉えている。

1.3 本論文の構成

第1章は序論であり、本研究の背景とその目的、意義について述べる。

第2章では、図形言語とソフトウェア視覚化についての考察を行なう。この章では、まず図形言語の特徴について基本的な事項をまとめ、次に本研究で述べるところのソフトウェア視覚化という用語について基本的な立場を明らかにしておく。その後でソフトウェア視覚化に関する従来の研究を概観し、視覚化ツールに関する考察を行なう。

第3章では、3次元コンピュータ・グラフィックスのソフトウェア視覚化への適用に関し考察を行なう。3次元CGは他分野において積極的に利用されているが、これをソフトウェア視覚化に適用する利点について、幾つかの実例をあげながら示していく。また、実際に視覚化ツールを構成するにあたり考慮すべき点に関する考察を行なう。

第4章では、著者が開発した汎用的情報量制御手法、Generalized Fractal Views について述べる。図形ツールの持つ大きな問題の一つは、表示される図形数をいかに制御するかである。この章では、まず従来の制御手法について概観し問題点を指摘する。次に B. Mandelbrot によって提案されたフラクタルの概念を完全木から一般木へ、一般木から論理木へと拡張する。さらに拡張された論理木のフラクタル性を利用した情報制御手法について述べるとともに、計算機シミュレーションによってその有効性を検証する。

第5章では、本論文の主題である3次元視覚化を実現するために試作したシステム VOGUE (Visualization Oriented Generic User-interface Environment) について、その設計指針、システム構成、特徴点等について述べる。

第6章では、VOGUE を実際のソフトウェア視覚化に適用した例として、電力制御用ソフトウェアを扱う。まず静的視覚化の例として電力制御用ソフトウェアの階層的なモジュール構造とタスク間通信の様子を視覚化する。次に、時間軸を導入することで、実際の制御用計算機を走らせて得られたトレース情報を視覚化する。さらに、将来的に複数台の計算機間におけるメッセージ通信のトレースが得られた場合を想定して、分散マニピュレータの実行状態のトレースを視覚化する。従来の2次元ツールでは支援するのが困難であった視点が3次元視覚化によって明確に支援されることを示す。

第7章では、VOGUE の2番目の適用例として、オブジェクト指向言語のクラス・

ライブラリの視覚化を行なう。クラス階層とメソッド継承を同時に視覚的に表現することを試み、被験者実験によって2次元ツールに対する有効性を示す。次により複雑な例として、Flavorにおけるデーモン・メソッドの取り扱いを論じ、やはり被験者実験を行なうことによりその有効性を示す。さらに実際のSmalltalk-80のクラス・ライブラリの視覚化を行ない、大規模データの視覚化の際に生じる図形数の問題に対する図形数制御の重要性が示される。

第8章では、考察および展望として、視覚化システムの将来的な発展として、演繹データベースとの融合、人工現実感システムとの融合に関して述べる。また、第4章で開発したGeneralized Fractal Viewsの他の領域への適用例を述べる。

第9章では、結論を述べる。

第 2 章

図形言語とソフトウェア視覚化

2.1 緒言

図は文字同様、古来から人間の情報表現、伝達、記録手段の 1 つであり、また人間の思考を支援する目的としても利用されてきた。これはソフトウェア開発においても同様で、状態遷移図、ペトリネット、フローチャート等、数多くの図が利用されてきている。

ソフトウェア視覚化とは、基本的には従来紙上に表現されていた図を計算機ディスプレイ上に実現しソフトウェア開発の支援を行なうことであるが、図をプログラミングの主たる道具とするか、従来のプログラミングを側面から支援するものと捉えるかにより 2 つの概念に大別される。前者を Visual Programming と呼び、後者を Program Visualization と呼ぶのが一般的であり、序論で述べたように本論文では後者の立場をとっている。両者は概念的には明かに異なるが、抱える問題点には共通するものも多く、これは主に図の特質に起因するものであると考えられる。図に対する人間の認知に関する研究は主に心理学で行なわれ、得られている知見も多い。ユーザ・インタフェースとして図の利用を考える場合、図と認知の関係を把握しておく必要があると思われる。

以下、2.2 節において図形言語の分類と特徴についてまとめるとともに、ソフトウェア開発に利用されてきた図形の概観を行なう。2.3 節においてソフトウェア視覚化の概念の整理を行ない、続く 2.4 節において、従来のソフトウェア視覚化ツールについて概観する。最後に 2.5 節において図形言語と視覚化ツールに関する考察を行なう。ここでの考察は第 5 章で実現される 3 次元視覚化ツール設計の際の基礎となる。

2.2 図形言語

一般に絵や図に対する概念として文や式といった記述言語がある。これら記述言語の表現力、厳密さ、客観性等の長所は言うまでもなく、人間の思考はこれら記述言語によってなされてきたとも言える。にもかかわらず、 絵や図が情報の表現、伝達、記録の手段として重要視されてきたのは、情報によっては絵や図で表す方が、人間にとって効率的なものが存在するためである。絵や図の全ての性質に関して述べることは、本論文の主題から外れるため、ここでは図形言語の分類と人間の認知という側面から見た時の特徴について述べることにする。

2.2.1 図形言語の分類

出原ら [30] によれば、図形は図素とそれらの関係付けによって成り立ち、これら図形要素群を視覚的に結びつけて部分集合を表すために以下の 3 種の方法がある。

1. 配置による関係づけ
   特定の配置を行なうことで関係づけられ、部分集合が視覚的に識別される。
   (a) 遠近配置
   関係のある要素群を近くに集めて、文字どおりの部分集合を形成する。
   (b) 配列配置
   関係のある要素群を規則的にならべることでそうでない要素群から識別される。
2. 図形の指示機能による関係づけ
   要素群は線分などを書くことで関係づけられる。
   (a) 連結図形
   関係のある要素群を直接線分で結びつけていくことで部分集合を形成する。
   (b) 領域図形
   関係のある要素群の占める領域を線分などによって指示することにより部分集合を表現する。
3. 要素記号の表現による関係づけ
   要素群そのものに形態上の差異を与えて、部分集合を表す。

(a) 形状 (方向も含める)

(b) 大きさ

(c) 表面処理

こうした図形の関係づけの方法は、知覚心理学におけるゲシュタルト要因 (Table 2.1参照) と深い関係がある。Table 2.2は関係づけの基本手段と関連するゲシュタルト要因との関係を表したものである。

実際の図形においては以上の基本手段が組み合わせて用いられ、主となる表現方法によって以下の 4 つに分類することができる [30]。

- 連結系
  論理図、工程図、関連樹木図など一般に数学的なグラフに還元されるもの
- 領域系
  概念間の包含関係図やベン図など
- 座標系
  直交座標上の棒グラフ、折れ線グラフ、あるいは地図、等高線図など座標上に図素間の数量的位置関係を示すもの
- 配列系
  統計表、相関表、一覧表、分類表、対応表など一般に表と呼ばれるもの

この 4 種を図形言語と呼び、絵や写実画とは分けて考えなければならない。

2.2.2 図形言語の特徴

図形言語の特徴は人間の認知と深く関わり、文や式といった記述言語と比較されて述べられることが多い。以下では図の特徴について幾つか代表的なものを記す。

まず第 1 に、記述言語の理解が部分から全体へとボトムアップに進むのに対し、図の理解は逆に全体から部分に向かって (global-to-local) トップダウンに進むとされる。例えば、モザイク状に描画された図は至近距離ではその内容を理解できないが、ある程度の距離をおくことで認識できるようになる。また、Navon の実験によっても与えられた視覚刺激に対して、全体特徴が部分特徴より先に処理されることが示された (Figure 2.1)。

(1)	近接の要因	近いものはまとまりやすい
(2)	類同の要因	似たものはまとまりやすい
(3)	閉合の要因	互いに閉じ合うものはまとまりやすい
(4)	よい形の要因	単純な、規則的な、左右相称的な、同じ幅をもつような形はまとまりやすい
(5)	よい連続の要因	多くの連続の可能性があるとき、よい曲線が生じるようにまとまりやすい
(6)	割り切れの要因	残りを生じないようにまとまる傾向がある
(7)	共通運命の要因	変化、運動などにおいて共通の運命をもつものはまとまりやすい
(8)	客観的態度の要因	刺激の系列が継時的に示されるような場合、まとまりをつける傾向にある
(9)	経験の要因	たびたび経験したまとまりは他のものと一緒に与えられる時、まとまりやすい

Table 2.1: ゲシュタルト要因

関係づけの手段		関連するゲシュタルト要因
配置	遠近配置	(1)
	配列配置	(5)
図形の指示機能	連結図形	(5)
	領域図形	(3),(5)
要素記号の表現		(2)

Table 2.2: 関係づけの手段とゲシュタルト要因の関係

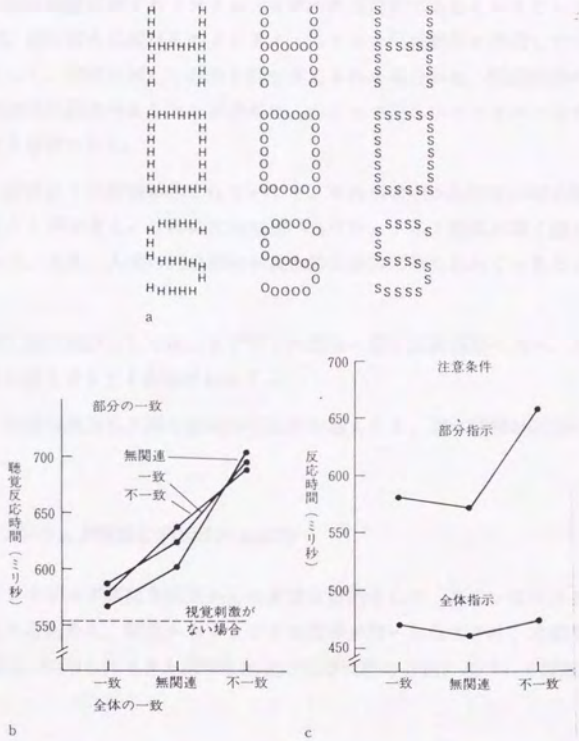


Figure 2.1: Navon の実験に使われた視覚刺激の例と実験結果

第 2 に、本研究の立場の節でも述べたが、図は問題を直感的に理解させることができる。代数と幾何の例によれば、実数解の有無は図から即座に答えることができる。ただし、詳細な解析は代数的手法、つまり記述言語によってなされなければならない。

第 3 に、図は問題に対するメンタル・モデルの具現化であるということができると思われるが、逆に言えば図はこのメンタル・モデルを反強制的に形成していると考えられる。よって、問題に対して適当な図が与えられた場合には、問題解決に有効である一方、不適当な図が与えられた場合には、かえって正しいメンタル・モデルの形成が困難になり有害である。

第 4 に、図は多くの情報が含まれていても、それらの中から特定の部分集合を即座に選別することができる。これには先に述べたゲシュタルト要因が深く関わっていると考えられる。また、人間の図の認知が基本的に並列に行なわれていることにも関係する。

これに対し図の短所としては、まず第 1 に図は一般に記述言語に比べ、より多くの表示領域を必要とすることが挙げられる。

第 2 に、図素が増加し人間の認知許容限界を越えると、逆に理解が困難になる点が挙げられる。

2.2.3 ソフトウェア開発に利用される図形

複雑なソフトウェア開発を側面からの支援を目的として、あるいはシステムの挙動をモデル化するために、従来からさまざまな図形が用いられてきた。本節では、ソフトウェア開発に利用されてきた図形を前述の図形言語の分類に基づいて概観する。

連結系

状態遷移図 (Figure 2.2) は、古くから用いられてきたシステムモデル化の図であり、状態を表すノードと 1 種類のリンクから構成される有効グラフである。

ペトリネット (Figure 2.3) は Petri によって互いに関連しあう同時進行的な要素からなるシステムをモデル化することを目的として作られた。場 (place) と遷移 (transition) との 2 種類のノードと、1 種類のリンクから構成される有効グラフである。

データフローグラフ (Figure 2.4) は、個々のデータの流れを主体にして、動作の半

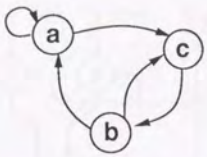


Figure 2.2: 状態遷移図

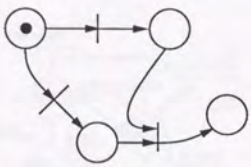


Figure 2.3: ペトリネット

順序関係を表現するもので、ペトリネットに以下のような変更を施して得られた。

- トークンに値を保持できるようにする
- 場所 (place) をフローリンク (データリンクと制御リンク) とする
- 遷移 (transition) を計算プリミティブとして、基本アクタに対応させる
- 基本アクタの全ての入力枝にトークンがあると、出力枝に (前の) トークンがなくなれば発火できる。発火してトークンを出力枝に送り出す

DFD(Figure 2.5) はバブルチャートとも呼ばれ、処理プロセスとデータのフローに着目した図である。現在 CASE ツールなどにおけるプロジェクト管理図等に用いられている。構成要素はプロセスを表すノードとデータフローを表すリンクである。

フローチャート (Figure 2.6) はアルゴリズムを記述するのに用いられる。手続きを表す四角形と条件分岐を表す菱形のノード及びリンクによって構成される。

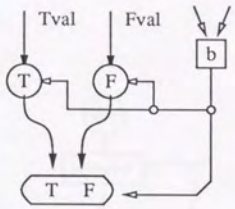


Figure 2.4: データフローグラフ

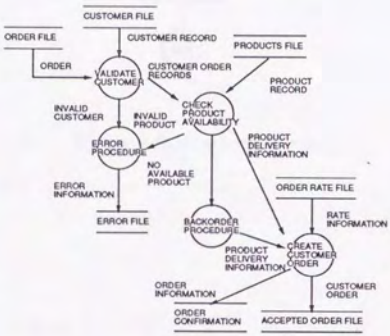


Figure 2.5: データフローダイアグラム

その他データ構造を表したり、関数の呼びだし関係等を表すために、木構造 (Figure 2.7) がよく利用される。また、データ間の接続関係、階層関係あるいは包含関係を表現するのに用いられる。

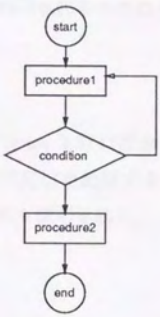


Figure 2.6: フローチャート

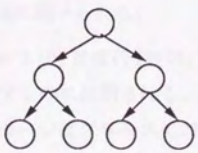


Figure 2.7: 木構造

領域系

領域系の図は包含関係を表現するのに優れるが、木や森として表現できない関係を表現するのは一般的に困難であり、連結系の図で代用されることが多い。従って、ソフトウェア開発における図として利用されるものは少ない。

座標系

時刻 - プロセスダイアグラム (Figure 2.8) は縦軸にプロセス、横軸に時間を取ったグラフで、プロセス実行状態の時間変化を把握するのに用いる。並列プロセスの場合にはプロセスの並列度を調べるのにも使用される。

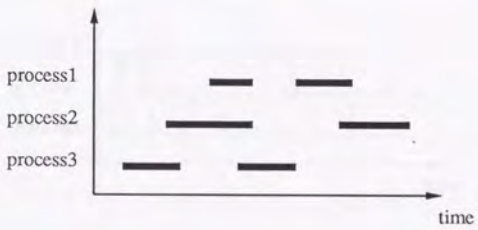


Figure 2.8: 時刻 - プロセスダイアグラム

工程管理グラフ (Figure 2.9) は縦軸にプロジェクト、横軸に時間軸をとってプロジェクトの日程見積りや並列度の把握に用いられる。

タイミング・チャート (Figure 2.10) は並行 (並列) プロセスにおけるメッセージ送信のタイミングを視覚的に表現するのに使用される。一方の軸を各プロセスにとり、他方の軸を時間軸にとる。メッセージはプロセスに対応する線を横切る矢印で表示される。

配列系

配列系の図としては、現在表計算ソフトウェアが一般化し、主にパーソナル・コンピュータ上で使用されている。また CASE ツールにおいてもデータベース検索のフォー

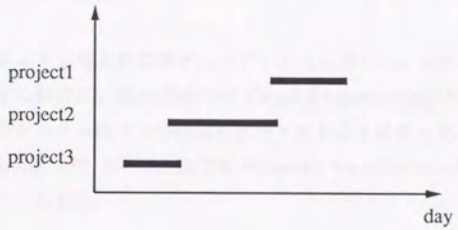


Figure 2.9: 工程管理グラフ

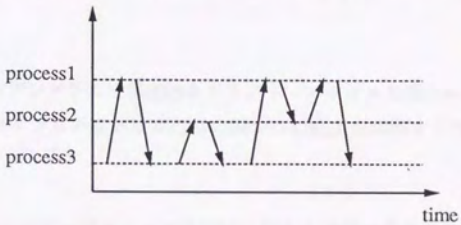


Figure 2.10: タイミング・チャート

ムシートとして、あるいは検索結果の表示用に利用されている。また、ウィンドウ・システムにおけるメニュー形式のツールも配列系の一つに含めることができる。

2.3 ソフトウェア視覚化

広義のソフトウェア情報を計算機ディスプレイ上に表示し、ユーザの理解を促進することを目的とする研究は、広い意味での Visual Programming と呼ばれるが、最近になってこれらのシステムは 2 つの概念に大別されるようになった。Myers は一方を Visual Programming(VP) と呼び、他方を Program Visualization(PV) と呼んで以下のように定義している [81]。

> "VP は 2 次元あるいはそれ以上でプログラムを表現するシステムであり、グラフィカル・プログラミング言語やフローチャートは含まれるが、絵を定義するためのプログラミング言語 (Sketchpad,PostScript 等) は含まない。"

> "PV はプログラムは伝統的なテキスト・スタイルで書かれ、グラフィックスはプログラムあるいは実行時の幾つかの視点を図解するために使用される"

前者は図形自体をプログラミングの手段と捉える立場であり、後者は従来方式のプログラミングを側面から支援する道具と捉える点が異なる。Shu は前者を Visual Language と呼び、後者を Visual Environment と呼んでいるが [94]、呼び方が異なるだけで両者の分類は矛盾しない。

本論文では序論でも述べたように後者の立場をとるが、VP 及び PV の定義からも判るように、従来の視覚化研究の対象は主にプログラムやそこで用いられるデータ構造である。一方、図形言語の特徴は記述されている情報量が多くても、全体の把握、部分集合の認識が即座に行なえることであった。比較的簡単な問題は図を使うまでもなく記述言語的アプローチによって解決できる。先の数学の例で言えば、簡単な問題は代数的手法で十分だとも言える。ただし、より問題が複雑になった時、代数的手法による問題解決を側面から支援する上での幾何的手法、つまり図形が必要になると思われる。従来はハードウェアの制約等により実現できなかった部分も多いが、計算能力、グラフィックス能力が発達した現在、より視覚化の対象を広げることが必要であると思われる。よって本論文では PV の概念を広く捉えたソフトウェア視覚化 (Software Visualization) を以下のように定義する。

“ソフトウェア視覚化 (Software Visualization) とはソフトウェア一般に関する情報を視覚的に表示すること”

以後、本論文でソフトウェア視覚化とは全てこの意味で用いる。

2.4 ソフトウェア視覚化システム概観

本節では前節で定義したソフトウェア視覚化の概念に基づき、広い意味での視覚化に関する従来の研究を概観する。

INTERLISP の MASTERSCOPE は、関数や変数の呼び出し関係を、関数名あるいは変数名をラベルに持つノードと、リンクからなる木構造として表示する。ノードを選択することによって対応する定義をエディタを用いて編集することが可能である。表示する木が大きくなり過ぎて、ウィンドウに入りきらない時は、左隅に木全体の縮小図が現われ、現在ウィンドウに表示されている部分を小さな四角で示している。

類似のシステムとして Smalltalk-80 上に実現されたクラス階層ブラウザ (Figure 2.11) がある [8]。Smalltalk-80 のクラス階層ブラウザは、クラスの階層構造をクラス名をノードとしてもつ木構造として表現する。ノードを選択すると、ポップアップ・メニューが出現し、そのクラスに属するメソッドのリストが現われる。任意のメソッドを選択すると、対応するメソッド定義を編集することが可能である。標準システム・ブラウザはクラス階層の把握を視覚的に支援してはいないが、クラス階層ブラウザはこの欠点を補うものである。ただし、木が大きくなった場合にはスクロール・バーで必要なクラスを探索しなければならない。前者は INTERLISP、後者は Smalltalk-80 という統合化環境の中で実現されているがゆえに実現は比較的簡単である。しかしながら、統合化環境はこれらの環境内で閉じた世界を作り易く、他の例えば UNIX 上におけるソフトウェア情報を視覚化するのには向かないと考えられる。

Macintosh の標準ファイル管理ツールのフォルダは、アイコンを用いたファイル構造の視覚化環境であり、1 つのフォルダは 1 つのディレクトリに相当しファイルを階層的に管理する。重要な点はディレクトリとその中身というソフトウェア情報を視覚的に表現することによって、インタフェースを向上させている点である。現在エンジニアリング・ワークステーションのインタフェースとしても利用され始めている。ただし、フォルダは開いてみないと中が判らない。また階層構造全体を視覚的に表現する機能がない。

Directory Browser はフォルダの概念を継承しつつ、UNIX のディレクトリ階層を木として視覚的に表現するツールで、各ディレクトリはアイコンとして表現される。アイコンをクリックするとそのアイコンに対応するディレクトリの構成要素が出現する。現在、作業中のディレクトリ付近の上下関係は視覚的に把握されるが、あまり大

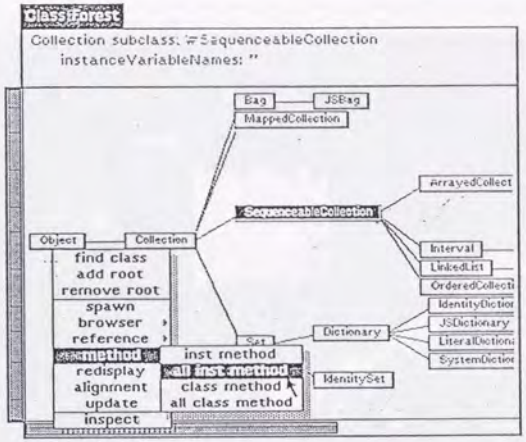


Figure 2.11: クラス階層ブラウザ

きな木は表示できない。

Spatial Data Management System (Figure 2.12) はディスプレイ上のアイコンを通して、INGRES に蓄積されている海軍の船舶のデータにアクセスするシステムである [44]。ジョイスティックを用いて見たい部分を指示すると、その部分がズームアップし、拡大図が表示される。さらにその中から特定のアイコンを選択すると、それが拡大されると共により詳細なデータが表示される。離れて見ればデータベースの全情報を漠然と把握することができ、ズームインすることで、詳細情報を得ることができる。また、データベース情報をアイコンとして空間的に配置することで、ユーザは 1 度検索したデータをその位置で記憶するため、再検索の速度が向上する。ある 1 つの関係表として表現できるデータに対しては、同様に適用可能であるが、1 つの関係表だけでは表現しきれない関係の表現には適用できないと考えられる。

BALSA [21] は Macintosh 上に実現されたアルゴリズム・アニメーション・システムである (Figure 2.13)。バブルソート、クイックソート等、各種のソート・アルゴリズムにおいて、その要素がソートされる様子をバーチャート、木などで視覚的に表示する。ユーザはアルゴリズムの違いを視覚的アニメーションとして認識し、アルゴリズムに対するメンタル・モデルを形成しやすくなる。

Figure 2.12: Spartial Data Management System

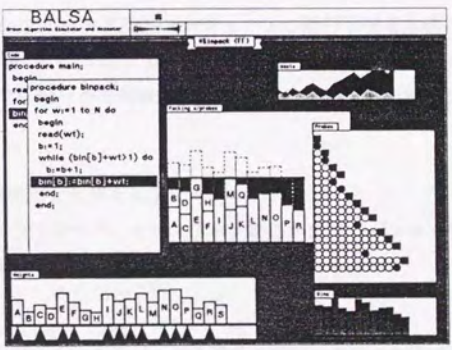


Figure 2.13: BALSA

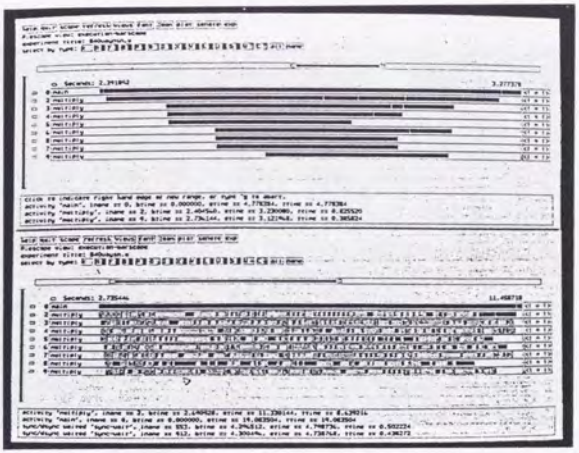


Figure 2.14: PIE

PIE [69] は Mach OS 上に実現された並列プロセスのパフォーマンス解析ツールで、プロセスの実行状態をバーチャートとして表す (Figure 2.14)。バーチャートを用いて Mach OS のカーネルの実現方式に対する評価等に適用している。

ADAPT [89] はソフトウェア開発プロジェクトの進行状況を木構造として視覚化するツールである (Figure 2.15)。プロジェクト情報を視覚化しようとすると、視覚化すべき対象が多くなり、何らかの対策が必要になる。ADAPT の対応策は以下のようなものである。

- ファイルの種類ごとに異なるアイコン
  作成されたソースコード、マニュアル、あるいはプロジェクト管理者からのメッセージ等が形状の異なるアイコンとして表現される。
- 色の利用
  アイコン間を結ぶリンクはその表す関係の種類に応じて色分けされている。
- 木の簡略化
  大きな木を表示する場合には、ノードのアイコンを表示しない。

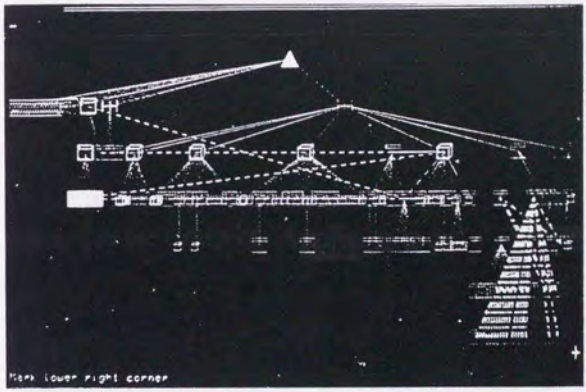


Figure 2.15: ADAPT

また、細部を見るためには対応する部分の拡大を行なう。

SemNet [31] は 3 次元グラフィックスを用いて、知識ベースを 3 次元グラフとして視覚化するシステムで、各知識要素をラベル付きノード、要素間の関係をリンクとして表示する (Figure 2.16)。知識要素間のセマンティックな情報を利用せずに、大規模な情報を効果的に表示する手法について述べている。

結論として、何らかの方法での情報削減の必要性があるとし、SemNet では以下の手法を用いている。

- 型によるサブセット戦略
  知識要素の型に着目し、ユーザの指示した型に属するノードは表示する
- 焼きなまし法による知識要素の再配置
  知識要素間の論理的関係により関係のある知識要素同士は近くに配置することにより、ネットワークの視覚的複雑さを削減する
- Fisheye View
  x、y、z 軸を各 2 等分し、8 つの部分領域に分割する。分割された部分領域を

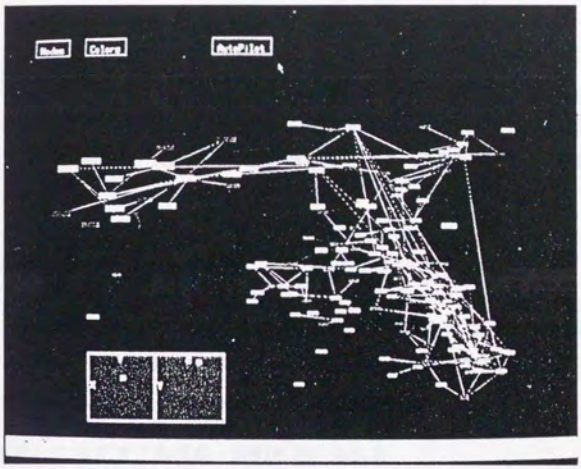


Figure 2.16: SemNet

再帰的に 8 分割し、最終的に 3 階層まで作る。最終的な葉は、各々の部分空間に属するノードである。そして、ユーザの注目ノードから、第 4 章で述べる Fish-eye View [36] を用いることで表示オブジェクト数を削減する。

2.5 図形言語とソフトウェア視覚化に関する考察

本節では前節までの考察を基に、図形言語とソフトウェア視覚化に関する考察を行ない、特に実際のソフトウェアを視覚化するために考慮すべき事項を述べる。ただし、図形のユーザに対して与える影響を考察するにあたり、部分的には定性的議論にならざるを得ないが、可能な限り認知科学等における知見を引用することで主観的になることをさける。

2.5.1 表現手法

図形言語に関する考察から、図形言語は連結系、領域系、座標系、配列系の 4 種類に分類されることを述べたが、2.2.3 項で行なったソフトウェア開発に利用される図形の概観から、これらに関して以下のことが言える。

- 連結系としてのグラフ表現が一般的に用いられる
- 領域系は描画が困難なため特別な場合に使用が限定され、グラフ表現で代用されることが多い
- 座標系としては 1 軸を時間にとったものが多用される
- 配列系は特殊な場合に限られる

1 番目の、グラフ表現が一般的に用いられる理由としては以下のものが挙げられる。

- ソフトウェアにおける論理的関係の表現にグラフ (木) が適すること
- グラフ理論が体系化されていること [18]
- グラフ描画のためのアルゴリズムが研究されていること [117, 91]

また、ノード・リンクを構成要素とするグラフ表現は、単純であるが故に汎用性が高い点も重要である。従来の視覚化システムの中で、グラフ表現以外をとっているものは、その領域に固有の表現方法を採用し汎用性が低い。ソフトウェア視覚化の枠組を考える場合、多くの例に適用可能な汎用性が重要であると思われる。グラフ表現は図形言語の分類に基づくと連結系に属するが、ノードを配置する座標を考慮することで座標系図への適応も可能である。さらに、ノードのスケールを変化させ、ノード内にノードを表示する入れ子構造をとることで簡単な領域図へも適応できる。

2.5.2 モデル化の機能

さらに汎用性という面から考えると、視覚化の対象は各々異なるデータ形式を持つ。従って、これらの異なるデータに対し柔軟に対応するためには、モデル化の機能をツールが持つ必要があると思われる。これにはシステムに簡単なデータベース機能を持たせることで対応可能である。SemNet はアプリケーションに対応する部分を Lisp にまかせることで、種々のアプリケーションに対応可能な汎用ツールとなった。

また、ADAPT や SemNet で行なった、型による情報削減を実現するために、データの型を記述できるものが望ましい。その時、上述したグラフ表現へのマッピングを考慮した場合、個々の対象とそれら間の関係を陽に記述することのできるデータ・モデルが良いと考えられる。このようなデータモデルとしては Entity-Relationship モデルやオブジェクト指向モデルがある。後者の場合、データの型はデータが属するクラスということになり、型に階層構造が定義される。つまり、オブジェクト指向モデルに基づくデータベースでは、思考の特殊化 / 一般化操作が他のデータベースに比べ容易に実現できると考えられる。

2.5.3 量への対応

視覚化の対象を一般のソフトウェア情報に拡大するにあたって、その本質的な変化は表示すべき対象の量の増加である。

視覚化の対象がプログラムの段階では、対象の量はせいぜい 10^1 のオーダであった。しかし、その対象を広げると 10^2 から 10^3 のオーダになるものと予想される。従って、多数の図形を表示できる能力が必要になると考えられるが、現段階におけるウィンドウ環境やパーソナル・コンピュータで一般的に利用できるグラフィックス機能では十分とは言い難い。視覚化対象のアニメーション表現や、スクロール、拡大縮小といった視点の変更にあたって、画面更新の高速さが要求される。

2.5.4 図形数と認知許容限界

一方、図素数が増加すると選択に要する時間も増加する。いま、ある図素の選択確率を p_i とした時、任意の図素を選択するまでに要する時間は、

$$H = \sum_{i=1}^{n} p_i \log \frac{1}{p_i}$$

となることが実験的に示されている [7]。各図形の選択確率が同じ、つまり $p_i = 1/n$ であるとすれば、上式は

$$H = \log n$$

と簡略化できる。つまり単純に考えると、図素の選択までに要する時間はその数に応じて単調増加するということができる。

図の複雑さに対して一般的に利用されている定量的な評価指数は存在しない。図素数が同じでも配置が工夫されていれば、その図は複雑でないと感じ、そうでない場合は複雑であると感じる。ここで、ノード・リンクによる一般のグラフ表現を考えた時、その図形の複雑さは要素数に従って単調増加するといっても差し支えないと思われる。よってここでは図形の複雑さと要素記号の多さとを同じ意味で用いることにする。

2.2 節で述べたように、図形の長所は図素数が増えた時に生かされる一方、図素数が増えると複雑さが増し認知能力が落ちてしまうというジレンマが生じる。こうした問題は、従来の視覚化ツールのように比較的小さな規模の場合には起こらなかった、あるいは無視され得た問題といえる。

さらに図形数の削減はシステムの反応性を高める効果を持つ。一般に表示される図形が増加するに従い、図形の選択、視点の変更といったシステムの反応性が悪くなる。特に、実際のソフトウェアではこれが深刻な問題となる。この観点からも図形数制御機能が必要である。

2.5.5 マルチウィンドウの乱用

ソフトウェア開発においては、その局面ごとに幾つかの視点からソフトウェアを見る必要性が生じる。現在の視覚化ツールは、そのほとんどが 2 次元的図形として実現されており、こうした異なる視点を視覚化するためには、別な図として提示されることになる。ウィンドウ環境内においては、これらは別々のウィンドウで提示される。

前述したように、図はユーザに対して反強制的にメンタル・モデルを形成させるため、ある図からは 1 つのメンタル・モデルを獲得し、別な図からは別なメンタル・モデルを獲得することになる。こうした場合ユーザは、自分自身の心の中で別々のメンタル・モデルを一つのメンタル・モデルとして再構成する必要が生じる (Figure 2.17)。

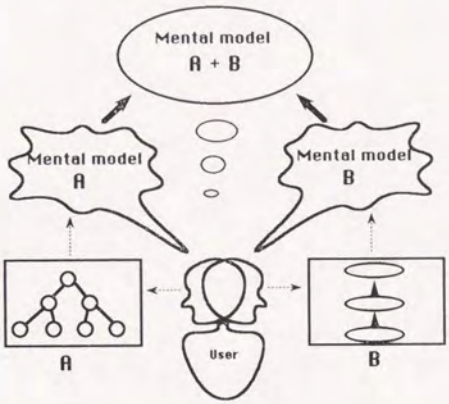


Figure 2.17: メンタル・モデルの再構成

2.5.6 本節のまとめ

以上の考察から、実際のソフトウェア視覚化を実現するために考慮すべき点として、以下のようなものが挙げられる。

- グラフ表現は汎用性が高く、かつ領域系図、座標系図への対応も可能である
- 異なる視覚化対象に対応するためのデータベース機能が必要である
- データモデルとしては、グラフ表現に対する整合性、階層化された型による思考の特殊化 / 一般化の支援という点からオブジェクト指向モデルが良い
- 高度なグラフィックス機能が必要である
- 図形制御機能が必要である
- マルチウィンドウの乱用は有害である

本節における考察は、第 5 章で実現されるソフトウェア視覚化ツール VOGUE の基礎となる。

第 3 章

ソフトウェアの 3 次元視覚化

3.1 緒言

本節では、3 次元コンピュータ・グラフィックス (CG) を用いたソフトウェアの 3 次元視覚化を提案する。

前章では図形言語とソフトウェア視覚化に関する考察を行ない、実際のソフトウェア開発を視覚的に支援するためには、従来の視覚化ツールでは表現力の点、機能面において不十分であることを述べた。またマルチウィンドウの乱用は悪戯にユーザの認知を阻害する点を指摘し、より人間の思考に自然な表現形態での情報の提示が重要であることを述べた。本章で提案する 3 次元視覚化はこれを考慮したものである。

ただしここで言う 3 次元とは、xyz 座標によって定義される 3 次元空間の意味で用い、要素記号の形態的差異による自由度のことではない。色・形等の次元要素は差異により分類された部分集合間の順序関係を示す能力が弱く、例えば、色は光のスペクトルのように 1 次元に並べる事は可能であるが、青と緑、あるいは赤と黄がどの程度近いかを表すことはできない。従って、これらの次元要素は上述した 3 次元表現を補助する目的に使用されるべきであると考えられる。

先にも述べたが、視覚化空間を 3 次元に拡張するという考え方自体は決してユニークなものではない。しかし、実際のソフトウェアへの適用を考慮し、その利用法、問題点等について解析したものは見当たらない。本章では 3 次元の利用法を実証的に示し、実現に向けて考慮すべき問題点等の考察を行なう。

まず 3.2 節で他分野における 3 次元 CG の利用法に関して分析を行なう。次に、3.3

節でソフトウェア視覚化における3次元の利用法について述べる。最後に3.4節において実際の3次元視覚化ツールを実現する上で考慮すべき点について考察する。

3.2 3次元CGの他分野での利用法

現在、3次元CGは機械系CAD、分子グラフィックス、Scientific Visualization等の分野において盛んに利用されている。

電気系CAD(Computer Aided Design)では2次元グラフィックスが主流であるのに対し、機械系CADでは3次元グラフィックスが主流である。これは電気系CADが平面的回路を扱うのに対し、機械系CADは立体的剛体を扱うことの違いによる。CGの導入により図面はデータベースに登録され、必要に応じて取り出すことができるようになった。つまり再現性が保証されるようになった[66]。また作成されたデータから完成予想図を3次元ソリッドモデルとして表示し、製品を実際に作成することなくそのイメージを得たり、部品同士の干渉を検出したりでき、さらに外側の物質を半透明表示することで、内部の部品を見たりできるようになった。また、アニメーションによって動かしてみることもできる。

分子グラフィックスにおいても比較的早くから3次元CGの導入が行なわれた。以前は示性式や分子モデルを用いて分子に対するイメージを獲得していたが、CGの導入により巨大分子の表示・保管・移動が便利になった[55]。また実際には存在しないファンデルワルス面等の表示が行なえる。さらに、分子のダイナミクスの表示や立体障害の視覚的把握が行なえるようになった。

Scientific Visualizationは、自然現象等を解析して得られる大規模な数値データを視覚的に表現することで、その解析に役立てようとするものである[85]。地形に対して気流の流れを仮想的に表示したり、ある物質の伝熱状態を視覚的に表示したりするのに利用されている。

以上に共通する3次元CG導入のメリットは次のような点である。

1. 大規模データへの対応
2. 仮想的な表示
3. コミュニケーションの円滑化
4. ダイナミクスの表現
5. 空間的な対象の空間的取り扱い

1は計算機能力に起因するものであり、2、3は一般に図形表現の特徴に起因するものである。また4はこの両者が組み合わせられて実現できる機能であり、ソフトウェアの視覚化においてもそのまま適用可能である。これに対し、5についてはソフトウェアには物理的形状が存在しない。以下では形状を持たないソフトウェアにおける3次元視覚化を利用する意義について考察する。

3.3 ソフトウェア視覚化における3次元の利用法

第2章において図形言語は連結系、領域系、座標系、配列系の4種類に分類されることを述べた。ここで、配列系は一般に表と呼ばれるもので特殊な形式の図形である。本論文ではこのような表形式の図は取り扱わないこととする。ただし、座標系と配列系には共通する部分もあり、特にある要素に対する単純なリスト形式の図は、座標系の一部に含めることができる。従って、このリスト形式の図は座標系として考える。

以上のように対象とする図形言語を整理した上で、物理的形状を持たないソフトウェアを3次元視覚化するにあたり、本論文ではその利用法を以下の2種類に分類する。

- 3次元の弱い利用法
- 3次元の強い利用法

それぞれについて以下に説明する。

3.3.1 3次元の弱い利用法

ソフトウェアが連結系図(グラフ)として表現された場合を仮定すると、3次元視覚化による利点としてまず挙げておかなければならないのは、グラフの視認性の向上である。一般のグラフをリンクの交差なしに平面上に描画することは不可能であり、リンク数が増加するとリンクの追跡に支障をきたすと考えられる[31]。一方、グラフを3次元に描画することでリンクの交差は極力回避され、リンクのトレースが容易になると考えられる。

参考までに、30から50個のノードを直列に接続したグラフを、平面上にノード同士が重ならないよう配慮しつつランダムに配置して、先頭ノードから終端ノードまで追跡させる実験と、同様のグラフを3次元空間上に配置し液晶シャッタ眼鏡を装着して同様に追跡させる実験を行なったところ、課題達成時の達成時間には差が生じなかったが、その課題達成率において2次元の場合にはわずか50 %程度であったが、3次元の場合にはほぼ100 %に近い成績を修めた。

この結果から、従来平面上に描画されていたグラフを空間的に描画することで、その視認性を向上させることが可能である。このように平面的な描画と本質的な差はなく、立体視による視認性の向上を目的とした利用法を3次元の”弱い”利用法と呼ぶ。

3.3.2 3次元の強い利用法

3次元の強い利用法の分類

ソフトウェアにおいて利用される図形に関する考察から、ソフトウェア開発で利用される図には連結系が多く、領域系はこれで代用されることを述べた。また座標系としては、1軸を時間軸にとるものが多いことも述べた。以上から本論文では以下のような3次元の利用法を提案しこれを前節での3次元の弱い利用法に対して、3次元の強い利用法と呼ぶ。即ち、

- 互いに干渉し合わない関係の分離
- ある構造の時間的変化(ダイナミクス)の表現

前者は、一般に階層構造等の関係は1つの2次元平面で記述できるが、他の関係を同時に表示しようとすると、これらの表現が複雑になり混乱をきたすため、これを回避するための利用である。後者は、2次元平面上で時間軸とともに表現されていたプロセス、プロジェクトといった要素間に存在する構造をより明らかな形で記述するものである。

この他に、例えばソフトウェア部品を3つの属性に従い整列させて、3次元空間にマッピングする方法も3次元空間の有効な利用法と考えられるが、本論文ではこれを扱わない。

3次元の強い利用法の意義

これらの関係を1つの平面図で見る場合には、関係が錯綜するため混乱を引き起こし、2つの平面図として見る場合には、先にも述べたように両者の整合性をとるために労力を要する。一方、3次元視覚化することで、2つの関係を同時に把握しつつ、片方の関係をより重点的に見たい時には、視点の移動という手段によってその目的が達成される。これが3次元の強い利用法の意義である。

この点に関しては、宮崎がApolloniusの円錐曲線の例 [77] を挙げて以下のように述べている。

Apolloniusの円錐曲線とは以下の式で表される2次曲線である。

$$ax^2 + 2hxy + by^2 + 2gx + 2fy + c = 0$$

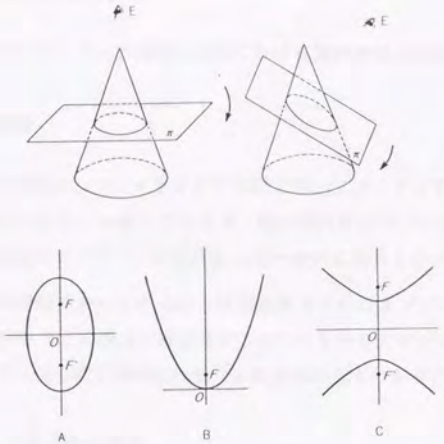


Figure 3.1: Apollonius の円錐曲線の例

点や直線の特殊な場合を除けば、$h^2 - ac < 0$ のとき楕円、$h^2 - ac = 0$ のとき放物線、$h^2 - ac > 0$ のとき双曲線となる。これらの背後に統一があり、それが上式で表現されるということはこのままでは理解し難い。しかし Figure 3.1のような円錐と平面を考えることによって、異なる視点間に存在する連続的関係を知り、理解が深まると宮崎は指摘している。上式で表現されるある概念に対し、楕円の図や放物線の図はある視点から見た局面の実体を視覚化している。しかし、メンタル・モデルを空間的にすることで、これら別々の図を包含する新たな図が導入され、理解が促進されると考えられる。ソフトウェアが持つ情報は唯一であるが、一方、それを視覚化する手段は複数存在する。こうした視点間をつなぐことは、見る人の理解を早め深めるのに役立つと考えられる。

3.3.3 3次元の強い利用法の実例

本節では、実際のソフトウェア開発の局面における複数の視点の実例を挙げる。

オブジェクト指向言語

オブジェクト指向言語は、プログラミング言語工学、ソフトウェア工学等の分野で、現在最も注目されているテーマの一つであり、数年後にはオブジェクト指向プログラミングが現在の構造化プログラミングに代わって一般的になるものと考えられている。

オブジェクト指向言語において中心的な役割を果たすのはオブジェクトという概念であり、Smalltalk-80 などの場合には存在する全てのものをオブジェクトとして扱う。こうしたオブジェクト間の相互関係には幾つかの基本的なものが存在する。

- **上位クラス・下位クラス関係**
  オブジェクト指向言語において最も特徴的なのは、クラスの階層構造である。下位クラスは上位クラスのインスタンス変数、及びメソッドを継承する。
- **クラス・インスタンス関係**
  データの概念的なテンプレートを表すクラスとその具象化であるインスタンスとの関係を表す。Smalltalk-80 等で導入されているメタクラスとクラスとの関係もこの関係である。
- **クラス・メソッド関係**
  一般のオブジェクト指向言語では、メソッドはクラスに属するものとされる。

これらの関係を図で表現する場合、オブジェクトをノード、関係をリンクとするグラフ表現が一般的に用いられる。複数の関係を同時に表現する場合、例えば上位クラス・下位クラス関係とメタクラス・クラス関係を表現するには、異なる関係をそれぞれ違うパターンの線として表示する。Figure 3.2は Smalltalk-80 のメタクラス・クラス・インスタンスの関係を平面的に記述したものである。一方、3次元空間の概念を採り入れると、この関係は Figure 3.3のように表現できる。次に、クラス・メソッド関係に着目すると、一般にメソッドはクラスに属するものとして扱われるが、メソッド継承の場合には、同じ名前を持つメソッドの上下関係が重視される。つまりクラス階層に注目しつつ、同じ名前のメソッドの並びにも注目する必要がある。従来はクラ

Figure 3.2: Smalltalk-80 のクラスとメタクラスの関係 (2 次元)

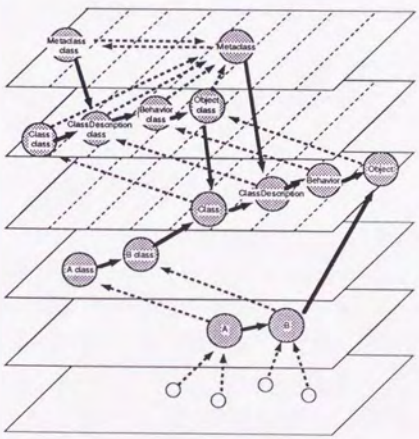


Figure 3.3: Smalltalk-80 のクラスとメタクラスの関係 (3 次元)

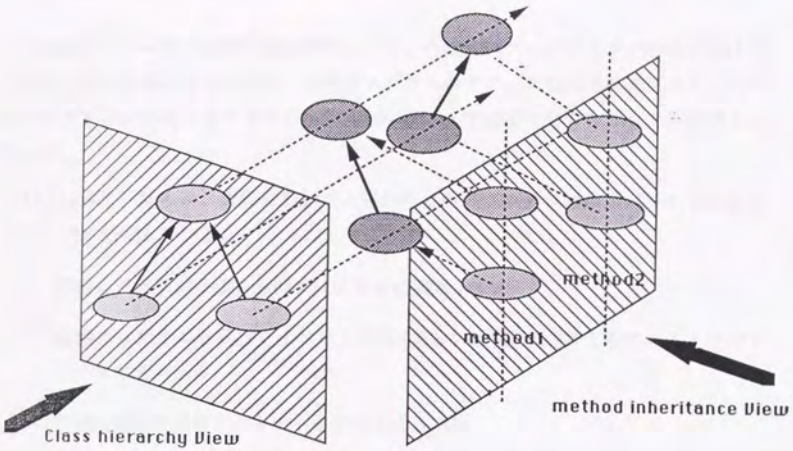


Figure 3.4: Smalltalk-80 のクラスとメソッドの関係 (3 次元)

ス階層の木とメソッドの起動順番の表という形で表現されることが多い。一方、3 次元空間の概念を採り入れることで、この関係は Figure 3.4のように表現できると思われる。

並列プログラムのデバッグ

計算機システムの大規模分散並列化により、今後並列プログラミングの必要性は増大する一方である。その一方で、並列プログラムのデバッグには逐次的プログラムのデバッグに用いられてきた手法(cyclic debugging)では不十分であることが指摘されている。

McDowell [74]らは、実際に有効だと思われる並列プログラムのデバッグ手法を以下の4つに分類している。

1. 従来のデバッグ手法を並列プログラムに適用する
2. 並列プログラムの実行をイベントの列(あるいは複数の列)としてみるイベントベースのデバッガ
3. 制御の流れや分散されたデータを表示する方法
4. データフロー解析に基づく静的解析手法

このうち3番目の表示手法として、

1. データのテキスト表現や制御の流れの表示
2. プログラムの実行を時間と個々のプロセスを両軸にとって表示した時間 - プロセス・ダイアグラム
3. プログラムの実行のアニメーション。表示はある瞬間のプログラムの状態のスナップショット
4. マルチウィンドウ。プロセス毎にウィンドウを割り当てる

をあげている。しかし、アニメーションはある瞬間のシステムの状態を観察するためには良いが、一定時間に渡った動作のパターンは明確に示されない。時間 - プロセスダイアグラムは一定時間に渡った動作のパターンを表示できるが、その代わりに各時刻の各プロセスに関する情報は少ない。結論として、複雑な並列システムに理想的なデバッガはアニメーションと時間 - プロセス表示の両方をサポートするものであり、この両者をそれぞれウィンドウを用いて表示する方法が考えられる、としている(Figure 3.5)。

一方、3次元空間の概念を用いると、Figure 3.6のように表現できる。

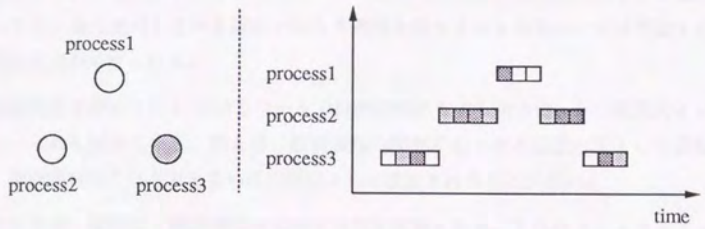


Figure 3.5: 並列プロセスのデバッグに利用する図 (2 次元)

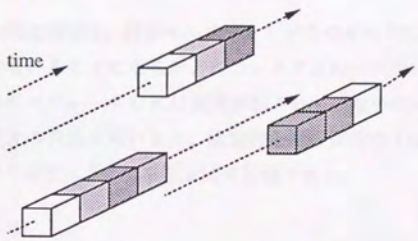


Figure 3.6: 並列プロセスのデバッグに利用する図 (3 次元)

版管理・構成管理

ソフトウェア開発、特に大規模なソフトウェア開発においては、版管理や構成管理が非常に重要である。

UNIX 上では版管理には SCCS や RCS、構成管理には Make が用いられている。これらは版管理、あるいは構成管理ツールとして、機能的にはほぼ十分であるが、残念ながらインタフェースが不十分である。特にこれらの情報を視覚的に表示する機能がないため、現在使用している版がいかなる過程を経てきたものかユーザが把握するのが困難な点があげられる。

最近注目を集めている CASE ツールでは機能的にもより強力で、かつ視覚的インタフェースにも優れている。例えば、版情報は時間軸に沿った木構造の図として表現され、構成情報はやはり木あるいは領域図として表現されることが多い。

ところで、版情報と構成情報は本来不可分な関係にある。1つのソフトウェア・システムは幾つかのモジュールから構成されるが、このシステムとしての版は必ずしも個々のモジュールの版とは一致しないのが普通である。あるモジュールは変更の必要がなく、前回のコードがそのまま再利用され、一方、別なモジュールは新たに作成されたコードからなる場合が多々ある。

こうした版情報と構成情報は、現在マルチウィンドウでそれぞれ Figure 3.7,Figure 3.8のような図として表示されることになるが、このシステムのバージョン1からバージョン2への移行においてモジュール C には変更が施されていないことは直感的にはわからない。一方、3次元の概念を用いると、版情報と構成情報は Figure 3.9のように表現でき、モジュールの変更についても視覚的に把握できる。

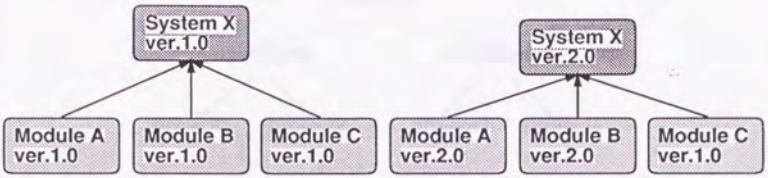


Figure 3.7: 構成情報の視覚化 (2 次元)

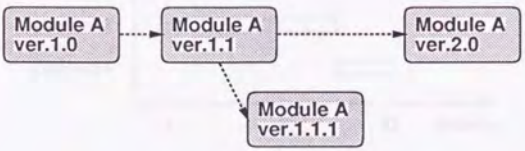


Figure 3.8: 版情報の視覚化 (2 次元)

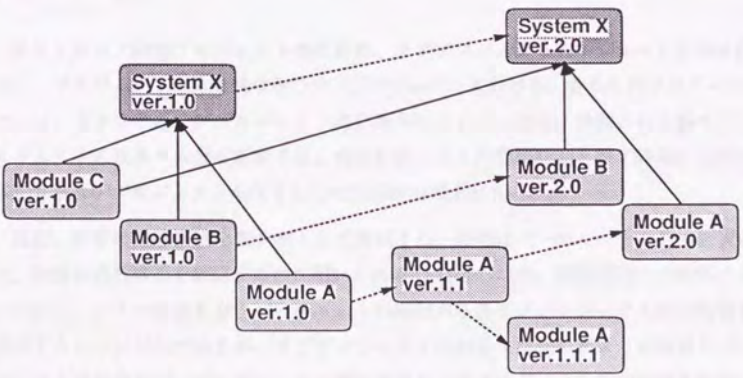


Figure 3.9: 版情報と構成情報の視覚化 (3 次元)

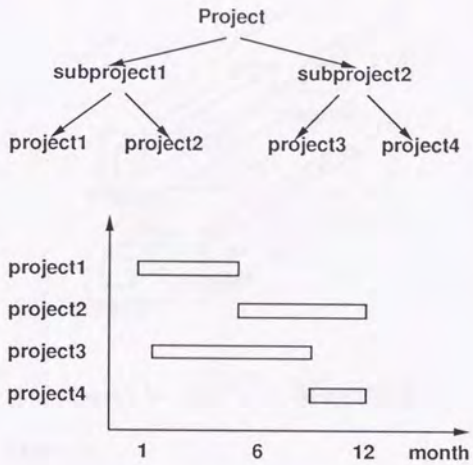


Figure 3.10: プロジェクト管理 (2次元)

プロジェクト管理

ソフトウェア開発プロジェクトの設計は、まずプロジェクトのモジュール分割を行ない、プロジェクト全体を幾つかのサブプロジェクトに分ける。さらに各サブプロジェクトは、より小さなサブプロジェクトに分割されることもある。分割された各サブプロジェクトには各々人員が配置され、内容に応じてその開始時期と終了時期の見積りが立てられ、プロジェクト全体としての計画が出来上がる。

現在、前者は階層構造を表す木として表現され、後者はバーチャートとして表現され、作業の並列度等を確認するのに用いられる (Figure 3.10)。階層構造の比較的小さいプロジェクトの場合には、バーチャートの縦軸から各サブプロジェクト間の関係を把握することは可能であるが、サブプロジェクトの細分化が進んだより大規模なプロジェクトの場合には、バーチャートの情報だけから各プロジェクト間の関係を把握することは困難になると考えられる。一方、3次元の概念を導入するとこれらは、Figure 3.11のように表現できる。

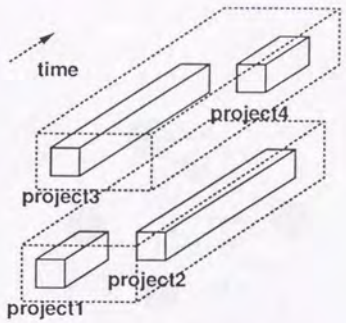


Figure 3.11: プロジェクト管理 (3 次元)

ニューラル・ネットワーク

ニューラル・ネットワークでは、各セルの物理的位置関係や各層間の接続関係が重要である。例えば、Rumelhurt のバック・プロパゲーション・モデル [92] は、ネットワークが Figure 3.12のように表現される。そして、ある時刻における各セルの状態は、その図において発火しているセルと発火していないセルを色分けすることで、認識することができる。

ところで、ニューラル・ネットワークにおいては、任意の時刻におけるネットワークの状態よりは、ネットワークの学習過程、つまり初期状態から収束状態に至る間のネットワーク全体の発火の様子、つまりネットワークのダイナミクスが重要な場合が多い。このような場合には、トレースを記録しておいて画面上でアニメーションとして表示している。

もちろんこれは、システムのダイナミクスを直感的に把握するのには十分であるが、例えば別なトレース結果と比較検討を行なうといった要求には対応できないと思われる。

一方、3 次元空間を用いることで、ネットワーク形状とその時間的変化を Figure 3.13のように表現できる。この図により、学習過程を視覚的イメージとして捉えることが可能となり、結合度の初期値や入力の違いによる学習過程の比較・分析が行なえると思

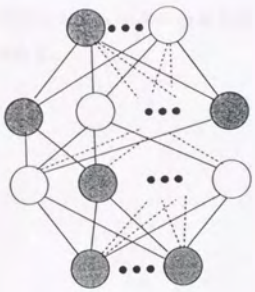


Figure 3.12: ニューラルネットワーク

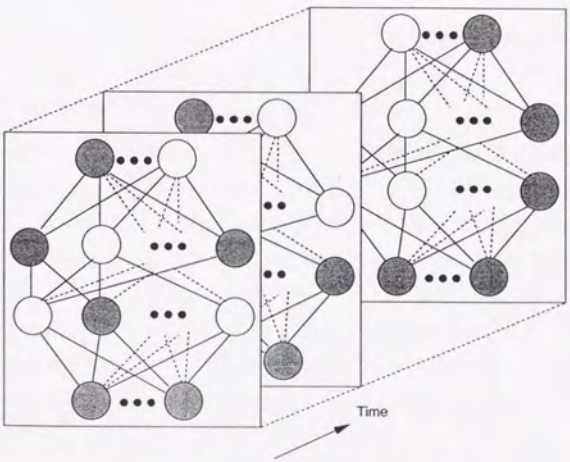


Figure 3.13: ニューラルネットワーク (3 次元)

われる。

ただし、実用的ニューラル・ネットワークにおいてはそのセルの数は膨大な数にのぼるため、このような大規模なネットワークの視覚化には、かなり高度なグラフィックス能力が必要となる。

ここで述べた視覚化手法は並列コンピュータにおける各プロセッサの挙動を視覚化するのにも応用できる考え方である。

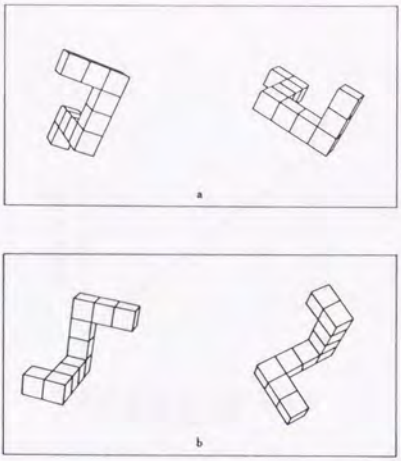


Figure 3.14: メンタル・ローテーションの実験に使用された図

3.4 3次元視覚化ツールを実現する上で考慮すべき点

3次元CGを利用することで、従来2次元平面上の1つの図で、あるいは複数の図で表現していた関係を、1つの図として表現できることを示した。本節では、この3次元視覚化の枠組を実際に実現する場合に考慮しなければならない点を考察する。

3次元空間上に実現された図は、視点を移動することで幾つかの側面を見ることができることを示したが、その際まず考慮しなければならないのは、視点の移動方法である。Shepard [93] らの実験によれば、ある空間的物体を幾つかの角度から見た透視図が同じ物体であるかどうか判断する時間は、図形の角度差に比例して大きくなることがわかった。これから人間は心の中でこれらの図を回転させ(メンタル・ローテーション)一致するかどうか調べていることが実験によって示された訳である。この結果から、ある3次元図形に対して、異なる視点からの図を表示する場合、前の図との不連続さが大きいほど、人間に対する負荷は高まるということができる。可能な限り、連続移動を実現することが必要だと思われる。

次に、3次元CG上に描画された図は、そのままではあくまでも2次元投影図に過ぎないため、図形の前後関係等を把握する際に誤認する可能性がある。これはネッカー・

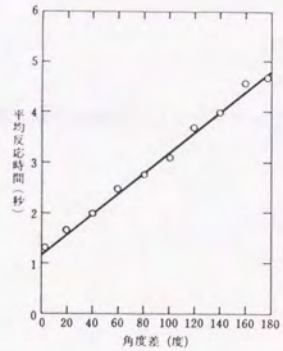


Figure 3.15: メンタル・ローテーションの実験結果

キューブの図 (Figure 3.16) で有名である。このような誤認を防ぐためには、3次元図形を3次元図形として認識できる機能が必要である。

さらに視覚化空間を3次元化するということは、単純に考えると描画できる図素の増加を意味するが、3次元視覚化自体は図形数の増加による図形の複雑さの問題に対して何の解決も与えない。従って、図形制御手法が必要である。

以上まとめると、3次元視覚化を実際のソフトウェアに適用するにあたり考慮しなければならない点は、

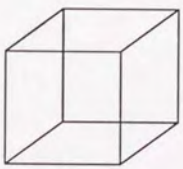


Figure 3.16: ネッカーキューブ

- 視点の連続的移動
- 3次元立体視の支援
- 図形数の制御

であると考えられる。

本章における考察は、第5章で実現されるソフトウェア視覚化ツール VOGUE の基礎となる。

第 4 章

表示情報量制御に関する考察 − Generalized Fractal Views の開発 −

4.1 緒言

図形に限らず必要以上に多量の情報の提示は、人間の認知を阻害する。問題はいかに着目点近傍を効果的に表示し、非着目点を表示しないかである。その時に注意すべき点として、焦点の移動に伴う提示情報量の変化の起伏が大き過ぎないことが挙げられるが、これは従来見過ごされていた点である。

本章ではこうした点を踏まえ、提示情報量をほぼ一定に保ちながら着目点近傍を効果的に表示する手法として著者が開発した Generalized Fractal Views について述べる。本手法は、後に提案する 3 次元視覚化ツールにおいて図形数制御に利用されるが、ここではその定式化を行なう。

以下、4.2 節では従来の情報量制御手法のうち代表的なものを概観し、その問題点を指摘する。4.3 節ではこれらの考察を基にフラクタル次元に基づく情報量制御手法 Generalized Fractal Views について述べ、数値シミュレーションにより、その有効性を示す。

v	n	output
0	1	#
1	1	(if...)
1	2	(if # ...)
1	3	(if # # ...)
1	4	(if # # #)
2	1	(if ...)
2	2	(if (member x ...) ...)
2	3	(if (member x y) (+ # 3) ...)
3	2	(if (member x ...) ...)
3	3	(if (member x y) (+ (car x) 3) ...)
3	4	(if (member x y) (+ (car x) 3) '(foo . #(a b c d ...)))

Table 4.1: LISP オブジェクトの表示例

4.2 表示情報量制御に関する従来の研究

4.2.1 LISP のプリンタ

木構造は、現在計算機上で取り扱うデータ構造の基本的なものの一つである。実際には木構造ではなくネットワークである場合にも、便宜上木構造と考えて処理する場合が多い。こうした木構造に基づいた情報量削減手法として、従来から用いられてきたものとしては、木の深さと枝の数に着目したものがある。身近なものとしては LISP におけるプリンタが上げられる。

例えば Common Lisp[99] の場合、 *print-level*、*print-length* という 2 つの大域変数がオブジェクトの表示数を制御する。前者は入れ子になったデータオブジェクトが印字する深さのレベルを制御し、後者はあるレベルにある要素の印字個数を制御する。例として、以下のようなオブジェクトの場合がある。

```
(if (member x y) (+ (car x) 3) '(foo . #(a b c d ''Baz'')))
```

これは *print-level* = v、*print-length* = n のさまざまな値に対して Table 4.1のように印字される。

フレキシブルな値の設定が可能であるが、それらはユーザに委ねられ、毎回設定を変更しなければならない。

4.2.2 Generalized Fisheye Views

G. W. Furnas は、局所的な情報と大局的な情報のバランスを考慮した汎用的な情報量制御手法 Generalized Fisheye Views [36] を提案した。

彼は、ある階層構造の定義できるある情報構造において、木の各ノードのルート・ノードからの距離 $API(x)$ と、現在着目しているノードからの距離 $D(x,y)$ を用いて、各ノードが持つ重要度 $DOI_{fisheye}(x|=y)$ を以下のように定義した。

$$DOI_{fisheye}(x|=y) = API(x) - D(x,y)$$

Figure 4.1はある木における $API(x)$ と $D(x,y)$、および $DOI_{fisheye}$ を示している。

そして、閾値 k を変化させることでその閾値以上の DOI を持つノードを表示するというストラテジーである。 Figure 4.2は、閾値 k を変化させた時にこの木構造をいかに表示するかを示したもので、順番に zero order fisheye、first order fisheye、second order fisheye と呼ばれる。

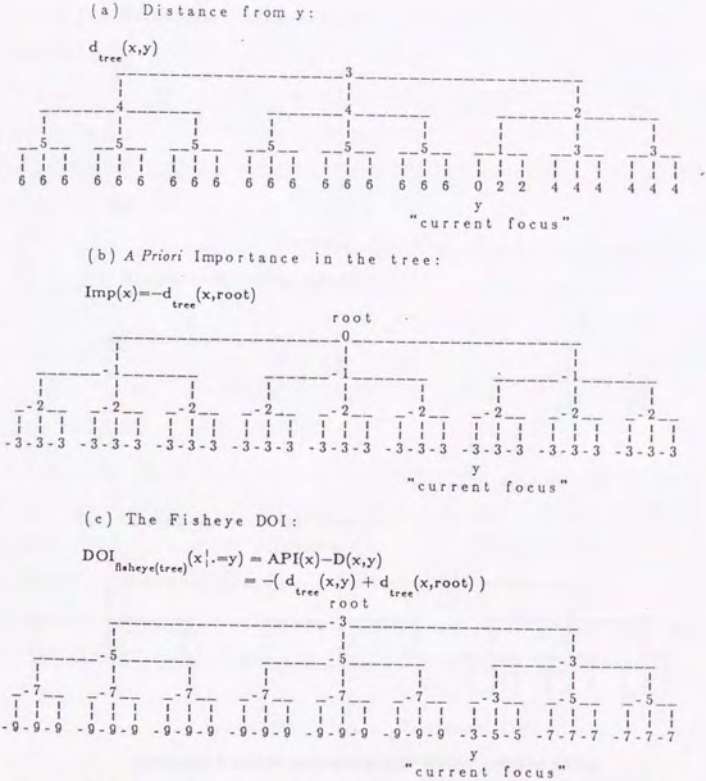
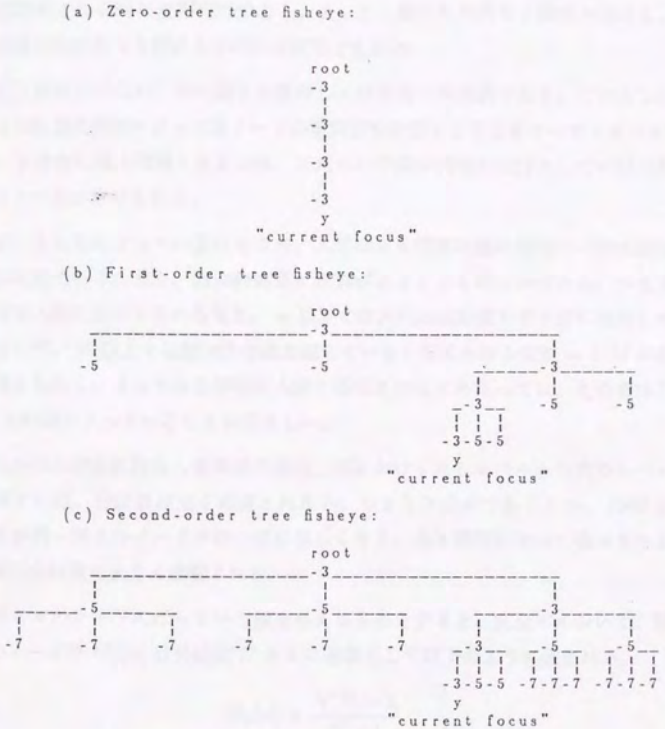


Figure 4.1: Fisheye View の計算方法



. Zero-order, first-order and second-order fisheye views for a tree.

Figure 4.2: Fisheye View の例

4.2.3 従来の手法に関する考察

LISP のプリンタで用いられる手法は最も原始的な手法であるが故に、柔軟な出力をできる反面、変数の設定をいちいちユーザが行なう点が問題といえる。

Generalized Fisheye Views はより洗練された手法ではあるが、ある一つの明かな階層構造が定義される時に使用可能である。よって、幾つかの異なる関係が存在し、それら間の重要性にあまり差がない時には使用できない。

こうした従来の手法は、木の深さを重視している点では共通である。このように注目点からの論理的距離に従って各ノードの重要度を決定する手法をユーザ・インタフェースに用いる場合に最も問題となるのは、これらの手法が情報の全体としての提示量を制御できない点が挙げられる。

個人差、また日によっての差はあるが、人間はある程度の量の情報を一時に認知することが可能である。また、認知許容量に限界があることも明らかである。つまり、ある情報を人間に提示するにあたり、m 以下では人間の認知能力を十分に利用しているとは言えず、M 以上では認知許容量を越えていると考えられる変数 m と M が存在すると考えられる。よってある情報が人間に提示されるにあたっては、その量はこの m と M の範囲に入っていることが望ましい。

ところが木の深さに注目した手法の場合、木におけるあるレベルから次のレベルに値を伝播する際、分岐数は全く考慮されない。つまり 2 分木であろうが、1000 分木であろうが同一深さのノードが持つ値は等しくなり、ある閾値によって提示される情報量は木の分岐数に大きく影響される。

今、深さ n のノードには n という値を与えるものとすると、完全木において、閾値 k 以下のノード数 $M_k(n)$ は分岐数 N と k の関数として以下のように表される。

$$M_k(n) = \frac{N^{k+1} - 1}{N - 1}$$

Figure 4.3: 分岐数の異なる木の場合

横軸に分岐数 N、縦軸にしきい値 k 以下の値を持つノードの総数 $M_k(n)$ をとり、k を変化させたときのグラフを Figure 4.4 に示す。$k = 1$ の時には上式は $M_k(n) = N + 1$ と簡単化され、グラフにおいて右上がりの直線として表される。また、$k > 1$ の時には分岐数が増加するに従い $M_k(n)$ は指数関数的に増大する。つまり木の形に依存して、表示される情報量が極端に変化するという欠点がある。結局、木の深さだけを考えた手法では、このような欠点は避けられない。しかし、ユーザ・インタフェースへの利用という点から見た時に必要なのは、着目点の移動によって提示される情報量が極端に変化しないことである。次節ではこの点を考慮した手法について述べる。

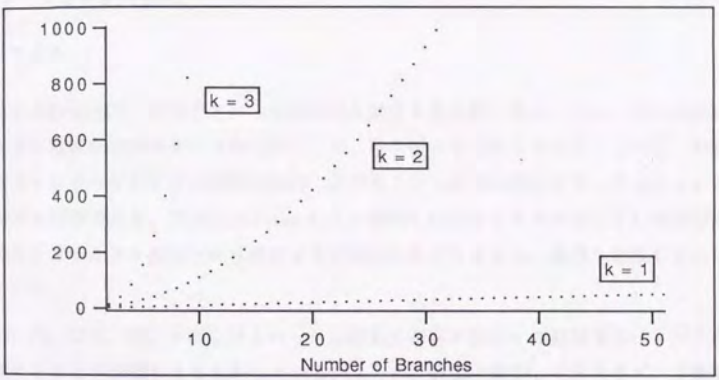


Figure 4.4: 分岐数と表示されるノード数の関係

4.3 Generalized Fractal Views

前節での考察から、必要なのは表示情報量をほぼ一定に保つことのできる情報量制御手法であることを述べた。本節では、著者が開発した情報量制御手法について述べる。

まず、B. Mandelbrot によって提唱されたフラクタルの概念 [72] を完全木から一般木へ、そして論理的一般木へと概念拡張した後、Generalized Fractal Views について述べる。さらに計算によってその有効性を検証する。実際のデータにおける有効性は第 7 章で述べられる。

4.3.1 フラクタル概説

フラクタル

従来の数学は円、長方形といった理想的な図形を主に取り扱い、その一方で海岸線のような複雑な図形はその他の図形として、取り扱いを拒否していた。しかし、実際には我々の身のまわりには理想的な円、長方形といった形は存在せず、そのほとんどが複雑な図形である。B.Mandelbrot により提唱されたフラクタルはこうした図形の複雑さをフラクタル次元という値により定量化することにより、複雑さを取り扱いやすくした。

山、川、樹木、雲、宇宙などといった自然界に存在する多くの複雑な形はフラクタルであることが研究によりわかっている。さらに、株価の変動、1/f ノイズ、浸透現象などもフラクタルであることがわかっており、フラクタルは複雑さ一般を扱う概念として注目を集めている。

自己相似図形と相似性次元

自己相似図形とは、全体と部分とが厳密に相似な図形のことで、コッホ曲線はその代表的なものである (Figure 4.5)。

従来、次元という概念はその空間における自由度の数として定義されていた。しかし、例えばペアノ曲線は 2 次元であるはずの正方形上の任意の点をたった一つの実数で表し得る。この矛盾を避けるために幾つかの次元が考えだされたが、相似性次元は

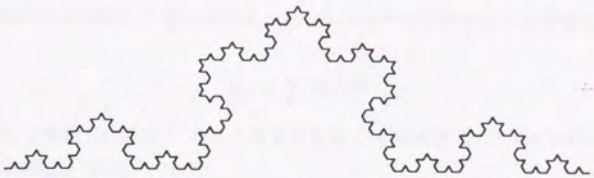


Figure 4.5: コッホ曲線

その一つである。

相似性次元は、ある図形が全体を $1/a$ に縮小した図形 b 個で構成されているとすると、

$$D = \log_a b$$

と定義される。これによれば、従来の線分、正方形、立方体の次元はそれぞれ 1、2、3 となり経験的な次元と一致する。また上述のコッホ曲線の場合は、

$$\log_3 4 = 1.26...$$

となり非整数値をとる。

この自己相似性次元は厳密な自己相似図形にしか適用されない。任意の図形の次元は次に述べる Hausdorff-Besicovitch 次元で定義される。

Hausdorff-Besicovitch 次元

いま空間内の点の集合 S を d 次元球 $h(\delta) = r(d)\delta^d$ で被覆することを考える。その時

$$M_d = \sum r(d)\delta^d$$

は、$d < D$ で発散し、$d > D$ で 0 となるような D が存在する。この時の D を Hausdorff-Besicovitch 次元と呼ぶ。

このような D は与えられた図形に対し唯一存在することが証明されている [33]。自己相似図形においては相似性次元と Hausdorff-Besicovitch 次元とは一致する。Mandelbrot は最初この Hausdorff-Besicovitch 次元がトポロジカル次元より大きいことをフラクタルの定義としていた。以下は Mandelbrot による最初のフラクタルの定義である。

> *"A fractal is by definition a set for which the Hausdorff-Besicovitch dimension strictly exceeds the topological dimension."*

統計的フラクタル

Hausdorff-Besicovitch 次元の欠点は、取り扱いが繁雑なことである。そこで一般によく用いられる方法は、特徴的な長さ r とその r から求められる物理量 N(r) を log-log プロットする方法である。これによれば、フラクタルは右さがりの直線となる。円、長方形などの特徴的な長さをもつ図形は直線にのらない。

海岸線を適当な r に従い log-log プロットすると、直線にのることがわかる。このように正確に自己相似ではないが、やはり特徴的な長さを持たないものを統計的フラクタルと呼ぶ。

Mandelbrot もその後フラクタルの定義を以下のように改めている。

> *"A fractal is a shape made of parts similar to the whole in some way."*

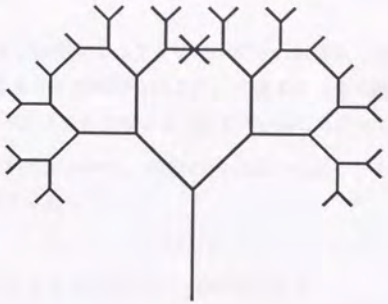


Figure 4.6: フラクタルな木

4.3.2 木のフラクタル性の概念拡張

完全木のフラクタル性

Mandelbrot がその著書「Fractal Geometry of Nature」の 16 章において述べているように、樹形構造はフラクタル性を持つ。Figure 4.6は 2 分木をある反復アルゴリズムによって表示した例であるが、分岐が無限に続く理想的な状態を仮定すれば、この木は厳密な意味での自己相似性を有している。

自己相似図形の代表としてしばしば例に取りあげられるコッホ曲線の場合、その相似性次元はあるレベルでの代表長さと次のレベルでの代表長さとの縮尺率 $r = 1/3$ と得られる図形の数 $N = 4$ とから、

$$D = -\log_r N = \log_3 4 = 1.26...$$

と計算された。この定義にならえば、Figure 4.6のような 2 分木の相似性次元は、あるレベルでの枝の長さがその前のレベルでの枝の長さの r 倍になっているとすると

$$D = -\log_r 2$$

で表される。N 分木の場合も同様にして

$$D = -\log_r N \tag{4.1}$$

で定義することができる。上式が完全木のフラクタル次元の定義である。

一般木のフラクタル性

本節では前節で定義した完全木のフラクタル次元の定義を一般の木に拡張する。一般木もフラクタルであることが期待されるが、完全木のように厳密な自己相似ではないので、ここでは log-log プロットによる方法を用いることとする。

Figure 4.6の完全 2 分木において、最初の枝の長さを仮に 1 とすると深さ n のノードから分岐する枝の長さ δ は、

$$\delta = r^n$$

である。よってこの深さにおける枝の長さの総和 $L(\delta)$ は

$$L(\delta) = (2r)^n$$

n が

$$n = \ln \delta / \ln r$$

で表されることから、$L(\delta)$ は以下のように表される。

$$L(\delta) = (2r)^n = exp(\frac{\ln \delta[\ln 2 + \ln r]}{\ln r}) = \delta^{1-D}$$

ただし、

$$D = -\ln 2 / \ln r$$

従って、その深さにおける枝の総数 $N(\delta)$ は、

$$N(\delta) = 2^n = 2^{\ln \delta / \ln r}$$

であるから、

$$N(\delta) = \delta^{-D}$$

となる。これを横軸に $\log \delta$、縦軸に $\log N(\delta)$ をとってプロットすると右さがりの直線となり、その傾きは $-D$ である。

さてここで、あるスケール・ファクタ r_2 をもつ 2 分木が世代 n からスケール・ファクタ r_3 を持つ 3 分木に変化するとする。この時、この木の log-log プロットは Figure 4.7のようになると考えられる。

いま、2 分木のプロット部分と 3 分木のプロット部分が 1 直線となるためには、その傾きが等しいこと、つまり、

$$\log_{r_2} 2 = \log_{r_3} 3$$

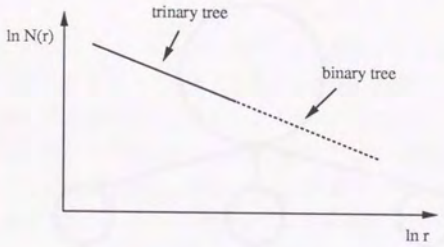


Figure 4.7: log-log プロット

が必要十分である。

よって任意の一般木において、分岐数 N とスケール r との間に

$$\log_r N = Constant$$

の関係があるならば、その log-log プロットは右下がりの直線となる。よってこのような一般木は統計的な意味でのフラクタルであるといえる。

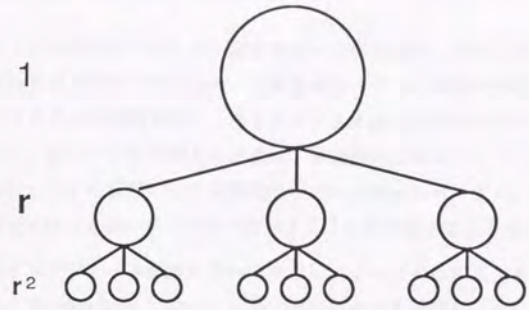


Figure 4.8: 論理木のフラクタル性

論理木のフラクタル次元

さて、上で求めた N 分木の相似性次元は、枝の長さという物理量を持つ木の相似性次元である。一方、計算機で取り扱う論理的な木の場合、図に描画された木の枝の長さのような物理的長さというものを持たないためそのままでは相似性次元の計算は不可能である。

しかしこの問題は、木のノードが持つある概念的な重みを仮定することによって概念拡張できる。つまり、木の各ノードはある概念的な重み (これをフラクタル値と呼ぶことにする) を持ち、木のレベルが深くなるにつれて、ある縮尺率 r により小さくなっていくときフラクタル性が実現する、とする。

この概念拡張によって、そのフラクタル値がスケール・ファクタ r によって制御される論理的な完全 N 分木の相似性次元は

$$D = -\log_r N$$

として定義することが可能となる。

前出のコッホ曲線は、$D = \log_3 4$ をもつ論理木の具現化の一形態であり、カントール集合は $D = \log_3 2$ をもつ論理木の具現化の一形態である。

4.3.3 Generalized Fractal Views の定式化

ある閾値によって表示オブジェクト数をほぼ一定に制御するためには、分岐が増えるほど伝播される値が小さくなるメカニズムを必要とする。前節での考察から、一般の木構造はフラクタル的性質を有し、あるフラクタル次元の下でフラクタル次元の定義式に基づいて、各ノードに伝播される値は、分岐数に依存して小さくなることがわかっている。従って、分岐数の多い木構造の場合には浅いレベルまで、分岐数の少ない木構造の場合にはより深いレベルまで表示することが可能となると考えられる。

この概念をインタフェース設計に用いるために、ここでその定式化を行なう。定式化にあたって、Generalized Fisheye Views が用いたのと同様に、ユーザが各ノードをどの程度着目しているかを表す、"Degree of interest"関数 (DOI) を用いることにする。Generalized Fisheye Views では DOI として、明示的重要度と距離を用いたが、ここでは、以下のような式を用いる。

$$DOI_{fractal}(x|=y) = Fv$$

ただし Fv はルートノードを 1 とし、以後あるノードにおける子ノードの数を N、任意のフラクタル次元を D とした時、

$$r = N^{-\frac{1}{D}}$$

という式によって伝播される値とする。伝播される値は木の下へ行く程小さくなり、また分岐が多い時ほど子ノードに伝播される値は小さくなる。任意の閾値 k を選び、それ以上のノードを表示することにすると、異なる表示を得ることができる。

4.3.4 数値シミュレーションによる有効性の検証

前節までの考察から一般木のフラクタル次元は、スケールファクタ r と分岐数 N を用いて

$$D = -\log_r N$$

と定義された。故に N は r と D によって

$$N = r^{-D}$$

と表される。一方、完全木において深さ n までの総ノード数 $M(n)$ は

$$M(n) = \frac{N^{n+1} - 1}{N - 1}$$

である。一方深さ n におけるノードのフラクタル値 Fv は

$$\begin{aligned} Fv &= r^n \\ &= N^{-\frac{n}{D}} \end{aligned}$$

これより

$$\begin{aligned} -\frac{n}{D} &= \log_N Fv \\ n &= -D \log_N Fv \end{aligned}$$

よって

$$\begin{aligned} M(n) &= \frac{N^{1-D\log_N Fv} - 1}{N-1} \\ &= \frac{NFv^{-D} - 1}{N-1} \end{aligned}$$

となる。

いま D、k をいろいろ変えた場合の N と $M(n)$ の関係をグラフにすると Figure 4.9から Figure 4.12のようになる。横軸が完全木の分岐数 N、縦軸がノード数 $M(n)$ である。D と k を決めると木の分岐数が大きくなっても表示されるノード数はほぼ一定に保たれることがわかる。

このグラフはあくまでも完全木における結果であるが、D と k を決定すると分岐数によらず提示されるノード数は一定になることから、これは一般の木においても有効であると考えられる。これより Generalized Fractal Views は以下の 2 つの特徴を持つことがわかる。

1. 木の形に関係なくノード数をほぼ一定に制御できる

2. 閾値を変化させることで、表示ノード数を滑らかに変化できる

実際の木における効果は第 6 章において検証される。また、図形以外の応用例に関しては第 8 章で述べられる。

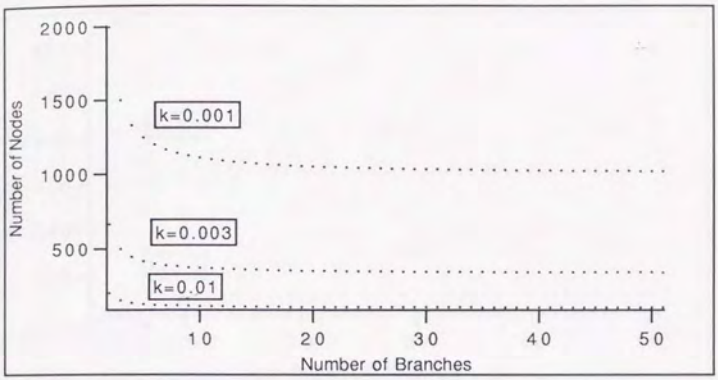


Figure 4.9: 分岐数 N とノード数の関係 (D=1)

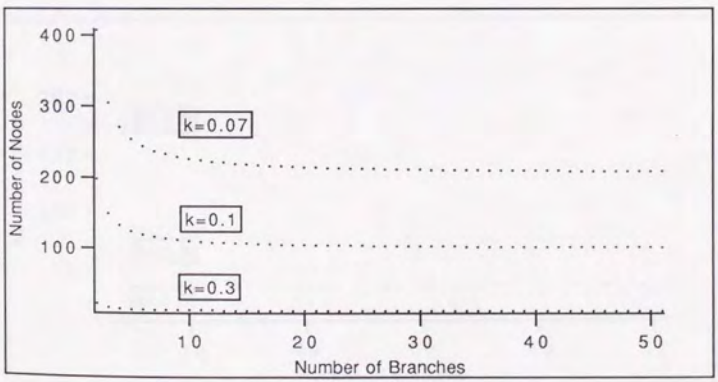


Figure 4.10: 分岐数 N とノード数の関係 (D=2)

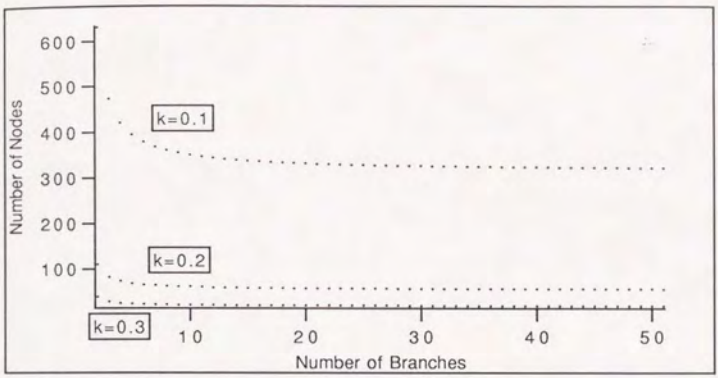


Figure 4.11: 分岐数 N とノード数の関係 (D=2.5)

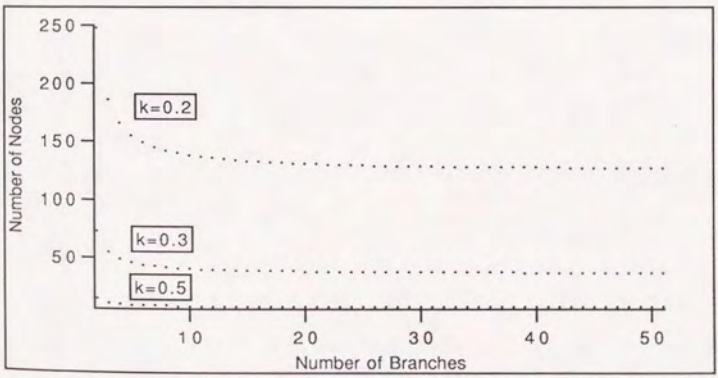


Figure 4.12: 分岐数 N とノード数の関係 (D=3)

第 5 章

試作システム VOGUE

5.1 緒言

本章では、ソフトウェアの3次元視覚化のための試作システムのインプリメントについて述べる。試作システムは VOGUE (Visualization Oriented Generic User-interface Environment) と呼ばれ、対象とするソフトウェアのモデル化を行なうためのデータベースとグラフィックス・ワークステーション上に実現された3次元 Grapher から構成される。そして、続く第6章、第7章においてそれぞれ実際のソフトウェアの3次元視覚化に適用する。

これまで、第2章において図形言語と従来の視覚化ツールについて概観し、実際のソフトウェアを視覚化する上で考慮すべき点を述べた。得られた知見は以下のようなものであった。

- 表現手法としてはグラフ表現が汎用性の点で優れており、領域系図、座標系図への対応も可能である。
- 異なるソフトウェアに対応するためには、モデル化の機能としてデータベースが必要である。また、データモデルとしてはオブジェクト指向モデルが良い。
- 視覚化対象の量に対応するためには、高度なグラフィックス機能が必要である。
- 図形数の制御が必要である。図形数の削減は、人間にとって図の複雑さを削減するとともに、システムの反応性を高める。
- マルチウィンドウの乱用は有害である。

また、第3章において3次元視覚化を提案し、3次元視覚化ツールが考慮すべき点について考察を行なった。ここで得られた知見は以下のようなものである。

- 視点の連続的移動を支援する必要がある。
- 3次元立体視の支援が必要である。
- 図形制御手法が必要である。

さらに第4章では、第2章、第3章での考察から図形数削減手法は視覚化において無視できない問題であるとの認識から、汎用的な情報制御手法 Generalized Fractal Views を開発した。実現された VOGUE は以上の考察を踏まえたものとなっている。

以下、5.2節において VOGUE の設計思想について述べ、続く 5.3節では VOGUE のシステム構成について述べる。5.4節では CLOS を拡張して実現したオブジェクト指向データベースについて述べ、5.5節ではグラフィックス・ワークステーション上に実現した3次元 Grapher について述べる。さらに、5.6節ではこの Grapher 機能を基本として実現された拡張機能について述べ、5.7節ではデータベースと Grapher の結合について述べる。最後に 5.8節で本章のまとめを述べる。

5.2 VOGUE の設計指針

本論文の目的の1つは、ソフトウェアを視覚表現するにあたり従来の2次元ツールには支援できなかった視点の存在を示し、それらの有効性を実証することである。こうした目的を果たすための実験システムとして VOGUE を試作した。試作にあたっては以下のような設計指針に基づいている。

- 不完全な3次元入力インタフェースは利用しない
- 汎用性の重視とモデル化の機能
- 視覚化のための対話的環境の提供
- 3次元立体視のハードウェア・サポート
- 図形制御機能の提供

それぞれについて以下に説明する。

不完全な3次元入力インタフェースは利用しない

将来的には、VOGUE によって示された3次元視覚化の枠組を用いて、3次元グラフィックスと直接対話する環境が実現されることが望まれるが、この機能を完全に実現するためには現段階では3次元入力インタフェースの技術が確立されていない。ここで言う3次元入力インタフェース技術とは具体的には以下のようなものである。

- 3次元の対象のポインティング
  3次元空間上に表現された対象物を正確にポインティングし操作する機能。2次元環境におけるマウスに対応する概念である。現在、2次元マウスによる方法、データグローブ等、幾つかの3次元ポインティング・デバイスが開発されているが、正確なポインティングを実現できるものはない。
- 3次元空間での視点の移動
  3次元空間上に表現された対象物に対して、ユーザの視点がある位置と着目する点の位置の移動。ユーザの視点の変更は2次元の場合基本的に縦横へのスクロー

ルと拡大・縮小による疑似的な奥行き方向への移動であるが、3次元の場合これに回転という要素が加わる。現在、3次元的スクロールやCADで用いられるダイヤルボックスによる視点の移動があるが、インタフェースの自然さという点から見ると不完全な技術である。

これらの問題が解決されない限り、独立したツールとしてエンド・ユーザに利用させるのは無理がある。本研究における3次元視覚化の有効性の評価に被験者を用いた実験を行なう場合、実験の結果が3次元視覚化の本質によるものか3次元インタフェースの未熟さによるものかが明らかでない。従って3次元空間上の対象を直接操作し、対話的に編集する機能は持たないこととする。ただし、現在までに開発された、あるいは将来的に開発が期待されるデバイスを用いた開発環境については、第8章で人工現実感システムとの融合として述べる。

汎用性の重視とモデル化の機能

ソフトウェアの3次元視覚化の有効性を実証する目的のためには、あるソフトウェアにだけ適用できる視覚化ツールを開発するだけでは不十分であり、異なる様々なソフトウェアに対してもツールのその基本的実現部分に変更を加えることなく対応できることが望ましい。従って実現にあたっては汎用性を重視し、表現手法としてはノード・リンクを構成要素とするグラフ表現をとるものとする。

さらに、異なるソフトウェアにおいては対象とするデータは様々であるため、これらのデータをユーザが自分なりにモデル化することによって対応できるような機能が必要であると考える。本論文で実現した3次元視覚化表現は、例えばPHIGS等の他の3次元グラフィックス・ライブラリを用いて実現することは可能であると思われるが、異なるアプリケーションに対しては、また新たに最初からプログラムを書き直さなければならない。しかも、VOGUEで採用したオブジェクト指向データベースは、クラスによってデータの型を定義することが可能なため、この型による情報削減機能が実現できる。

視覚化のための対話的環境の提供

先に述べたようにVOGUEでは3次元に描画された対象物を直接操作する機能は持

たないが、視覚化した対象のある部分に変更を加えたり、移動させたりという基本的な操作は必要である。しかも可能なかぎり対話的に、例えばユーザがコマンドを打ち込むことによって行なえることが必要であると考えられる。

また、ユーザとしてはある程度 VOGUE に対する知識を持った人間を想定することで、一連のグラフィックス操作を Lisp のプログラムとしてプログラムできるようにすることとする。

3次元立体視のハードウェア・サポート

3次元グラフィックスを用いて表現した対象物は、そのままではただの2次元投影図に過ぎず、3次元グラフにおけるリンクの交差という問題は解決されないままである。3次元的に描画された図を立体として認知するためにはソフトウェアだけによる解決は困難だと考えられ、ハードウェアによるサポートが必要となる。

図形数制御機能を提供する

前章までの考察から、実用的視覚化ツールが図形数制御機能を持つべき理由は以下の2つある。

1. 第2章での考察でも述べたとおり、図形内図素数の増加は人間の認知に悪影響を及ぼし、図形の持つ利点を失わせてしまう。この図形数の問題は実用的視覚化ツールが解決しなければならない問題であり、特に本論文のように実際のソフトウェア視覚化を行なう場合には、当然視覚化すべき対象の増加が予想されるため、図形削減機能は必要不可欠である。

2. 図形の増加はグラフィックスの反応性を悪化させる。図形の選択、視点の変更等の操作に対する応答性はツールの使いやすさに大きく影響する。このような意味からも図形数制御機能は必要である。

5.3 システム構成

5.3.1 ハードウェア構成

VOGUE は Figure 5.2に示すように2台の計算機から構成される。1台は SUN Microsystems 社製 SPARC Station 1 でこの上には、CLOS(Common Lisp Object System) [50] を拡張して実現したデータベースがある。もう1台は Hewlett Packard 社製 HP9000 350/SRX でこの上には3次元 Grapher 本体があり、これら2台のワークステ - ションはイ - サネットを介してプロセス間通信を行いメッセージの交換をする。Table 5.1はグラフィックス・ワークステーション HP9000 350/SRX の仕様であるが、Z バッファのサポートによる高速な隠面処理、ダブルバッファリングによる滑らかな動画の作成、多光源によるシェーディングなどが可能で、高速なグラフィックスの描画が可能である。また Starbase Graphics Library によって、カメラ・モデルを使った画像の生成が容易にできる。この HP9000 350/SRX には3次元ステレオ・ディスプレイが接続され、Solidray 社製液晶シャッタ眼鏡で3次元立体視が可能である。3次元立体視に関しては次節で述べる。さらに、このディスプレイ上部にはポリマス・センサが装着され、位置と姿勢の情報を HP9000 に送っている。

CPU	MC68020 (25MHz)
コプロセッサ	MC68881 (20MHz)
グラフィックス・アクセラレータ座標変換	180000 座標 /sec
スキャン・コンバージョン	1600 万ピクセル /sec
イメージ表示 (ラスタダンプ)	125 億ピクセル /sec

Table 5.1: HP9000 350/SRX の仕様

Figure 5.1: システムの全景

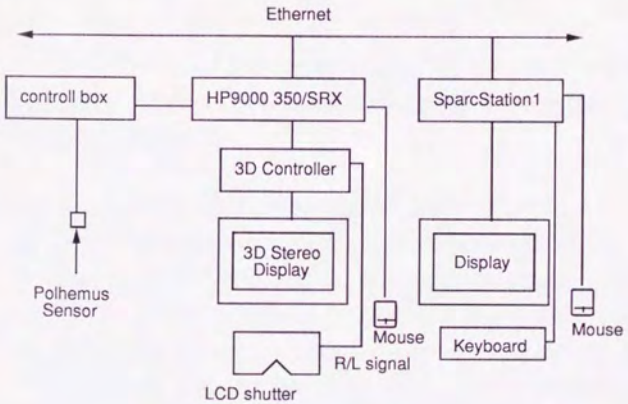


Figure 5.2: VOGUE のハードウェア構成

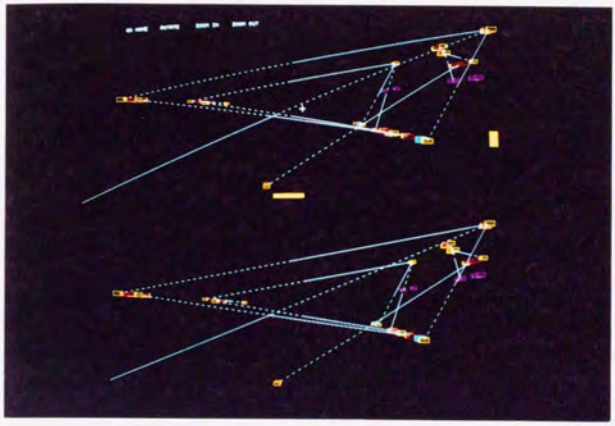


Figure 5.3: 左目用画像と右目用画像

3 次元立体視機能

VOGUE では、ソフトウェア情報を視覚化する枠組としてノード・リンクを構成要素とする 3 次元グラフを用いる。そして描画された 3 次元グラフを立体視するために、以下の 2 種類の方法を提供する。

- 液晶シャッタ眼鏡による両眼視差を利用した立体視
  HP9000 350/SRX の出力デバイスとして Solidray 社製立体視コントローラと専用の 3 次元ディスプレイを用いている。このディスプレイは垂直同期が 120Hz であり、画面の上半分に右目用画像、下半分に左目用画像を描画し (Figure 5.3)、これをスイッチの切替えによって画面全体に引き延ばし (Figure 5.4)、液晶シャッタ眼鏡 (Figure 5.5) によって立体視の映像を得る。1 秒間に 60 フレームの高品質な立体映像を得ることができる。

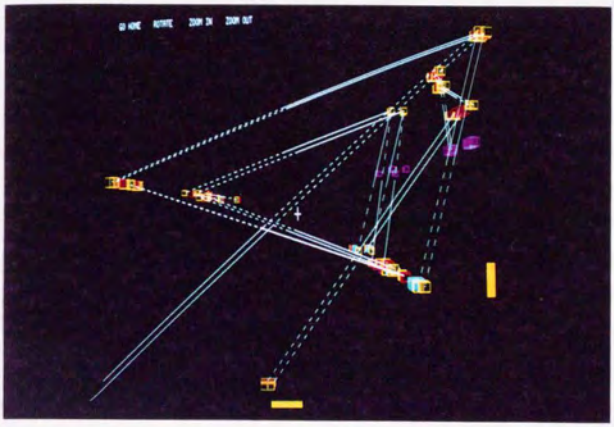


Figure 5.4: 3 次元立体視画像

Figure 5.5: 液晶シャッタ眼鏡

- ポリマス・センサを利用した運動視差による立体視

  ポリマス・センサは磁気を利用し、空間内における位置と姿勢の6自由度を計測できるが、その欠点として周囲の磁界の影響を大きく受けることがあげられる。特に金属が付近にあると、得られるデータは微妙に振動し安定しない。VOGUE ではこのポリマス・センサのノイズによる振動を逆に利用し、3次元グラフを微妙に振動させることで運動視差を用いた立体視機能を提供する。具体的には、普通 VPL 社製 Data Glove や頭部搭載型ディスプレイに装着されるポリマス・センサをグラフィック・ディスプレイ上部に装着し、センサから得られる姿勢情報を Grapher プロセスにフィードバックすることで、そのノイズをグラフの中心回りの縦揺れと横揺れに変換して出力する。結果として3次元グラフはかすかに振動し、これによりユーザは運動視差を用いた立体感を得ることが可能である。

  この運動視差を利用した立体感の表現に関しては、SemNet において縦揺れを利用する方法が紹介されている。実験をつづけるに従い液晶シャッタ眼鏡を用いる方法より効果が大きいことが判明した。ただし問題点としては、表示されるオブジェクト数が多くなるとグラフィックスの画面更新に要する時間が増大し、画面の反応性が悪くなる点である。よって、表示されるオブジェクト数が少ない時に限って有効な方法である。表示オブジェクト数と画面更新速度の関係は後述する。

この他にも、Starbase Graphics Library の機能を利用して、透視図法や光源モデルを用いて遠近感を高めている。

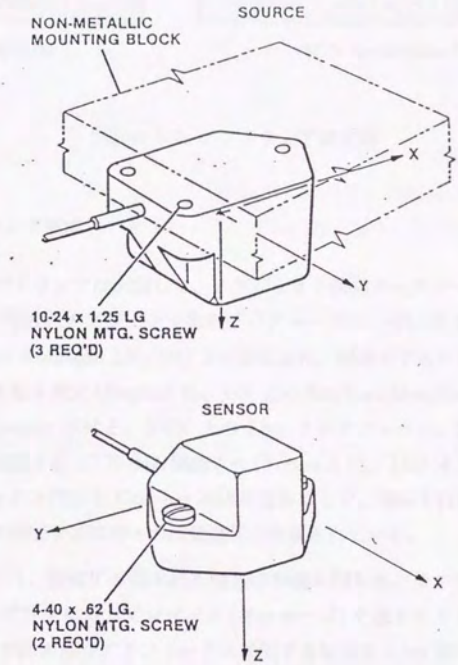


Figure 5.6: ポリマス・センサ

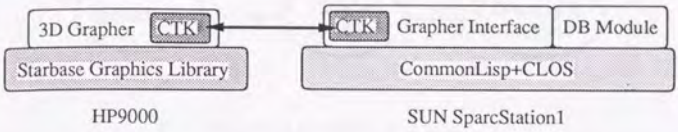


Figure 5.7: ソフトウェア構成図

5.3.2 ソフトウェア構成

VOGUE のソフトウェアは大別して、オブジェクト指向データベースと 3 次元 Grapher から構成される。オブジェクト指向データベースは、CLOS を載せて拡張した SUN 上の Allegro Common Lisp [35] 上に実現され、対象とするソフトウェアのモデル化を行なう。また 3 次元 Grapher は、HP 上の Starbase Graphics Library を用いて記述された Grapher 本体と、SUN 上の Lisp インタフェース、およびこれらをプロセス間通信で接続する CTK から構成され (Figure 5.7)、Lisp インタフェースの保持するグラフィックス情報を Grapher 本体に送ることで、描画を行なう。さらに、この基本的なグラフ機能を基に種々の拡張機能が実現されている。

HP 側での操作は、後述する基本的な視点の移動に限られ、その他の操作は全て、Lisp 側の概念的グラフに対してコマンド (メッセージ) を送ることで行なわれる。つまり、Grapher 本体が持つグラフィックスに関する情報を Lisp 側で管理していることになるが、こうした実現方式を採った理由として以下のようなものがある。

1. 3 次元的視点の連続的変更を行なうためには、Grapher 本体が 3 次元モデルを保持する必要がある。

2. VOGUE はそのフロント・エンドとして Common Lisp インタープリタを使用することで、対話的なグラフィックスの作成・編集を実現しているが、その際に編集の対象とするグラフ、ノード、リンクが CLOS のオブジェクトとして存在する方が、概念的に理解しやすいことが上げられる。

3. 基本的なグラフィックス関連コマンドを全て Lisp に埋め込むことによって、ユーザは一連の処理を Lisp Program の形式で記述することができる。

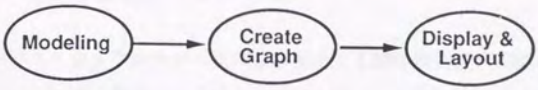


Figure 5.8: 視覚化プロセス

4. 後述するように視覚化ツールの実用化にはグラフィックス・ワークステーションの性能が大きく関係し、しかもグラフィックス・ワークステーションの発達は日進月歩である。よって、ハードウェアに依存する Grapher 本体の機能を必要以上に高めることは移植性を低下させることとなる。これに関しては第8章の VOGUE-AR のところで述べられる。

5. VOGUE は実験システムであり図形削減手法等の機能をインクリメンタルに実現していくにあたり、インタープリタ形式の Lisp での開発が有効であるためである。

VOGUE を利用した視覚化は以下のようなプロセスに従って行なわれる。

1. 対象とするソフトウェアのモデル化
2. グラフの作成とインスタンスの登録
3. ディスプレイ上への描画とレイアウトの検討

次節以降において各モジュールについて説明するとともに、上で述べた視覚化プロセスを例を挙げながら説明する。

5.4 オブジェクト指向データベース

対象とするソフトウェアのモデル化および型による図形数制御を実現するために、VOGUE はオブジェクト指向データベース機能を持つ。これには Lisp 上のオブジェクト指向機能拡張案として現在検討が進められている CLOS (Common Lisp Object System)[1]を拡張することで実現している。CLOS はクラスの概念、継承の概念といった一般のオブジェクト指向言語の性質に加え、多重継承、デーモン・メソッド、メソッド組み合わせ等の機能を有する。

VOGUE がデータベースとしてオブジェクト指向モデルを採用した理由は以下にまとめられる。

- VOGUE ではソフトウェアをノードとリンクのグラフで表現する。その際、各データを独立したオブジェクトとして扱うオブジェクト指向モデルとの整合性が良い。
- 型による情報削減を実現するにあたりクラスによって型を陽に記述できる。
- 将来的な拡張として、マルチメディアへの対応を考慮した場合、オブジェクト指向モデルが良い。

5.4.1 オブジェクト指向データベースの定義

従来データベースとしてはリレーショナル・データベース [25] が一般的であったが、近年マルチメディアへの整合性の良さ等からオブジェクト指向データベースが注目を集めている。

このオブジェクト指向データベースについてはまだ厳密な定義はなされていないが、1990 年のオブジェクト指向データベース宣言 [10] によればオブジェクト指向データベースが満たすべき条件として Tabel 5.2に示す項目が挙げられている。本データベースのようにオブジェクト指向言語を拡張して実現されたデータベースの場合、これらの条件の幾つかは自然に満たされるが、これらの定義自体に対するコンセンサスも得られていない状態でもあり、本論文では視覚化にあたり必要と思われる機能の実現にとどめた。

[1]正確には CLOS の仕様検討案として Xerox から配布されている PCL (Portable Common Loops) を使用しているが、本論文ではこれを CLOS と称する

必須条件	複合オブジェクト
	オブジェクト識別性
	カプセル化
	型 / クラス
	継承
	遅延束縛と一体化した再定義
	拡張可能性
	計算の完全性
	永続性
	2 次記憶管理
	並行処理制御
	障害回復
	アドホックな問い合わせ処理機構
付加条件	多重継承
	型検査と型推論
	分散
	設計トランザクション
	バージョン
選択条件	プログラミングパラダイム
	表現システム
	型システム
	統一性
必須または付加条件 (未定)	ビュー定義と導出データ
	データベース管理ユーティリティ
	完全性制約
	スキーマ進化機能

Table 5.2: オブジェクト指向データベースシステム宣言の各条件

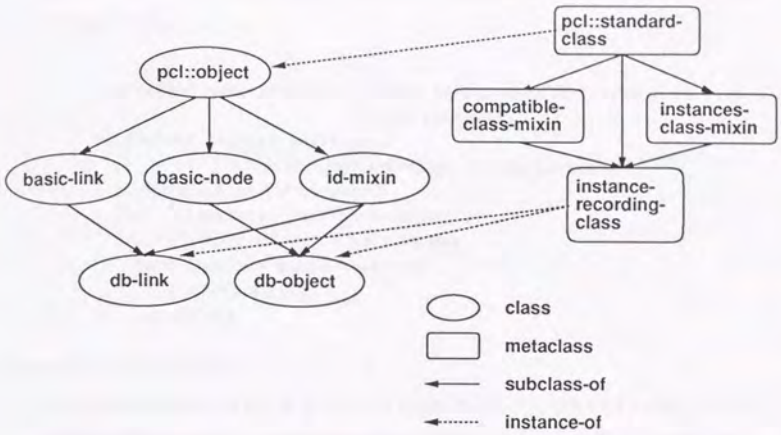


Figure 5.9: データベースを実現するクラス

5.4.2 データベースのクラス構成

Figure 5.9は、データベースを実現するにあたり作成されたクラスである。pcl::object と pcl::standard-class は CLOS(PCL) の組み込みクラスであり、それ以外はデータベース実現のために作成されたクラスである。各クラスに関して説明する。

pcl::object

CLOS(PCL) の組み込みクラスで、defclass によってクラス生成する際にスーパークラスを指定しない場合、このクラスのサブクラスとなる。

pcl::standard-class

pcl::object のメタクラスであり、インスタンス生成の make-instance は実際にはここで定義されている。

instances-class-mixin

pcl::standard-class のサブクラスとして定義され、スロットとしてインスタンス・テーブルを持つ。普通は pcl::standard-class で処理される make-instance を再定義し、メソッド組み合わせを用いてインスタンス生成の際に、そのインスタン

スをテーブルに格納する処理を行なう。make-instance の定義は以下のように行なっている。

```
(defmethod make-instance ((class instances-class-mixin)
                          &rest initargs)
  (declare (ignore initargs))
  ;; スーパークラス pcl::standard-class の make-instance を
  ;; 呼び出しインスタンスを生成する
  (let ((instance (call-next-method)))
    ;; クラス・テーブルにインスタンスを登録
    (add-instance class instance)
    ;; インスタンスを返す
    instance))
```

compatible-class-mixin

pcl::standard-class のサブクラスとして定義される。CLOS(PCL) ではメッセージ送信の際にメタクラスの整合性を調べる機構があり、この整合性を確保するための mixin クラスである。これは以下のように実現されている。

```
(defmethod pcl::check-super-metaclass-compatibility
  ((class compatible-class-mixin) (super pcl::standard-class)) t)
```

instance-recording-class

これら 2 つの mixin および pcl::standard-class のサブクラスとして実現されたクラスである。このクラスをメタクラスとして定義されるクラスは、インスタンス・テーブルを持ち、生成されるインスタンスは全てこのテーブルに登録される。

basic-node

ノードとしての基本的な機能を実現したクラスで、スロットとしては links スロットだけを持ち、接続されたリンクをリスト形式で保持する。後述する graph-node、db-object のスーパークラスとなる。リンクへのアクセサとして以下のものが定義されている。

- (links *node*)
  - *node* の持つ全リンクを返す
- (output-links *node*)

- *node* の持つ出力リンクを返す

- (input-links *node*)
  - *node* の持つ入力リンクを返す

basic-link

リンクとしての基本的な機能を実現したクラスで、スロットとしては source スロットと destination スロットを持ち、各々始点ノードと終点ノードを格納する。後述する graph-link、db-link のスーパークラスとなる。各スロットへのアクセサとしては以下のものが定義されている。

- (link-source *link*)
  - *link* の始点ノードを返す
- (link-destination *link*)
  - *link* の終点ノードを返す

id-mixin

インスタンス生成の際にシステムにユニークな識別子を生成し、インスタンスをこのシンボルに束縛するための mixin クラスである。インスタンスの識別子へのアクセサは以下のものがある。

- (id *object*)
  - *object* の識別子を返す

db-object

クラス db-object は、basic-node および id-mixin のサブクラスであり、かつ instance-recording-class をメタクラスに持つ。後述する define-class でクラス生成する際、特にスーパークラスを指定しない場合このクラスのサブクラスとなる。つまりデータベース内にユーザが定義したクラスは db-object をルートとする階層構造をなす。

db-link

クラス db-link は、basic-link および id-mixin のサブクラスであり、かつ instance-recording-class をメタクラスに持つ。後述する define-link でクラス生成する際、スーパークラスを指定しない場合このクラスのサブクラスとなる。ユーザが定義したリンクは、db-link をルートとする階層構造をなす。

以上のクラスによって実現されるデータベースは、基本的に CLOS の機能を全て継承し、メソッド定義、スロット・アクセス等はそれに従う。データ型として複素数型以外の数型、文字型、シンボル型、リスト型とコンス型、文字列型を扱うことができる。次節以降では各拡張点について説明する。

5.4.3 クラス定義

CLOS ではクラス定義を defclass で行なうが、本データベースではこれを拡張した define-class を用いる。define-class はメタクラスを instance-recording-class に設定し、かつ新たに作成されるクラスの上位クラスが指定されない場合には、db-object のサブクラスとする。上位クラス、スロット指定子等のシンタックスは defclass に準じるが、メタクラスの指定はできない。実際には define-class は以下の形式である。

- (define-class *class-name superclass-list slot-specifiers-list*)
  - *class-name* はシンボルである。*superclass-list* はスーパークラス名のリストで *slot-specifiers-list* はスロット指定子のリストである。これらに関する詳細は文献 [50] に譲る。

例えば、クラス flavor-method、及びそのサブクラスとして flavor-after-method、flavor-primary-method、flavor-before-method というクラスを定義するには以下のようにする。

```
(define-class flavor-method ()
   ((name :initarg :name :initform nil :accessor method-name)) )
(define-class flavor-primary-method (flavor-method) ())
(define-class flavor-before-method (flavor-method) ())
(define-class flavor-after-method (flavor-method) ())
```

これによりデータベース内には Figure 5.10のようなクラス階層ができる。

5.4.4 インスタンス生成とインスタンスへの識別子の付加

インスタンスの生成は CLOS 同様 make-instance によってなされるが、上述のとおり作成されたインスタンスは以後各クラスによって管理される。インスタンスへのアクセス方法については後述する。

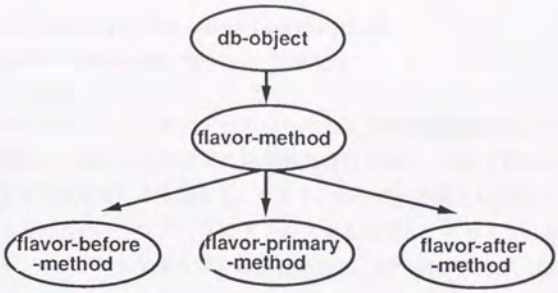


Figure 5.10: データベース内のクラス階層

作成されたインスタンスにはシステムにユニークな識別子が付加され、この識別子と同じシンボルに束縛される。例えば上で定義したクラス flavor-primary-method のインスタンスを生成するには、

```
> (make-instance 'flavor-primary-mehtod :name "display")
#<FLAVOR-PRIMARY-METHOD :ID DB402>
```

とする。これにより name スロットに "display" を持つ flavor-primary-method のインスタンスが生成された。返り値に識別子が付加されているが、以後このインスタンスにアクセスするにはこのシンボルを評価する。

```
> db402
#<FLAVOR-PRIMARY-METHOD :ID DB402>
```

5.4.5 インスタンスのリストアップ

あるクラスの全インスタンスへのアクセスには以下の形式の all-instances を用いる。

- (all-instances *class-name*)
  - *class-name* で指定されたクラスのインスタンスをリストとして返す

上で定義した flavor-primary-method の場合、

```
> (all-instances 'flavor-primary-method)
(#<FLAVOR-PRIMARY-METHOD :ID DB402>)
```

さてここで、オブジェクト指向データベースのように階層構造によってデータの管理を行なう場合、上位のクラスはより抽象的な概念を表し、逆に下位のクラスはより具象的な概念を表現していると言える。つまり、ユーザの要求として考えられるのは、個々のクラスの独自のインスタンスにアクセスする場合と、そのクラスを含め下位クラス全部のインスタンスにアクセスする場合である。先の例で言えば、flavor-method のインスタンスだけにアクセスしたい場合と、 flavor-method 以下のクラス全部のインスタンスにアクセスしたい場合が考えられる。例えば以下のように “display” という after メソッドがあるとする。

```
> (make-instance 'flavor-after-method :name "display")
#<FLAVOR-AFTER-METHOD :ID DB403>
```

この時、ユーザがクラス flavor-method のインスタンス全部を得たいとしても、先に述べた all-instances はクラス flavor-method の固有のインスタンスだけをリストアップする。つまり、以下の式を評価しても返される値は NIL である。

```
> (all-instances 'flavor-method)
NIL
```

よってオブジェクト指向データベースの場合、ある指定したクラスのサブクラス全部のインスタンスまでリストアップする機能が必要となる。これを行なうのが all-instances* である。

- (all-instances* *class-name*)
  - *class-name* で指定されるクラスとそのサブクラスのインスタンスをリスト形式で返す

上の例で言えば、

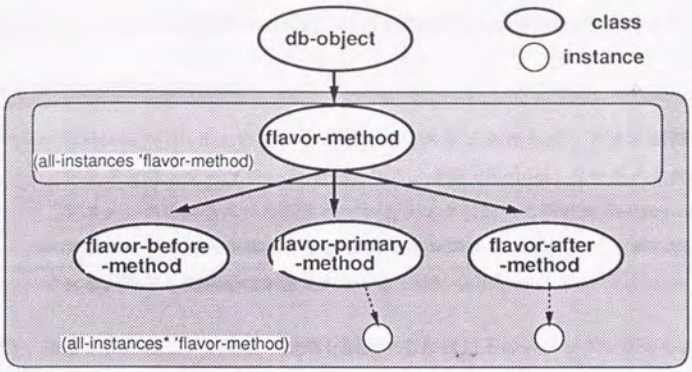


Figure 5.11: インスタンスのリストアップ

```
> (all-instances* 'flavor-method)
(#<FLAVOR-PRIMARY-METHOD :ID DB402> #<FLAVOR-AFTER-METHOD :ID DB403>)
```

となる。これらの概念を Figure 5.11に示す。

5.4.6 インスタンスの消去

作成したインスタンスを消去するには、以下の形式の destroy-object を用いる。

- (destroy-object *instance*)
  - *instance* が db-link のサブクラスのインスタンスならば、クラスが持つインスタンス・テーブルから消去される。また db-object のサブクラスのインスタンスの場合には接続されているリンクに対して同様に destroy-object メッセージを送った後、クラスが持つインスタンス・テーブルから消去される。ただし識別子は解放されない。

また、あるクラスのインスタンス全部を消去するには以下のメソッドを用いる。

- (delete-instances *class-name*)
  - 指定したクラスの全インスタンスを消去する
- (delete-instances* *class-name*)
  - 指定したクラスとそのサブクラスの全インスタンスを消去する

つまり

```
> (delete-instances* 'db-object)
```

は db-object のサブクラスの全インスタンスの消去を意味する。

5.4.7 リンク作成

本データベースではインスタンス間の関係を記述することができ、これをリンクと称する。このリンクは内部的には CLOS のクラスとして実現されているので、やはり階層的に管理する事が可能である。リンクの定義には defclass を拡張した以下の形式の define-link 、また実際のリンクの生成には connect を用いる。

- (define-link *link-name superclass-list slot-specifiers-list*)
  - 上位リンク、スロット指定子等の構文は defclass に準じ、作成されたリンク・クラスは組み込みクラス db-link のサブクラスとなる。
- (connect *instance1 instance2 link-name*)
  - 始点を *instance1*、終点を *instance2* とする *link-name* のインスタンスを生成する

例えば、has-same-name-link というリンクを作成するには以下のように行なう。

```
> (define-link has-same-name-link () ())
```

これで db-link のサブクラスに has-same-name-link が作られる。さらに db402 と db403 に束縛されているインスタンス同士を has-same-name-link で接続するには、

```
> (connect db402 db403 'has-same-name-link)
```

とする。データベース内では Figure 5.12のようなリンクができている。

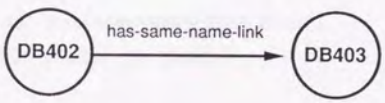


Figure 5.12: リンクで結合されたインスタンス

5.4.8 インスタンスの検索

ある条件に従いインスタンスを検索するには以下の形式のメソッド solve を用いる。

- (solve *obj1 relation obj2 &optional class-name*)
  - *obj1* および obj2 は、インスタンス、スロット値、パターン変数のいずれかで、*relation* はスロット名かリンク名、*class-name* は検索するクラス階層を指定する。

solve の特徴はインスタンスの属性値とリンク先との扱いを同等に考えている点である。これまでの flavor-method の例で言えば、シンボル db402、db403 に束縛されているインスタンスはそれぞれ名前 "display" を内部スロットとして持ち、これらは has-same-name-link で結合されている。今、db402 の name スロットを調べるには以下のように行なう。

```
> (solve db402 'name '(? x))
("display")
```

また db402 と has-same-name-link で結合されているオブジェクトを調べるには以下のようにする。

```
> (solve db402 'has-same-name-link '(? x))
(#<FLAVOR-AFTER-METHOD :ID DB403>)
```

逆に db403 と has-same-name-link で結合されているオブジェクトを調べるには以下のようにパターン変数を逆に用いる。

```
> (solve '(? x) 'has-same-name-link db403)
(#<FLAVOR-AFTER-METHOD :ID DB402>)
```

同様に次のような使用法も可能である。

```
> (solve '(? x) 'name "display")
(#<FLAVOR-AFTER-METHOD :ID DB402> #<FLAVOR-AFTER-METHOD :ID DB403>)
```

検索すべき対象クラスが明らかな場合にはそのクラス名を指定することもできる。

```
> (solve '(? x) 'name "display" 'flavor-before-method)
NIL
```

この場合、指定された flavor-before-method クラスには "display" という name スロットを持ったインスタンスが存在しないことがわかる。all-instances の場合と同様に、指定したクラスのサブクラスのインスタンスまで考慮する必要がある。その場合には solve* を用いるがシンタックスは solve と同じである。

5.4.9 ファイル入出力

ファイルへの入出力には各々以下の形式の save-db および load-db を用いる。

- (save-db *filename*)
  - *filename* で指定されたファイルに全インスタンスのスロットデータを書き込む。
- (load-db *filename*)
  - *filename* で指定されたファイルからデータを読み込みインスタンスを生成する。

複合オブジェクトの取り扱い

オブジェクト指向データベースにおいて考慮されなければならない点の 1 つに複合オブジェクト (Figure 5.13) の取り扱いがある。複合オブジェクトとは、あるインスタンスの内部スロットが別なインスタンスになっている場合である。特に問題となるの

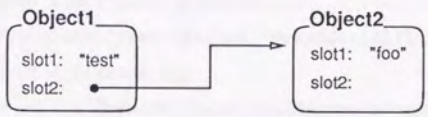


Figure 5.13: 複合オブジェクト

は、この参照関係がサイクリックになっている場合である (Figure 5.14)。本データベー

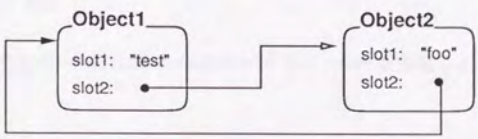


Figure 5.14: 複合オブジェクト (サイクリックな場合)

スでは、これを以下のように実現している。各インスタンスのスロットに関するデータは、数型等プリミティブな型の場合にはそのままファイルに書き込むが、ユーザ定義クラスの場合には後述するメソッド db-slots によってインスタンス・スロットをリストアップし、順番にファイルに書き込む操作を行なう。save-db が呼ばれると、空のインスタンス表を作成し、1度ファイルに書き込んだインスタンスは表に記録する。そして1度出現したインスタンスが他のオブジェクトのスロットとして再び出現すると、このインスタンスの識別子だけをファイルに書き込む操作を行なう。load-db も同様に読み込んだインスタンスを表に記録し、識別子を読み込むとこれを評価し、その識別子に束縛されているオブジェクトをスロットに格納する。

書き込むスロットの指定

基本的にクラスで定義されたスロットは全てファイルに書き込むが、クラスによってはスロットを記録したくないものも存在する。例えば、後述する graph-node の場合、書き込んだデータから再び図形として描画する時スロット graphic-id は NIL でなければならない。このような場合には、以下のメソッド db-slots を再定義する。

```
(defmethod db-slots ((node graph-node))
  (let ((slots (call-next-method))
        (omit-slots (list 'figure 'graphic-id)))
    (remove-if #'(lambda (x)
                   (member (pcl::slotd-name x) omit-slots))
               slots)))
```

3行目の omit-slots の部分に、書き込みたくないスロット名をリスト形式で記述することにより、これらのスロットは書き込みの際に無視される。

5.4.10 組込みクラス

データベースには db-object,db-link 以外にクラス color を用意してある。

- color
  色情報を持つクラスで (make-color R 値 G 値 B 値) で生成される。また、シンボル red, blue, green, yellow, magenta, cyan, gray80 等にはこれらの色に対応する color のインスタンスがあらかじめ束縛されている。

5.5 3次元 Grapher

3次元 Grapher は、ノードとリンクによって構成される3次元グラフの描画、グラフィックス属性変更といった基本機能を有し、Grapher 本体、Lisp インタフェースから構成され、CTK [67] というプロセス間通信ツールキットを用いてメッセージ交換を行なう。3次元グラフの作成は、Lisp 上に作成された概念的なグラフに対して Lisp 関数の形で描画命令を送ることによって実現され、ディスプレイ上で図形を直接操作することによりグラフィクスの変更を行なうことはできない。これは設計指針でも述べたように、現段階で3次元入力インタフェースの実際的利用は時期尚早と判断したためである。以下の各節において、各々のモジュールとその機能について解説する。

5.5.1 Grapher 本体

処理フロー

Grapher 本体は HP 上に、ANSI 仮想装置コンピュータ・グラフィックス・インタフェース (VD/CGI) 標準の拡張版である Starbase Graphics Library(以後 Starbase と略記) を用いて C で記述されている。このプログラムは無限ループ状態で、Lisp インタフェースからの Table 5.3に示すようなメッセージの到着と、ディスプレイ上におけるマウス・イベントの監視を行なっている。メッセージの到着があった場合には解釈・実行し、マウス・イベントが起こった場合には対応する処理を行なう。

	メッセージ文字列	処理内容
ノード関連	AddData	新しいノードの生成
	DeleteData	ノードをデリートする
	DispData	ノードを表示する
	HideData	ノードを消去する
	MoveData	ノードを移動する
	ChangeColor	ノードの色を変更する
	ChangeShape	ノードの形を変更する
	ChangeScale	ノードの大きさを変更する
	ChangeTransparency	ノードの透明度を変更する
	ChangeLabelColor	ノードのラベルを変更する
	ChangeLabelSize	ノードのラベルの大きさを変更する
	DispLabel	ノードのラベルを表示する
	HideLabel	ノードのラベルを消去する
リンク関連	AddLink	新しいリンクの生成
	DeleteLink	リンクをデリートする
	DispLink	リンクを表示する
	HideLink	リンクを消去する
	ChangeLinkColor	リンクの色を変更する
視点関連	MoveCameraAbsolutely	カメラ位置を絶対座標で指定する
	MoveCameraRelative	カメラ位置を相対座標で指定する
	GetCameraPosition	カメラ位置を得る
	GoHome	ホームポジションに戻る
	RotView	図形を参照位置回りに回転する
その他	Reset	セグメントを初期状態にする
	RefreshScreen	画面の更新を行なう
	PolhemusSwitch	ポリマス・モードにする

Table 5.3: Grapher 本体が受けるメッセージと処理内容

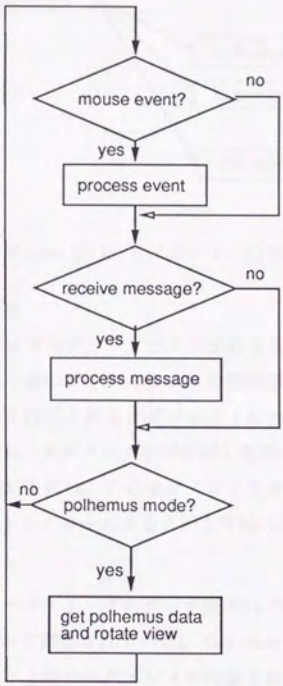


Figure 5.15: Grapher のフローチャート

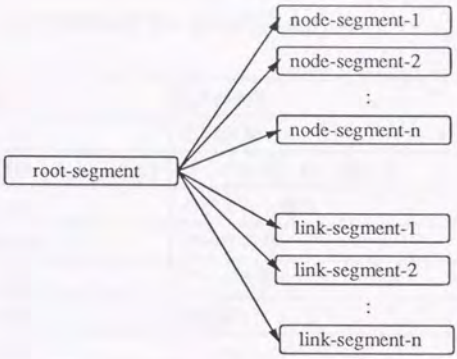


Figure 5.16: セグメントの概念

ノード・リンクのデータ構造

Starbase は多くのプリミティブなグラフィックス関数を提供するが、このプリミティブ関数を組み合わせたある 1 連のグラフィックス処理手順をセグメントとして登録することが可能である。新しく作成されるオブジェクトはルート・オブジェクトの子としてその描画手続きを登録し、セグメントを呼び出した時にこの手続きが実行される。グラフィックスに変更を加えるには、このセグメントをオープンし、対応する部分の手続きを変更し、セグメントをクローズするという手順が踏まれる。これらは全て Starbase の機能である。

Grapher 本体におけるノード・リンクのデータ構造は Table 5.4,Table 5.5に示す形式であり、セグメントを用いて記述されている。Grapher 本体はメッセージ AddData を受けるとノードを作成し、上述のセグメントが作成される。作成されたセグメントは 1 次元配列のテーブルに登録され、そのインデックスを Lisp プロセスにメッセージとして送り返す。同様に AddLink メッセージを受けるとリンクを作成し、セグメントが作成される。作成されたセグメントはリンクテーブルに登録され、Lisp にインデックスを返す。他のグラフィックス属性変更メッセージは、メッセージの第 1 引数にこのインデックスを持ち、各々のテーブルから対応するセグメントを検索し、書き換えを行なう。

リンクは始点ノードと終点ノードを認識するために、その中点から終点ノード側を

破線で表現する。この他に矢印や細い円錐等で表現する方法が考えられるが、これらはかえってグラフ全体の視認性を低めるからである。

x,y,z	x,y,z 座標
r,g,b	R,G,B 値
width,height,length	ノードの縦、横、奥行き
segno	セグメント番号
dtnum	データ番号
stype	ノードの形
trans	透明度
label	ラベル
in_link_num	入力リンク数
in_link	入力リンク
out_link_num	出力リンク数
out_link	出力リンク
visible	表示されているかどうかのフラグ

Table 5.4: Grapher 本体内でのノードのデータ構造

r,g,b	R,G,B 値
segno	セグメント番号
lknum	データ番号
ltype	ノードの形
src	始点ノード
dst	終点ノード
visible	表示されているかどうかのフラグ

Table 5.5: Grapher 本体内でのリンクのデータ構造

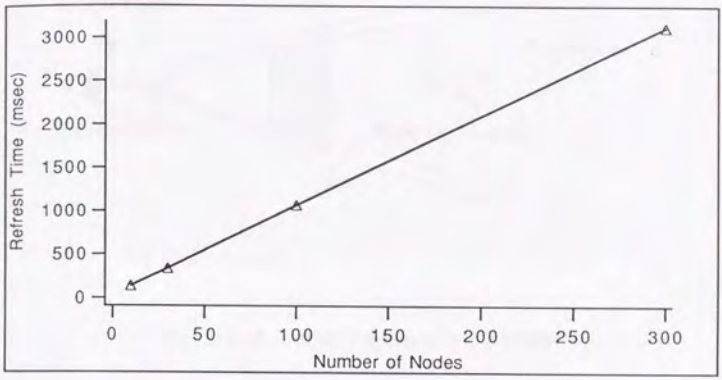


Figure 5.17: 表示ノード数と画面更新時間の関係

画面更新のタイミング

セグメントが更新されただけでは表示画面は変化せず、各セグメントに対してセグメント呼び出しがあって初めて表示画面は更新される。この更新速度はセグメント数に比例し増加するため、多数のセグメントが登録されている場合には、この更新速度が画面の反応性に重大な影響を及ぼす。Figure 5.17は表示されるノード数とその時の画面更新に要する時間の関係である。画面更新の時間はほぼノード数に比例することがわかる。例えばアニメーションによって図形の時事刻々の変化を見たい時には、セグメントが書き換えられる毎にセグメントの更新をする必要があるが、他の場合にはセグメントの書き換えを終了してから更新をしても何ら問題はない。従って、セグメント更新のタイミングはユーザに委ねることとし、Grapher 本体としては、スライダー等による視点変更時を除いて自動的に更新を行なうことはしない。

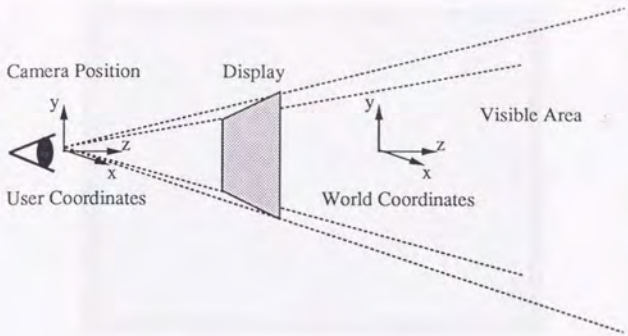


Figure 5.18: 3次元 Grapher における座標系

3次元座標系と視点の制御

3次元 Grapher の座標系を Figure 5.18に示す。初期状態ではディスプレイに対し水平右方向に x 軸、垂直上方向に y 軸、奥行き方向に z 軸をとっている。3次元空間のクリッピング処理は行なわない。

3次元グラフィックスにおける主たる問題点の1つは視点の制御である。視点に関連する変数としては、以下の2つを考慮しなければならない。

- カメラ位置 (cx, cy, cz)
  ユーザの視点をカメラに例えた時の座標。
- 参照位置 (rx, ry, rz)
  上の仮想カメラが参照している座標。初期状態では世界座標系の原点にある。

この2つの座標の変更によって視点の移動が行なわれ、この操作の正確さ、容易さは3次元ツールのインタフェースに大きく影響するが、現在までのところ決定的なデバイスはない。従って、視点制御用メッセージとしては、カメラ位置の絶対移動、相対移動、参照位置を中心とした回転、カメラ位置の獲得だけに絞られている。この視点の移動はよりリアルタイム性が要求されるため、ディスプレイ上でマウスによる操作を実現している。これに関しては後述する。

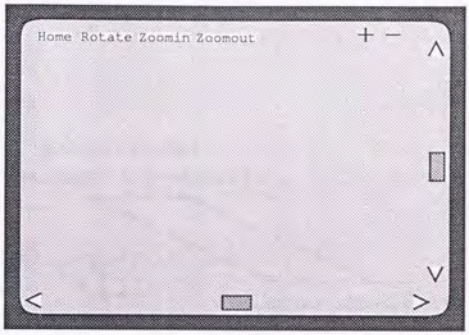


Figure 5.19: ディスプレイ上のボタン

5.5.2 ディスプレイ上でのユーザ・インタフェース

設計指針でも述べたとおりVOGUEでは3次元図形を直接に操作する機能は提供せず、基本的なグラフィックス操作は、ユーザによって全てLispへのコマンドとして入力されるが、視点の変更やノードの選択といった、特にリアルタイム性を要求される機能に関してだけは、Grapher本体側での操作で実現できることが必要である。これらはHP側のマウスによって操作され、Grapher本体プロセスはこのためにマウス・イベントを監視している。

視点の移動

HPのディスプレイはFigure 5.19のようになっていて、幾つかのボタンによる基本的な視点変更が可能である。

- 並進移動
  他の2軸に関する変数を固定し、残りの各1軸に関する変数を同じ値だけ増減する。例えばx軸方向へのd移動は、$(cx+d, cy, cz)$,$(rx+d, ry, rz)$として表される(Figure 5.20)。これは画面上の6個の並進移動ボタンによって行なわれる。

- 図形の回転
  参照位置を固定し、カメラ位置をその回りに任意の角度だけ回転させる。ディ

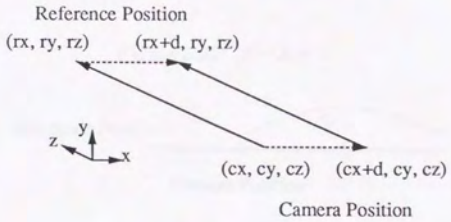


Figure 5.20: 並進移動

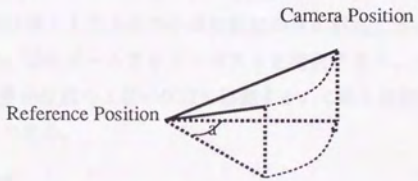


Figure 5.21: 回転

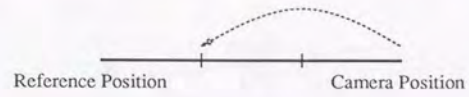


Figure 5.22: ズームイン

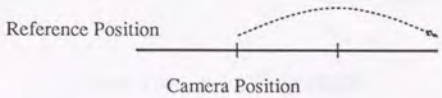


Figure 5.23: ズームアウト

スプレイ下方と右方に各々スライダー・ボタンがあり、下方のスライダー・ボタンをスライドすると画面上の図形が、参照点を中心として横方向に回転する。これはカメラ位置をz座標はそのままで、参照点回りに回転させることで実現されている。同様に右方のスライダー・ボタンをスライドすると、図形は参照点を中心として縦方向に回転する。

- ズームイン・ズームアウト
  ノードを選択すると、参照位置はノードの中心に自動的にセットされる。この状態で画面上部のズームイン・ボタンを選択すると、カメラ位置を現在のカメラ位置と参照位置の3等分点の参照位置側の点に移動する。ただし、参照位置は変化しない。逆にズームアウト・ボタンを選択すると、カメラ位置を現在のカメラ位置と参照位置の3倍の位置に移動する。これらは画面左上部のメニュー・ボタンで実行できる。

- 初期位置へ戻る
  SemNetでも指摘されているが、3次元空間での視点移動は自分の位置を見失い易い。このような時のために初期位置へ戻る機能がある。これもメニュー・ボタンで実現できる。

ここで実現した視点の変更方法は、2次元ツールにおけるインタフェースの延長上にある技術であり、3次元空間におけるインタフェースとしては、必ずしも有効では

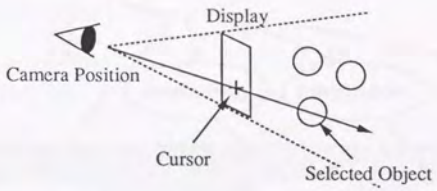


Figure 5.24: ノード選択の概念図

ないが、本論文で行なった視覚化に関する限り、許容できる範囲である。この視点の変更のためのデバイスに関する議論は第8章の考察で行なうこととする。

ノードの選択

HP のディスプレイ上でマウスの左ボタンをクリックすると、Starbase のプリミティブ関数によってマウスの2次元座標は適当な3次元ベクトルに変換されて、そのベクトル上において視点からみて最初に出現したノードが選択される (Figure 5.24)。ノードが選択されると、選択されたノードのエッジの色が変わり、選択されたことが認識できるようになっている。また Lisp 上の Grapher インタフェースに、選択されたノードの ID とともに (LEFTBUTTONPRESSED ID) というメッセージを送る。右ボタンが押された場合には、ノードのエッジの色は変化しないが、(RIGHTBUTTONPRESSED ID) というメッセージを送る。これらのメッセージは、Lisp の Grapher インタフェースで処理のカスタマイズに利用される。

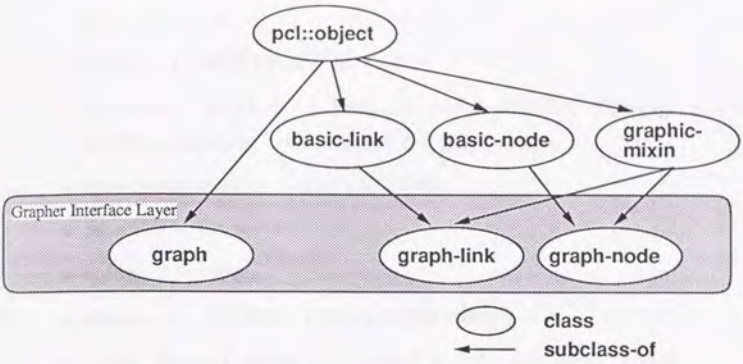


Figure 5.25: Grapher インタフェースのクラス階層

5.5.3 Grapher インタフェース

Grapher インタフェースは、Lisp 内から HP 上の3次元グラフを対話的に作成・変更するためのモジュールで、CLOS を用いて記述され Figure 5.25に示す各クラスから構成される。描画命令は Lisp の関数あるいは CLOS のメソッドとして記述され、CTK を利用して Grapher 本体に送られる。各クラスについて以下に説明する。

graph-node

graph-node はノードとしての基本機能を持つ basic-node とグラフィックス情報に関するスロットを持つ mixin クラスである graphic-mixin のサブクラスとして実現され、グラフィックスに関する情報とリンク情報を持つ。先に述べたプロセス間通信を実際に実現しているクラスであり、graph-node へのメッセージは CTK を介して Grapher 本体に送られる。主なスロットには以下のようなものがある。

- figure: ノードとして表現しているデータベース内のインスタンス。後述する add-object により自動的に埋められる。
- graphic-id: Grapher 本体プロセス内のノードのインデックス。ノードが display メッセージを受けると3次元 Grapher プロセスに各スロット情報

をソケットを通じてメッセージの形式で送った時 Grapher から返される値。

- links: リンク情報を保持する。
- focused-p: 現在このノードが focus されているかどうかの情報。NIL の場合には display メッセージを受けても表示されない。
- xpos,ypos,zpos: ノードの 3 次元座標。
- label: ノードのラベル。
- color: ノードの色。
- shape: ノードの形状。1 から 9 の整数で指定する。
- width,height,length: ノードの大きさ。デフォルトは 0.2 である。
- transparency: ノードの透明度。デフォルトは 3 である。
- displayed-p: 表示されているか否かのフラグ。

graph-node に実現されているメソッドには以下のようなものがある。

- (display *node &key with-link link-destination with-label*)
  - インスタンスに対して初めて display メッセージが送られると、CTK のプロセス間通信機能を用いて Grapher 本体に AddData というメッセージと共にグラフィックス情報を送る。Grapher 本体はノードを生成し、セグメントを作成するとセグメントのインデックスをやはり CTK によって送り返す。この値は graphic-id スロットに保持され、他のグラフィックス関数がインスタンスに送られる時に、このインデックスと引数をメッセージにして Grapher 本体へ送る。2 度目からの display メッセージは displayed-p スロットを見て、NIL の場合にだけ、Grapher 本体に対して DispData メッセージを送り、ノードを表示する。キーワード引数は以下のとおり。
    - *with-link* - 接続されているリンクを同時に表示する指定。デフォルトは NIL。
    - *link-destination* - *with-link* 指定時のリンクの方向。
    - *with-label* - ラベルを同時に表示する指定。デフォルトは T。
- (erase *node &key with-link link-destination*)

- erase メッセージが送られると displayed-p スロットを見て、T の場合にだけ 3 次元 Grapher に対して HideData メッセージを送りノードを消去する。キーワード引数は display と同じ。

- (move *node x y z*)
  - ノードを移動するには引数として 3 次元座標とともに move を用いる。move が送られると Grapher 本体に対し MoveData というメッセージを送る。
- (move-by *node &key x y z*)
  - ノードを指定したキーワード引数の方向に相対移動する。
- (change-attribute *node &key xpos ypos zpos color shape transparency width height length*)
  - *node* のグラフィックス属性を変更する。各キーワード引数の指定方法は graph-node のスロットと同じである。キーワード引数の指定によって、グラフィックス属性変更の各メッセージに展開されて Grapher 本体へ送られる。
- (display-label *node*)
  - *node* のラベルを表示する。
- (erase-label *node*)
  - *node* のラベルを消去する。

graph-link

graph-link はリンクとしての基本機能を持つ basic-link と graphic-mixin のサブクラスとして実現され、プロセス間通信を実現している。主なスロットは以下のとおりである。

- figure: リンクとして表現しているデータベース内のインスタンス。
- graphic-id: Grapher 本体プロセス内におけるリンクのインデックス。
- source: リンクの始点ノード。
- destination: リンクの終点ノード。
- focused-p: 現在このリンクが focus されているかどうかの情報。NIL の場合には display メッセージを受けても表示されない。

- color: リンクの色。

メソッドとしては以下のようなものがある。

- (display *link*)
  - graph-link のインスタンスに対して初めて display メッセージが送られると、CTK を利用して Grapher 本体に AddLink というメッセージを送る。Grapher 本体はリンクを生成し、セグメントを作成するとそのインデックスを送り返す。このインデックスは graphic-id スロットに保持され、以後インスタンスに対して他のグラフィックス操作関数を送ると、このインデックスと引数をメッセージとして Grapher 本体に送る。2度目以降の display メッセージは、graph-node 同様 displayed-p スロットが NIL の場合に限り、Grapher 本体に DispLink メッセージを送る。
- (erase *link*)
  - erase メッセージが送られると displayed-p スロットを見て、T の場合にだけ3次元 Grapher に対して HideLink メッセージを送りノードを消去する。
- (change-attribute *link &key color*)
  - *link* のグラフィックス属性を変更する。ただしリンクの場合は色だけである。

graph

graph は graph-node、graph-link を管理するテーブルを内部スロットに持ち、以下のアクセサによってスロットにアクセスできる。

- (nodes *graph*)
  - *graph* 内の全ノードをリスト形式で返す
- (links *graph*)
  - *graph* 内の全リンクをリスト形式で返す
- (elements *graph*)
  - *graph* 内の全ノードと全リンクをリスト形式で返す

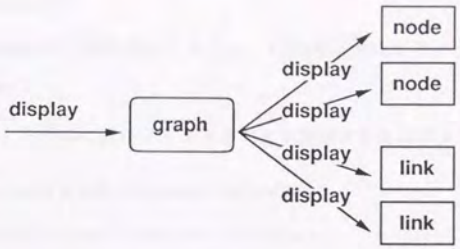


Figure 5.26: グラフ要素へのメッセージの送信

グラフへのノード・リンクの登録には以下のメソッドを用いる。これらの仕様法に関する詳細は次節のグラフの作成例のところで述べられる。

- (add-node *graph node*)
- (add-link *graph link*)
  - *graph* に *node* あるいは *link* を登録する。
- (remove-node *graph node*)
- (remove-link *graph link*)
  - *graph* から *node* あるいは *link* を削除する。

表示関係のメソッドとしては以下のようなものがある。

- (display *graph*)
  - *graph* に display メッセージが送られると、このグラフに登録されている全ノード、リンクへ display メッセージが送られる (Figure 5.26)。
- (erase *graph*)
  - *graph* に erase メッセージが送られると、このグラフに登録されている全ノード、リンクへ erase メッセージが送られる。
- (move-by *graph* &key *x y z*)
  - *graph* に move-by メッセージが送られると、このグラフに登録されている全ノードへ同一引数で move-by メッセージが送られ、結果としてグラフ全体が平行移動を行なう。

- (reset *graph*)
  - *graph* を初期化するとともに、Grapher 本体に Reset メッセージを送る。

以下は指定した条件に従いノード・リンクを検索するためのメソッドである。

- (assoc-node *graph slot-name slot-value*)
- (assoc-nodes *graph slot-name slot-value*)
  - *graph* の全ノード中 *slot-name* として *slot-value* を持つノードの内最初にマッチしたノードが返される。マッチする全ノードを得るには assoc-nodes を用いる。
- (assoc-link *graph slot-name slot-value*)
- (assoc-links *graph slot-name slot-value*)
  - *graph* の全リンク中 *slot-name* として *slot-value* を持つリンクの内最初にマッチしたリンクが返される。マッチする全リンクを得るには assoc-links を用いる。

グラフの各要素に対して同一処理を行なうためには、以下のマクロを用いる。

- (map-nodes (*node-var graph*) . *body*)
  - *graph* 内のノードを順次 *node-var* に束縛し、*body* を評価する。
- (map-links (*link-var graph*) . *body*)
  - *graph* 内のノードを順次 *link-var* に束縛し、*body* を評価する。
- (map-elements (*element-var graph*) . *body*)
  - *graph* 内のノードを順次 *element-var* に束縛し、*body* を評価する。

画面関係のメソッド

その他に画面を制御するために以下のメソッドがある。

- (amove-camera *cx cy cz*)
  - カメラ位置の絶対移動。
- (rmove-camera *cx cy cz*)

- カメラ位置の相対移動。

- (rotate-view *axis degree*)
  - 参照位置を中心とした回転。*axis* はシンボルで x か y を指定する。参照位置まわりに *degree* 度回転する。

- (refresh *hp*)
  - 画面を更新する。*hp* は画面を表す定数である。

- (polhemus-mode)
  - ポリマス・モードの ON/OFF スイッチ

5.5.4 グラフの作成例

グラフを作成するには、まずグラフの生成を以下のように行なう。

```
> (setq g (make-instance 'graph))
```

グラフのインスタンスが 1 個生成される (Figure 5.27)。

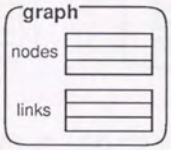


Figure 5.27: グラフのインスタンスの作成

作成されたグラフに対してノードを登録するには、まず graph-node のインスタンスを生成する。

```
> (setq n1 (make-instance 'graph-node))
#<GRAPH-NODE 46509848>
```

figure、graphic-id 以外の各スロットは初期値に設定されたノードのインスタンスが生成される (Figure 5.28)。

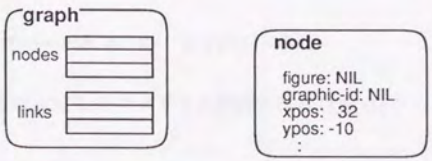


Figure 5.28: ノードのインスタンスの作成

次にグラフにこのノードを登録する。

```
> (add-node g n1)
#<GRAPH-NODE 46509848>
```

グラフのノード表にノードが登録される (Figure 5.29)。

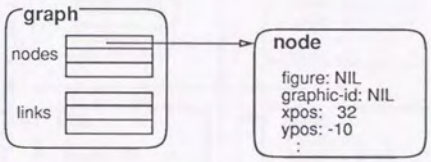


Figure 5.29: ノードのグラフへの登録

リンクの生成も同様に make-instance で行なわれるが、始点ノードと終点ノードが各々指定されなければならない。

```
> (setq n2 (make-instance 'graph-node))
#<GRAPH-NODE 46509846>

> (setq l1 (make-instance 'graph-link :source n1 :destination n2))
#<GRAPH-LINK 46509844>
```

ノードの links スロットおよびリンクの source、destination スロットが埋められる (Figure 5.30)。これは以下の式と同値である。

```
> (setq l1 (connect n1 n2 'graph-link))
```

作成されたリンクは add-link でグラフに登録される (Figure 5.31)。

```
> (add-node g n2)
#<GRAPH-NODE 46509846>

> (add-link g l1)
#<GRAPH-LINK 46509844>
```

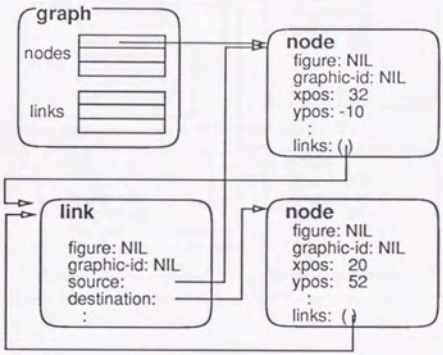


Figure 5.30: リンクの生成

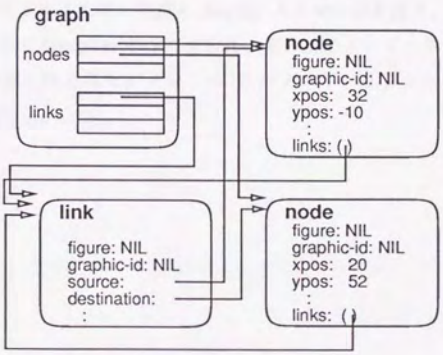


Figure 5.31: リンクの登録

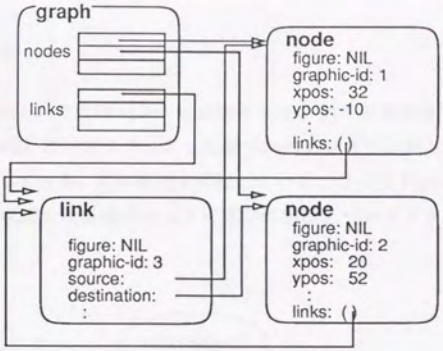


Figure 5.32: 表示後のグラフ

以上は Lisp 上に概念的なグラフを作成しただけで、ディスプレイ上には表示されない。グラフをディスプレイに表示するには、グラフに対し display メッセージを送る。

```
> (display g)
```

グラフは各ノード・リンクに対し同様に display メッセージを送り、これらのメッセージはソケットを通じて Grapher 本体に送られ、HP 上にノード・リンクが表示される。Grapher 本体から返された各セグメントの ID でノード、リンクの graphic-id スロットが埋められる (Figure 5.32)。

5.6 拡張機能の実現

5.6.1 クラス構成

VOGUE は前節の 3 次元 Grapher を基本として、種々の機能を実現している。具体的にはクラス graph のサブクラスとして幾つかのクラスを定義し、各々の機能をメソッドとして実現している。拡張機能を実現しているクラスは Figure 5.33に示すクラスがある。以下の各節では各機能とともに実際に実現しているクラスに関する説明を行なう。

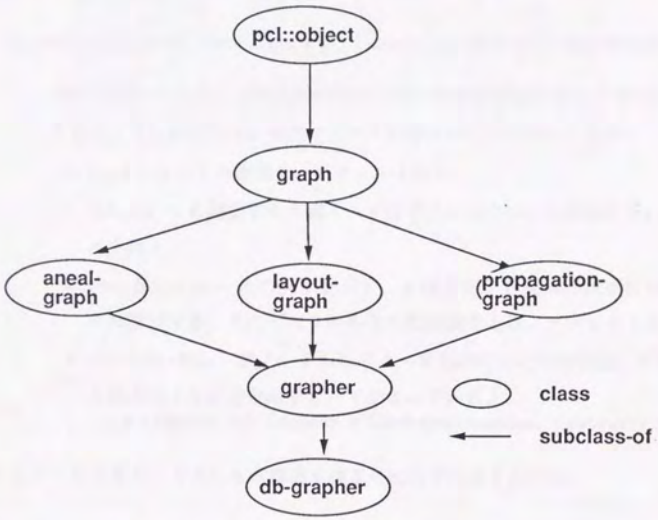


Figure 5.33: 拡張グラフのクラス階層

5.6.2 レイアウト機能の実現

レイアウト機能はクラス layout-graph で実現されている。ただし、これらレイアウト・アルゴリズムの空間的経済性、計算速度等は重視していない。以下に各レイアウトについて説明する。

木構造配置

効率的な木構造を得るための描画アルゴリズムは従来から数多く研究されているが [117, 91]、本研究で使用したアルゴリズムは文献 [8] で使用されているものと同じである。木構造レイアウトには以下の形式のメソッドを用いる。

- (layout-as-tree *graph node &key x y z balance tree-direction child-function*)
  - *node* をルートとし、*child-function* で得られる木構造をルートが (x, y, z) になるように描画する。各キーワード引数は以下のとおりである。
    - *x,y,z* – ルートの座標で、デフォルトは 0
    - *balance* – t に設定すると親ノードは子ノードの中心に位置する。デフォルトは t
    - *tree-direction* – 木の向きを示す。x 軸方向から反時計回りに 90 度ごとに設定でき、それぞれ 0 から 3 の整数値をとる。デフォルトは 3。
    - *child-function* – 親ノードから子ノードを求めるための関数。デフォルトは次のように全部の子ノードになっている。

```
#'(lambda (n) (mapcar #'link-destination  (output-links n)))
```

例えばノード r をルートとした木構造を得るには以下のようにする。

```
> (layout-as-tree g r)
```

Figure 5.34は 4 階層の 2 分木を木構造配置した例である。

クラスタ構造配置

現在までのところ、3 次元クラスタ構造を得るためのアルゴリズムに関する研究は少ない。本論文では、第 4 章で開発した Generalized Fractal Views の考え方を基本とした 3 次元クラスタ構造生成手法を用いている。実際に 3 次元クラスタ構造に配置するには以下の形式のメソッドを用いる。

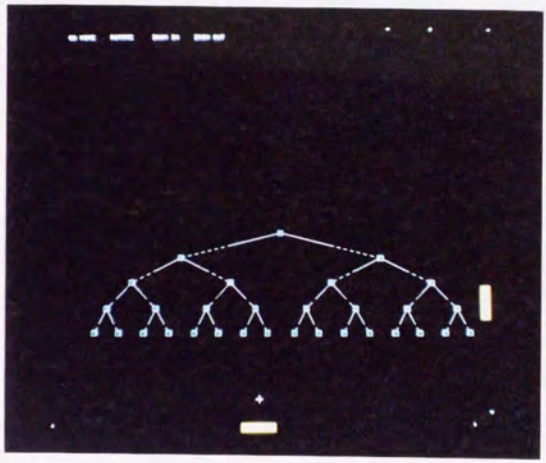


Figure 5.34: 2 分木の木構造配置例

- (layout-as-cluster *graph node*
  &key fd x y z scale transparency-value level child-function 2d)

  – *node* をルートとした 3 次元クラスタ構造を描画する。キーワード引数は以下のとおり。

  * *x,y,z* – 最終的なルートノードの中心座標。デフォルトは 0。
  * *scale* – ルートノードの大きさ。デフォルトは 5.0。
  * *transparency-value* – ルートノードの透明度。デフォルトは 5。
  * *level* – どの深さまで描画するか指定する。デフォルトは 100。
  * *child-function* – 親ノードから子ノードを求めるための関数。デフォルトは次のようになっている。

    ```
    #'(lambda (n) (mapcar #'link-destination  (output-links n)))
    ```

  * *2d* – t にすると xy 平面上にクラスタ構造が描画される。デフォルトは nil

例えば r をルートとし、isa-link による階層構造を 3 次元クラスタとして表示するには以下のようにする。

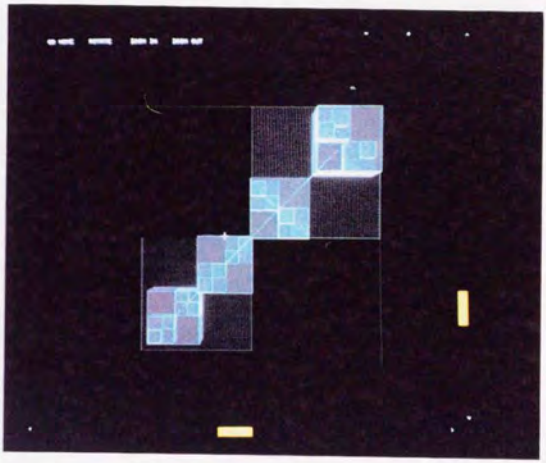


Figure 5.35: 2 分木のクラスタ構造配置例 (フラクタル次元 =1.0)

```
> (layout-as-cluster g r
    :child-function
      #'(lambda (n)
          (mapcar #'link-destination
            (remove-if-not #'(lambda (l)
                               (typep (figure l) 'isa-link))
    (output-links n)))))
```

Figure 5.35は 4 階層の 2 分木をクラスタ構造配置した例である。また Figure 5.36は引数として与えるフラクタル次元を 0.7 とした時の表示例で、各クラスタの縮小率が大きくなる。

ここで用いられたアルゴリズムに関して説明する。第 4 章での考察によれば、一般木はフラクタル性を有し、物理形状を持たない論理木の場合も各ノードが持つ概念的な値を考慮することで、フラクタル性を持つと考えることができた。ここでの 3 次元クラスタアルゴリズムは、この概念的大きさをノードの大きさに具体化したものである。つまり、木として表現されるノードの階層構造に対し、ノードの大きさをルート

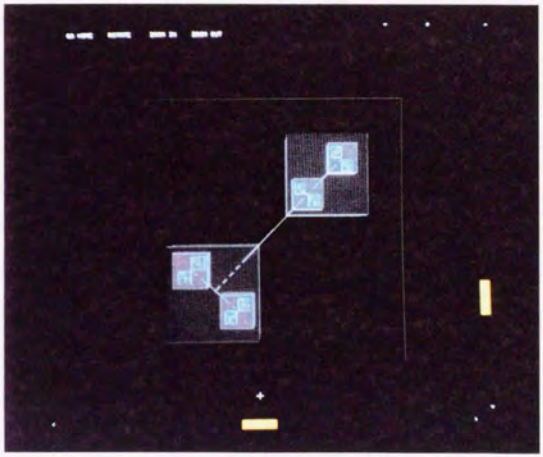


Figure 5.36: 2 分木のクラスタ構造配置例 (フラクタル次元 =0.7)

ノードから

$$r = N^{-\frac{1}{D}}$$

の式に基づき次第に縮小する。実際に用いられたアルゴリズムは以下に示すとおりであり、Figure 5.37がその概念図である。

1. ルートノードを中心が (x,y,z) の一辺 L の立方体としてに描く。
2. 子ノードが n 個存在する時、この立方体を一辺が L/n の小立方体 n^3 個に分割する。
3. 3 次元の各行列が他のノードのそれと重複しない領域の中心に子ノードを配置する
4. 子ノードの大きさを $r = n^{-\frac{1}{D}}$ に基づき縮小する。
5. 以上の操作を子ノードに対しても再帰的に繰り返す

各ノードの集散度は引数として与えられるフラクタル次元 D によって制御され、D を $D > 0$ の範囲で小さくすればノードに伝播される縮尺も小さくなり、各ノードの集

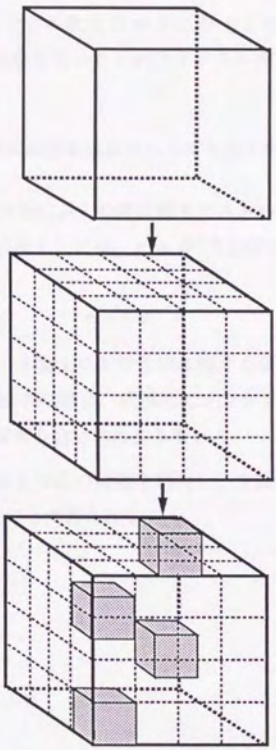


Figure 5.37: 3 次元クラスタ描画アルゴリズムの概念図

散度は高まる一方、クラスタ間の間隙も大きくなる。また $D < 3$ の範囲で大きくすればノードに伝播される縮尺は大きくなり、各ノードの集散度、クラスタ間の間隙共に小さくなる。これはフラクタルの性質から当然と言える。

このアルゴリズムには以下の2つの特徴がある。

1. 引数として与えるフラクタル次元 D が $0 < D \leq 1$ の時には、同じ木の深さのクラスタ同士が決して重ならないことがフラクタル次元の定義によって保証される
2. 立方体を6個のどの面に垂直な位置から見ても全てのクラスタが重ならない

ただし問題点としては、木の分岐が1の時にはクラスタの大きさが変化しないことが挙げられる。この問題の解決法としては、$r = N^{-\frac{1}{D}}$ の式を1以下の正定数 C によって

$$r = CN^{-\frac{1}{D}}$$

とすることで解決できるが、本論文における例に関する限り問題はないので、そのまま利用している。また一般の木の場合、経験的にフラクタル次元は1.3程度まで大きくしても、クラスタ同士の重なり合いはあまりない。

2次元クラスタ構造は上の2つ目の特徴を利用し、1度3次元クラスタを作成してから xy 平面に投影変換することで得られる。

5.6.3 焼きなましによる自動配置機能の実現

焼きなまし機能はクラス aneal-graph で実現されている。この機能はリンクで結合されたノード同士は他のノードより関係が深いと考え、これらリンクで結合されたノード同士を近くに配置する機能であり、以下のメソッドによって実行される。

- (heuristics *graph ncycles*)
  - *graph* に登録されている全ノードに対して、*ncycles* 回焼きなまし操作を実行する

焼きなましの具体的アルゴリズムは以下に、またその概念図を Figure 5.38と Figure 5.39に示す。

1. グラフ上の各ノードについて以下のことを行なう
2. あるノードとリンクで結合されているノード全体の重心を求める
3. このノードを中心とする指定された半径内のある点を求める
4. この点の回りに他のノードが存在せず、かつ2で求めた重心に接近する方向ならばノードを移動する

各ノード間の最小距離は aneal-graph の内部スロット interval、移動できる最大半径は max-move によって決定される。これらは setf によって変更可能であり、デフォルト値はそれぞれ5、20である。interval を小さくすると、各ノード間の間隔は狭くなる。

この焼きなまし手法は SemNet において用いられた手法で、セマンティックな情報無しで、関連するノード同士を近くに配置しグラフ全体の視認性を高めるために用いられたが、同一の情報から必ずしも同じグラフの形が得られるとは限らない点や計算速度の問題等がある。本論文の第6章、第7章の例では利用されていないが、第8章において考察がなされる。

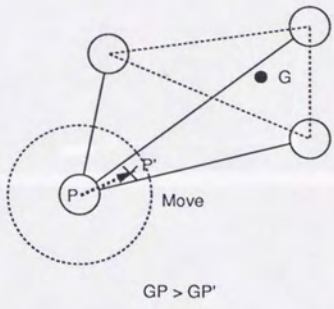


Figure 5.38: 焼きなましの概念図 (動く場合)

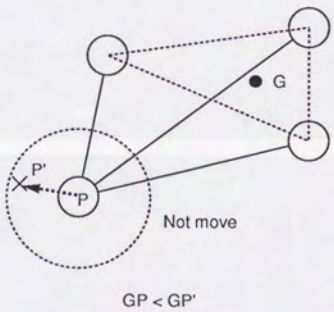


Figure 5.39: 焼きなましの概念図 (動かない場合)

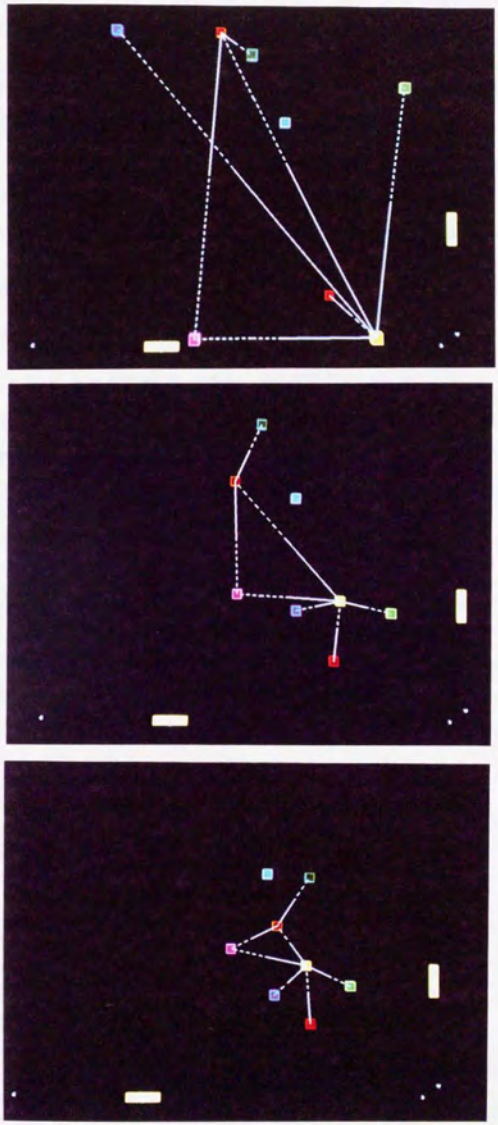


Figure 5.40: 焼きなましの例

5.6.4 図形数制御機能

着目ノードをルートノードとする木構造に基づいた図形数制御手法はクラス propagation-graph で実現されている。各々について以下に説明する。

フラクタル次元に基づいたノード数制御

第4章で開発した Generalized Fractal Views に基づいたノード数の制御は以下の形式のメソッド fractal-display によって行なわれる。

- (fractal-display *graph node D &key threshold link-destination*)
  - *node* をルートとする木を考え、横型探索によって $r = N^{-\frac{1}{D}}$ の式によって決定される値を各ノードに伝播し、閾値 *threshold* 以上の値を持つノードは表示し、それ以外は消去する。各キーワード引数は次のとおりである。
    - *threshold* – 閾値。デフォルトは 0.1
    - *link-destination* – リンクの方向を指示する関数。デフォルトは links。

この手法は Generalized Fractal Views における各ノードの概念的な値を、着目点に対する各ノードの重要度と考えている。第4章での計算機シミュレーションの結果から、閾値をある値に固定すると対象とする木構造がいかなる形であっても、表示されるノード数はほぼ一定に制御される。そしてこの表示ノード数は閾値を変化させることで、フレキシブルに変化させることが可能である。Figure 5.41はその概念図である。

本手法は第7章において適用され、一般の木構造における閾値と表示図形数との関係が考察される。

木の深さに基づくノード数制御

木の深さに基づくノード数制御は以下の形式のメソッド normal-display でなされる。

- (normal-display *graph node level &key link-destination*)
  - *node* をルートとする木を考え、縦型探索によって深さ *level* のノードまでは表示し、それ以外は消去する。キーワード引数 *link-destination* のデフォルト値は links である。

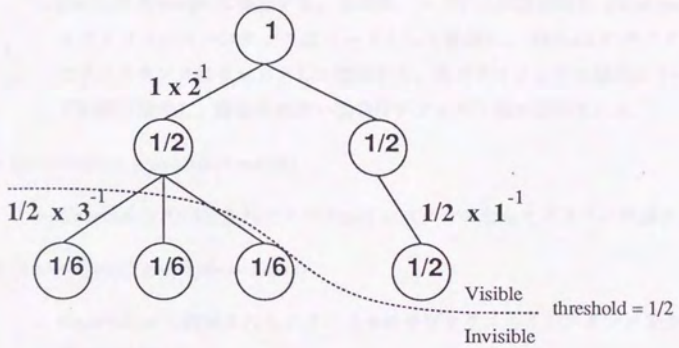


Figure 5.41: フラクタル次元に基づくノード表示の概念図

5.7 データベースとの結合

5.7.1 db-grapher

データベースのインスタンスを登録するためのクラスは db-grapher で、上で述べた grapher のサブクラスとして定義され、各拡張機能を全て継承している。

データベース内のインスタンスをグラフに登録するには、以下のメソッドを用いる。

- (add-object *graph instance &key xpos ypos zpos color shape width height length transparency label*)
  - *instance* を *graph* に登録する。その際、システムが自動的に db-object のサブクラスのインスタンスはノードとして登録し、db-link のサブクラスのインスタンスはリンクとして登録する。各グラフィックス属性はキーワード引数で指定し、指定されない場合はデフォルト値が適用される。
- (add-objects *graph class-name*)
  - *class-name* で指定されたクラスのインスタンスを全てグラフに登録する。
- (add-objects* *graph class-name*)
  - *class-name* で指定されたクラスとそのサブクラスのインスタンスを全てグラフに登録する。

Figure 5.42は、add-object をする以前のグラフとインスタンスで、この状態で

```
> (add-object g db406)
```

を実行すると、Figure 5.43のように、figure スロットにこのインスタンスを持った graph-node が1つ作成されて、グラフに登録される。

5.7.2 データの階層性による図形制御機能の実現

VOGUE では階層管理されたモデルに基づき、ノード・リンクの表示を制御することができる。これには以下の形式のメソッドを用いる。

Figure 5.42: add-object 実行前

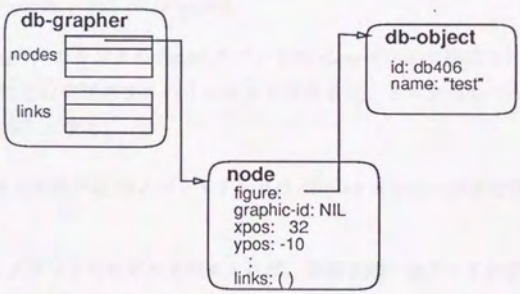


Figure 5.43: add-object 実行後

- (focus-node *graph class-name*)
  - *graph* 内のノードの figure スロットが *class-name* で指定されたクラスあるいはそのサブクラスのインスタンスならば、ノードの focused-p スロットを T にする。
- (unfocus-node *graph class-name*)
  - *graph* 内のノードの figure スロットが *class-name* で指定されたクラスあるいはそのサブクラスのインスタンスならば、ノードの focused-p スロットを NIL にする。
- (focus-link *graph class-name*)
  - *graph* 内のリンクの figure スロットが *class-name* で指定されたクラスあるいはそのサブクラスのインスタンスならば、リンクの focused-p スロットを T にする。
- (unfocus-node *graph class-name*)
  - *graph* 内のリンクの figure スロットが *class-name* で指定されたクラスあるいはそのサブクラスのインスタンスならば、リンクの focused-p スロットを NIL にする。

focused-p スロットが NIL のノード・リンクは display メッセージを受けても表示されない。

先の Flavor メソッドのモデルを例にとれば、初期状態で全ノードが表示されている時、flavor-primary-method だけを消去したい場合には

```
> (unfocus-node g 'flavor-primary-method)
```

とすることにより達成される (図 5.44)。 unfocus-node によりグラフ g は保持する全ノードを調べ、ノードの figure スロットで指示されるインスタンスのクラスが指定されたクラスかあるいはそのサブクラスの場合には、ノードの focused-p スロットを NIL に変更する。display メッセージはこの focused-p スロットが t の場合に限りノードを表示するので、unfocus されたノードは以後表示されなくなる。

また、unfocus されたクラスを表示するためには、

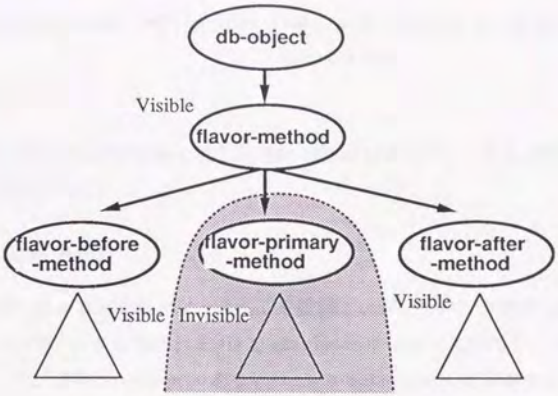


Figure 5.44: (unfocus-node g 'flavor-primary-method) 実行後

```
> (focus-node g 'flavor-primary-method)
```

を実行する。これにより figure スロットのインスタンスのクラスが flavor-primary-method かそのサブクラスなら、focused-p スロットは t になる。

5.7.3 その他の機能

ノード選択時の動作のカスタマイズ

3 次元 Grapher 上でノードが選択された場合、Grapher 本体から (LEFTBUTTON-PRESSED ID) あるいは (RIGHTBUTTONPRESSED ID) というメッセージが返ってくる。これを用いて、ノードが選択された場合の処理をノードに対応するインスタンスのクラスごとにメソッド left-button-proc、right-button-proc を定義することでカスタマイズできる。デフォルトではこれらは以下のように定義されておりノードが選択されても何もしない。

```
(defmethod left-button-proc ((obj pcl::object)(g db-grapher)
                             &rest args)
```

```
    'nil)
(defmethod right-button-proc ((obj pcl::object)(g db-grapher)
                              &rest args)
    'nil)
```

ただし、この機能を使用する時には Lisp 側で以下の形式のメソッドを起動し、ループ状態に入る必要がある。

- (main-loop *graph*)
  - Grapher 本体からのメッセージを監視し、メッセージが到着した場合にはメッセージに付加された ID を graphic-id スロットに持つインスタンスに対して、left-button-proc あるいは right-button-proc を起動する。

例えば、flavor-primary-method に対応するノードがマウスの左ボタンで選択された時、ノードの色を赤く変えるには、以下のような定義をする。

```
(defmethod left-button-proc ((obj flavor-primary-method)(g db-grapher)
                             &rest args)
  (change-attribute (assoc-node g 'figure obj) :color red))
```

5.8 本章のまとめ

ソフトウェアの3次元視覚化のための試作システム VOGUE のインプリメントについて述べた。VOGUE のソフトウェアはオブジェクト指向データベースと3次元 Grapher から構成され、対象とするソフトウェアのモデル化の機能、自動配置機能、図形数削減機能、基本的な3次元インタフェース機能を有する。

次章以降では、試作した VOGUE を用いて実際のソフトウェアの3次元視覚化表示をし、議論を行なう。