平成 12 年度

筑波大学第三学群情報学類

卒業研究論文

題目：３次元ビジュアルプログラミングシステムにおける
仮想空間内でのマウスによる移動操作方法の研究

主専攻	情報科学
著者名	甲斐 健太郎
指導教員	電子・情報工学系 田中 二郎

要旨

本論文では、次世代コンピューティング環境の一つの例として３次元環境を取り上げ、その基礎技術の一つである３次元ビジュアルプログラミングシステムにおける仮想空間内での視点移動操作手法について述べる。

我々の研究グループでは３次元ビジュアルプログラミングシステム “3D-PP” の開発を進めている。しかしながら、ビジュアルプログラミングシステムの３次元化には仮想３次元空間内における視点の移動操作がユーザに大きな負担を与えてしまう、という問題を解決しなければならない。本研究では、３次元ビジュアルプログラミングシステムに特化した視点移動操作を提案することでこの問題の解決を図り、また実際に “3D-PP” に実装した。また、ユーザに仮想３次元空間内の情報を与え、ユーザの空間の把握を助ける “3D-PP” の機能を拡張した。

目次

第 1 章

序論

近年、 VRML 等で記述されたウェブページや、３次元環境を題材としたゲーム等の普及に伴ない、数多くの一般のコンピュータのユーザが３次元環境でのインタラクションを経験する機会が増えてきている。３次元グラフィックスはもはや特殊な技術ではなく、映画やゲーム、テレビなどで誰でもが日常的に接するものとなってきている。我々は、このような次世代コンピューティング環境では、コンピューティング・パワーの多くはコンピュータ・ヒューマン・インタラクション (CHI) に注力されるべきと考え、次世代コンピューティング環境の１つとして、３次元コンピューティング環境について注目している。

現在、我々の研究グループでは３次元環境でのビジュアルプログラミングに関しての研究を行っている。近年、コンピュータは低価格化・高性能化の一途を辿っており、それに伴って図形の再描画を頻繁に行うビジュアルプログラミングを、実際にアプリケーションとして使用することが現実的になってきている。また、オブジェクト指向方法論の発展に伴い、アプリケーションの開発者は、自らプログラムコードを書く代わりに、あらかじめ他人の手によって書かれた拡張性のあるプログラムを再利用し、それらを上手く組み合わせることによってアプリケーションを開発して行く傾向が強くなってきている。我々はさらにこの次の段階として、テキストコードを書く必要が従来のプログラミングに比べて少ないビジュアルプログラミングが現在よりもさらに重要視されていくのではないかと考えている。

既存のビジュアルプログラミングシステム（VPS）の多くは２次元表示によるプログラムの視覚化を行っている [1, 16]。しかしながら、 VPS においては「ノード」や「エッジ」等のビジュアルプログラムのプログラム要素は図形で表示されるため、２次元表示ではプログラム要素の数が多くなるにつれて VPS の表示領域内がプログラム要素で埋め尽くされてしまい、それ以上プログラム要素を追加できなくなってしまう、という問題があった。２次元表示に基づく VPS では、このような貧弱なスケー

ラビリティが原因で、規模の大きい実用的なビジュアルプログラムは記述は困難であり、結局はトイ・プログラムしか記述できなかった。

我々は３次元表示に基づく VPS こそ、 VPS のスケーラビリティ問題の解決への大きなブレイク・スルーになると考えており、実際に３次元 VPS“3D-PP” の開発を進めている。単純な例で考えても、ある面積の表示領域を持つ２次元 VPS において、最大で $10 \times 10 = 100$ 個のノードを２次元空間内に配置可能な場合、同じ面積の表示領域を持つ３次元 VPS では奥行き方向をノードの配置に利用できるため、 $10 \times 10 \times 10 = 1000$ 個ものノードを配置することが可能である。

このように、 VPS を３次元化するということは、 VPS に関する非常に大きなパラダイムシフトを引き起こす可能性があるものであると考える。

また、既存の VPS では、プログラミング初心者を対象とした、プログラミングの学習用として開発されたものもあり、小規模のビジュアルプログラムが記述できれば十分有用とされている場合 [2] もあるが、本研究では “3D-PP” がターゲットとするエンド・ユーザとしてプログラマも想定しているので、規模の大きなプログラムが記述できるということは重要な目標の１つである。

しかし、 VPS を３次元化するには、仮想３次元空間内における視点の移動操作が、ユーザにとって大きな負担となってしまうという問題を解決しなければならない。これは、３次元では２次元に比べ自由度が格段に増大するのに対し、入力デバイス、出力デバイスが、それぞれマウス、ディスプレイといった２次元のデバイスを用いることに起因する。特殊な入出力デバイス用いて解決する方法も考えられるが、通常考えられるシステム上での使用を考えた場合、これは避けるべきである。ただし、２次元マウスでは自由度が不足しており、なんらかの工夫が必要となる。

本論文の構成は以下の通りである。まず、第 2章で我々が開発を進めている３次元 VPS“3D-PP” を紹介する。続く第 3章では VPS の３次元化に伴って発生する問題を明らかにし、その解決手段として、３次元 VPS に特化した視点移動操作を提案する。最後に第 4章で、３次元 VPS に特化した視点移動操作を実装し、さらにユーザの仮想空間の把握を助けるための機能を拡張した新しい “3D-PP” について述べる。

第 2 章

３次元ビジュアルプログラミングシステム “3D-PP”

我々のグループでは現在、特に VPS のの３次元環境化について興味があり、実際に３次元 VPS“3D-PP” を設計・開発している [4, 5, 6, 7, 13, 14]。

2.1 “3D-PP” の概要

“3D-PP” は、並列論理型言語の１つである GHC [17] を対象として開発されている [7]。

“3D-PP” は従来の視覚化手法である２次元の視覚化 [2, 16] ではなく、３次元による視覚化の新しい手法を取り入れたビジュアルプログラミングシステムであり、並列論理型言語 GHC の特徴であるゴールのリダクションの過程や論理変数の具体化に着目した表現手法を実装している。

“3D-PP” において、ユーザは３次元表現されたノード・エッジ・アイコン等の「プログラム要素」を組み合わせていくことによりプログラムを記述することができる (図 2.1)。

2.2 “3D-PP” のシステム構成

2.2.1 言語のモデル化

“3D-PP” は、次のような理由から並列論理型言語の１つである GHC に基づいて開発されている。

- 『言語を構成する基本要素の数が少なく，規則がシンプルである』

  GHC を含む宣言型の言語は、データ構造の要素となる意味単位 (アトム) や演算子、そしてそれらのデータ構造間の関係規則によって構成される。また実行

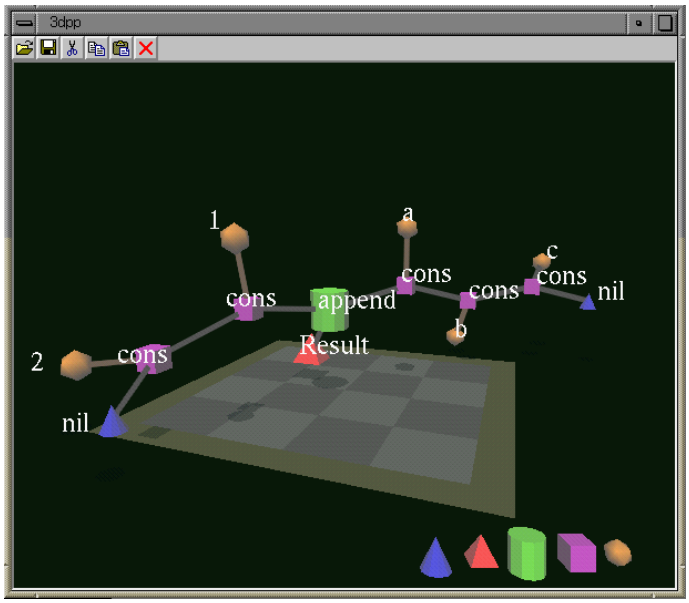


図 2.1: “3D-PP” の実行画面

規則も、ゴールをサブゴールに置換するというもののみで、非常に簡潔なものである。

このため、手続き呼び出し・変数・制御構造など多くの要素・規則を含む手続き型言語などに比べて視覚化が容易である。

- 『データ及びプログラムが単一の表現から成る』

  GHC のプログラムは、データ構造から新しいデータ構造への変換規則を定義するものであり、データ構造と同じ構成要素を用いて記述される。

  従ってビジュアルプログラム内のプログラム要素を操作する場合、操作対象による表現の違いを意識するような必要がなく、プログラマの負担を軽減できる。

2.2.2 ユーザとのインタラクション

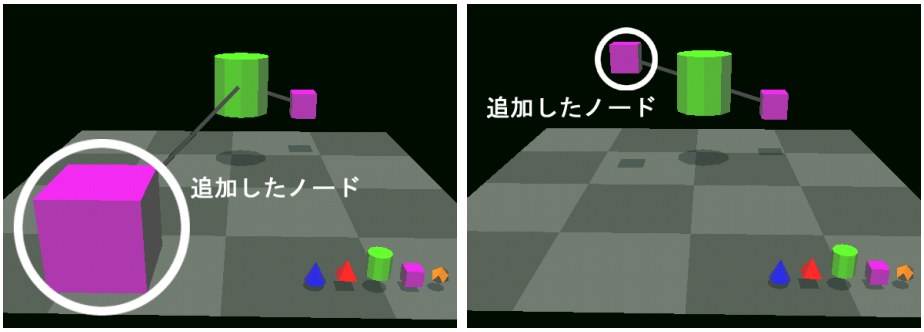
“3D-PP” は、ユーザがビジュアルプログラムの作成や編集を行う「ビジュアル・エディタ部」と、 GHC に基づいて設計された並列論理型言語である KL1 [3] の処理系との通信を行う「エンジン部」の、 2 つのサブシステムから構成されている。ビジュアルプログラムは内部表現を用いて表現され、 2 つのサブシステム間は内部表現を介してやりとりを行う。

実際にユーザが “3D-PP” を使用する際には、 “3D-PP” の「ビジュアル・エディタ部」によりウィンドウ上に表示された仮想 3 次元空間内で、「ノード」等のビジュア

ルプログラムのプログラム要素を配置、編集していくことによって、ビジュアルプログラムを構築していく。

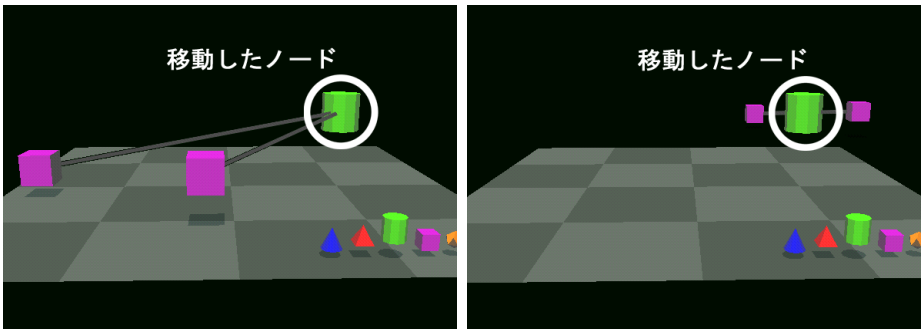
ユーザが空間内でノードを移動させる際には、画面上のノードを直接ドラッグ＆ドロップすることによって行うことが可能である。このため、３次元空間内の情報と直接インタラクションを行うことができ、ユーザは直観的にビジュアルプログラミングを行うことができる。

また、スプリングモデルを用いた自動レイアウト機能 [5] により、それぞれのノードは適切な位置に自動的に移動する。この機能により、ユーザはノードを追加する際にはドラッグ＆ドロップによってノード同士の結線を行うだけで、図 2.2a のように離れた位置にあるノードを追加しても、図 2.2b のように適切な位置まで自動的に移動を行わせることが可能である。また、結線されたノード群全体を移動する際には、図 2.3a のように一つのノードを移動させるだけで、図 2.2b のように他のノードが自動的に追随するため、全てのノードを移動する必要はない。このように、“3D-PP” は自動レイアウト機能により、少ない手間でビジュアルプログラミングを行うことが可能となっている。



(a) 適用前　　(b) 適用後

図 2.2: 自動レイアウトをノードの追加に適用した例



(a) 適用前　　(b) 適用後

図 2.3: 自動レイアウトをノード群全体の移動に適用した例

2.3 “3D-PP” の操作例

実際に “3D-PP” を用いてビジュアルプログラムを記述する例を示す。

例題として

- リスト $[1,2]$ とリスト $[a,b,c]$ とを結合する

というプログラムを作成する。このプログラムは KL1 で記述すると次のようになる。

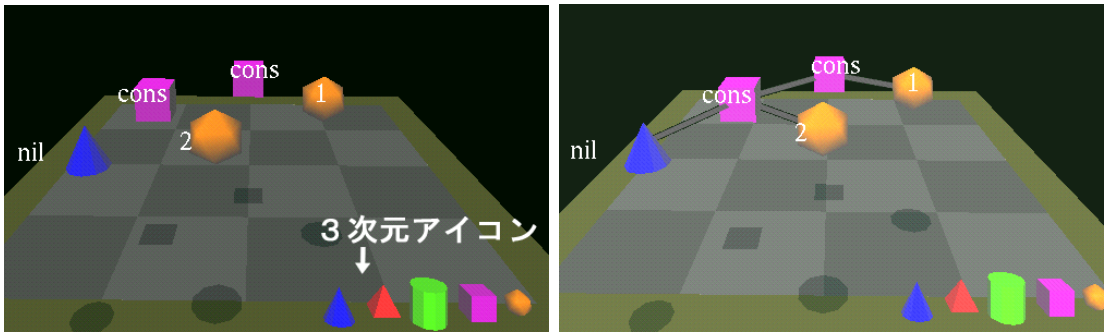
```
:- module main.

main :- append([1,2],[a,b,c],X), io:outstream([print(X),nl]).

append([],In2,Out):- Out = In2.
append([Msg|In1],In2,Out):-
   Out = [Msg|OutTail],
   append(In1,In2,OutTail).
```

以下、このプログラムを “3D-PP” を用いて記述する例を順を追って説明する。

2.3.1 リストの作成

結合するためのリストを用意する。まず、リスト $[1,2]$ を作成する。図 2.4a のように、リストの構成要素である、1、2、*nil*、*cons* の各ノードをそれぞれ用意する。ノードを追加する際には、画面内の 3 次元アイコンをクリックすることによって新たに空間内にノードを出現させる。

次に図 2.4b のように、それぞれをエッジで結線する。この際、エッジで結線したい二つのノード上でドラッグ&ドロップを行うことにより、二つのノードがエッジで結線される。これらの操作によりリスト $[1,2]$ が作成された。また、リスト $[a,b,c]$ も同様に作成する。



(a) リストの構成要素を用意する　　(b) それぞれエッジで結線する

図 2.4: リスト $[1, 2]$ の作成

2.3.2　プロセス *append* の呼び出し

二つのリストを結合するプロセス *append* を呼び出す。また、出力データを結果として格納するためのノードである *result* も用意する (図 2.5)。

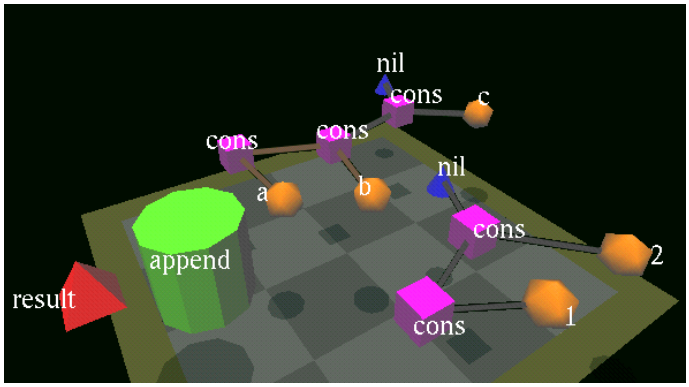


図 2.5: プロセス *append* の呼び出し

2.3.3　リストの結合

用意した二つのリストを図 2.6のようにドラッグ&ドロップによってプロセス *append* とエッジで結線し、さらに、プロセス *append* と *result* とを結線する。

また結合された二つのリストを構成するそれぞれのノードは、自動レイアウト機能により適切な位置に自動的に移動する。

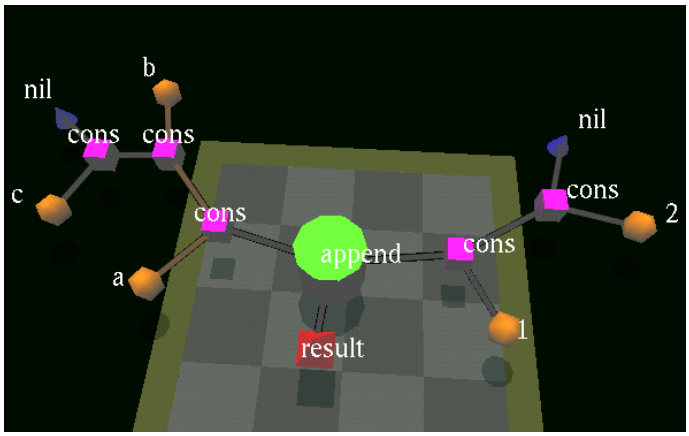


図 2.6: 二つのリストの結合

これらの操作により、リスト $[1, 2]$ とリスト $[a, b, c]$ とを結合するビジュアルプログラムが作成された。

第 3 章

３次元ビジュアルプログラミングシステムにおける視点移動操作

３次元 VPS において、プログラマは仮想３次元空間内のプログラム要素を操作することによって、ビジュアルプログラミングを記述していく。

しかし、“3D-PP” 等の３次元 VPS を含めた３次元グラフィクス環境での共通の問題の一つとして、視界の狭さの問題がある [8]。

一般に、仮想３次元空間は、図 3.1のように、限られたサイズの２次元のディスプレイの画面にクリッピングされて表示されるため、その３次元空間全てを一度に画面に表示することは不可能である。

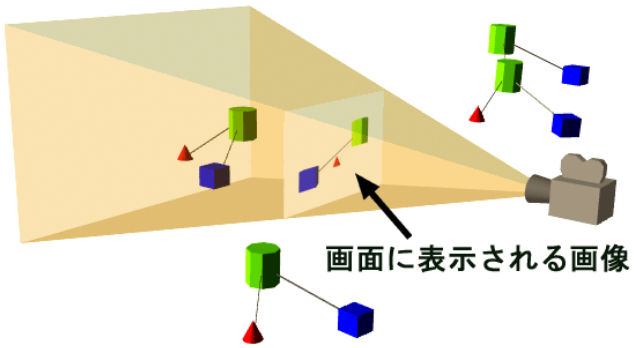


図 3.1: ３次元空間のクリッピング

画面表示の方法として、画面表示を歪ませるなどして、より多くの空間を一度に画面内に納めるといった方法 [10] もあるが、これはユーザにとって自然な見え方ではないので、空間の把握を妨げるおそれがある。特に自分で３次元空間内を移動し、３次元オブジェクトの直接操作によってシステムとのインタラクションを行うことを前提とした “3D-PP” のようなシステムでは、ユーザに３次元空間をより正確に把握させ

る必要があるため、表示される画面はより人間の視界に近い、自然な画像である必要があると考えられる。

そのため、ユーザには何らかの操作によって自らの視点を移動して、見ようとする空間が実際の画面に表示されるように調整することが求められる。

また、ユーザと３次元グラフィクス環境とのインタラクション手法として、３次元入力デバイス [20] や没入型のディスプレイ [9] といった特殊な入出力デバイスを用いるという手法もあるが、我々が通常利用できるシステム上での使用を前提とすると、コストや技術の面で現実的ではない。本研究では入力デバイスとして２次元マウス、出力デバイスとして通常のディスプレイを用いることとする。

3.1 代表的な３次元空間内での視点移動操作の問題点

３次元空間内ではユーザの視点を、図 3.2のように、３次元カメラのような３次元オブジェクトとみなし、その位置に関して x, y, z 方向の３自由度、姿勢に関してロール、ピッチ、ヨーの３自由度の計６自由度という多くのパラメータをそれぞれ指定する必要がある [10]。しかし、入力デバイスとして２次元の自由度を持つマウスを用いるため、全ての自由度を同時に指定することはできない。

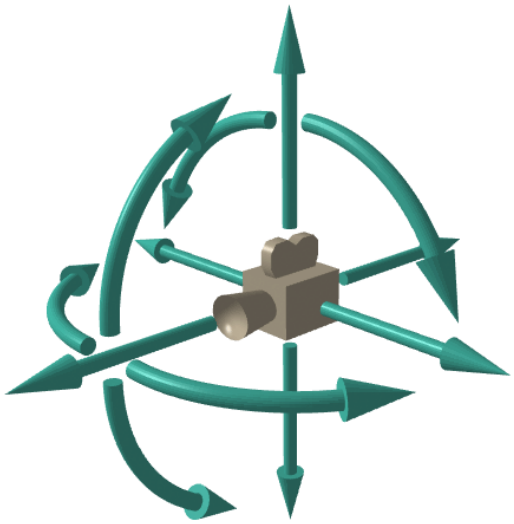


図 3.2: ３次元空間内での６自由度

これらの自由度の指定において、最も単純な方法は、視点の６自由度に対してそれぞれスライダ等の GUI 部品を対応させ、これらを操作することで視点を変更する手法である。しかしこの手法は、パラメータが多すぎるために操作が複雑になってしまい、ユーザの求める視点を得るのは難しい。

また、VRML ブラウザに代表されるような、旋回と移動を組み合わせた手法もある。この手法は、

- 目標の進行方向に達するまで、旋回する
- 目標に達するまで前進などの移動を行う

というように、旋回と移動とを組み合わせたものである。しかし、それぞれの動作が終了するまでボタンを押し続ける、マウスでドラッグし続けるといった継続的な操作が求められるものであるため、目標が遠ければ遠いほど、操作を行うユーザに対して大きな負担を与えてしまう。

また，仮想３次元空間内での視点の移動に関しての問題点の一つとして、３次元空間を自由に移動させると、ユーザは容易に迷子になってしまう、という問題もある [11]。

一般にこれらの問題は、視点移動操作の自由度を非常に高くしているため、かえって操作性が低いものになってしまう、という点に起因するものである。つまり、「どの方向を向きながらでも、どこにでも」移動できてしまう、という視点移動操作の自由度の高さは、逆にユーザに大きな負担と困惑を与えてしまう。

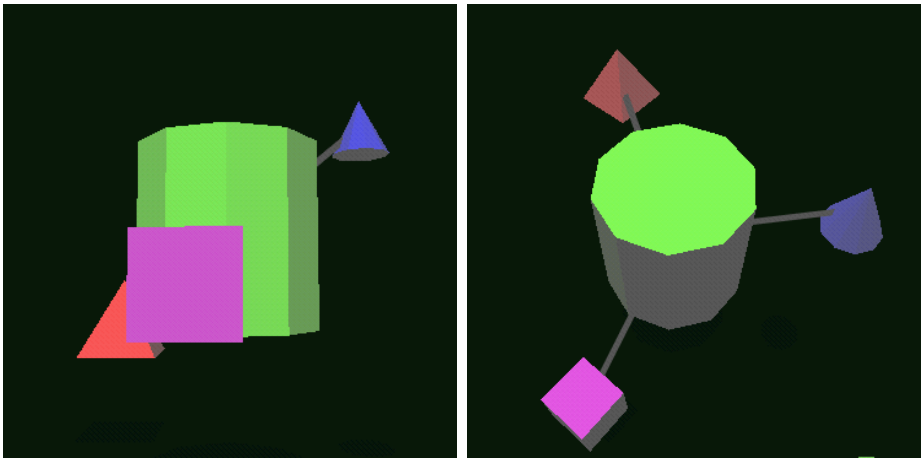
しかしながら、現在開発している “3D-PP” をはじめとした３次元 VPS のように、ユーザの行動がある程度限られてくるアプリケーションに関しては、視点移動に関して前述のような非常に高い自由度を求める必要はない。逆に、３次元 VPS に必要とされる視点移動のみを実現し、それにより操作性の向上を図ることで、VPS の３次元化に伴って発生する、３次元空間内の移動操作のわずらわしさといった、ユーザにかかる余計な負担を減らし、ビジュアルプログラミングそのものに集中させることが可能になり、これにより VPS の３次元化のメリットが活かされるものと考える。

3.2　３次元 VPS に求められる視点移動操作

第 3.1節で述べた問題点を解決するために、３次元 VPS に用いる事に特化した視点移動操作方法を提案する。この節では、その前段階として、３次元 VPS に求められる視点移動とは何かを考察する。

まず、３次元 VPS では、２次元の画面に３次元の空間を表現しているために、３次元空間の奥行き情報が欠落している。このためユーザは、ある一点からの視点からの情報だけでは３次元空間内の完全な情報を得ることができない。これは、ユーザの３次元空間の把握を妨げる重要な問題である。このため、ユーザは違った角度の視点から同じ部分空間を見る、という視点の移動が必要となる。具体例として、図 3.3a のような視点からの情報だけでは、それぞれのノードの奥行きに関する位置関係や、そ

れぞれがどのように関係しているかなど、空間内に存在する情報を全て正確に把握することは難しい。しかし、図 3.3b のように違う視点から同じノード群を見ることによって隠れていた情報を知ることが可能になり、その問題が解決できる場合がある。



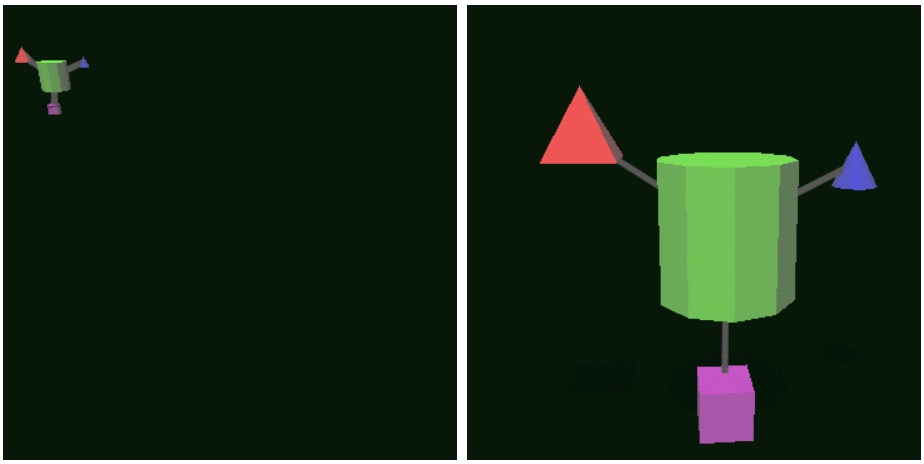
(a) ある視点からの画像　　(b) 別の視点からの画像

図 3.3: 別の視点からの情報が必要な例

また、VPS を 3 次元化することにより、一度に画面に表示できるノードの数は飛躍的に増加させることができるが、現在開発を進めている “3D-PP” のような 3 次元 VPS では、ユーザにより自然な視界を与えるために、レンダリングの際には遠近法に基づいた透視変換を用いているため、3 次元空間内で遠くにあるノードは小さく表示される。このため、図 3.4a のように視点から離れた位置にあるようなノードを編集したり、閲覧を行う際には、図 3.4b のようにその目標のノードの近くまで移動する、もしくはそのノードを視点の近くまで移動させる必要がある。しかしながら、前節で述べた通り、一般に 3 次元空間内での視点の移動はユーザに非常に負担のかかる操作を伴う。しかし、現在開発している “3D-PP” はビジュアルプログラミングを行うためのシステムであり、実際のプログラミング作業以外のユーザへの負担はできる限り軽減すべきである。そのためユーザが空間内の離れた位置にも、容易に視点移動を行える操作法が求められる。

次に、実際にビジュアルプログラムを記述、編集する際には、細かい部分を編集するためにノードを大きく画面に表示したい場合や、より大きいレベルで編集を行うためにできるだけ多くのノードを画面に表示したい場合がある。しかしながら現在の 3 次元 VPS では限られた大きさの画面を利用しているため、なんらかの工夫によって、一度に表示できる範囲を自由に変化させることができる、ということも必要である。

以上より、3 次元 VPS には、



(a) 視点から遠いノード群　　(b) 視点に近いノード群

図 3.4: 視点とノードとの距離の違いによる見え方の違い

- 『同じ部分空間の様々な方向からの検証』
- 『遠くのノードの位置への容易な移動』
- 『一度に表示する空間の範囲の変更』

の 3 点を可能にする視点移動操作方法が求められる。

3.3　3 次元 VPS のための移動操作手法の提案

前節で述べた視点移動操作を実際に 3 次元 VPS に適用するための新しい視点の操作手法を提案する。この節では、この手法の仕組みを説明する。

提案する移動操作手法は、以下の 3 つの要素から成り立っている。

- 『空間内の任意の一点を中心とした回転』

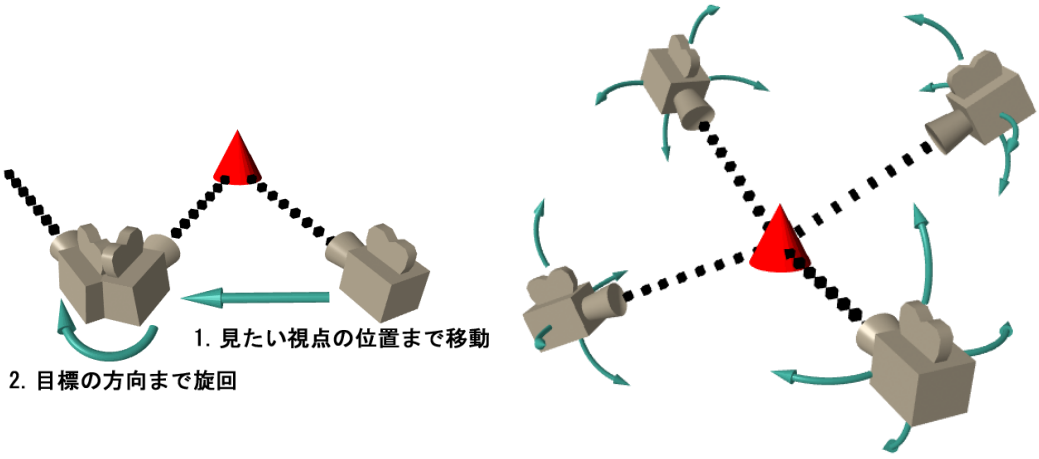
  基本的なユーザの視点移動操作として、3 次元空間内におけるユーザの視点は

  - 空間内の注目する一点（注目点）の方向を常に向いた状態で、
  - 注目点との距離を一定に保ったまま、
  - 注目点を中心とした

  回転運動を行う。

  従来までの多くの視点移動操作手法における視点の回転運動は、第 3.1章で述べた通り、自らを中心とした自転運動であった。このため、注目点を別の視点

から見るには、図 3.5a のように、まず姿勢を保ったまま見たい視点の位置まで移動し、次に目標の方向まで旋回するというような手間のかかる操作が必要であり、移動の際に目標が視界から出てしまう可能性もあった。この視点の回転の中心を、視点の位置ではなく注目点の位置へと変更し、視点の回転を自らを中心とした自転運動から、図 3.5b のように、注目点を中心とした公転運動にする。これにより任意のノード群を視界に入れたまま様々な角度から検証することを容易にし、ユーザに３次元空間内の情報をより正確に伝えることが可能となる。



(a) 回転の中心が視点の位置の場合　　(b) 回転の中心が目標の位置の場合

図 3.5: 任意の一点を中心とした回転運動

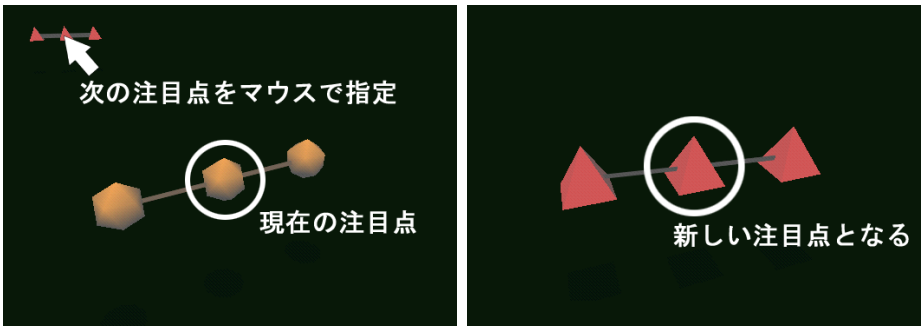
この操作は、空間をドラッグするという操作によって行う。空間をあたかも一つの３次元物体であるかのように扱うことで、ユーザは空間を画面内で回転させているような感覚で、注目点を中心とした自由な回転移動を直観的に行うことが可能である。

- 『クリックによる注目点の変更』

  基本となる回転運動の中心である注目点の変更を行う。

  注目したいノード、及びノード群を画面上からマウスボタンのクリックにより直接選択させる。選択されたノード、及びノード群の中心が以後新しい回転運動の中心点となる。

  例として図 3.6a のように遠くにあるノードを編集したい場合、マウスで目的のノードを直接指定し、クリック操作を行うことによって、図 3.6b のように、選択したノードを以後の注目点とする。



(a) 移動前　　(b) 移動後

図 3.6: 注目点の変更

また、この変更の際に、注目点をいきなり変更してしまうのではなく、現在の注目点から新しい注目点までの移動を自動的にアニメーションで表現する。これにより、ユーザへ突然の空間の位置の変化による混乱を与えてしまうことを防ぐ移動と同時に姿勢に関しても徐々に目標となる方向を向くようにすることにより、旋回、移動といった段階をおいた移動に比べてより自然な視点の移動を可能にする。

この操作により、画面上から移動先となる目標を直接指定するという直観的な操作だけで移動することが可能となり、仮想 3 次元空間内の移動において問題であった、操作が難しい、任意の位置への移動が難しい、また空間内での位置、方向の把握が難しい等といった問題が解決され、ユーザは、移動操作に関して特に考える事なく、プログラミングに集中することができる。

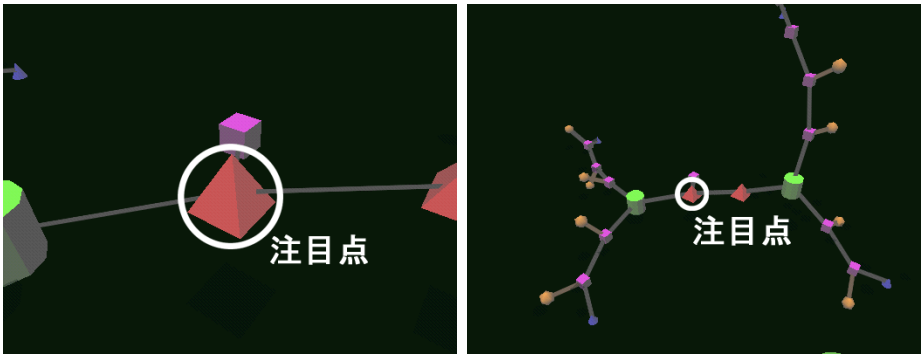
また、目標とするノード位置と視点からの距離に関わらず、マウスを一度クリックするだけで移動が可能なため、空間内の移動にボタンを押し続ける等といった連続的な操作が不必要になり、ユーザへの負担を軽減できる他、付加的な効果として、反応速度の遅い、低能力計算機上での利用にも有効であると思われる。

- 『視点の回転半径の変更』

  基本的な視点の動きである、注目点を中心とした回転運動の半径を変更することにより、一度に画面に表示されるノードの数を変更可能にする。

  ユーザが細かい部分を編集したい場合には、図 3.7a のように、回転の半径を小さくする、つまり注目点と視点との距離を短くすることで目標となるノードを大きく表示する。また、より大きいレベルで作業したい場合には逆に図 3.7b のように注目点と視点との距離を長くすることで、一度に多くのノードを表示す

ることが可能となる。



(a) 距離を短くした場合　　(b) 距離を長くした場合

図 3.7: 注目点と視点との距離の変更による表示領域の違い

この操作も前述の２つの操作同様に、スクロールバーといった新たな GUI 部品を画面上に配置すること無く、画面上の空間を縦方向空間にドラッグするという、非常にシンプルな操作で行うものとする。

また、注目度に応じて各ノードの表示の大きさを変えることにより、より多くのノードを一度に画面に表示する方法に比べ、より人間の視界に近い自然な画面表示のままで、表示するノードの数を変更できるため、ユーザは３次元空間の把握を妨げること無く大小様々なレベルでの編集、閲覧が可能となる。

これら三つの視点移動操作の要素を組み合わせることにより、簡易な操作でありながら、３次元 VPS に適した視点の移動操作を行うことが可能となる。また実際に我々の研究グループが開発している “3D-PP” にこの視点移動操作方法を実装することで、３次元空間についての知識が少ないユーザであっても直観的に空間内を移動することができ、ビジュアルプログラムそのものに専念することが可能となっている。

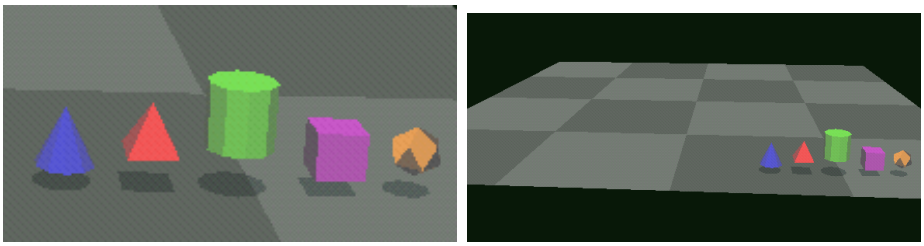
第 4 章

３次元ビジュアルプログラミングシステム “3D-PP” への実装

第 3章で述べた視点移動操作方法を、我々の研究グループが開発している３次元 VPS“3D-PP” に実装し、また視点移動操作に関してユーザを補助する機能の拡張を行った。

“3D-PP” では仮想３次元空間内で、図 4.1a に示すような、ノードのミニチュア立体アイコンから必要なノードを生成し、それらのノードをマウスのドラッグ＆ドロップによって結線していくことによってビジュアルプログラミングを記述していく。

また、ユーザは「地面」と呼ぶ図 4.1b に示すような台の上で作業を行い、空間をドラッグすることで、３次元空間を回転し、自由な方向からの視点で作業をすることができる。



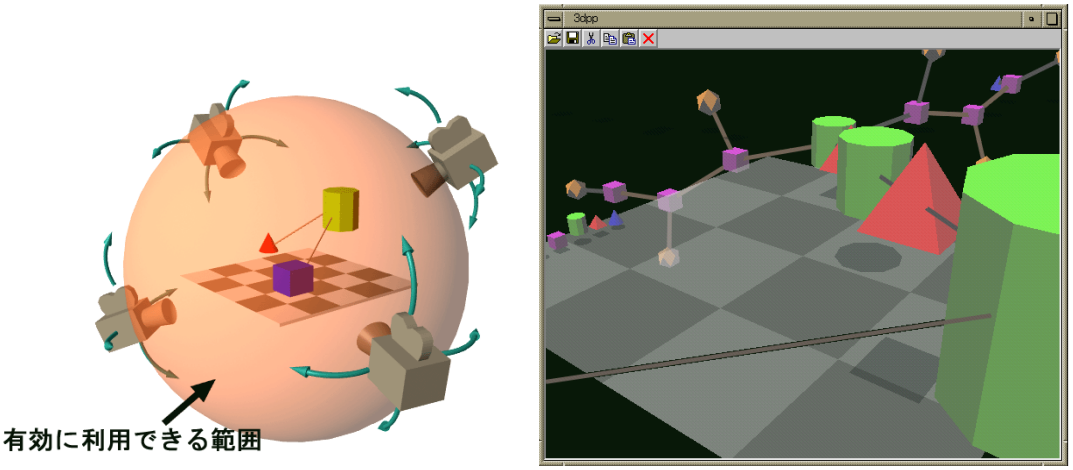
(a) ３次元アイコン　　(b) 地面

図 4.1: ３次元アイコンと地面

4.1 “3D-PP” の視点移動の拡張

従来までの “3D-PP” に実装されていた視点の移動操作手法では、視点は常に地面の中心方向を向いた状態で、視点の移動は地面の中心を中心点とした回転運動のみであり、また中心点と視点との距離は固定されていた。そのため、実際にユーザがビジュ

アルプログラミングに有効に利用できる作業空間は図　4.2a で表されるような、視点が回転運動を行う球状の部分空間の内側のみに制限されてしまっていた。この制限により、一定の大きさを越えてしまうような大規模なプログラムになると、図 4.2b のように画面内で編集を行うことが非常に難しくなってしまい、規模の大きい実用的なビジュアルプログラムを記述することは困難であった。



(a) 利用できる範囲が限定されている　　(b) 大きいプログラムだと編集ができない

図 4.2: 現在までの “3D-PP” の視点移動の問題点

このように、現在までの “3D-PP” では、視点移動操作が貧弱であったために、 VPS の３次元化のメリットを活かしきれていない、というのが現状であった。

これは従来までの “3D-PP” が、実際に大規模なビジュアルプログラムを記述することを前提としておらず、とりあえず小さなテスト・プログラムを記述できるようにする、といった段階であったため、あまり大きな作業空間を必要としていなかったことに起因する。

しかしながら、今後 “3D-PP” を実用的な３次元 VPS として完成させていくためには、大規模なビジュアルプログラムを記述できる、ということは必要不可欠である。このため、視点の移動操作の拡張による作業空間の拡大は重要な意味を持つと考えられる。

この問題を解決するために、第 3.3節で述べた３次元 VPS での利用に特化した視点移動操作手法を “3D-PP” に実装した。

- 『一点を中心とした回転』

  任意のノード、もしくはそのノードが含まれるノード群の中心を注目点とした回転運動を行う。これにより、ノードの位置、関係等の情報を容易に得ることが可能になった。

- 『クリックによる注目点の変更』

  画面上から任意のノードを直接指定することで、注目点の変更を行う。これにより、今まで固定されていた回転の中心を容易に変更できるようになった。

- 『注目点との距離の変更』

  今まで固定されていた注目点と視点との距離を変更可能にした。これにより、一度に表示できるノードの数を自由に変更することが可能となった。

これらの視点の移動の拡張により、今までの “3D-PP” に比べ、図 4.3のように、視点を移動できる範囲が格段に大きくなり、ビジュアルプログラミングをより広い作業空間を用いて行うことが可能となった。

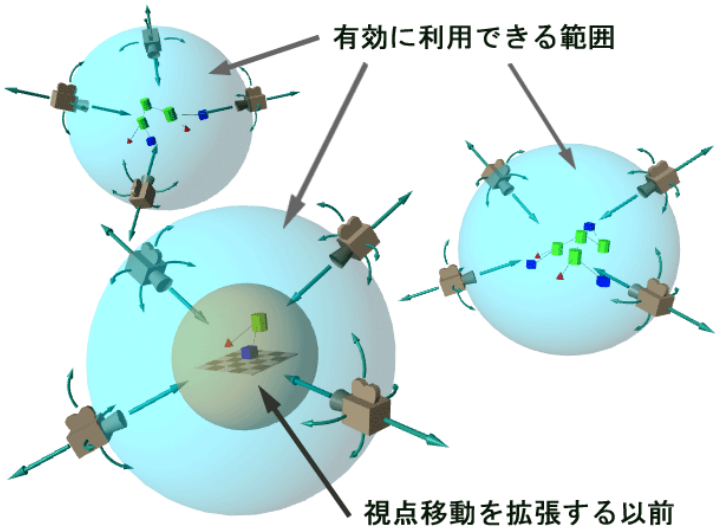


図 4.3: 視点移動を拡張した “3D-PP” の視点移動の概念図

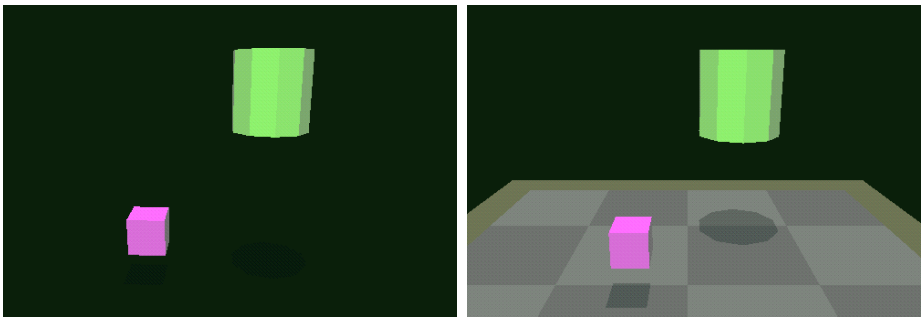
4.2 二つの地面

図 4.4a のように何もない空間にノードが配置されている状態では、それぞれのノードの奥行き、高さ等の位置情報を正確に把握することは難しい。これはノードが配置されている空間が３次元であるのに対して、その空間を表示している画面が２次元であるために、奥行き情報が無くなってしまっていることが原因である。

そこで、なんらかの手法によってより多くの３次元空間の情報を与えることは、ユーザの３次元空間の把握に非常に重要な意味を持つ。

そこで、 “3D-PP” では、この解決策として、地面と呼ぶ台を用いる。図 4.4b のように空間内に台を配置し、そこにそれぞれのノードの影を投影させることにより、

各々の奥行き、高さ等の位置情報を表現させることができるため、ユーザに空間を把握させやすくすることが可能となる。



(a) 地面なし　　(b) 地面あり

図 4.4: 地面による位置情報の付加

さらに、視点移動を拡張した新しい “3D-PP” では、地面の持つもう一つの特性に着目した。

“3D-PP” では画面上に表示される全てのノードはドラッグ操作により移動可能であり、さらに自動レイアウト機能により、その位置は一定でない。しかし、 “3D-PP” では地面はドラッグ操作でも移動させることは不可能であり、自動レイアウトの影響も受けない。つまり、地面は常に３次元空間内での絶対座標と姿勢を保ち続ける。このため、３次元空間内を移動するユーザにとって、地面は現在の視点の位置と方向を認識させるランドマーク的役割も果たすものと考えられる。

このように地面は、ユーザに空間の情報を与え、２次元の画面上に表示される “3D-PP” の３次元空間の把握を助ける重要なオブジェクトである。

しかしながら、視点移動を拡張した新しい “3D-PP” では、注目点を自由に変更できるようになったため、今までの “3D-PP” のような空間内の原点にだけ存在する地面のみでは、図 4.5のようにすぐに地面は視界の外に出てしまい、地面の機能を活かすことができない。

そこで、二つの地面を用意することによってこれらの機能を実現する。ここではそれぞれの地面をデスクトップ的地面、ランドマーク的地面と呼ぶことにする。図 4.6a のように、初期状態では二つの地面とも空間内の原点に重なって位置する。また視点の注目点を変更し、空間内を移動する際には、図 4.6b のように、デスクトップ的地面を注目点の移動に伴って移動させ、常に画面内に存在させる。一方、ランドマーク的地面は原点から移動せず、元の位置にとどまる。

デスクトップ的地面は視点の位置に関わらず、常に注目したノード群の真下に位置するため、注目したノード群の位置情報を常にユーザに与えることが可能である。

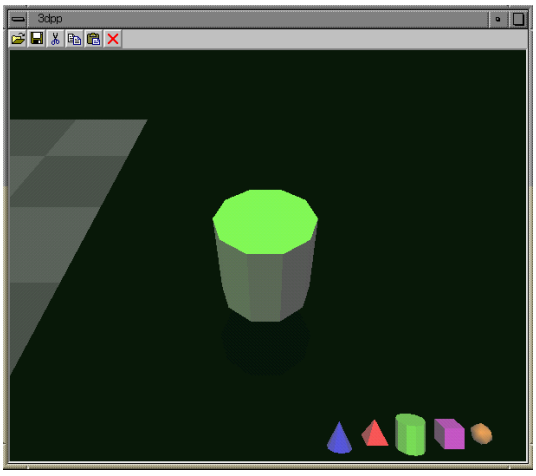


図 4.5: 画面からはみだしてしまう地面

(a) 初期状態　　(b) 注目点を変更した場合

図 4.6: “3D-PP” での二つの地面

また一方のランドマーク的地面は常に “3D-PP” の 3 次元空間内での原点にとどまり続けるため、ユーザは相対的に自らの視点の位置を知ることができ、それによりユーザの 3 次元空間の把握を助けることが可能である。

また、注目点の変更による視点移動の際に、常に原点に固定されているランドマーク的地面を選択することにより、注目点を初期状態である空間の原点に戻す。これにより、空間内で迷ってしまっても、すぐに復帰することが可能である。

以上のような地面の機能の強化により、ユーザに “3D-PP” の仮想 3 次元空間を、より自然に利用させることが可能となる。

4.3 視点移動操作を拡張した “3D-PP” の操作例

視点移動操作を拡張した “3D-PP” を用いて、実際にビジュアルプログラム作成する例を示す。

例題として、差分リストを用いてクイックソートを行うプログラムの一部を作成する。プログラム全体は KL1 で記述すると次のようになる。

```
:- module main.
main :-
klicio:klicio([stdout(Res)]),
check_stream(Res).

check_stream(normal(St)) :-
list50(L),
qsort(L,S),
St = [putt(S),nl].
check_stream(abnormal).

list10(L) :- true |
   L = [27,74,17,33,94,18,46,83,65, 2].

qsort(L, R) :- true | qsort(L, R, []).

qsort([X|L],R,R0) :- true |
    partition(L, X, L1, L2), qsort(L2, R1, R0), qsort(L1, R, [X|R1]).
qsort([],R,R0) :- true | R = R0.

partition([X|L],Y,XL1,L2) :- X=<Y | XL1 = [X|L1], partition(L,Y,L1,L2).
partition([X|L],Y,L1,XL2) :- X>=Y | XL2 = [X|L2], partition(L,Y,L1,L2).
partition([],_,L1,L2) :- true | L1 = [], L2 = [].
```

上記のプログラムのうち、視点移動操作を拡張した “3D-PP” を用いて、プロセス *qsort* を作成し、リストを入力する部分を作成する例を順を追って説明する。

4.3.1 プロセス qsort の作成

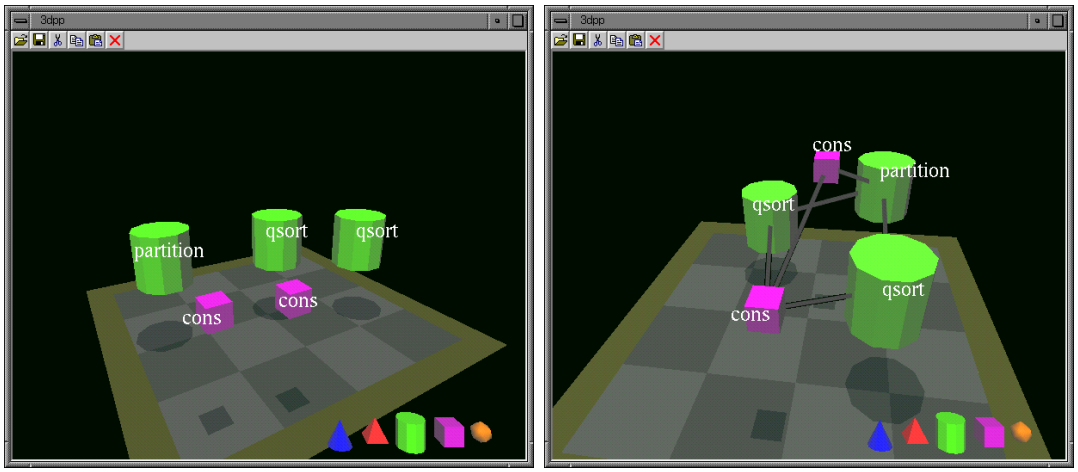
クイックソートを行うプロセス *qsort* の

```
qsort([X|L],R,R0) :- true |
    partition(L, X, L1, L2), qsort(L2, R1, R0), qsort(L1, R, [X|R1]).
```

の部分を作成する。

図 4.7a のように、 *partiton* を一つ、 *cons* と *qsort* をそれぞれ二つずつ用意する。

次に適切なノード同士をエッジを用いて、図 4.7b のように結線する。結線にはドラッグ&ドロップによって行う。



(a) 必要なノードの用意　　(b) エッジによる結線

図 4.7: プロセス *qsort* の作成過程

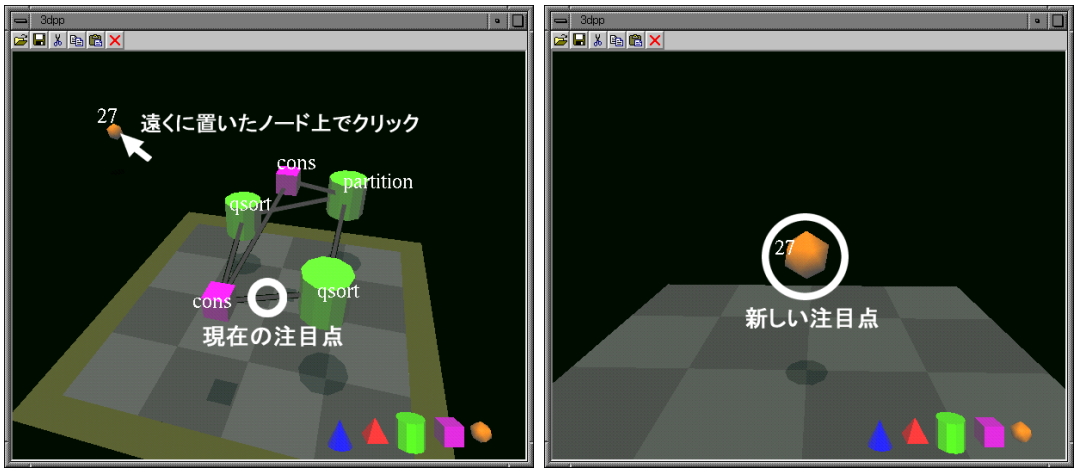
4.3.2 入力するリストの作成

プロセス *qsort* に入力するリストを作成する。

従来の “3D-PP” では、回転の中心が固定であったため、この例のように、リストやプロセスを全て作成することは困難であったが、新しい “3D-PP” では注目点の変更が可能となったため、いくつかの部分空間を用いることで、複数の部分的なプログラムを記述していくことができるようになっている。

リストを先程のプロセス *qsort* とは違う部分空間で作成するために、まずリストを構成するノードのうち一つを画面内に作成し、適当な離れた位置に配置する。次に、図 4.8a のように配置したノードをクリックして選択することにより、視点とデスクトップ的地面を移動させる。以後、視点は図 4.8b のように選択したノードを中心と

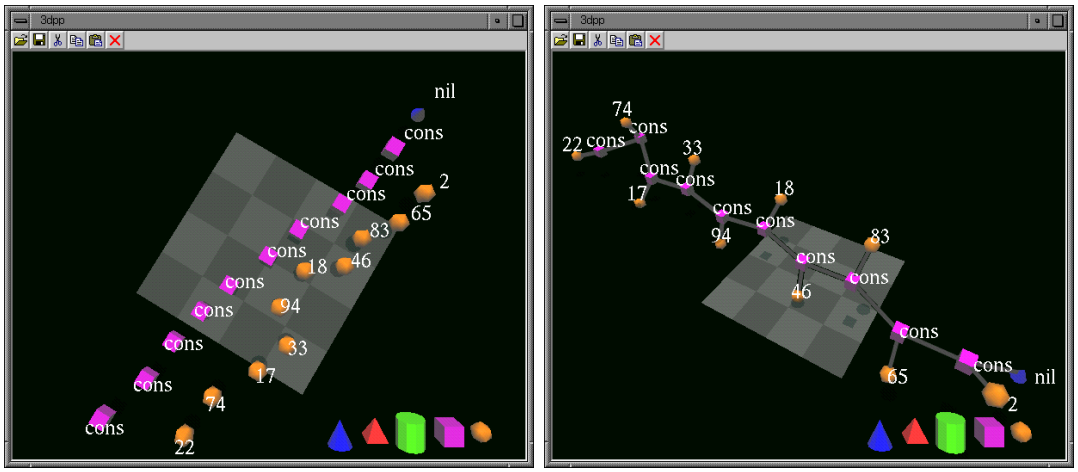
した回転運動が行えるようになる。また、移動の際は、一度目標を選択するだけで、後は自動的に視点の移動を行う。



(a) 適当な位置にノードを配置　　(b) 注目点を変更

図 4.8: 入力するリストの作成過程 1

この状態で、改めて入力するリストに必要なノードを図 4.9a のようにそれぞれ用意する。さらに図 4.9b のように各ノードをエッジて結線することにより、入力するリストが作成される。この際、作成されるリストは比較的大きくなるが、この視点移動操作を拡張した “3D-PP” では、視点と注目点との距離を大きくすることが可能であるため、以前の “3D-PP” で問題であった、全てを一度に見ることができない、という点が解決されている。

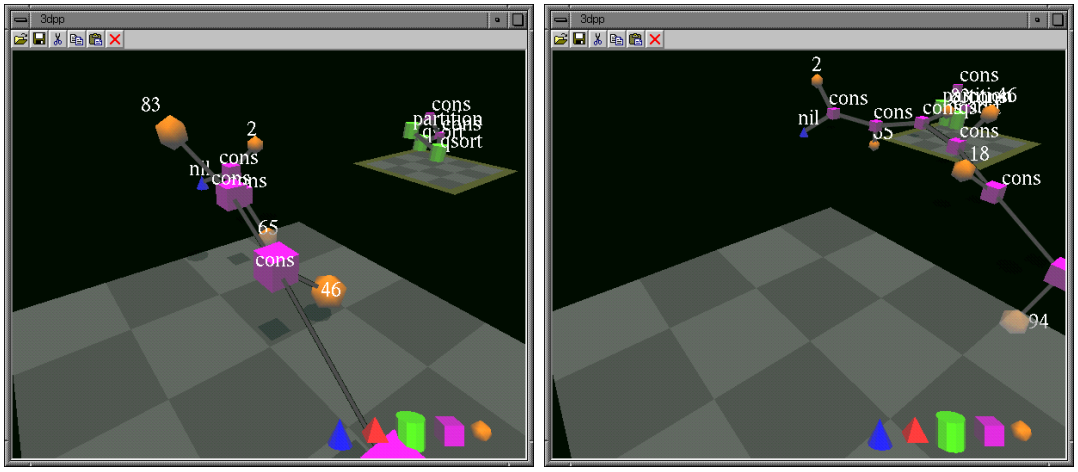


(a) リストに必要なノードの用意　　(b) エッジによる結線

図 4.9: 入力するリストの作成過程 2

4.3.3 プロセス *qsort* に入力

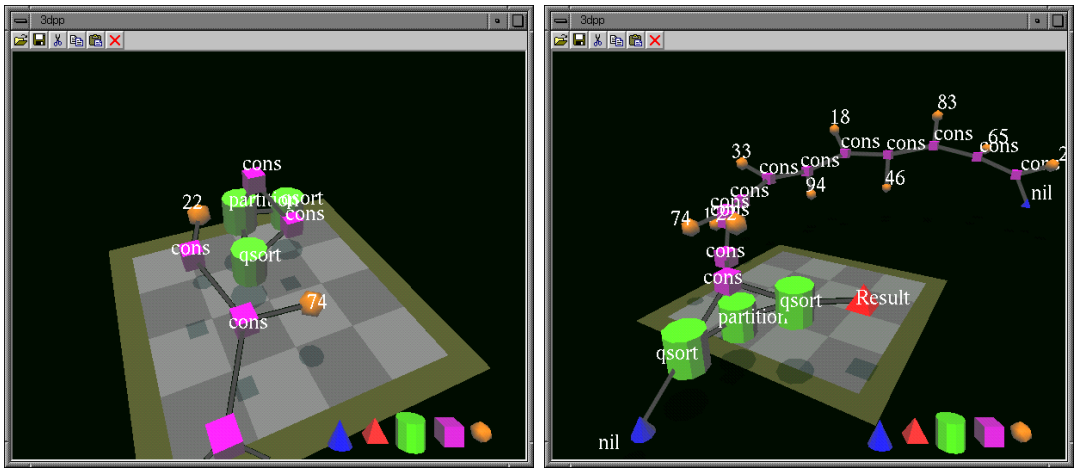
図 4.10a のように最初に作成したプロセス *qsort* が視界に入るように視点を回転し、先程作成したリストをプロセス *qsort* の方向に移動する。自動レイアウト機能により、一つのノードを移動するだけで、リストに含まれる全てのノードを移動することが可能である（図 4.10b）。



(a) プロセス *qsort* を視界に入れる　　(b) リストを移動

図 4.10: リストをプロセス *qsort* に入力する過程 1

次に、注目点を再び最初の位置まで戻すために、ランドマーク的地面を選択し、視点を移動させる（図 4.11a）。



(a) 注目点を最初の位置に変更　　(b) プロセス *qsort* に入出力を与える

図 4.11: リストをプロセス *qsort* に入力する過程 2

最後にリストをプロセス *qsort* とエッジで結線することにより、プロセス *qsort* に入力を与える。また同様にノード *nil* を作成し、プロセス *qsort* に入力として与える。

最後に結果を格納するノードである *Result* を用意し、プロセス *qsort* の出力として結線する。

これらの操作により、図 4.11b のようなビジュアルプログラムが作成された。

4.4 考察

現在の “3D-PP” では、まだ含有関係やノードからの引数等を完全に表現する能力がないため、実際に実用的なビジュアルプログラムの全てを表現することはできない。

しかし、視点移動操作の拡張を行ったことで、従来の “3D-PP” に比べて格段に大規模なビジュアルプログラムを記述できるようになった。これにより “3D-PP” は実用的な 3 次元 VPS に一歩近づいたと考えられる。

第 5 章

関連研究

３次元表示に基づく VPS の関連研究として、 PrologSpace [18] や VisuaLinda [15] がある。 PrologSpace は論理型言語 Prolog をベースにしている。 VisuaLinda は並列言語 Linda をベースにしている。 PrologSpace は VisualProlog を用いて実装されている VPS である。 VisualProlog は X ウィンドウ・ウィジェット・３次元表示・アニメーション・オーディオ等がサポートされている Prolog の一種である。 PrologSpace は最低でも 4 つのサブウィンドウから構成されている。これに対し、 “3D-PP” は 1 つのウィンドウのみによって構成されている。

VisuaLinda は並列言語 Linda のプログラム実行の状態を視覚化する VPS であり、複数のプロセスの動作を時間軸に沿って視覚化されている。 VisuaLinda での視点の変更は 3 つのスクロールバーを操作することによって行われる。それぞれのスクロールバーは３次元空間の x 軸 y 軸 z 軸に対応している。スクロールバーを操作して視点を変更することは間接的な操作であるため、ユーザにとっては直観的ではない。これに対し、 “3D-PP” ではマウスで空間内のドラッグ操作などによって直接的に視点を変更することが可能である。また、 VisuaLinda はプログラムの実行過程を視覚化するのみであり、プログラムの編集過程は視覚化しない。これに対し、 “3D-PP” ではプログラムの編集過程と実行過程の両方を視覚化することができる。

また、３次元視覚化における視点の移動手法の関連研究としては、 Information Visualizer プロジェクトの Point of Interest 方式 [12] や Path Drawing [19] があげられるる。

Point of Interest 方式は、ユーザが着目点を選択すると着目オブジェクト方向に視点が移動する。これは “3D-PP” の移動操作の一つである移動先の画面からの直接選択による視点の移動手法と近いものである。

Path Drawing はユーザが希望する移動経路を２次元の画面上にマウスを用いて直接自由ストロークとして描き、その上を辿るようにして視点が３次元空間内を移動す

るものである。この手法の最も大きな特徴は、移動の目的地だけでなく、移動の際の経路も同時に指定できることである。これは特に3次元マップ等のウォークスルー環境における視点の移動操作方法として非常に有効である。

第 6 章

まとめ

次世代のコンピューティング環境の１つとして３次元コンピューティング環境に着目し、３次元ビジュアルプログラミングシステム “3D-PP” の開発を進めている。

本論文では、ビジュアルプログラミングシステムを３次元化する上で解決しなければならない問題として、仮想３次元空間内における視点の移動操作がユーザに大きな負担を与えてしまう、という問題を取り上げた。またその原因がユーザが仮想３次元空間とのインタラクションに用いるデバイスが２次元であることによる、自由度の不足であることを述べ、その解決方法として、３次元ビジュアルプログラミングシステムでの利用に特化した視点の移動操作を提案した。さらにこの視点移動操作を “3D-PP” に実装し、従来の “3D-PP” で問題であった、その貧弱な視点移動操作によって非常に狭く制限されていた “3D-PP” で利用可能な空間を、操作を複雑にすることなく拡張した。さらに地面という考えを用いて３次元空間の情報をユーザに与えることで空間の把握を助ける “3D-PP” の機能を拡張し、実装した。

謝辞

本研究にあたり、田中二郎先生、志築文太郎先生からは、貴重なご指導、助言などをいただきました。心より感謝致します。

また田中研究室のみなさんからは、ゼミ等を通じて多くのアドバイスを頂きました。特に 3D-PP グループの小川徹さん、中須正人さん、劉学軍さん、山田英仁さんとは、卒業研究にあたり、多くの有益な議論を得られました。ここに感謝の意を表します。

最後に、自分を精神的、物理的に支えていただいた両親をはじめ家族やすべての友人に心より感謝致します。

参考文献

[1] P. T. Cox, F. R. Giles and T. Pietrzykowski: Prograph: A Step towards Liberating Programming from Textual Conditioning, *1989 IEEE Workshop on Visual Languages*, Rome, pp.150–156, 1989.

[2] Ken Kahn: ToonTalk — An Animated Programming Environment for Children, *Journal of Visual Languages and Computing*, pp.197–217, June, 1996.

[3] KLIC 講習会テキスト -KL1 言語編 -, 財団法人 新世代コンピュータ技術開発機構 作成, 財団法人 日本情報処理開発協会開発研究室 改訂, 1995.

[4] 大芝 崇: 直観的な操作に基づく 3 次元モデリングツールと 3 次元ビジュアルプログラミングシステムの構築, 平成 11 年度 筑波大学大学院修士課程理工学研究科修士論文, 2000.

[5] 宮下 貴史: 三次元ビジュアルプログラミング編集環境の構築, 平成 11 年度 筑波大学大学院修士課程理工学研究科修士論文, 2000.

[6] 宮城 幸司: 三次元ビジュアルプログラミング環境の構築, 平成 10 年度 筑波大学大学院修士課程理工学研究科修士論文, 1999.

[7] 宮城 幸司, 大芝 崇, 田中 二郎: 三次元ビジュアル・プログラミング・システム 3D-PP, 日本ソフトウェア科学会第 15 回大会論文集, pp.125–128, 1998.

[8] 岡田 潤: 実時間三次元映像における注目度情報による視点の自動制御, 平成 8 年度 筑波大学第三学群情報学類卒業論文

[9] C.Cruz-Neira, D.J. Sandin, T.A. DeFanti: Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE, *Computer Graphics*, pp.135-142, 1993.

[10] 小池 英樹: インタラクティブ 3 次元情報視覚化, コンピュータソフトウェア, Vol.11, No.6, pp.20-31, 1994.

[11] Darken, R. P. and Sibert, J. L.: A Toolset for Navigation in Virtual Environments, *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '93)*, ACM Press, 1993.

[12] Mackinlay, J. D., Card, S. K., and Robertson, G. G.: Rapid Controlled Movement through a Virtual 3D Workspace, *SIGGRAPH 1993 Conference Proceedings*, pp.171-176, 1990

[13] Takashi Oshiba and Jiro Tanaka: "3D-PP": Visual Programming System with Three-Dimensional Representation, In *Proceeding of International Symposium on Future Software Technology (ISFST '99)*, pp.61–66, Nanjing, China, October 27th to 29th, 1999.

[14] Takashi Oshiba and Jiro Tanaka: "3D-PP": Three-Dimensional Visual Programming System, In *Proceeding of 1999 IEEE Symposium on Visual Languages (VL'99)*, pp.189–190, IEEE Computer Society Press, Tokyo, Japan, September 13th to 17th, 1999.

[15] 高田 哲司, 小池 英樹: VisuaLinda: 並列言語 Linda のプログラムの実行状態の３次元視覚化, 竹内彰一編, インタラクティブシステムとソフトウェア II, 日本ソフトウェア科学会 WISS'94, 近代科学社, pp.215–223, 1994.

[16] Masashi Toyoda, Buntarou Shizuki, Shin Takahashi, Satoshi Matsuoka and Etsuya Shibayama: Supporting Design Patterns in a Visual Parallel Dataflow Programming Environment, In *Proc. 1997 IEEE Symposium on Visual Languages*, 1997.

[17] Kazunori Ueda: Guarded Horn Clauses, *ICOT Technical Report*, TR-103, Institute for New Generation Computer Technology, 1985.

[18] Masoud Yazdani and Lindsey Ford: Reducing the Cognitive Requirements of Visual Programming, In *Proceeding of 1996 IEEE Symposium on Visual Languages (VL'96)*, pp.225–262, 1996.

[19] Takeo Igarashi, Rieko Kadobayashi, Kenji Mase, Hidehiko Tanaka: Path Drawing for 3D Walkthrough, *11th Annual Symposium on User Interface Software and Technology, UIST'98*, pp.173-174, 1998.

[20] T. G. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, Y. Harvill: A Hand Gesture Interface Device, *Proceedings of the ACM CHI + GI '87 Conference on Human Factors in Computing Systems and Graphics Interface*, ACM Press, pp.189-192, 1987.

付録 A

システムのソースコード

“3D-PP” のソースコードの一部を付録として添付する.

glbox.cpp

```
/*********************************************************************************
** "glbox.cpp" in Three-Dimensional Visual-Programing System "3D-PP"
**
** This source code made first by T.Miyashita in 1999 is extended and written by K.Kai.
**
** And the OpenGL code is mostly borrowed from Brian Pauls "spin" example
** in the Mesa distribution.
*********************************************************************************/
#include <qapplication.h>
#include <qkeycode.h>
#include <qlayout.h>

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "glbox.h"
#include "rmdfile.h"
#include "rmddraw.h"
#include "polycalc.h"
#include "material.h"
#include "select.h"
#include "normcalc.h"
#include "screen.h"

#include "models.h"
#include "models-gui.h"
#include "layout.h"

#include <qobject.h>
#include <qtimer.h>
#include <GL/glut.h>
```

```
void drawBitmapFont(GUINodeCore* guinode){
  if(guinode->ID>6 &&  !(guinode->EdgeStartID != 0)){
    glDisable(GL_LIGHTING);
    glDisable(GL_LIGHT0);

    int i;
    char *text;

    glColor4f(1,1,1,1);
    glRasterPos3f(300 * cos(PI*2/360*camSpin.y) * sin(PI*2/360*-(camSpin.x+15)),
  -1 * 300 * sin(PI*2/360*camSpin.y),
  -1 * 300 * cos(PI*2/360*camSpin.y) * cos(PI*2/360*-(camSpin.x+20)));

    for(i=0;i<(int)strlen(guinode->guilabel);i++)
      {
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,guinode->guilabel[i]);
      }

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
  }
}

static vector2f spin, now,zoom,zoom_tmp;

//ラベル書き換え用ダイアログボックス
GLBox::GLBox( QWidget* parent, const char* name )
  : QGLWidget( parent, name )
{
  popup1 = new QFrame( this ,0, WType_Popup);
  popup1->setFrameStyle( QFrame::Panel | QFrame::Raised );
  popup1->setLineWidth( 4 );
  popup1->resize(150,100);
  textbox = new QLineEdit( popup1 );
  connect( textbox, SIGNAL( returnPressed() ), this, SLOT(closeTextwindow()  ) );
  textbox->setGeometry(10,10, 130, 30);
  textbox->setFocus();

  QPushButton *tmpB = new QPushButton("ok", popup1);
  connect( tmpB, SIGNAL( clicked() ), this, SLOT( closeTextwindow() ) );
  tmpB->setGeometry(10, 50, 60, 30);

  QPushButton *tmpC = new QPushButton("cancel", popup1);
  connect( tmpC, SIGNAL( clicked() ), this, SLOT( cancelTextwindow() ) );
  tmpC->setGeometry(80, 50, 60, 30);

  scale = 1.25; // default object scale
  object = 0;
  last.x = 0; last.y = 0;
  setMouseTracking( TRUE );
```

```
  mouse_button_left = FALSE;
  mouse_button_middle = FALSE;
  mouse_button_right = FALSE;

  QTimer *timer = new QTimer( this );
  connect( timer, SIGNAL(timeout()), this, SLOT( NodeLayout() ) );
  timer->start(0);
}

GLBox::~GLBox()
{
    glDeleteLists( object, 1 );
}


void GLBox::button1Clicked(QMouseEvent *e){

  popup1->move( e->globalPos ()  );
  popup1->show();

}

void GLBox::closeTextwindow(){

  popup1->close();
  strcpy(gui_pick->guilabel,textbox->text());

}

void GLBox::cancelTextwindow(){

  popup1->close();

}

void GLBox::mousePressEvent(QMouseEvent *e) {

  //マウスのボタンを押した

  NodeDescriptor  d;
  NodeID nid;
  GUINodeCore* guiedge;
  //内部情報記述テスト
  if (e->button() == LeftButton) { //■左ボタン

    int polynum;

    pushground=false;
    touchground=false;

    //一度地面に触れているかチェックする
```

```
    polynum = CalcTouchPolygon2(gui_list_start,
e->x(), e->y(), gui_world,
&gui_operate, &gui_pick);

    if(gui_operate != NULL && gui_operate->ID == 1){
      pushground=true;
    }

    polynum = CalcTouchPolygon2(gui_list_start->NodeNext,
e->x(), e->y(), gui_world,
&gui_operate, &gui_pick);


    if (gui_operate == NULL) {
      past.x = e->x(); past.y = e->y();
    }
    else {

      pushground=false;    //地面越しの作業を可能にするため

      moved = FALSE;
      printf("ID %d\n",gui_operate->ID);
      printf("iconShape %d\n",gui_operate->iconShape);
      printf("GrpID==> %d\n",gui_operate->GrpID);
      printf("Force_X==> %d\n",gui_operate->Fx);

      if (gui_pick != NULL) {
if (gui_pick->parent == gui_operate) {
  gui_operate = gui_pick;
}
      }

      // 3次元アイコンをクリック → 物体生成
      NodeID tmpID = gui_operate->ID;

      vector3d tmpVector =gui_operate->shape.vector;

      if (--tmpID <= ICON_MAX) { //地面があるので --tmpID する
//新しいノードの箱（中身は空）を生成
printf("Node Create!!\n");
gui_operate = new GUINodeCore;
gui_list_end->NodeNext = gui_operate;
gui_operate->NodeNext = NULL;
gui_operate->NodePrev = gui_list_end;
gui_list_end = gui_operate;
NodeCore* n;
GoalNode* aaa;
switch (tmpID) {

case ICON_ATOM:
  icon->AddNodeToGUINodeCore(gui_operate, ICON_ATOM,1.0,
```

```
    LOOKAT/2  * sin(PI*2/360*-(camSpin.x-10-(camSpin.y*10/90)))+viewPoint.x,
    -100+viewPoint.y,
    -1 * LOOKAT/2 * cos(PI*2/360*-(camSpin.x-10-(camSpin.y*10/90)))+viewPoint.z
    );
  sprintf(gui_operate->guilabel,"atom");
  break;

case ICON_LIST:
  icon->AddNodeToGUINodeCore(gui_operate, ICON_LIST,1.0,
    LOOKAT/2  * sin(PI*2/360*-(camSpin.x-15-(camSpin.y*15/90)))+viewPoint.x,
    -100+viewPoint.y,
    -1 * LOOKAT/2 * cos(PI*2/360*-(camSpin.x-15-(camSpin.y*15/90)))+viewPoint.z
    );
  sprintf(gui_operate->guilabel,"list");
  break;

case ICON_GOAL:
  icon->AddNodeToGUINodeCore(gui_operate, ICON_GOAL,1.0,
    LOOKAT/2  * sin(PI*2/360*-(camSpin.x-20-(camSpin.y*15/90)))+viewPoint.x,
    -100+viewPoint.y,
    -1 * LOOKAT/2 * cos(PI*2/360*-(camSpin.x-20-(camSpin.y*15/90)))+viewPoint.z
    );
  sprintf(gui_operate->guilabel,"goal");
  break;

case ICON_BLTIN:
  icon->AddNodeToGUINodeCore(gui_operate, ICON_BLTIN,1.0,
    LOOKAT/2  * sin(PI*2/360*-(camSpin.x-30-(camSpin.y*15/90)))+viewPoint.x,
    -100+viewPoint.y,
    -1 * LOOKAT/2 * cos(PI*2/360*-(camSpin.x-30-(camSpin.y*15/90)))+viewPoint.z
    );
  sprintf(gui_operate->guilabel,"bltin");
  break;

case ICON_VAR:
  icon->AddNodeToGUINodeCore(gui_operate, ICON_VAR,1.0,
    LOOKAT/2  * sin(PI*2/360*-(camSpin.x-35-(camSpin.y*20/90)))+viewPoint.x,
    -100+viewPoint.y,
    -1 * LOOKAT/2 * cos(PI*2/360*-(camSpin.x-35-(camSpin.y*20/90)))+viewPoint.z
    );
  sprintf(gui_operate->guilabel,"Var");
  break;
}
     }

     start.x = e->x(); start.y=e->y();
     stock.x = gui_operate->shape.vector.x;
     stock.y = gui_operate->shape.vector.y;
     SetModelView(&(gui_world->shape), &(gui_world->shape));
     glGetDoublev(GL_MODELVIEW_MATRIX,(GLdouble*)&matrixst);
     tnormal0.x = 0.0;
```

```
      tnormal0.y = -1.0;
      tnormal0.z = 0.0;
      targetpos0 = targetpos;
      CalcCrossPosition(&tnormal0, &targetpos0, &matrixst,
e->x(), e->y(), &targetpos1);

      gui_operate->DrawMode(FALSE, FALSE, TRUE);
      updateGL();
    }
    mouse_button_left = TRUE;
  }

  if(e->button() == MidButton){ //■中ボタン

    int polynum = CalcTouchPolygon(gui_list_start->NodeNext, e->x(), e->y(),
   &gui_operate,
   gui_world);
    if((void *)gui_operate == NULL){
      past.x = e->x(); past.y = e->y();
      moved = FALSE;
    }
    else{

      NodeEdgeID = 0;//初期化
      NodeEdgeID = gui_operate->ID;//エッジの始点となるIDを抽出

      moved = FALSE;
      start.x = e->x();start.y=e->y();
      stock.x = gui_operate->shape.vector.x;
      stock.y = gui_operate->shape.vector.y;
      gui_operate->DrawMode(FALSE, FALSE, TRUE);
      updateGL();
    }
    mouse_button_middle = TRUE;
  }

  if(e->button() == RightButton){ //■右ボタン
    zoom_tmp.y=e->y();

    int polynum = CalcTouchPolygon2(gui_list_start->NodeNext, e->x(), e->y(),
    gui_world,
    &gui_operate, &gui_pick);
    if(gui_operate == NULL){
      past.x = e->x(); past.y = e->y();
    }
    else{
      moved = FALSE;

      if(gui_pick != NULL){
if(gui_pick->parent == gui_operate){
  gui_operate = gui_pick;
```

```
}
      }

      start.x = e->x();start.y=e->y();
      stock.x = gui_operate->shape.vector.x;
      stock.y = gui_operate->shape.vector.y;
      SetModelView(&(gui_world->shape), &(gui_world->shape));
      glGetDoublev(GL_MODELVIEW_MATRIX,(GLdouble*)&matrixst);

      tnormal0.x = -gui_world->shape.matrix.m[2];
      tnormal0.y = -gui_world->shape.matrix.m[6];
      tnormal0.z = -gui_world->shape.matrix.m[10];

      targetpos0 = targetpos;
      CalcCrossPosition(&tnormal0, &targetpos0, &matrixst, e->x(), e->y(),
&targetpos1);
      gui_operate->DrawMode(FALSE, FALSE, TRUE);
      updateGL();
    }
    mouse_button_right = TRUE;
  }
}


void GLBox::mouseReleaseEvent(QMouseEvent *e) {
  //マウスのボタンをはなしたときの処理
  int DoubleID,ID1,ID2;
  GUINodeCore* guinode;
  GUINodeCore* tmpguinode;
  GUINodeCore* grpidnode;
  GUINodeCore* guiedge;
  if (e->button() == LeftButton) { //■左ボタン
    mouse_button_left = FALSE;
    last.x = end.x; last.y = end.y;

    if(pushground && !touchground ){ // 地面左クリックなら原点に向かって移動

      changeViewPoint(0,0,0,2500);

    }else if (gui_operate != NULL) { // 何かに触れている

      if (moved) { // マウスは動いた
if (gui_operate->focus) {
  gui_operate->DrawMode(FALSE, TRUE, FALSE);
}
else {
  gui_operate->DrawMode(TRUE, FALSE, FALSE);
}
updateGL();
gui_operate = NULL;
      }
```

```
      else { // 動いていない

gui_operate->FocusMode(FALSE);
gui_operate->DrawMode(TRUE, FALSE, FALSE);

gui_operate->ChildUpdate();

changeViewPoint(gui_operate->shape.vector.x,
gui_operate->shape.vector.y,
gui_operate->shape.vector.z,
1500);                        //視点の移動処理

updateGL();
gui_operate = NULL;
      }
    }
  }

  if (e->button() == MidButton) { //■中ボタン

    mouse_button_middle = FALSE;
    last.x=end.x; last.y=end.y;
    if (gui_operate != NULL && gui_pick != NULL
&& !gui_operate->shape.IconOrNot
&& !gui_pick->shape.IconOrNot) { // 何かに触れている

      if(!moved){ //中ボタンでなにかノードをクリックー＞ラベルの変更ダイアログの出現

textbox->setText(gui_pick->guilabel);
textbox->setSelection ( 0, strlen(gui_pick->guilabel));

button1Clicked(e);
      }
      if(!(gui_operate->focus)){
gui_operate->DrawMode(TRUE, FALSE, FALSE);
printf("MOUSE_TEST\n");
      }else {
gui_operate->DrawMode(FALSE, TRUE, FALSE);
      }
      moved = FALSE;
      //二重結線防止処理
      DoubleID = 0;
      for(guinode = gui_list_start->NodeNext;
  guinode != NULL;
  guinode = guinode->NodeNext){
if( guinode->EdgeStartID != 0 && guinode->EdgeEndID != 0)
  {
    if(  (guinode->EdgeStartID == NodeEdgeID && guinode->EdgeEndID == gui_pick->ID)
 ||(guinode->EdgeStartID == gui_pick->ID && guinode->EdgeEndID == NodeEdgeID)){
      //結線削除処理
```

```
      guinode->EdgeStartID = 0;
      guinode->EdgeEndID = 0 ;
      tmpguinode = guinode;

      guinode = tmpguinode->NodePrev;
      guinode->NodeNext = tmpguinode->NodeNext;
      if(tmpguinode->NodeNext != NULL){
guinode = tmpguinode->NodeNext;
guinode->NodePrev = tmpguinode->NodePrev;
      }
      else{
gui_list_end = guinode;
      }
      delete tmpguinode;
      DoubleID = 1;
      break;
    }
  }
      }

      if(DoubleID == 0 && NodeEdgeID != gui_pick->ID){
guiedge = new GUINodeCore;
gui_list_end->NodeNext = guiedge;
guiedge->NodeNext = NULL;
guiedge->NodePrev = gui_list_end;
gui_list_end = guiedge;
guiedge->EdgeStartID = NodeEdgeID;//エッジの始点となる ID を抽出

guiedge->EdgeEndID = gui_pick->ID;//エッジの終点となる ID を抽出

printf("EdgeID!!!%d\n",guiedge->ID);
printf("EdgeID->EdgeStart==>%d\n",guiedge->EdgeStartID);
printf("EdgeID->EdgeEnd==>%d\n",guiedge->EdgeEndID);
printf("Not!!DoubleID!!!\n");
      }
      updateGL();
      printf("MiddleButton_MOUSERELE=>pick_ID= %d,op_ID= %d\n",gui_pick->ID,gui_operate->ID);

      printf("gui_operate=>%d,gui_pick=>%d\n",gui_operate->ID,gui_pick->ID);
      if(gui_operate->ID > gui_pick->ID){
gui_operate->GrpID = gui_operate->ID;
gui_pick->GrpID = gui_operate->ID;
      }
      else{
gui_operate->GrpID = gui_pick->ID;
gui_pick->GrpID = gui_pick->ID;
      }
      gui_operate = NULL;
    }
    else{
```

```
      if(gui_operate != NULL){
gui_operate->DrawMode(TRUE, FALSE, FALSE);
      }
      updateGL();
      gui_operate = NULL;
    }
  }

  if (e->button() == RightButton) { //■右ボタン
    mouse_button_right = FALSE;
    last.x=end.x; last.y=end.y;

    if (gui_operate != NULL) { // 何かに触れている
      if (moved) { // マウスは動いた
if (gui_operate->focus) {
  gui_operate->DrawMode(FALSE, TRUE, FALSE);
}
else {
  gui_operate->DrawMode(TRUE, FALSE, FALSE);
}
updateGL();
gui_operate = NULL;
      }

      else { // 動いていない
//ノードを右クリック―>そのノードが含まれるノード群の座標の中心に移動
int thisGrpID;
thisGrpID=gui_operate->GrpID;
vector3d GrpPosMax=gui_operate->shape.vector;
vector3d GrpPosMin=gui_operate->shape.vector;

for(guinode = gui_list_start->NodeNext;
    guinode != NULL;
    guinode = guinode->NodeNext){

  if(guinode->GrpID == gui_operate->GrpID)
    {
      if(GrpPosMax.x<guinode->shape.vector.x)
GrpPosMax.x=guinode->shape.vector.x;
      if(GrpPosMax.y<guinode->shape.vector.y)
GrpPosMax.y=guinode->shape.vector.y;
      if(GrpPosMax.z<guinode->shape.vector.z)
GrpPosMax.z=guinode->shape.vector.z;

      if(GrpPosMin.x>guinode->shape.vector.x)
GrpPosMin.x=guinode->shape.vector.x;
      if(GrpPosMin.y>guinode->shape.vector.y)
GrpPosMin.y=guinode->shape.vector.y;
      if(GrpPosMin.z>guinode->shape.vector.z)
GrpPosMin.z=guinode->shape.vector.z;
    }
```

```
}

gui_operate->FocusMode(FALSE);
gui_operate->DrawMode(TRUE, FALSE, FALSE);

gui_operate->ChildUpdate();

double tmp=1500/1.2;
if(GrpPosMax.x-GrpPosMin.x>tmp)
  tmp=GrpPosMax.x-GrpPosMin.x;
if(GrpPosMax.y-GrpPosMin.y>tmp)
  tmp=GrpPosMax.y-GrpPosMin.y;
if(GrpPosMax.z-GrpPosMin.z>tmp)
  tmp=GrpPosMax.z-GrpPosMin.z;

changeViewPoint((GrpPosMax.x+GrpPosMin.x)/2,
(GrpPosMax.y+GrpPosMin.y)/2,
(GrpPosMax.z+GrpPosMin.z)/2,tmp*1.2);

updateGL();
gui_operate = NULL;
      }
    }
  }
}

void GLBox::mouseMoveEvent(QMouseEvent *e){

  //マウスを動かしたときの処理
  vector3d v1;
  GUINodeCore* gui_temp;
  now.x = e->x();now.y = e->y();
  moved = TRUE;
  int polynum;

  polynum = CalcTouchPolygon2(gui_list_start, e->x(), e->y(),
      gui_world, &gui_pick, &gui_temp);

  if(gui_pick != NULL && gui_pick->ID == 1){
    touchground=true;
  }

  polynum = CalcTouchPolygon2(gui_list_start->NodeNext, e->x(), e->y(),
      gui_world, &gui_pick, &gui_temp);

  if(gui_pick == NULL){
    updateGL();
  }
  else{
    if(gui_temp != NULL){
      if(gui_temp->parent == gui_pick){
```

```
gui_pick = gui_temp;
      }
    }
    CalcTouchPosition(&(gui_pick->shape), polynum, e->x(), e->y(),
      &targetpos, &tnormal, &(gui_world->shape));
    gui_pick->wireframe = TRUE;
    updateGL();
    gui_pick->wireframe = FALSE;
  }

  if(mouse_button_left){ //左ボタンを押しているなら
    if(gui_operate == NULL){// 何も触っていない → 背景を回す
      spin.x = last.x + 360.0 * (now.x - past.x)/500;
      spin.y = last.y + 360.0 * (now.y - past.y)/500;

      if(spin.x != end.x || spin.y != end.y){
end.x=spin.x;end.y=spin.y;
camSpin.x=spin.x;
camSpin.y=spin.y;

updateView();
      }
    }
    else { // 何かに触れているー>ノードの xz 平面上の移動処理
      CalcCrossPosition(&tnormal0, &targetpos0, &matrixst,
(int)now.x, (int)now.y, &v1);

      gui_operate->shape.vector.x += (v1.x - targetpos1.x)*LOOKAT/2000;
      gui_operate->shape.vector.y += (v1.y - targetpos1.y)*LOOKAT/2000;
      gui_operate->shape.vector.z += (v1.z - targetpos1.z)*LOOKAT/2000;

      targetpos1 = v1;
      for(int i=0;i<8;i++){ // 子ノードも一緒に移動させる
if(gui_operate->child[i]!=NULL){
  gui_operate->child[i]->shape.vector = gui_operate->shape.vector;
}
      }
      updateGL();
    }
  }

  else if(mouse_button_middle){ //中ボタンを押している
    if((void*)gui_operate == NULL){// 何も触っていない
      //処理未定義
    }
    else{
      updateGL();
    }
  }

  else if(mouse_button_right){ //右ボタンを押している
```

```
    if(gui_operate == NULL){// 何も触っていない ー>視点と注目点との距離の変更
      zoom.y = 20 * (now.y-zoom_tmp.y) ;
      zoom_tmp.y=now.y;

      if(zoom.y!=0){

if(LOOKAT+zoom.y>=100)LOOKAT+=zoom.y;  // 距離を最短を設定し,近づきすぎを防ぐ

updateView();
      }
    }
    else { // 何かに触れているー>ノードの xy 平面上の移動処理
      CalcCrossPosition(&tnormal0, &targetpos0, &matrixst,
(int)now.x, (int)now.y, &v1);

      gui_operate->shape.vector.x += (v1.x - targetpos1.x)*LOOKAT/2500;
      gui_operate->shape.vector.y += (v1.y - targetpos1.y)*LOOKAT/2500;
      gui_operate->shape.vector.z += (v1.z - targetpos1.z)*LOOKAT/2500;

      targetpos1 = v1;
      for(int i=0;i<8;i++){// 子ノードも一緒に移動
if(gui_operate->child[i]!=NULL){
  gui_operate->child[i]->shape.vector = gui_operate->shape.vector;
}
      }
      updateGL();
    }
  }
  else {
  }
}


void GLBox::paintGL(){

  GUINodeCore* guinode;
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  vector3d first_axis, second_axis;

  CubeDraw(); //背景を描画

  for (guinode = gui_list_start->NodeNext; //透明じゃないものを描く
       guinode != NULL;
       guinode = guinode->NodeNext) {
    guinode->EdgeDraw(guinode,gui_world,gui_list_start,matrixst);

    if (guinode->surface) {
      ModelDraw(guinode, gui_world);

      drawBitmapFont(guinode);            //ノードの文字情報の描画
```

```
  }
}

glMatrixMode(GL_MODELVIEW);
matrix16d vmatrix;
glGetDoublev(GL_MODELVIEW_MATRIX,(GLdouble*)&vmatrix);

//線の描画
glPushMatrix();
glLoadIdentity();

glRotatef(VROT, 1.0, 0.0, 0.0);

glTranslated(gui_list_start->shape.vector.x,
     gui_list_start->shape.vector.y,
     gui_list_start->shape.vector.z);

SetModelView(&(gui_world->shape), &(gui_world->shape));
glGetDoublev(GL_MODELVIEW_MATRIX,(GLdouble*)&matrixst);

glPopMatrix();

//デスクトップ的地面の描画
if(desktop->shape.vector.x != gui_list_start->shape.vector.x
   || desktop->shape.vector.y != gui_list_start->shape.vector.y
   || desktop->shape.vector.z != gui_list_start->shape.vector.z){
  glDisable(GL_CULL_FACE);
  glPolygonMode(GL_FRONT,GL_FILL);
  glPolygonMode(GL_BACK,GL_FILL);
  SetModelView(&(desktop->shape), &(gui_world->shape));
  glCallList(desktop->displaylist1);
  glEnable(GL_CULL_FACE);
}

glDisable(GL_CULL_FACE);
glPolygonMode(GL_FRONT,GL_FILL);
glPolygonMode(GL_BACK,GL_FILL);
SetModelView(&(gui_list_start->shape), &(gui_world->shape));
glCallList(gui_list_start->displaylist1);
glEnable(GL_CULL_FACE);

for(guinode = gui_list_start->NodeNext; //透明なものを描く
    guinode != NULL;
    guinode = guinode->NodeNext){
  if(guinode->transparent){
    ModelDraw(guinode, gui_world);
  }
}
for(guinode = gui_list_start->NodeNext; //ワイヤフレームを描く
    guinode != NULL;
    guinode = guinode->NodeNext){
```

```
    if(guinode->wireframe){
      ModelDraw(guinode, gui_world);
    }
  }

  glDisable(GL_LIGHTING); //影を描く
  glColor4ub(0,16,16,64);

  for(guinode = gui_list_start->NodeNext;
      guinode != NULL;
      guinode = guinode->NodeNext){
    if(!guinode->invisible && !guinode->shape.IconOrNot){
      ShadowDraw(guinode, gui_world);
    }
  }
  glDepthMask(GL_TRUE);
  glColor3ub(255,255,255);
  glEnable(GL_LIGHTING);
}

/*!
  Set up the OpenGL rendering state, and define display list
  */

void GLBox::initializeGL()
{
  InitScreen();

  glClearColor( 0.05, 0.1, 0.05, 0.0 ); //背景色を黒に指定
  glShadeModel( GL_SMOOTH );

  SetLight_MaterialParameter();
  SetLightModel();
  SetLight();
  SetMaterial();

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_NORMALIZE);

  glPolygonMode(GL_FRONT,GL_FILL);
  glPolygonMode(GL_BACK,GL_FILL);
  glCullFace(GL_BACK);                  //背面ポリゴンの廃棄
  glEnable(GL_CULL_FACE);

  glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);

  //直線ポリゴン / 点ポリゴンの設定
  glLineWidth(5);
```

```
  glHint(GL_LINE_SMOOTH_HINT,GL_FASTEST);
  glEnable(GL_LINE_SMOOTH);
  glPointSize(5.0);
  glHint(GL_POINT_SMOOTH_HINT,GL_FASTEST);
  glEnable(GL_POINT_SMOOTH);

  // 光源行列を初期化
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glRotatef(-45.0,5.0,0.0,0.0);
  glLightfv(GL_LIGHT0,GL_POSITION,Light0.position);

  glClear( GL_COLOR_BUFFER_BIT );

  object = makeObject(); // Generate an OpenGL display list

}

/*!
  Set up the OpenGL view port, matrix mode, etc.
*/

void GLBox::resizeGL( int w, int h ){

  glViewport( 0, 0, (GLint)w, (GLint)h );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective(60.0,(GLfloat)w/(GLfloat)h,ZNEAR,ZFAR);

  gluLookAt(viewPoint.x,viewPoint.y,-LOOKAT+viewPoint.z,
    viewPoint.x,viewPoint.y,viewPoint.z, 0.0,-1.0,0.0);

  glMatrixMode( GL_MODELVIEW );
  scr.width = w;
  scr.height = h;
}

/*!
  Generate an OpenGL display list for the object to be shown, i.e. the box
  */

GLuint GLBox::makeObject(){

  GUINodeCore* guinode;
  GUINodeCore* guinodetmp;
  stmode* mp;
  int i;

  gui_world = new GUINodeCore;
  gui_world->shape.vector.z = -10.0;
```

```
//デスクトップ的地面生成 begin
guinode = new GUINodeCore;
mp = &(guinode->shape);

desktop = guinode;

guinode->NodePrev = NULL;
guinode->NodeNext = NULL;
if (rmdload("/rmd/desktop.rmd", mp)) return 2;
polyresize(mp,1.05);
MakeDisplayList(guinode->displaylist1, mp, DRAW_MODE_TRANSPARENT);
//end

//ランドマーク的地面地面生成 begin
guinode = new GUINodeCore;
mp = &(guinode->shape);

gui_list_start = guinode;
gui_list_end = guinode;
guinode->NodePrev = NULL;
guinode->NodeNext = NULL;
if (rmdload("/rmd/floor3.rmd", mp)) return 2;
polyresize(mp,1.05);
MakeDisplayList(guinode->displaylist1, mp, DRAW_MODE_TRANSPARENT);
//end

//地面の上に3次元アイコンを生成 begin
guinode = new GUINodeCore;
gui_list_end->NodeNext = guinode;
guinode->NodeNext = NULL;
guinode->NodePrev = gui_list_end;
gui_list_end = guinode;
icon->AddNodeToGUINodeCore(guinode, ICON_ATOM, 0.5, 200, 500, -1500);
guinode->shape.IconOrNot=true;
guinode = new GUINodeCore;
gui_list_end->NodeNext = guinode;
guinode->NodeNext = NULL;
guinode->NodePrev = gui_list_end;
gui_list_end = guinode;
icon->AddNodeToGUINodeCore(guinode, ICON_LIST, 0.3, 300, 500, -1500);
guinode->shape.IconOrNot=true;

guinode = new GUINodeCore;
gui_list_end->NodeNext = guinode;
guinode->NodeNext = NULL;
guinode->NodePrev = gui_list_end;
gui_list_end = guinode;
icon->AddNodeToGUINodeCore(guinode, ICON_GOAL, 0.2, 400, 500, -1500);
guinode->shape.IconOrNot=true;

guinode = new GUINodeCore;
```

```
  gui_list_end->NodeNext = guinode;
  guinode->NodeNext = NULL;
  guinode->NodePrev = gui_list_end;
  gui_list_end = guinode;
  icon->AddNodeToGUINodeCore(guinode, ICON_BLTIN, 0.4, 510, 500, -1500);
  guinode->shape.IconOrNot=true;

  guinode = new GUINodeCore;
  gui_list_end->NodeNext = guinode;
  guinode->NodeNext = NULL;
  guinode->NodePrev = gui_list_end;
  gui_list_end = guinode;
  icon->AddNodeToGUINodeCore(guinode, ICON_VAR, 0.3, 600, 500, -1500);
  guinode->shape.IconOrNot=true;

  //地面の上に3次元アイコンを生成 end

  printf("gui_list_end->ID %d\n", gui_list_end->ID);

  return guinode->ID;
}


//注目点変更処理
void GLBox::changeViewPoint(double x,double y,double z,float LOOKAT_to){

  if(viewPoint.x!=x || viewPoint.y!=y || viewPoint.z!=z){

    vector3d tmp_viewPoint;
    tmp_viewPoint=viewPoint;
    int i;
    float tmp_LOOKAT=LOOKAT;

    printf("Now Moving The ViewPoint to (%f,%f,%f).\n",x,y,z);

    int moveTimes=50;    //移動にかかるアニメーションのコマ数（マシン等に応じて設定）

    for(i=0;i<moveTimes;i++){
      viewPoint.x+=(x-tmp_viewPoint.x)/moveTimes;
      viewPoint.y+=(y-tmp_viewPoint.y)/moveTimes;
      viewPoint.z+=(z-tmp_viewPoint.z)/moveTimes;

      LOOKAT+=(LOOKAT_to-tmp_LOOKAT)/moveTimes;

      desktop->shape.vector=viewPoint;
      updateView();
    }
    printf("moved\n");
    viewPoint.x=x;viewPoint.y=y; viewPoint.z=z;
    pre_viewPoint=tmp_viewPoint;
```

```
  }
}

void GLBox::updateView(){   // 注目点を中心とした回転運動

  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective(60.0,(GLfloat)scr.width/(GLfloat)scr.height,ZNEAR,ZFAR);
  gluLookAt(viewPoint.x,viewPoint.y,-LOOKAT+viewPoint.z,
    viewPoint.x,viewPoint.y,viewPoint.z, 0.0,-1.0,0.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(viewPoint.x,viewPoint.y,viewPoint.z);
  glRotatef(camSpin.y,1.0,0.0,0.0);
  glRotatef(camSpin.x,0.0,-1.0,0.0);
  glTranslatef(-viewPoint.x,-viewPoint.y,-viewPoint.z);

  glTranslated(gui_world->shape.vector.x,
       gui_world->shape.vector.y,
       gui_world->shape.vector.z);

  glGetDoublev(GL_MODELVIEW_MATRIX,(GLdouble*)&gui_world->shape.matrix);

  updateGL();
}


void GLBox::CubeDraw(){     //背景の描画
  glDisable(GL_LIGHTING);
  glDisable(GL_LIGHT0);

  glPopMatrix();
  glShadeModel(GL_SMOOTH);
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective(60.0,(GLfloat)scr.width/(GLfloat)scr.height,ZNEAR,ZFAR);
  gluLookAt(viewPoint.x,viewPoint.y,-LOOKAT+viewPoint.z,
    viewPoint.x,viewPoint.y,viewPoint.z, 0.0,-1.0,0.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(viewPoint.x,viewPoint.y,viewPoint.z);
  glRotatef(camSpin.y,1.0,0.0,0.0);
  glRotatef(camSpin.x,0.0,-1.0,0.0);
  glTranslatef(-viewPoint.x,-viewPoint.y,-viewPoint.z);

  static GLfloat CubeSize=2000000 /2 ; //背景となる立方体のサイズ

  static GLfloat NorthWestTop[2][3]={
    { 200/255, 255/255, 145/255 },
```

```
  { -1*CubeSize, -1*CubeSize, CubeSize}};
static GLfloat SouthWestTop[2][3]={
  { 255/255, 145/255, 145/255 },
  { -1*CubeSize, -1*CubeSize, -1*CubeSize}};
static GLfloat SouthWestBottom[2][3]={
  { 66/255, 21/255, 21/255 },
  { -1*CubeSize, CubeSize, -1*CubeSize}};
static GLfloat NorthWestBottom[2][3]={
  { 43/255, 66/255, 21/255 },
  { -1*CubeSize, CubeSize, CubeSize}};

static GLfloat NorthEastTop[2][3]={
  { 145/255, 255/255, 255/255 },
  { CubeSize, -1*CubeSize, CubeSize}};
static GLfloat SouthEastTop[2][3]={
  { 200/255, 145/255, 255/255 },
  { CubeSize, -1*CubeSize, -1*CubeSize}};
static GLfloat SouthEastBottom[2][3]={
  { 43/255, 21/255, 66/255 },
  { CubeSize, CubeSize, -1*CubeSize}};
static GLfloat NorthEastBottom[2][3]={
  { 21/255, 66/255, 66/255 },
  { CubeSize, CubeSize, CubeSize}};

// 画面向かって左の面の描画
glBegin(GL_POLYGON);
glColor3fv(NorthWestTop[0]);
glVertex3fv(NorthWestTop[1]);
glColor3fv(SouthWestTop[0]);
glVertex3fv(SouthWestTop[1]);
glColor3fv(SouthWestBottom[0]);
glVertex3fv(SouthWestBottom[1]);
glColor3fv(NorthWestBottom[0]);
glVertex3fv(NorthWestBottom[1]);
glEnd();

// 画面向かって奥の面の描画
glBegin(GL_POLYGON);
glColor3fv(NorthEastTop[0]);
glVertex3fv(NorthEastTop[1]);
glColor3fv(NorthWestTop[0]);
glVertex3fv(NorthWestTop[1]);
glColor3fv(NorthWestBottom[0]);
glVertex3fv(NorthWestBottom[1]);
glColor3fv(NorthEastBottom[0]);
glVertex3fv(NorthEastBottom[1]);
glEnd();

// 画面向かって右の面の描画
glBegin(GL_POLYGON);
glColor3fv(SouthEastTop[0]);
```

```
  glVertex3fv(SouthEastTop[1]);
  glColor3fv(NorthEastTop[0]);
  glVertex3fv(NorthEastTop[1]);
  glColor3fv(NorthEastBottom[0]);
  glVertex3fv(NorthEastBottom[1]);
  glColor3fv(SouthEastBottom[0]);
  glVertex3fv(SouthEastBottom[1]);
  glEnd();

  //画面向かって手前の面の描画
  glBegin(GL_POLYGON);
  glColor3fv(SouthWestTop[0]);
  glVertex3fv(SouthWestTop[1]);
  glColor3fv(SouthEastTop[0]);
  glVertex3fv(SouthEastTop[1]);
  glColor3fv(SouthEastBottom[0]);
  glVertex3fv(SouthEastBottom[1]);
  glColor3fv(SouthWestBottom[0]);
  glVertex3fv(SouthWestBottom[1]);
  glEnd();

  //画面向かって上の面の描画
  glBegin(GL_POLYGON);
  glColor3fv(SouthEastTop[0]);
  glVertex3fv(SouthEastTop[1]);
  glColor3fv(SouthWestTop[0]);
  glVertex3fv(SouthWestTop[1]);
  glColor3fv(NorthWestTop[0]);
  glVertex3fv(NorthWestTop[1]);
  glColor3fv(NorthEastTop[0]);
  glVertex3fv(NorthEastTop[1]);
  glEnd();

  //画面向かって下の面の描画
  glBegin(GL_POLYGON);
  glColor3fv(NorthEastBottom[0]);
  glVertex3fv(NorthEastBottom[1]);
  glColor3fv(NorthWestBottom[0]);
  glVertex3fv(NorthWestBottom[1]);
  glColor3fv(SouthWestBottom[0]);
  glVertex3fv(SouthWestBottom[1]);
  glColor3fv(SouthEastBottom[0]);
  glVertex3fv(SouthEastBottom[1]);
  glEnd();

  glPushMatrix();

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);

}
```