３．システム開発者の手引き

３－１．はじめに

建物や町並を記録した三次元データに、このデータ形式を解読するためのメタファイルをセットにした保存データを利活用する方法は、上記で例示した携帯端末による現場表示や PC による表示にとどまらずに、「任意形式の三次元データに、解読方法を記述したメタファイルを添付した保存データ」を核とした様々な利活用方法の開発が可能である。

本章では、これまでに作成した４種類の実装形態の開発過程を解説することを通じて、将来新たな処理系を開発しようとするシステム開発者のための解説資料とする。

多様な利活用のシステムに共通する部分は、メタファイルに基づいて、データを入力・解釈するコンバータを生成するコンパイラである。このコンパイラ部は、C 言語で記述されており、PC の OS である Windows のための開発環境 VS2005 と、携帯端末の OS である Android のための開発環境 ndk において、共通であり、このコンパイラのソースコードには互換性がある。

一方、様々な利活用を実現するための実装部分は、使用する機器・OS に適した処理内容を実装する関数を、共にコンパイル・ビルドすることにより作成した。

実際の利活用に当たっては、全てのシステムに共通して、まずメタファイルをコンパイルして実行形式を生成する処理を行う。次にこの実行形式を実行して建物や町並の形状を記述したデータファイルを解読し、様々な利活用の目的に応じた処理を実行する。

従って、新たな利活用方法を開発するためには、コンパイラ部分のソースコードをそのまま利用し、ライブラリ関数の処理内容を利活用目的に応じで書き換えると共に、全体をパッケージ化するためのユーザーインターフェースやマシンコントロールの外皮部分を、ターゲットとするマシンやＯＳ等、目的に応じた開発環境で作成することとなる。

本章では、現在までに試作した４の利活用形態の開発手順を記録することにより、今後の新たな利活用システム開発のための解説資料とする。

表 3-1-1 実装形態別概要

名称	処理概要	ハード	OS	開発環境	外形	ライブラリ
①VC-1C	ファイル出力	PC	Windows	VS2005	コンソール	LIBC
②VC-2V	仮想現実表示	PC	Windows	VS2005	景観シミュレータのプラグイン	MFC
③VC-3M	AR 表示	スマホ、タブレット	Android	sdk+ndk	アプリ	SDK(Java)
④VC-4D	データ保管庫	サーバ	Windows +SQL	VS2005	WEB アプリ	C, Java

３－２．仮想コンバータのコンパイラ・インタプリタ基幹部分の開発

様々な利活用システムに共通となるコンパイラ部分は、林晴比古「コンパイラ・インタプリタ開発」[51]の添付 CD-ROM に収録・公開されたソースコードを出発点とした。この解説書は、各読者がこのソースコードに様々な機能を追加することを前提として提供されていた。そこでまず、出版元であるソフトバンク・クリエイティブ株式会社から、自由に拡張したものをアプリケーションとしてフリーで配布すること、出典を明記した上でソースコードを公開することの許可を得た上で作業に着手した（2012 年 3 月 14 日）。

このコンパイラはソースコードを解釈して、仮想スタックマシンのための機械語を配列上に生成する。次に、実行段階でインタープリタがこの機械語を解釈して処理を実行する。インタープリタも、上記のコンパイラ部分と同様に C 言語で記述されており、ビルド時点でそれぞれの CPU、OS のための実行形式を生成する。従って、原理的には異なる CPU、OS の上への移植を制約する条件がない。

実行形式の機械語は、固定長の配列の上に生成される。また、ポインタ型の変数を記号表アクセスに限定し、機械語がアクセスできるメモリも、固定長の配列の範囲に限定されている。従って、ファイルの読み書きを超えて、実行している仮想スタックマシンの OS のメモリ領域にアクセスして不正な処理を行うようなプログラムは作成不可能である。

また、動的なメモリ管理は行っていない。このことはコンバータを作成する上では一つの制約条件ではあるが、メタファイル（プログラム）が処理するデータファイルが特定されていることから、メタファイル作成段階で、当該データファイルを処理するために十分な長さの固定長の配列を用意することにより、動的なメモリ管理を行う必要はない。

この処理系は、機械語の種類は少なく、高度な最適化等を行っておらず、処理系が単純であるため、大容量のデータを高速処理するような実務には向かないとされているが、しかし長期保存データを解釈するためのメタファイルを記述するための処理系としては、処理速度よりもむしろ単純明快さが求められ、処理は遅くとも確実であることが求められることから、適していると考えた。

（１）数値型の増補と、それに伴う仮想マシンの機械語の増補

数値型としては、最初から提供されていた 32 ビット整数に、32 ビット浮動小数と、256 ビット四元数を加えて、四則演算もこれに合わせて拡充した。まず、建物等を記述する三次元データを処理するためには、32 ビット整数型の変数だけでは不十分であるため、32 ビット浮動小数型も追加した。これは、１ｍｍの精度で、団地や集落のスケールの空間を記述するためには十分な精度である。一方、GPS 座標等、地球の半径に相当するスケールの中で１ｍｍの精度の位置や形状を記述するためには不十分である。このことから、地球的尺度の位置を計算する目的で、256 ビット四元数型を用意した。

メタファイルで使用される変数や関数に関して定義される数値型、printf 関数や scanf 関数の書式文字列の中で使用される変換等の数値型は、ヘッダファイル cci_h の中で DtType として定義されている。これも原著に対して、増補を行った。

[原著]

NON_T 型無し（解析にのみ使用）

VOID_T 種別 Void 宣言「void」（戻り値のない関数の型宣言に使用）

INT_T 種別 Int 宣言「int」

[増補]

FLOAT_T 浮動小数 Float 宣言「float」

QUAT_T　四元数（クォータニオン） Quat 宣言「quat」

STR_T　文字列範囲

MULTI_T　可変型

NON_T は無で、例えば、関数呼び出しの解析において、引数が無い場合、引数の型としてされる。

[例] `x = t sin(  );`

VOID_T は空で、戻り値のない関数の型宣言に用いる。

[例] `void exit(void );`

scanf 関数の書式文字列の解析においては、「%s」,「%[」が検出された場合の書式文字列の型を示すために使用している。但し、関数の void 型の引数として void 型関数が明示的に参照されたような場合には、エラーとしている。

[例] `exit( exit() );`

STR_T は、数値型としては使用しない。

scanf 関数の書式文字列の解析において、%n が検出された場合の型分類に用いている。

MULTI_T は、数値型としては使用しない。

printf, logf 関数の引数として、記号表がポインタ型として参照された場合に解析結果として使用している。C 言語のポインタのような書式で変数の前に「＊」が付された引数が printf 関数等で参照された時には、書式文字列が要求する型の要素が処理のために使用される。具体的には、%s には変数名が、%d には格納アドレスに格納された値が参照される。

メタファイルの中で変数や関数型宣言に用いられる予約語と、トークン要素は、cci_tkn.c で定義された予約語一覧である、KeyWdTbl[]で、トークンを定義する文字列と、それぞれに対応したトークン要素（整数値による ID）の組として定義されており、この中の前半部分で型宣言の文字列と、トークン要素が定義されている。これも本処理系のために増補した。この増補分トークン要素は、cci.h におけるトークン要素の定義に増補した。

[原著]

void トークン要素 Void

int トークン要素 Int

[増補]

float トークン要素 Float

quat トークン要素 Quat

このような数値型の追加を行った上で、数値演算に関する機械語を増補した。オリジナルの機械語と、本処理系のために増補した機械語を、表 3-2-1 に示す。

インタープリタによるこれら機械語の実行処理は、cci_code.c に含まれる execute 関数の中で行われる。それぞれの機械語の処理内容はこの関数の中で直接定義されている。

機械語のプログラムは、命令を示す８ビットのコード、８ビットのフラグ及び処理対象を指示する３２ビットのデータが附属している。データは、処理に用いる値や、メモリ上のアドレスを記述している。処理対象とするアドレスは、大域変数や配列に対応するグローバルアドレスを指す場合と、局所変数を示すスタック上のアドレスを指す場合があり、コンパイラが変数・配列の大域／局所に応じて付与するフラグにより識別されている。グローバル変数の場合 memory 配列上の opData が直接指示する番地であり、ローカル変数の場合、その変数を用いる関数が実行段階で呼び出された時点で定まるベースアドレスからの opData 分のオフセットを加えた番地である。参考として、各コードが実行する処理の概要を付記した。

表 3-2-1 機械語一覧　　* sp の欄はコード実行後のスタックポインタの増減を示す。

コード	処理内容	sp*	参考
①原著の機械語			
DEL	スタックから削除	-1	–stp
STOP	OS 戻り	0/-1	if(stp>0) return POP(); else return 0;
JMP	ジャンプ	0	Pc=opData
JPT	真ジャンプ	-1	if(stack[stp--]) Pc = opData
LIB	組込関数（関数種別） （ライブラリ関数含む）	別表	library(opData) （表 3-2-2 参照）
LOD	メモリの値を PUSH	+1	PUSH(IntMem(adrs))
LDA	アドレスを PUSH	+1	PUSH(adrs)
LDI	値を PUSH	+1	PUSH(opData)
STO	スタックのトップを メモリに格納	-1	ASSIGN(adrs、stack[stp]; --stp;
ADBR	フレーム確保	0	baseReg += opData
NOP	何もしない	0	
ASS	スタックのトップを、 スタックのセカンドの アドレスに格納	-2	ASSIGN(stack[stp-1],stack[stp]); stp -= 2; 整数型の変数、配列に値を書き込む
ASSV	スタックのトップを、 スタックのセカンドに 格納し値をトップに残す	-1	ASSIGN(stack[stp-1],stack[stp]); stp--; 代入式(変数=式)の値が更に参照される場合。参照されない場合には ASS に変更。

VAL	スタックのトップのアドレスの値をスタックのトップに	0	stack[stp] = IntMem(stack[stp]); 整数型の変数、配列の値を取得するために使用
EQCMP	スタックトップと値を比較し結果をトップに(switch 用比較)	0	if(opData==stack[stp]) stack[stp]=1; else PUSH(0);
CALL	戻り番地を保存してジャンプ	+1	PUSH(Pc); Pc = opData;
RET	戻り番地の復活	-1	Pc = POP();
INC	トップが示す変数に１を足し結果をトップに	0	stack[stp]++; stack[stp]=IntMem(stack[stp]);
DEC	トップが示す変数から１を減じ結果をトップに	0	stack[stp]--; stack[stp]=IntMem(stack[stp]);
NOT	トップを否定	0	stack[stp] = ! stack[stp];
NEG	トップを符号反転	0	stack[stp] = - stack[stp];
DIV	整数除算	-1	stack[stp-1] =stack[stp-1]/stack[stp]; stp--;
MOD	整数剰余	-1	stack[stp-1] =stack[stp-1]%stack[stp]; stp--;
ADD	整数加算	-1	stack[stp-1] =stack[stp-1] + stack[stp]; stp--;
SUB	整数減算	-1	stack[stp-1] =stack[stp-1] - stack[stp]; stp--;
MUL	整数乗算	-1	stack[stp-1] =stack[stp-1] * stack[stp]; stp--;
LESS	整数未満	-1	stack[stp-1] =stack[stp-1] < stack[stp]; stp--;
LSEQ	整数以下	-1	stack[stp-1] =stack[stp-1] <= stack[stp]; stp--;
GRT	整数超	-1	stack[stp-1] =stack[stp-1] > stack[stp]; stp--;
GTEQ	整数以上	-1	stack[stp-1] =stack[stp-1]>=stack[stp]; stp--;
EQU	整数一致	-1	stack[stp-1] =stack[stp-1]==stack[stp]; stp--;

NTEQ	整数不等	-1	stack[stp-1] =stack[stp-1]!=stack[stp]; stp--;
AND	整数論理和	-1	stack[stp-1] =stack[stp-1]&&stack[stp]; stp--;
OR	整数論理積	-1	stack[stp-1] =stack[stp-1]||stack[stp]; stp--;
②本処理系のための機械語命令追加分			
F2I	トップが指す浮動小数変数を整数に変換	0	stack[stp]= (int)(*((float*)(&stack[stp])));
I2F	トップが指す整数変数を浮動小数に変換	0	*(float*)&stack[stp]= (float)stack[stp];
I1F	セカンドが指す整数変数を浮動小数に変換	0	*(float*)&stack[stp-1]= (float)stack[stp-1];
INCF	浮動小数に 1.0 を加算	0	FloatMem(stack[stp])+=1.0f; stack[stp]=IntMem(stack[stp]);
DECF	浮動小数に 1.0 を減算	0	FloatMem(stack[stp])-=1.0f; stack[stp]=IntMem(stack[stp]);
NEGF	浮動小数符号反転	0	UNI_OPF(-);
DIVF	浮動小数除算	-1	ZERO_CHKF(); FLOAT_OP(/); *(float*)&stack[stp-1] = *(float*)(&stack[stp-1])/ *(float*)(&stack[stp]),--stp;
ADDF	浮動小数加算	-1	FLOAT_OP(+);
SUBF	浮動小数減算	-1	FLOAT_OP(-);
MULF	浮動小数乗算	-1	FLOAT_OP(*);
LESSF	浮動小数未満	-1	FLOAT2I_OP(<);
LSEQF	浮動小数以下	-1	FLOAT2I_OP(<=);
GRTF	浮動小数超過	-1	FLOAT2I_OP(>);
GTEQF	浮動小数以上	-1	FLOAT2I_OP(>=);
EQUF	浮動小数一致	-1	FLOAT2I_OP(==);
NTEQF	浮動小数不一致	-1	FLOAT2I_OP(!=);
ANDF	浮動小数 AND	-1	FLOAT2I_OP(&&);
ORF	浮動小数 OR	-1	FLOAT2I_OP(||);
PI	円周率	+1	*(float*)&stack[++stp] = 3.14159208f;

SIN	正弦	0	*(float*)&stack[stp] = (float)sin(*(float*)(stack+stp));
COS	余弦	0	*(float*)&stack[stp] = (float)cos(*(float*)(stack+stp));
TAN	正接	0	*(float*)&stack[stp] = (float)tan(*(float*)(stack+stp));
ATAN	逆正接	0	*(float*)&stack[stp] = (float)atan(*(float*)(stack+stp));
SIND	正弦、度表示	0	#define PI180 (3.14159208f/180.0f) *(float*)&stack[stp] = (float)sin(*(float*)(stack+stp));
COSD	余弦、度表示	0	*(float*)&stack[stp] = (float)cos((*(float*)(stack+stp)) *PI180);
TAND	正接、度表示	0	*(float*)&stack[stp] = (float)tan((*(float*)(stack+stp)) *PI180);
ATAND	逆正接、度表示	0	*(float*)&stack[stp] = (float)atan((*(float*)(stack+stp)) *PI180);
SQRT	平方根	0	*(float*)&stack[stp] = (float)sqrt(*(float*)(stack+stp));
EXPQ	四元数指数	0	expq((double*)&stack[stp-7]);
LNQ	四元数対数	0	lnq((double*)&stack[stp-7]);
CONCAVE	凹ポリゴンフラグ変更		setconcave(opData);

ASSQ	四元数型の変数・配列への書込。	-9	stp-=7;
			(quat)(&memory[stack[stp-1]])=
			(quat)(&stack[stp]);
			stp-=2;

ASSQV	四元数型の変数・配列へ書込み、値をスタックに残す	-1	quat *q;
			q = (quat*)(&memory[stack[stp-8]]);
			*q = *(quat*)(&stack[stp-7]);
			(quat)(&stack[stp-8]) = *q;
			stp--;

LODQ	四元数変数読出	+8	*(quat*)(&stack[++stp]) = *(quat*)(memory+adrs); stp+=7;
STOQ	四元数変数書込(引数からローカル変数へ)	-8	stp-=7; *(quat*)(memory+adrs)= *(quat*)(&stack[stp]);

			stp--;
VALQ	四元数変数読出	+7	*(quat*)(&stack[stp])= *(quat*)(memory+stack[stp]); stp+=7;

ADDQ	四元数加算	-8	quat *q1,*q2;
			q1 = (quat*)(&stack[stp-15]);
			q2 = (quat*)(&stack[stp-7]);
			q1->t += q2->t;
			q1->x += q2->x;
			q1->y += q2->y;
			q1->z += q2->z;
			stp -= 8;

I2Q	整数を四元数に変換	0	何もしないこととなった

SUBQ	四元数減算	-8	quat *q1,*q2;
			q1 = (quat*)(&stack[stp-15]);
			q2 = (quat*)(&stack[stp-7]);
			q1->t -= q2->t;
			q1->x -= q2->x;
			q1->y -= q2->y;
			q1->z -= q2->z;
			stp -= 8;

MULQ	四元数乗算	-8	quat *p,*q,r;
			p = (quat*)(&stack[stp-15]);
			q = (quat*)(&stack[stp-7]);
			r.t = p->t*q->t - p->x*q->x - p->y*q->y - p->z*q->z;
			r.x = p->t*q->x + p->x*q->t + p->y*q->z - p->z*q->y;
			r.y = p->t*q->y + p->y*q->t + p->z*q->x - p->x*q->z;
			r.z = p->t*q->z + p->z*q->t + p->x*q->y - p->y*q->x;
			*p = r;
			stp -= 8;

DIVQ	四元数除算	-8	quat *p,*q,r;
			double l;
			p = (quat*)(&stack[stp-15]);
			q = (quat*)(&stack[stp-7]);
			r.t = p->t*q->t + p->x*q->x + p->y*q->y + p->z*q->z;
			r.x = -p->t*q->x + p->x*q->t - p->y*q->z + p->z*q->y;
			r.y = -p->t*q->y + p->y*q->t - p->z*q->x + p->x*q->z;
			r.z = -p->t*q->z + p->z*q->t - p->x*q->y + p->y*q->x;
			l = q->t*q->t + q->x*q->x + q->y*q->y + q->z*q->z;
			p->t = r.t/l;
			p->x = r.x/l;
			p->y = r.y/l;
			p->z = r.z/l;
			stp -= 8;

BCH	配列の添字の範囲検査	0	stack[stp]<0 なら実行エラー「配列の添字が負」 opData<=stack[stp]（配列宣言時の長さ）なら実行エラー「配列の添字が過大」
DELQ	スタックデータ削除	-8	quat 型の式の値が使用されない場合、削除する。DEL（整数）の処理を数値型に合わせて多様化する。
TVAL	記号表の変数格納域	0	stack[stp]を添字とする記号表に登録された変数の格納域を stack[stp]に置く
TNAME	記号表の変数名	0	stack[stp]を添字とする記号表に登録された変数の名称を stack[stp]に置く

（２）入出力に用いる組込関数の増補

上記文献で提供されているソースコードは、入力手段としてコンソールから数値を取得

する input 関数と、コンソールに文字列を出力する printf 関数しかもたない。そこで、この入出力機能に、コンバータの記述に適した入出力関数（ライブラリ関数、ないしライブラリ関数）を増補することにより、データファイルの読み取りや、処理結果の出力が簡単に記述できるような言語となる。このような組込関数は、機械語の LIB 命令に、データとして関数タイプを示す数値を付して定義されており、実行は上記の execute 関数の中で LIB 命令が検出された段階で、cci_code.c の中で定義された library 関数に関数タイプを引数で渡して行われる。それぞれのライブラリ関数に引数として渡す数値は、関数毎にあらかじめ決められた長さのスタックを用いて受け渡す。

library 関数から、関数タイプに従って分岐して実行されるそれぞれの組込関数の実体は、引数と戻り値を持たず、スタックから引数情報を受け取り、結果をスタックに積んで終了する関数群であり、処理系により処理内容は異なり、ビルドに利活用目的に応じたソースコードを含めることにより実装されている。

例えば、メタファイルにおいて法線ベクトルを定義する NORMAL(x,y,z);というコマンドが要求されると、コンパイラは x,y,z の値を評価した結果をスタックに積んだ上で、「LIB NORMAL_F」という機械語命令を生成する。実行段階でインタープリタの execute 関数は、「LIB NORMAL_F」という機械語命令の処理に当たって、NORMAL_F を引数として組込関数を呼び出す：library(NORMAL_F);

library 関数は、引数として与えられた関数タイプに従って分岐し、NORMAL_F が引数である場合には、NORMAL()関数を呼び出す。

VC-1C を実装している cci_ip.c においては、この NORMAL 関数は、スタックから３の法線ベクトル値 X,Y,Z を取り出し、これをソートした上で ID 番号を取得し、これを用いて、

printf(fp, “N%d=NORMAL(%f,%f,%f);¥n”,ID,X,Y,Z);

という１行を実行し、引数式を評価した値を用いて出力ファイルに１行出力すると共に、この ID 値を、スタックを通じて返す。

VC-2V、VC-3M を実装している cci_dml.c においては、同じ NORMAL 関数が、スタックから３の法線ベクトル値 X,Y,Z を取り出し、これを表示に使用するメモリ上の配列に格納した上で、その ID（配列の添字）を、スタックを通じて返す。

VC-4D を実装している cci_sql.c においては、同じ NORMAL 関数が、スタックから３の法線ベクトル値 X,Y,Z を取り出してＳＱＬ文を生成して実行し、データベース上の normal テーブルから同一の法線ベクトル値を登録したレコードがあるかどうか検索し、あればその ID を、また無ければ新規登録した上でその新たな ID を、スタックを通じて返す。

各実装形態に共通して生成されるスタックトップの ID 値は、メタファイルの記述における次の処理（例えば、変数への値の代入）に使用される。

上記の各処理系の処理内容は、ファイル出力、メモリ上の配列へのデータ登録、SQL データベースへの登録と、それぞれ異なっている。しかしいずれの NORMAL 関数もスタックから３の引数値を取得し、各様の処理を行った結果をスタックのトップで返している。

現在までに試作した４の利活用形態に共通するコンパイラ・インタプリタ処理系のために増補した組込関数を表 3-2-2 に示す。処理系毎に実行内容が異なるものをライブラリ関数と呼んで区別し、ここではテンプレート（引数構成、戻り値、処理内容）を示す。

関数タイプの欄は、処理系内部で各関数に共通の機械語 LIB を呼び出す際に、関数タイプを識別するために opData として添付される ID であり、cci.h の中で宣言されている。

概要の欄には組込関数の処理内容を簡単に説明した。

sp の欄は、組込関数の呼び出し前と後でのスタックポインタの増減を示した。例えば-4+8であれば引数として４取得し結果を８残す（結果的に sp は４増加する）。

メタファイル中で使用する関数と用例の欄には、この組込関数を起動するためのメタファイル中での記法を解説した。メタファイルの中で使用されるそれぞれの組込関数の名称(トークン)は、cci_tkn.c に宣言されている。組込関数の戻り値の数値型は、cci_pars.c の factor 関数において switch~case で識別し、グループ別にプログラムで直接指定している。組込関数の引数の数と数値型は、cci_pars.c の sys_fncCall 関数の中でその整合性をチェックしている。

メタファイルの中で組込関数に対して指定された引数は、それぞれを式として評価・計算する機械語に展開された上で、最終的にはスタックを通じ個別の組込関数の処理へと渡され、結果（戻り値）はスタックを介して返される。引数の型の内、変数（var）と示したものは、メタファイル作成者がメタファイルの中で定義する関数にはない、組込関数の引数にのみ使用される数値型であり、メタファイルの中で組込関数の呼び出しにおいては変数名または配列の１要素が直接記述される。これは C 言語の関数の引数において、「＆変数」の形式で参照されるものに相当する。

メタファイルの中で組込関数が使用された場合には、コンパイラの文法解析段階では、組込関数は、固有のトークンとして認識され、これに対応する関数タイプを opCode として随伴する機械語の LIB 命令を生成する。実行段階で機械語 LIB が実行されると、opCode を引数として library 関数が呼び出される。その中で、opCode 毎の処理が行われる。

様々な処理系に共通の簡単な処理は library 関数の中に直接記述されている。一方、図形定義などを記述する組込関数を特に「ライブラリ関数」と呼び、利活用処理系により異なる実装を定義することができる。この実際に処理を行う各関数では、スタックから引数情報を受け取り、処理結果が void でなければ、数値型に応じたサイズでスタックのトップに結果を積んで終了する。この呼び出しと実行の手順だけは関数テンプレートとして定めているが、長期保存されたデータとメタファイルを用いて、本稿で例示した cci_ip.c（ファイル出力）、 cci_dml.c（仮想現実表示）、及び cci_sql.c（データベース構築）とは異なるライブラリ関数の様々な実装形態を、将来における新たな利活用処理系のニーズや開発環境に応じて作成することにより実現することを想定した。プログラマのための参考とする。

個別のライブラリ関数が果たすべき役割とメタファイルにおける用法の詳細に関しては、資料４－２で解説する。

表 3-2-2 組込関数一覧　　* sp の欄はコード実行後のスタックポインタの増減を示す

関数タイプ	概要	sp*	メタファイル中で使用する関数と使用例
原著に含まれている組込関数			
EXIT	プログラムの強制終了	-1+0	exit(int); execute 関数の中で関数タイプが EXIT であった場合、library 関数を呼び出すことなく、直ちに処理を終了する。
INPUT_F	コンソール入力	+1	仮想コンバータでは使用しない
PRINTF1_F	引数のない出力	-1+0	printf(“文字列”);
PRINTF2_F	引数のある出力	-2+0	printf(“書式文字列”,式);
仮想コンバータで追加した組込関数（データファイルへのアクセスと、ログ出力）			
SCANF_F	ファイル入力	-1+1	int scanf(“文字列”); int scanf(“文字列”,var);
LEN_F	ファイルサイズ取得	-0+1	int LEN();
SIORI_F	栞位置を返す	-0+1	int SIORI();
SEEK_F	栞位置を設定する	-1+0	SEEK(int);
GETC_F	１文字取得	-0+1	int GETC();
GETINT_F	整数値の取得	-1+1	int GETINT(int);
GETS_F	行長さの取得	-0+1	int GETS();
LOGF1_F	引数のないログ出力	-1+0	logf(“文字列”);
LOGF2_F	引数のあるログ出力	-2+0	logf(“書式文字列”,式); (式の数値型は書式文字列に従う)
仮想コンバータで追加した組込関数（座標系に関する数値関数）			
_Q_F	四元数の構築	-8+8	quat _Q(float,float,float,float); (引数が int 型の場合、float 値に変換)
_QS_F	文字列による四元数の定義	-1+8	quat _QS(“文字列”); [例]q = _QS(“0.12, 0.23, 0.43, 0.76”);
_Qt_F	四元数 T 成分取得	-8+1	float _Qt(quat);
_Qx_F	四元数 X 成分取得	-8+1	float _Qx(quat);
_Qy_F	四元数 Y 成分取得	-8+1	float _Qy(quat);
_Qz_F	四元数 Z 成分取得	-8+1	float _Qz(quat);
IKE2DES_F	緯度経度→地球座標	-8+8	quat = IKE2DES(quat);
DES2IKE_F	地球座標→緯度経度	-8+8	quat = DES2IKE(quat);
IKE2NAT_F	緯度経度→国家座標	-8+8	quat = IKE2NAT(int, quat);
NAT2IKE_F	国家座標→緯度経度	-9+8	quat = NAT2IKE(int, quat);

IKE2WLD_F	緯度経度→世界座標	-9+8	quat = IKE2WLD(int, quat);
WLD2IKE_F	世界座標→緯度経度	-9+8	quat = WLD2IKE(int, quat);
仮想コンバータで追加したライブラリ関数（LSS-G 系コマンド、三次元形状の定義）			
TEXTURE_F	テクスチャ定義	-1+1	int TEXTURE(“文字列”);
MATERIAL_F	マテリアル定義	-1+1	int MATERIAL(“文字列”);
COORD_F	座標定義	-3+1	int COORD(float,float,float); (引数が int 型の場合、float 値に変換)
COORDQ_F	座標定義	-8+1	int COORD(quat);
VERTEX_F	頂点定義	-5+1	int VERTEX(var,var,var,var,var); 座標,法線,テクスチャ座標,カラー,座標 (定義しない var は省略可能)[註]
TCOORD_F	テクスチャ座標定義	-2+1	int TCOORD(float,float);
NORMAL_F	法線ベクトル定義	-3+1	int NORMAL(float,float,float); （引数は順に X 成分、Y 成分、Z 成分）
COLOR3_F	カラー定義	-3+1	int COLOR(float,float,float); （引数は順に R、G、B 値とし 0.0～１.0）
COLOR4_F	カラー定義(α値付)	-4+1	int COLOR(float ,float, float, float); （引数は順に R､G､B､α 値とし 0.0～１.0）
FACE_F	頂点列により面を定義	-X+1	int FACE(var,var,……); ライブラリ関数の呼び出しに際しては全ての引数をスタックにプッシュした上で総数をトップに置く。
GROUP0_F	グループ宣言	-0+1	int GROUP();
GROUP_F	属性付グループ宣言	-1+1	int GROUP(“属性文字列”);
GROUP_FACE_F	グループに面を追加	-X+0	GROUP_FACE(var,var,….);
GROUP_LINE_F	グループに線を追加	-X+0	GROUP_LINE(var,var,….);
FACE_NORMAL_F	面に法線を定義	-2+0	FACE_NORMAL(var,var);
FACE_COLOR_F	面に色彩を定義	-2+0	FACE_COLOR(var,var);
FACE_TEXTURE_F	面にテクスチャを定義	-2+0	FACE_TEXTURE(var,var);
FACE_MATERIAL_F	面にマテリアルを定義	-2+0	FACE_MATERIAL(var,var);
LINE_F	頂点列により線（折れ線）を定義	-X+1	int LINE(var,var,….);
LINE_NORMAL_F	線に法線を定義	-2+0	LINE_NORMAL(var,var);
LINE_COLOR_F	線に色彩を定義	-2+0	LINE_COLOR(var,var);
LINE_TEXTURE_F	線にテクスチャを定	-2+0	LINE_TEXTURE(var,var);

	義		
LINE_MATERIAL_F	線にマテリアルを定義	-2+0	LINE_MATERIAL(var,var);
GROUP_TEXTURE_F	グループにテクスチャを定義	-2+0	GROUP_TEXTURE(var,var);
GROUP_MATERIAL_F	グループにマテリアルを定義	-2+0	GROUP_MATERIAL(var,var);
LINK_F	親グループと子グループの間のリンクを定義	-2+1	int LINK(var,var);
LINK_XFORM_F	リンクの相対的位置関係を定義	-X+0	LINK_XFORM(var,LOAD/PRE/POST, IDENTITY); TRANSLATE,var,var,var); ROTATE_X/_Y/_Z,var); ROTATE_A,var,var,var,var); MATRIX,var,var,var,var, var,var,var,var, var,var,var,var, var,var,var,var);
FACE_VERTEX_F	定義済の面に頂点を追加	-2+0	FACE_VERTEX(var,var); (引数は面と頂点。LSS-G にない形式)
LINE_VERTEX_F	定義済の線に頂点を追加	-2+0	LINE_VERTEX(var,var); (引数は線と頂点。LSS-G にない形式)
GROUP_ATTR_F	グループにデータファイル中の文字列を属性として追加	-3+0	GROUP_ATTRIBUTE(var,int,int); (引数は、グループ id, 先頭アドレス、文字数。LSS-G にない形式)
仮想コンバータで追加したライブラリ関数（LSS-S 系コマンド、時刻、視点等の記録）			
TIME_F	時間を定義する		int TIME(float); （単位：日）

CAMERA_F	カメラアングルを定義する	-13+1	int CAMERA(float,float,float, //視点
			float,float,float, //注視点
			float,float,float, //上方ベクトル
			float, //視野角
			float, //アスペクト比
			float, float); //zNear, zFar
			[例]

			4.59636449813843, 7.70967817306519, 4.25015115737915, 4.48174047470093, 6.71801471710205, 4.19128608703613, -0.00861490704119205, -0.0582613088190556, 0.998264610767365, 35.8967247009277, 1.77777802944183, 0.100000023841858, 317.575653076172);
LIGHT_F	光源を定義する	-8+1	int LIGHT(int, float, float, float, //位置 int, float, float, float); //色彩
LIGHTGROUP_F	光源グループを定義する	-X+1	int LIGHTGROUP(var,var,…); (引数は光源を定義した変数とし最大８)
MODEL_F	モデルを定義する	-1+1	int MODEL(“メタファイル名?データファイル名”);
IMAGE_F	画像を定義する	-1+1	int IMAGE(“背景画像ファイル名”);
EFFECT_F	効果を定義する	-5+1	int EFFECT(IKE,"0.060198","9.449400","0.000000","/storage/emulated/0/VirtualConverter/meta/LSSG.cmm?/storage/emulated/0/VirtualConverter/data/hamakaze6t.geo")
EFFECTGROUP_F	効果グループを定義する	-X+1	int EFFECTGROUP(var,var,…); （引数は効果とする）
SCENE_F	シーンを定義する	-0+1	int SCENE();
仮想コンバータで追加したライブラリ関数（制御に関する特殊関数）			
(トークン処理)	コメント制御	-	void COMMENT(ON または OFF);
(トークン処理)	凹多面体処理制御	-	void CONCAVE(ON または OFF);
仮想コンバータで追加したライブラリ関数（出力のための形状要素取得）			
G_F	グループを順次選択	-0+1	int G();
Gid_F	選択グループの ID を取得	-0+1	int Gid();
Gattribute_F	選択グループの属性を取得	-0+1	int Gattribute();
Gmaterial_F	選択グループのマテリアルを取得	-0+1	int Gmaterial();
Gtexture_F	選択グループのテク	-0+1	int Gtexture();

	スチャを取得		
F_F	選択グループの面を順次選択	-0+1	int F();
Fshp_F	選択面のシェープを取得	-0+1	int Fshp();
Fnormal_F	選択面の法線を取得	-0+1	int Fnormal();
Fcolor_F	選択面の色を取得	-0+1	int Fcolor();
Fmaterial_F	選択面のマテリアルを取得	-0+1	int Fmaterial();
Ftexture_F	選択面のテクスチャを取得	-0+1	int Ftexture();
F3_F	面の三角形分割し順次取得	-0+1	int Ft();
V_F	選択面の頂点を順次選択	-0+1	int V();
Vcoord_F	選択頂点の座標を取得	-0+1	int Vcoord();
Vcolor_F	選択頂点の色彩を取得	-0+1	int Vcolor();
Vnormal_F	選択頂点の法線を取得	-0+1	int Vnormal();
Vtcoord_F	選択頂点の座標を取得	-0+1	int Vtcoord();
Vvirtual_F	選択頂点の仮想線フラグを取得	-0+1	int Vvirtual();
S_F	全ての頂点座標を順次選択	-0+1	int S();
Sx_F	選択座標の X 値を取得	-0+1	float Sx();
Sy_F	選択座標の Y 値を取得	-0+1	float Sy();
Sz_F	選択座標の Z 値を取得	-0+1	float Sz();
Sq_F	選択座標を四元数として取得	-0+1	quat Sq();

N_F	全ての法線を順次選択	-0+1	int N();
Nx_F	選択法線の X 値を取得	-0+1	float Nx();
Ny_F	選択法線の Y 値を取得	-0+1	float Ny();
Nz_F	選択法線の Z 値を取得	-0+1	float Nz();
Nq_F	選択法線を四元数として取得	-0+1	quat Nq();
C_F	全ての色を順次選択	-0+1	int C();
Cr_F	選択色の R 値を取得	-0+1	float Cr();
Cg_F	選択色の G 値を取得	-0+1	float Cg();
Cb_F	選択色の B 値を取得	-0+1	float Cb();
Ca_F	選択色のα値を取得	-0+1	float Ca();
Cq_F	選択所を四元数で取得	-0+1	quat Cq();
T_F	全てのテクスチャ座標を順次選択	-0+1	int T();
Tx_F	テクスチャの X 座標を取得	-0+1	float Tx();
Ty_F	テクスチャの Y 座標を取得	-0+1	float Ty();
Tq_F	テクスチャ座標を四元数で取得	-0+1	quat Tq();
L_F	全てのリンクを順次選択	-0+1	int L();
Lp_F	親グループを取得	-0+1	int Lp();
Lc_F	子グループを取得	-0+1	int Lc();
M_F	マトリクス読出	-0+1	float Lm();
Mi_F	単位マトリクスか？	-0+1	int Li();
Lt_F	リンクの並進成分	-0+1	quat Lt();
Lr_F	リンクの回転成分	-0+1	quat Lr();
Ls_F	リンクの拡大成分	-0+1	quat Ls();
D_F	全ての表示グループ	-0+1	int D();

	を順次選択		
Dl_F	選択した表示の階層を取得	-0+1	int Dlevel();
Df_F	選択した表示の実体が初出なら 1	-0+1	int Dfirst();

組込関数が数値型を有する場合には、演算結果がスタックのトップに残される。入出力処理(printf など)において組込関数の戻り値を参照する必要がない場合には、コンパイル段階で、statement として扱い、実行時に組込関数がスタックトップに残す戻り値を削除する機械語を追加して、スタックの一貫性を維持している。このような場合、組込関数の先に式が続くような場合、コンパイル・エラーとして処理している。

[例] printf(“first”) + 3;

数式(expression)も一般的には、別の変数に代入が可能である。例えば、a=b=c; は、式「b=c」の値（c に等しい）を a に代入する。この式全体の値は、更に代入する先がない場合には、to_leftVal 関数による機械語の変換が行われている。

関数の戻り値が参照されなければ、処理が意味を持たないような組込関数が文頭に記述された場合に関しては、statement 関数で、エラーとして処理する。

[例] sin(0.0); ・・・代入先や参照がない

（3）メタファイルの文法における標準 C 言語との違い

三次元形状を記述するデータファイルの解読方法を指示するメタファイルを記述するに当たって、以下のように関数定義の方法などを C 言語の標準的な書法から変更した。

①main 関数の省略

通常の場合プログラムは、main 関数を起点として実行する。main 関数から、他の関数を呼び出すことができ、その戻り値を用いて続く処理を実行することができる。他の関数はプログラムの中で定義することができるが、ライブラリ関数と同じ名称の関数を定義しようとするとエラーとしている。また、他の関数の中であっても、exit 命令が実行されると、main 関数に戻ることなく処理を終了する。

main 関数から呼び出す他の関数がない場合には、メタファイルにおいて main 関数の宣言を省略することができることとした。言い換えると、直ちにコマンド列が始まるようなプログラムは、main(){}の宣言が省略された、{} の内部だけの表現として解釈する。

これにより、LSS－G 形式のファイルを、この処理系の開発初期におけるテスト用メタファイル（既知の固定形状を記述）として利用することができるようになった。

②引数の数が可変のライブラリ関数

例えば、

F=FACE (V1, V2, V3);

は、3 の頂点 V1,V2,V3 を頂点とする三角形を定義するコマンドであるが、4 頂点以上の

場合には、引数は４以上となり、その上限はない。このように、引数の数が不定の関数もライブラリ関数として増補できるようにするため、処理系の修正を行った。

なお、引数の数が不定の関数は、一部のライブラリ関数のみであって、メタファイルの中で定義される関数をこのような関数として定義することは許容していない。

③引数を省略できるライブラリ関数

例えば頂点を定義する VERTEX 関数は、全ての引数を使用する場合、

V=VERTEX(S, N, T, C, Sv);

（但し、S は座標、N は法線ベクトル、C は色彩、T はテクスチャ座標,Sv は S が仮想線の起点の座標である場合の、終点の座標）

という書法となるが、法線ベクトル、色彩等を明示的に定義しない場合に省略することができ、

V=VERTEX(S…Sv); （N,C,T を省略）

V=VERTEX(S); （N,C,T,Sv を省略）

等と簡略に記述することができる。引数リストの二つの「,」の間が空白である場合には、エラーとせず、引数が省略されている表記として解釈するように変更した。

具体的には、コンパイル段階で引数リストを取得するオリジナルの args()関数に代わって、新たに作成した argsV()関数で、空白の場合も含めて、引数リストが終了するまで全ての引数の取得を行うように変更した（cci_pars.c）。

④LSS-G コマンドを補足するライブラリ関数

メタファイルの中で、例えば上記②の FACE コマンドのように、引数（頂点数）が不定の関数呼び出しを処理するためには、C 言語の処理系では引数リストを配列に格納して関数に渡すが、本処理系では、メタファイルの記述を容易化するために、一度定義した面の頂点リストの末尾に頂点を追加する FACE_VERTEX 関数を追加した。

FACE_VERTEX(F,V);

この時、F が 25 角形であれば、末尾に新たな頂点 V を追加した 26 角形となる。

以下、これまでに開発した 4 種類の処理系（①VC-1C②VC-2V③VC-3M④VC-4D）を例に用いて、開発プロセスを解説する。コンパイラとインタープリタから成る基幹部分と数値演算関数等の組込関数は、上記の４の実装例に共通である。一方、ライブラリ関数の実際の動作を定義するソースコードは、表 3-2-3 に示すように実装例毎に異なっている。

表 3-2-3 実装形態別の、ソースコード利用状況

ソースコード名（主な機能）	ステップ数		実装形態別使用状況			
	原作	現状	①	②	③	④
cci_code.c (コンパイラのコード生成、実行)	344	1,377	○	○	○	○
cci_face.c （面の生成処理、三角形分割等）	0	859	○	○	○	○
cci_file.c (データファイルへのアクセス)	0	581	○	○	○	○

cci_misc.c (エラー処理、文字コード処理)	75	377	○	○	○	○
cci_pars.c （メタファイルの構文解析)	682	2,553	○	○	○	○
cci_quat.c （四元数演算、測地系変換)	0	1,075	○	○	○	○
cci_tbl.c (記号表処理)	107	226	○	○	○	○
cci_tkn.c (メタファイル解析におけるトークン処理)	222	511	○	○	○	○
cci_io.c (書式付入力)	0	970	○	○	○	○
cci.c （main 関数を含む原本のエントリ部分)	21	109	○	×	×	×
cci_ip.c （ライブラリ関数実装：LSSG 形式ファイル出力)	0	631	○	×	×	×
cci_dummy.c（処理を行う必要がない空のライブラリ関数)	0	14	○	×	×	×
cci_dml.c (ライブラリ関数実装：メモリ上のデータ構築)	0	2,015	×	○	○	×
cci_sml.c (LSSS 形式のファイルの処理)	0	533	×	×	○	×
cci_sql.c (ライブラリ関数実装：SQL コマンドの発行)	0	1,782	×	×	×	○

（4）ライブラリ関数の追加

コンパイラのソースコードに以下の増補を行った。追加する新しいライブラリ関数をfunc とすると、その追加の手順は以下の通り。

a. 関数に対応する関数種別：Func_Fをenumに追加する(cci.h)

b. 関数に対応するトークン種別：QFuncをenum TknKindに追加する(cci.h)

c. 関数に対応するダンプ名称を，ssLibCode[]に追加する(cci.h)

d. 変換テーブルKeyWdTbl[]に、関数名リテラルとTknKindのペアを登録する(cci_tkn.c)

e. compile関数のswitch-case に、新たなTknKind QFuncを追加する(cci_pars.c)

f. statement関数に、新たなTknKind QFuncを加える(cci_pars.c)

関数の戻り値を使用せず、行頭に来る可能性がある場合→正常処理

関数の戻り値を必ず使用し、行頭に来る可能性がない場合→エラー処理

g. factor関数に、新たなTknKindを加える（戻り値のある場合、cci_pars.c)

関数の戻り値を使用する場合→正常処理

関数の戻り値がない場合→エラー処理

h. sys_fncCall関数に、新たなTknKindに対する処理を加える(cci_pars.c)

i. library関数に、case Func_F FUNC(); 等の処理を加える(cci_code.c)

j. ヘッダファイルに、この関数 FUNC()を追加登録する(cci_prot.h)

k.処理系に応じた FUNC()の実際の処理をプログラムする(処理系別のソース)

VC-1C では、cci_ip.c, VC-2V と VC-3M では、cci_dml.c, VC-4D では、cci_sql.c に、ライブラリ関数を実装した。

但し、これらに共通するライブラリ関数は、cci_face.c, cci_file.c, cci_quat.c および cci_vector.c に集約し共通で利用している。

なお､原著でコンパイラとインタープリタの処理系の入り口となっていた main 関数を含

む cci.c は、コマンドラインで起動する初期の VC-1C においては増補しつつ利用したが、以後の処理系においては、ダイアログベースの処理となったため、ビルドからは外している。

以上を理解するために、コンパイラの処理について関連する部分だけを概略解説する（詳しくは原著参照）。

コンパイル処理は、compile()関数(cci_pars.c)配下の関数群により実行される。この時、nextTkn()関数により、メタファイルの構文が解析される。

nextTkn 関数が文字列比較により検出するトークンは、KeywdTbl 配列に、トークンの文字列(keyName)とトークンの種別コード(keyCode)のペアとして、宣言されている。この内、文字列はこの配列の中で直接定義され、種別コードは、cci.h の中で enum 宣言されたトークン種別の一つが参照されている。文字列の内、「(」などの特別な文字は文字コードで直接宣言され、数値型(Void, Int, Float, Quat)、プログラム制御等の文字列(If, For…)、及びライブラリ関数名等は、150 から始まる整数値として定義されている。本処理系のソースコードの中ではこの宣言名を直接使用しており、整数値を直接参照することはない。新たなライブラリ関数を追加する場合には、cci.h の TknKind にトークン種別を加える（b.）と共に、KeywdTbl 配列に、トークンの文字列とこのトークン種別のペアを追加する（d.）。ライブラリ関数は、関数宣言の中か、式の中にしか現れないため、compile 関数の中で直接検出されることはない。通常の処理を行うのは、関数宣言の冒頭にある数値型とセミコロンであり、数値型(void, int 等)の場合には、関数宣言の処理(fncDecl 関数)に進む。

しかし、本処理系の場合には、main 関数を省略した形を含めているため、ここでライブラリ関数が検出された場合には、main 関数の内部として再処理している。そこで、compile 関数の中の switch-case のリストの中に、文頭に現れることのできるライブラリ関数のグループに加える（e.）。compile 関数の中で、文頭に直接ライブラリ関数が検出された場合には、main 関数が省略された形として処理している。

次に、通常の関数宣言形式で構成されたメタファイルの処理であるが、型宣言、関数名、引数リストの後、「{」から「}」までの間をプログラムの記述として処理する。

プログラムの記述には、変数宣言、プログラム制御のキーワード(if,for,do,swtich 等)、変数への代入、関数の実行等が含まれる。それぞれのキーワード毎の処理は、nextTkn()関数で検出されたトークンの種類のコードにより switch されている。

statement 関数において nextTkn 関数によりライブラリ関数の名称が検出された場合には、TknKind でそれぞれの関数の処理を行う（f.）。

ライブラリ関数が戻り値を使用しない命令として利用できる関数の場合には、トークン種別を引数として sys_fncCall 関数を呼び出して、機械語に翻訳し、正常に処理を終了する。

戻り値を必ず使用するライブラリ関数（例えば COORD 関数）がここで検出された場合（つまり文頭に検出された場合）には、エラーとして処理する。

ライブラリ関数が式の中や、関数の引数として呼び出された場合には、expression 関数配下の関数の中で処理される。expression は、8 の階層を有する term から構成され、最も

原始的な単位が factor である。これは数値、変数、関数、括弧で括られた式である。

term のレベルは次の通りである。レベルの高い二項演算が優先的に実行される。

リスト 3-2-1 式(term)の強さのレベル

```
レベル８ : factor
レベル７ : ＊, / , %
レベル６ : ＋,－
レベル５ : ＜,＜＝,＞,＞＝
レベル４ : ＝＝, ！＝
レベル３ : &&
レベル２ : ｜｜
レベル１ : ＝
```

factor には以下のものが含まれる。

リスト 3-2-2 因子(factor)

```
++変数, --変数, +変数, -変数（＊変数には配列も含む）
*変数, &変数
数値
関数
ライブラリ関数
（expression）　//括弧で括られた expression
```

factor 関数の中でライブラリ関数が要求された場合に、sys_fncCall(トークン種別)を呼び出す（g.)。

sys_fncCall 関数(cci_pars.c)の中では、トークン種別で switch-case を行い、それぞれのライブラリ関数毎に引数をスタックに積む処理を追加し(h.)、機械語 LIB に、定義された関数種別をオペランドとして付して処理を終了する。この関数種別は、cci.h の中で enum 宣言されているため、追加する関数をこの宣言に追加する（a.)。

実行処理は、execute 関数（引数はデータファイル名）の中で行われる。この関数は、機械語を順次読み出して実行するが、ライブラリ関数を起動する LIB 命令の実行は、オペランド（関数種別）を引数として、libary 関数を呼び出す。

library 関数(cci_code.c)においては、switch-case で引数の関数種別毎に処理を行う。この際に、利活用処理系により異なる処理が必要となる、図形定義の関数が要求された場合には､その処理の全てを処理系別のソースコードで定義されたそれぞれの関数を作成し(k.)、これにより処理する。更に、この関数のプロトタイプを、cci_prot.h に追加すると共に、ダンプリストを作成する際に使用する文字列を ssOpCode[]配列に追加する。この配列は、関数種別の定義と同じ順序・総数となっていなければならない（c.)。

例えば、メタファイルで座標値を定義する COORD 関数が用いられた場合には、LIB COORD_F という機械語が実行される。すると、library 関数において COORD__F が要求され、そこから、COORD（）関数が呼び出される。

この COORD()関数の実装は、利活用処理系により異なる。例えば、ファイルを出力する VC-1C のための cci_ip.c においては、以下のような処理を行っている。

リスト 3-2-3 VC-1C における COORD ライブラリ関数の実装

```
void COORD(){
        int i;
                float x,y,z;
                z = *((float*)&stack[stp--]);//元データをメモリと見て、送り先に書き込む
                y = *((float*)&stack[stp--]);
                x = *((float*)&stack[stp--]);
        for(i=0;i<NP;i++){
                if( x != X[i]) continue;
                if( y != Y[i]) continue;
                if( z != Z[i]) continue;
                break;
        }
        if(i == NP){
                X[i] = x, Y[i] = y, Z[i] = z;
                NP++;
                fprintf(stberr,"P%d=COORD(%f,%f,%f);¥n",i,x,y,z);
        }
        stack[++stp] = i;
}
```

VC-2V 及び VC-3M でメモリ空間上にデータを構築する cci_dml.c では、以下のように処理している。

リスト 3-2-4 VC-2V, VC-3M における COORD ライブラリ関数の実装

```
int fCOORD(float x, float y, float z){
        /*cci_dmlの場合、配列に蓄積するのみで、ファイル出力もデータベース登録もしない*/
        int i,beda;
//      static int kiri[MP], kanan[MP];
        static int HIT,FAIL,RANDOMFLAG;
        if(NP==0) HIT=FAIL=RANDOMFLAG=0;
        if(RANDOMFLAG){
                i = NP;
                goto nocheck;
        }
        for(i=0;;){
                if(NP==0) break;
                beda = banding(i,x,y,z);
                if(beda<0){
                        if(!kiri[i]){
                                kiri[i] = NP;
                                i = NP;
                                break;
                        }else{
                                i=kiri[i];
                                continue;
                        }
                }else if(0<beda){
                        if(!kanan[i]){
                                kanan[i] = NP;
                                i = NP;
```

```
                                        break;
                                }else{
                                        i = kanan[i];
                                        continue;
                                }
                        }
                        HIT++;
                        break;
                }
nocheck:
                if(i == NP){
                        if(NP%MP==0){
                                if(NP==0){
                                        //X = sX, Y = sY, Z = sZ;
                                        X = (float*)Malloc((NP+MP)*sizeof(float));
                                        Y = (float*)Malloc((NP+MP)*sizeof(float));
                                        Z = (float*)Malloc((NP+MP)*sizeof(float));
                                        kiri = (int*)Malloc((NP+MP)*sizeof(int));
                                        kanan = (int*)Malloc((NP+MP)*sizeof(int));
                                }else{/*注意：Reallocの場合には、解放されるブロックと新たに取得するブロック
を合わせた量の空きメモリが必要*/
                                        X = (float*)Realloc(X,(NP+MP)*sizeof(float));
                                        Y = (float*)Realloc(Y,(NP+MP)*sizeof(float));
                                        Z = (float*)Realloc(Z,(NP+MP)*sizeof(float));
                                        kiri = (int*)Realloc(kiri,(NP+MP)*sizeof(int));
                                        kanan = (int*)Realloc(kanan,(NP+MP)*sizeof(int));
                                }
                                if(!kanan){
                                        exe_err("オーバーフロー,fCOORD");
                                        return 0;
                                }
                        }
                        X[i] = x, Y[i] = y, Z[i] = z;
                        kiri[i]=kanan[i]=0;
                        NP++;
                        if(NP == 1000){//データのランダム性をチェック
                                if(HIT < 10) RANDOMFLAG = 1;
                        }
                }
                return i;
}


void COORD(){
                int i;
                float x,y,z;
                z = *((float*)&stack[stp--]);//元データをメモリと見て、送り先に書き込む
                y = *((float*)&stack[stp--]);
                x = *((float*)&stack[stp--]);
                if(MP <= NP){
                        exe_err("オーバーフロー,COORD");
                        stack[++stp] = -1;
                        return;
                }
                i = fCOORD(x,y,z);
                stack[++stp] = i;
}
```

SQL データベースを構築する VC-4D においては、次のように処理している。

リスト 3-2-5 VC-4D における COORD ライブラリ関数の実装

```
int coord(double v[3]){//一致するものがあればそのIDを、なければ追記し新しいIDを、返す（目標）
          int susun;
          Sel(hS);
          susun = TABLESIZE("coord WHERE X = %lf and Y = %lf and Z = %lf",v[0],v[1],v[2]);
          if(0 < susun){
                    Q("SELECT id FROM coord WHERE X = %lf and Y = %lf and Z = %lf",v[0],v[1],v[2]);
                    return Vint("id");
          }
          Q("INSERT coord VALUES(%lf,%lf,%lf)",v[0],v[1],v[2]);//
          Q("SELECT IDENT_CURRENT('coord') as hasil");
          return Vint("hasil");
}

void COORD(){
          int i;
//        float v[3];
          double v[3];
          v[2] = *((float*)&stack[stp--]);//元データをメモリと見て、送り先に書き込む
          v[1] = *((float*)&stack[stp--]);
          v[0] = *((float*)&stack[stp--]);
          i = coord(v);
          stack[++stp] = i;
}
```

なお、このCOORD関数は、メタファイルの書法においては、３の浮動小数を引数とすることも、１の四元数を引数とすることもできる。COORD関数の引数リストの解析は、

```
void sys_fncCall(TknKind kd); /* ライブラリ関数呼出 (cci_pars.c)*/
```

の中で実行しており、整数または浮動小数が３個であれば COORD 関数（上記の例）を、四元数が１個であれば、COORDQ 関数を呼び出し、それ以外の場合はエラーとしている。

上記のように、この COORD 関数を定義しているソースコードを、利活用処理系により３種作成して、ビルドにこのいずれかを含める方法を採った。

一方、本稿までに試作した４の利活用処理系に共通する処理は、以下の共通のライブラリ関数で記述し、全てのビルドに含めている（表 3-2-3）。

①cci_file.c： ファイル入力処理を行うライブラリ関数

②cci_face.c：頂点と面を定義する処理を行うライブラリ関数

③cci_quat.c：四元数の演算処理を行うライブラリ関数

④cci_sml.c：LSS-S 形式のファイルを入出力するためのライブラリ関数群

（５）機械語レベルで実装した数値計算の関数の書法

座標計算等に使用する超越関数等については、処理系に依存せず、高速処理する効果が高いため、機械語を追加する方法で実装した。メタファイル中の書法としてはライブラリ関数と同様のライブラリ関数として呼び出す。

表 3-2-4 仮想コンバータで追加した数値関数（超越関数）

機械語	円周率定数を取得	メタファイル中での書法

PI	円周率	float PI();
EXP	指数関数（浮動小数）	float exp(float);
EXPQ	指数関数（四元数）	quat exp(quat);
LN	対数関数（浮動小数）	float ln(float)
LNQ	対数関数（四元数）	quat ln(quat);
SIN	正弦（ラジアン）	float sin(float);
COS	余弦（ラジアン）	float cos(float);
TAN	正接（ラジアン）	float tan(float);
ATAN	逆正接（ラジアン）	float atan(float);
SIND	正弦（度）	float SIN(float);
COSD	余弦（度）	float COS(float);
TAND	正接（度）	float TAN(float);
ATAND	逆正接（度）	float ATAN(float);
SQRT	平方根	float sqrt(float);

（６）コンパイル時のメモリ

①記号表

コンパイル時には、メタファイルにおいて新たな変数や関数が宣言される度に、記号表に追加されていく。このテーブルの構造は、cci.h において次のように定義されている。

リスト 3-2-6 記号表を定義する構造体

```
typedef struct {                /* 記号表構成                    */
  char    *name;                /* 変数や関数の名前              */
  SymKind nmKind;               /* 種類                          */
  DtType  dtTyp;                /* 変数,関数の型                 */
  int     aryLen;               /* 配列長。0:単純変数            */
  char    level;                /* 定義レベル 0:大域 1:局所      */
  char    args;                 /* 関数の場合の引数個数          */
  int     adrs;                 /* 変数,関数の番地               */
} SymTbl;
```

記号表は、cci_tbl.c で宣言されており、サイズは 2000（固定）としている。これを超える数の変数名等を使用するメタファイルを扱う場合には、サイズを大きくする必要がある。

リスト 3-2-7 記号表の最大サイズの定義

```
#define TBL_MAX 2000
SymTbl table[TBL_MAX+1];      /* 記号表                */
```

このテーブルに登録される変数や関数の名前の文字列は、s_malloc 関数でシステムメモリから取得している。記号表に登録されるメモリブロックは、コンパイル終了に伴い全て解放される。

本処理系では、s_malloc 関数から、Malloc 関数を呼び出している。

この関数では、デバッグのために、「_DEBUG」が宣言されている場合には、メモリブロックをシステムから malloc 関数で取得する度に、memlist 配列にアドレスを登録している。メモリが解放された場合には、登録されたアドレスを検索して NULL に書き換えている。解放済のメモリブロックを再度解放しようとしたり、メモリブロックが存在しないアドレスを用いて解放しようとしたりすると、エラー処理する。また、システム終了時に解放されていないメモリブロック（アドレス）を検査し、ログファイルに出現順位を保存する。次にシステムを開始する際に、前回終了時に検出されたこのメモリーリークをロードしておき、同じ順位のメモリ要求に際してブレークを掛けることで、問題個所を特定している。このデバッグは、新たなライブラリ関数の追加などのシステムの改良に際して行う手順であり、メタファイルの作成時には必要ではない。

「_DEBUG」が宣言されていない場合には、単純に malloc 関数を実行するのみである。

②コード

コンパイルの結果として生成する機械語は、Inst 構造体の配列に格納されていく。構造体は、cci.h で定義されている。

リスト 3-2-8 機械語命令を格納する配列の構造

```
typedef struct {              /* 命令語      */
  unsigned char opcode;       /* 命令コード */
  unsigned char flag;         /* フラグ      */
  int           opdata;       /* 数値か番地 */
} Inst;
```

opcode(命令コード)は、表 3-1 に示した 1 バイトの機械語であり、本処理系では原著に対して大幅に増補している。

opdata は命令コードに添付する数値またはアドレスである。命令コードがオペランドを必要としない場合には、使用していない。この点は、マイクロプロセッサの機械語において多くの場合オペランドが機械語に続くメモリ上のアドレスに格納されているのと異なっている。

フラグは、opdata が変数のアドレスである場合に、大域アドレスか、局所アドレスかを識別している。 0 ならば大域アドレス、 1 ならば局所アドレスである。

機械語を格納する固定長配列は、cci_code.c でリスト 3-2-9 のように宣言されている。

リスト 3-2-9 機械語命令格納領域の配列宣言

```
Inst code[CODE_SIZ+1];        /* コード格納                    */
```

この固定長配列の長さは、cci.h でリスト 3-2-10 のように定義されている。

リスト 3-2-10 機械語命令格納領域の配列の長さの定義

```
#define CODE_SIZ 200000       /* コード格納領域サイズ 初版20000   */
```

機械語が長大になるのは、固定形状を記述する大きなメタファイルをコンパイルするよう

な場合であり、データ構造が単純であれば、データファイルが巨大であってもメタファイルは短く記述できるため、この配列を小さくすることができる。

③変数領域

実行時の変数や配列の値は、メモリ上のアドレス上に読み書きされる。このメモリは、cci_code.c において、char 型配列としてリスト 3-2-11 のように宣言されている。

リスト 3-2-11 定数、変数や配列を格納するメモリの配列宣言

```
char memory[MEM_MAX+1];      /* memory[0]～[MEM_MAX]がメモリ領域  */
```

このサイズは、cci.h において、リスト 3-2-12 のように定義されている。

リスト 3-2-12 定数、変数や配列を格納するメモリの配列の長さの定義

```
#define MEM_MAX 0xFFFC       /* メモリサイズ*/
```

memory 配列は、３の部分に分けて使用される。

a. 一番先頭の４バイトには、静的領域サイズ（整数）が記憶される。この長さは整数の長さ(sizeof(int):処理系に依存)である。

b. これに続く領域に、リテラル定数、グローバル変数、グローバル変数がコンパイル時に割り当てられていく。コンパイル開始時には、先頭アドレス４がポインタ globlalAdrs の初期値として設定され、コンパイル処理が進むに従い、この値が増加する。コンパイル終了時には、この末尾の値が先頭の４バイトに格納され、実行時の処理に使用される。

グローバル変数等の割り当ては、mallocG(int)関数により行われ、先頭から順に変数等にメモリが割り当てられる。

mallocS(char*)関数により、printf 関数、scanf 関数、logf 関数の書式文字列などの、メタファイル中で使用されたリテラル定数もこの領域に確保される。

globalAdrs が指し示すこの領域の末尾は、コンパイル処理中は次第に成長（アドレスが増加）し、コンパイルが終了すると確定する。その値は①に書き込まれる。

メタファイル中で使用される printf, scanf,logf 等の関数の書式文字列は、グローバル変数領域に、リテラル定数（固定的な文字列）として格納され、その先頭アドレスを用いて参照される。

c. 関数の中で定義され使用される局所変数と局所配列は、実行時にメモリ末尾から動的に割り当てられるため、コンパイル段階では仮のアドレスのみが決定されている。

（７）実行時のメモリ

①スタック

スタックは、整数の固定長配列として、cci_code.c においてリスト 3-2-13 のように宣言されている。

リスト 3-2-13 スタックの配列宣言

```
int stack[STACK_SIZ+1];      /* オペランドスタック                    */
```

この配列のサイズは、cci.h でリスト 3-2-14 のように定義されている。

リスト 3-2-14 スタック配列の長さの定義

```
#define STACK_SIZ 10000        /* スタックのサイズ                    */
```

実行時の数値計算はスタックを用いて行われる。単項演算の場合には、スタックのトップに対して行われた演算結果がスタックのトップに返され、スタックポインタ stp は変化しない。二項演算（例えば加算）の場合には、スタックのトップとセカンドの間で演算が行われ、結果がトップに残され、stp は 1 減少する。

関数（サブルーチン）の呼び出しに際して、戻り番地（呼び出し元のプログラムカウンタ）もスタックにプッシュされ関数のアドレスにプログラムカウンタがセットされてジャンプする。関数の処理が終了すると RET 命令により、スタックから元の番地が取り出され、プログラムカウンタにセットされ、元の文脈の中で処理が続行される。関数が戻り値を有する場合には、数値型に応じたサイズのデータとしてスタックのトップに結果が残される。

②局所変数、配列

実行時に個別の関数の中で使用される変数や引数に割り当てられる局所的なメモリには、上記の memory 配列で、コンパイル時に大域変数などに割り当てられた領域の末尾から後の残りの領域が使用される。局所変数および配列のアドレスは、関数呼び出しに際して動的に確保されるフレームの中に定義され、フレームの先頭アドレスを指す beseReg からのオフセットとしてアクセスされる。この baseReg は、memory 配列の末尾(MEM_MAX)で初期化され、関数が呼び出される度に baseReg が必要な値だけ引き下げられ、このその差分の領域に、ローカルメモリが割り当てられる。関数呼び出しの都度行われるフレーム確保の際に、メモリの先頭に記憶されているグローバル変数領域の末尾(コンパイル時に globalAdrs の終了時として memory の先頭 4 バイトに格納され、実行開始時に start_localMEM 変数に取得される)との比較を行い、領域侵犯を防いでいる。ローカル変数として長大な配列を宣言した場合や、再帰的な関数呼び出しで再帰回数が多くなったような場合に生じうる。

大きな三次元データを扱う場合には、処理系によっては大量のメモリが必要となる場合がある。しかしながら、基幹部分であるコンパイラ・インタプリタ系のメモリ必要量はメタファイルにおける変数の使用状況に依存するため、単純なデータ構造の場合には必ずしも大きなメモリを必要としない。例えば、VC-1C においては、大きなデータファイルを処理すると、大きな変換結果ファイルが作成される。VC-2V,VC-3M においては、表示処理系の大量メモリ容量の問題となる。VC-4D においては、データベースのテーブルサイズの問題となる。基幹部分のメモリ必要量はむしろデータ構造の複雑さ、バッファリングの必要性などに依存する。

（８）エラー処理

メタファイルのコンパイル時のエラーは、原著においては、err_ss 関数(cci_misc.c)により処理されている。この関数は二つの文字列を引数として受け取り、

行番号：文字列１（文字列 2）

の形でコンソールに出力を行う。派生して、第二引数を省略した err_s 関数、および書式付で整数値を表示する err_fi 関数が用意されている。

更に、エラーメッセージが出力される度に、変数 err_ct を用いて数え、総数が MAX_ERR で定義された値（cci_misc.c の冒頭部分で「#define MAX_ERR 10」と定義）を超えると強制終了している。

コンパイル・エラーが発生した場合には、compile 関数は 0 を返す。その場合には、実行段階に進む意味がないため、エラー処理に進むように各処理系でプログラムする。

コンパイルに成功し、１が返された場合には、実行段階に進む。実行段階でエラーが発生する度に、exe_err_ct に１が追加される。

実行時のエラーは、execute 関数の戻り値にも反映される。

インタープリタ処理系内部でエラーが発生した場合（スタックオーバーフローやゼロ除算等）は、処理が直ちに終了し、０を返す。

メタファイル中で、exit(int)が実行された場合には、引数が戻り値となる。

メタファイルの main 関数の中で、return int が実行された場合には、引数が戻り値となる。

メタファイルの main 関数が終了した場合には、0 を返す。

エラーメッセージは全て、プログラム中に日本語でリテラル定数として書き込まれている。処理系に依存しない共通のメッセージを cci_kanji.h で

```
#define KANJI008 "#コンパイル失敗¥n
```

の形式で定義し、これを利活用処理系で必要とする文字コード（Shift-JIS, utf-8, EUC 等）で保存して、エラーメッセージ出力箇所から参照している。

本処理系では、エラーメッセージの出力先を、利活用処理系によって指定できるようにした。多くの場合、処理に先立って予めログファイルを開いておき、そのファイルハンドルを渡して、各利活用処理系のエラー処理関数の出力先をファイルに行い、エラー個数が１以上の場合にこのファイルを表示するように処理している。

実行時のエラーは、原著においては、exe_err 関数(cci_code.c)により処理されている。この関数では、引数として渡されたメッセージ文字列を、プログラムカウンタおよびその時に実行していた機械語と共に表示する。

本処理系においては、上記のコンパイル・エラーと同様に、実行時エラーの出力先も実行前に準備したログファイルに出力するようにした。この exe_err 関数は、各種利活用処理系で実装するライブラリ関数からも呼び出すことができ、各処理系のトラブル（表示装置

の異常、データベースの不具合等）などを表示することもできる。但し、仮想現実等の表示エラーを、エラーを生じている表示装置を用いて行おうとすると、多重エラーが発生して収拾がつかなくなる恐れがあるので、注意を要する。

更に、メタファイルの中で、デバッグ用のメッセージや、メタファイル実行時に検出したデータファイルの問題点を指摘できるように、LOGF 関数を用意し、これの出力先を exe_err 関数の出力先と同じくログファイルに行うようにしている。

cci_prot.h で、outfile と stberr の二つの出力ファイルハンドルを定義している。

エラー出力先 stberr は、全ての利活用処理系において、compile 関数を呼び出してコンパイルを開始する前に開いておかなければならない。上位の処理系から引数等として渡されたファイル名が無効の場合には、必ず作成可能な固定名称等の一時的ファイルを開いて、stberr をファイルハンドルとする。

インタープリタで実行する段階では、二つのファイルが開いていれば、logf 関数の出力先を stberr とする。printf 関数の出力先は outfile ファイルハンドルとし、これが使用できない場合には stberr ファイルハンドラにエラーメッセージを出力する。

（９）ヘッダファイル

原著では、コンパイラーインタープリタ処理系で簡潔するソースコード群のためのヘッダファイルとして作成されていたが、様々な利活用処理系の中に組み込むことを想定して、以下のように再整理した。

① cci.h

コンパイラ処理系に必要なインクルード

コンパイラ処理系に共通の定数、構造体等の定義

② cci_prot.h

ライブラリ関数のプロトタイプ定義

ここでプロトタイプ定義された関数は、全ての利活用処理系において実装されていなければならず、関数型は一致していなければならない。当該処理系が必要としない関数は、ダミーとして設定する。

[例]実装例である cci_ip.c, cci_dml.c, cci_sql.c においてインクルードしている。

③cci_kanji.h

コンパイラ処理系の２バイト系メッセージ

（処理系が使用する文字コードに変換して保存する）

④cci_export.h

仮想コンバータを利活用処理系の API から利用する入口の関数（compile 関数、execute 関数等）のプロトタイプ宣言を含む。利活用アプリケーションからこの処理系を呼び出すソースコードでインクルードする。

[例]2Vwnd.cpp、mobile.c、VC-4DDlg.cpp、 VC-4D_exe.cpp

（１０）記号表ポインタ型変数

本処理系においては、標準の C 言語で利用できるような、システムのメモリ空間にアクセスできるようなポインタ型変数の使用を認めていない。しかし、内部に変数を有するデータファイルのためのメタファイルの記述において記号表を使用することは便利である。そこで、記号表へのアクセスに限定して、ポインタ型の記述ができるようにしている。

＊（式）の形の表現は、式を評価した結果の整数値がランタイムに生成する記号表の範囲（１～末尾）である場合に、記号表に登録された変数名と変数値にアクセスするために使用される。

①スキャン関数による記号表登録

rc = scanf(“%s”, word);

機械語は、LIB SCANF_F である。その処理内容は、int SCANF()関数(cci_file.c)に記述されている。引数の書式文字列のメモリ上の番地と、word のアドレスがスタックを通じて渡される。

この関数により、データファイルから文字列（例えば”HOUSE1”）が取得され、記号表の登録番号が word に格納され、整数型のデータ格納域が固定的に確保される。この文字列が既に登録されている場合には、その番号が返され、未登録の場合には新たな番号となる。

記号表にはスキャンされた文字列「HOUSE1」の名前を有する整数型変数が登録され、対応する 32 ビットのメモリが割り当てられる。この時、コンパイル時点で記号表に登録された”word”の名称はコンパイル終了時点で既にクリアされ、word はアドレスとしてのみ存在している。この word の格納域に、新たに登録された変数の ID 番号が格納される。この番号が仮に３であれば、*word により、この記号表の要素 table[3]つまり「HOUSE1」にアクセスすることができる。内部的には、ポインタ*word は、MULTI_T という特殊な型として処理されている。

②ID 値

printf(“%d”,word)では、word に格納された値「３」が出力される。これが、新たに記号表に登録された、ランタイムの変数の記号表における ID（添字）である。

③記号表へのアクセス

例えば上記のように新規に登録され word に格納された記号表のアドレス（配列の添字）が３である時に、 printf(“%s”, *3), printf(“%s”,*word)は、記号表に登録された「HOUSE1」を出力する。また、printf(“%d”,*3)は、ランタイムに登録された変数 HOUSE1 に対応する値を出力する。

*word = 123; で、HOUSE1 変数に１２３が代入される。よって、

*word = GROUP();

というコマンドを実行することにより、HOUSE1 変数に、GROUP の ID が登録される。

これを用いて、GROUP_FACE(*word, *face1); のように、記号表[face1]に登録されている面を、記号表[word]に登録されているグループオブジェクトに対して割り付ける。

④ポインタの内部処理と機械語

コンパイル終了時点では、メモリ上には、メタファイルが使用するグローバル変数の格納域と、リテラル文字列が格納されている。その末尾の後に、ランタイムに新たに記号表に登録された変数の格納域が追加されていく。

マシン語レベルでは、例えば VERTEX 関数は、頂点座標、カラーID 等を記入した変数を引数とする。この時、それぞれの引数のアドレスが、LDA スタックに積まれて、機械語

LIB VERTEX_F

が実行される。この引数としてメタファイル中の変数名が直接列挙されている場合には、引数は、LDA　アドレスの並びとしてスタックに積まれる。一方、引数として、*word の表現が行われた場合には、

LOD word　・・・word 変数に格納されている値をスタックトップに

TVAL　・・・記号表[値]のデータ格納アドレスをスタックトップに

以後、このアドレスの値を取得し、値を書き込むことができる。

記号表を書き換え、登録されたデータ格納アドレスを変更する機械語は存在しない

LOD word

TNAME　・・・記号表[値]の文字列格納アドレスをスタックトップに

スキャンした文字列を表示してデバッグ等に使用することができる。

文字列自体を書き換えるような機械語は存在しない。

この機械語は、例えば

printf(“%s”, *word);

のような形式で、記号表の文字列が書式文字列で要求した場合に、コンパイラにより生成される。

*word = 12345;

では、記号表[値]が指示するデータ格納域に格納された値が変更されるのみである。

printf(“%d”, *word);

のような形式でこの値を出力することができる。

ix = *word;

により、データ格納域の値を別のメタファイル変数 ix に代入することができる。

if(*word <10000000)｛・・・｝

のように条件判定に使用することができる。

func(*word);

のように、関数呼び出しの整数型引数として使用することができる。

⑤関連ソースコード

コンパイル段階で、expression 関数(cci_pars.c)の中で、ポインタ型の変数を処理し、MULTI_T という型を与える。

printf(“%s”, *(式)) ; の形式で評価された場合にのみ、記号表の文字列を参照する。この

場合には、機械語の TNAME が追加される。

その他の参照が行われた場合には、整数型として記号表の変数格納域の値を参照する。この場合、＊に対応して、機械語の TVAL が追加される（表 3-2-1、p.3-8)。

実行段階で、expression 関数の冒頭で、記号表がゼロクリアされ、スタティック変数に、コンパイル時に生成されたリテラル文字列やメタファイル中のグローバル変数の格納域の末尾がローカル変数 start_localMEM に格納されランタイムの間記憶される。

ここを先頭としてランタイムで生成される記号表の登録文字列や変数格納域が成長し、その末尾がグローバル変数 globalAdrs に格納される。ポインタのアクセス先がこの範囲の外にある場合には、実行時エラーとしている。

記号表のサイズは、グローバル変数 tableCnt に記憶され、ランタイムに 1 に初期化されている。ランタイムには、ランタイム変数の登録に従いこの値が増加する。ポインタ型での参照が行われた場合には、アドレスが 1 以上 tableCnt 以下であることをチェックし、この範囲外である場合には、実行時エラーとしている。

３－３．VC-1C の開発

（1）試作段階でのコンソール・アプリケーション

開発の初期の段階で、コンパイラ、インタープリタの動作を確認すると共に、個々のライブラリ関数を作成し、既知のサンプルデータを用いて動作確認を行うために使用した。

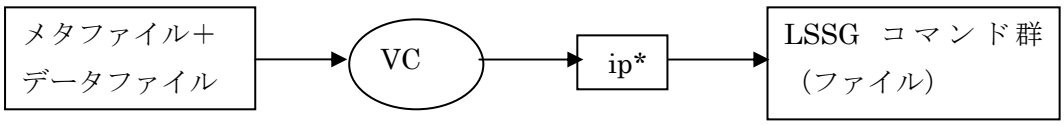


図 3-3-1　VC-1C による処理 (*ip は、cci_ip.c によるライブラリ関数の実装)

VC-1C の実行形式は、VC-1C.exe というコンソール・アプリである。コンソール（Windows では command.com または cmd.exe）からキーボードを用いてコマンドを入力する。

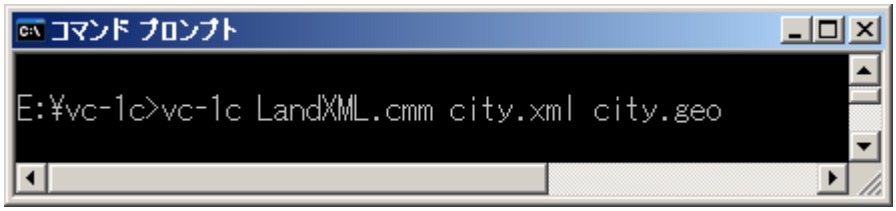


図 3-3-2　コンソールからの VC-1C の起動（変換実行）

(city.xml というデータファイルを、city.geo に変換する)

コマンドラインの引数は 3 である。

通常の場合、第一引数はメタファイル、第二引数はデータファイル、第三引数は、処理結果やエラーメッセージを出力するファイルである。main 関数の中では、第一引数に指定されたメタファイルをコンパイルし、実行形式を生成する。コンパイルに失敗するとここ

で終了する。コンパイルに成功すると、次の段階に進み、生成された機械語が仮想マシンの中で実行され、第二引数で指定されたデータファイルを解釈する。その結果は第三引数で指定されたファイルに格納される。

デバッグ時には、第二引数として、「--code」という文字列を入力し、コンパイルの結果生成する機械語をコードダンプして第三引数で指定したファイルに出力する。

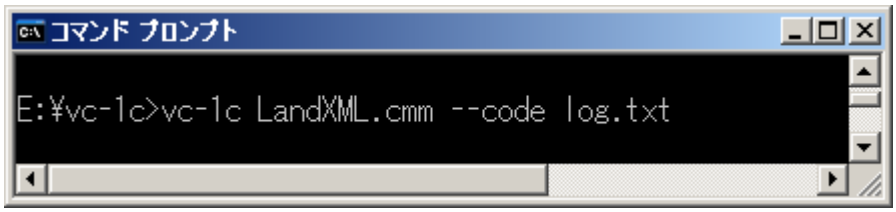


図 3-3-3　コンソールからの VC-1C の起動（デバッグ）

（LandXML.cmm というメタファイルをコンパイルし、ダンプリストを log.txt に出力する）

なお、cmd.exe は、MS-DOS において command.com がユーザーキーボード入力した指令を文字列として解釈して実行していた操作環境を、Windows 上の一つのサブ・ウィンドウとして継承したもので、その後 PowerShell.exe に発展している。

開発環境としては、VS2005 を使用し、OS は Windows Vista を主に使用し(2012 年まで)、後に Windows8.0 でデバッグを行った(2013 年以降)。なお、VS2005 の後継である開発環境 VS2008 は、操作性等において VS2005 から大きく変化しておらず、新たに追加された機能を開発に必要としなかったため、VS2005 を継続使用した。完成後の可搬性検証段階では、最新のものを含む様々な開発環境の上でテストすることは重要であるが、初期の開発着手から一定の安定動作を実現するまでの間は、たとえ古くとも、枯れて安定した開発環境を使用する方が能率的である。開発完了後に新しい開発環境に移行することは容易（ほぼ自動的）であるが、逆に新しい開発環境における成果を古い開発環境に戻してテストすることは、ビルドの再定義等の手入力が必要である。

LSS-G 形式のファイルは、データファイルにかかわりなく固定形状を記述した一種のメタファイルと見なすことができる。その内部においては変数や関数は存在せず、定数値を引数とするライブラリ関数だけから構成されている。LSS-G 形式のファイルをメタファイルとして読込んだ結果出力されるファイルは、景観シミュレータにおいて同じ表示を与えるものでなければならない。

ある形式の三次元データファイルを解読するメタファイルを作成し、解読結果を VC-1C におけるライブラリ関数の実装形態で処理することにより、他の実装形態（VC-２V 等）において「データファイル＋メタファイル」のセットと同様の作用を、メタファイル単独で行うことのできるような「固定形状を記述するメタファイル」を得ることができる。

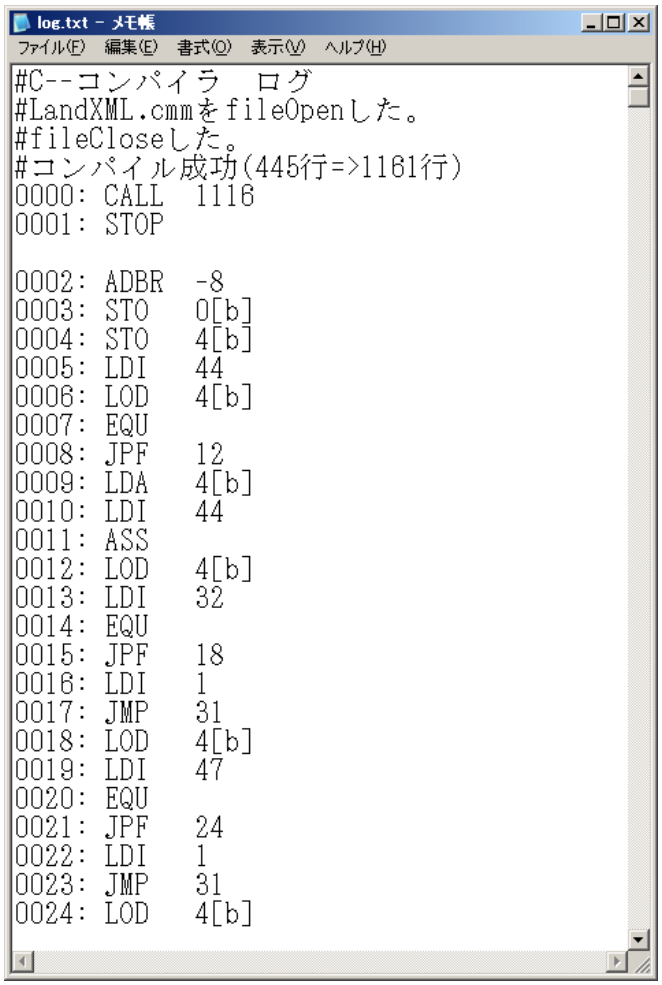


図 3-3-4 デバッグモードで作成した、機械語ダンプリストの冒頭部分

VC-1C におけるライブラリ関数の動作は、cci_ip.c の中で定義されている。LSSG コマンドをファイル出力する。これを景観シミュレータでファイルとして読み込み、正しく変換されていることを確認する。

例えば、P0=COORD(1.0, 2.0, 3.0);　というメタファイルの１行が実行されると、出力ファイルに、同じ内容の１行が出力される。正しく変換されていれば、このファイルを、景観シミュレータの LSS-G ファイルとして読み込むことができ、形状を確認することができる。

この実装形態は、仮想コンバータ開発の最も初期にコンパイラの開発とテストのために作成したものである。VS2005 のデバッガを用いて実行状況をトレースすることにより、バグがコンパイラ処理系によるものか、インタープリタによるものか、メタファイルによるものかを切り分けて解決する作業を行った。

（２）景観シミュレータの外部関数としての活用

現在は実用的な意味を持たせるために､パラメータ入力用のダイアログ vc-1c_D.exe を付して、景観シミュレータの外部関数の一つとして追加してある。

外部関数は、ユーザー定義による各種パラメトック部品の実行形式であり、景観シミュレータ本体から起動する。景観シミュレータの実行形式を変更してバージョンを多様化することなしに、選択的な機能を追加するための、初期から用意した仕組みである。

関数は「外部関数のリスト」(ext.tab)に追記することにより、景観シミュレータのメニュー[形状生成][オプション]から選択できるようになる。

外部関数を景観シミュレータから起動するためには二つの方法がある。

1) ＬＳＳ－Ｇ形式の外部ファイルの中での FILE 記述による起動

以下のような形式で外部関数を起動するコマンドを記述することにより、このファイルをロードする過程で子プロセスとして外部関数を実行する。

リスト 3-3-1 VC-1C の外部関数としての起動

```
（一般形）
グループ名称=FILE(外部関数名, パラメータ１, パラメータ 2, ・・・・);
（コンバータ STL2LSS.exe の場合）
グループ名称=FILE(STL2LSS,変換元ファイル名.stl);
（VC-1C.exe の場合）
グループ名称＝FILE(VC-1C, STL.cmm, 変換元ファイル名.stl);
```

外部関数には小数のパラメータから複雑な形状を生成するためのパラメトリック部品とならんで従来、各種のファイルコンバータ（入力元のファイルはそれぞれの固定的な形式）が登録されている。外部関数名と、パラメータの数およびデータ型を、外部関数と同じディレクトリに配置されたテキスト形式の登録テーブル ext.tab の中に１行を追加登録する。

リスト 3-3-2 外部関数 VC-1C の ext.tab への登録

```
（一般形）
FILE(外部関数名,パラメータ１の型,パラメータ２の型,・・・・);
（コンバータ STL2LSS.exe の場合）
FILE(STL2LSS, STRING);
（VC-1C の場合）
FILE(VC-1C, STRING, STRING);
```

この表に登録されていない外部関数が要求された場合には、エラーとして処理する。

2) パラメータ設定ダイアログを通じた起動

パラメータ設定用のダイアログの実行形式(Windows アプリ)を、外部関数実行形式(「外部関数名.exe」という名称のコンソール・アプリケーション)とは別に「外部関数名_D.exe」

という名称で用意した(図 3-3-8)。これは各種の外部関数と同様の扱いである。

起動に際しては、景観シミュレータのプルダウン・メニューから[形状生成][オプション]をまず選択する(図 3-3-5)。

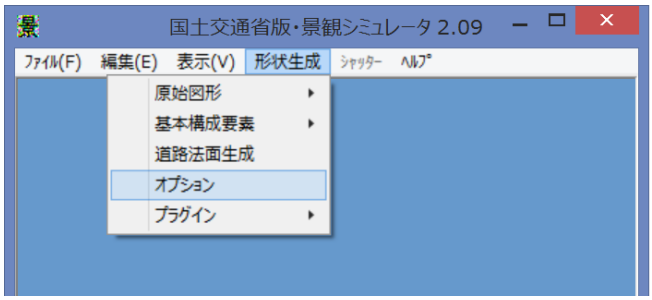


図 3-3-5 景観シミュレータの[形状生成][オプション]で外部関数の一覧を表示

この時、ext.tab に登録された関数の一覧が表示されるので、VC-1C を選択すると、VC-1C_D.exe が起動し、パラメータ設定画面が開く(図 3-3-8)。パラメータを入力した上で、このダイアログの[実行]ボタンの操作により、VC-1C.exe が起動する。

実行ボタン操作後、このダイアログに入力されたパラメータから、リスト 3-3-1 に示した形式の 1 行を持つ一時的なファイル VC-1C.geo がテンポラリディレクトリ(ksim/temp)に作成され、ダイアログが終了し、景観シミュレータに制御が戻る。景観シミュレータは引き続きローダを起動する。ローダはこの行を実行して、1)の場合と同じ形状生成の処理を行い、現在表示されているシーンの中に生成したオブジェクトを追加して終了する。

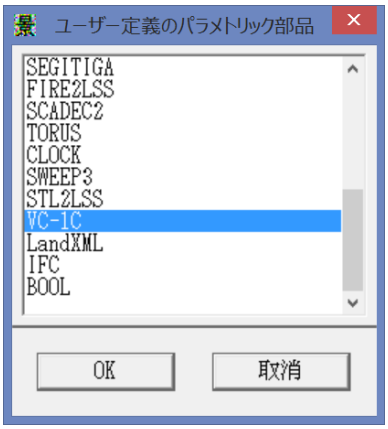


図 3-3-6 ユーザー定義のパラメトリック部品から、VC-1C を選択

3)生成したオブジェクトの選択によるパラメータの再設定

過去に外部関数を用いて生成した物体が景観シミュレータの画面に表示されている時に、これを画面から左クリックで選択し強調表示された状態で、更に右クリックによりポップアップするメニューから、「オプション」を選択すると(図 3-3-7)、パラメータ設定用のダイアログの実行形式が再度起動され、現在設定されているパラメータを表示する(図 3-3-8)。

これにより、修正したパラメータ(VC-1C の場合には、形式定義=メタファイル名または形状記述=データファイル名)を用いて解読→形状生成の処理を再実行することができる。

景観シミュレータから起動するコマンドの指定の中には、変換結果を出力するファイル名が明示的に示されていないが、これは内部的に外部関数名から、VC-1C.g という一時的なファイル名を自動生成した上で、実行形式 VC-1C.exe を起動する段階でパラメータに第3引数(出力ファイルの名称)として付け加えていることによる。

ダイアログ部でパラメータが決定され、実行ボタンが押された時点で生成した VC-1C.geo という、上記の FILE コマンドの１行だけから成るファイルは、変換が失敗しダイアログ部が再度開いた段階で、VC-1C_D.exe がファイル名入力欄に初期表示する文字列に使用する。また、既存の変換結果を選択して、パラメータを再設定する場合にも、このファイルが作成され、ダイアログ部が開いた時点での表示の初期化に使用される。

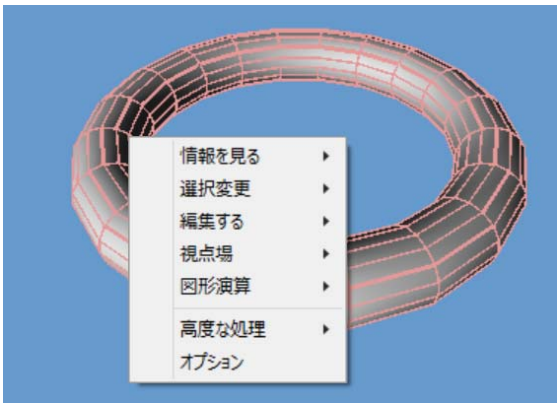


図 3-3-7 既存の変換結果を選択して、パラメータを再編集する場合の操作

ダイアログからパラメータを設定して生成した物体を含むデータをＬＳＳ－Ｇ形式で保存すると、1)で示した形式の１行としてこの物体の保存が行われる。

VC-１C は特殊なファイルコンバータであり、入力元のファイルは任意形式であり、メタファイルで解読方法定義した任意のファイル形式のデータを入力することができる。

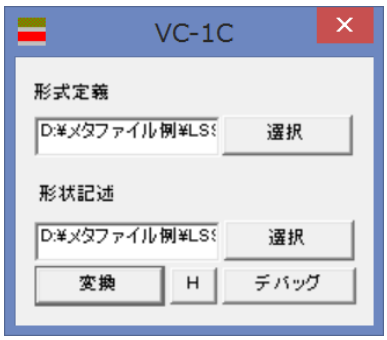


図 3-3-8 パラメータ設定ダイアログ(VC-1C_D.exe)

図 3-3-8 に示したダイアログ画面は従来の他の「ユーザー定義のパラメトリック部品」のダイアログ部と同様に、VC-1C_D.exe という独立した実行形式であり、VC-１C.exe という変換処理を行う実行形式に受け渡す引数を入力するだけの機能を有する。

右下のデバッグを選択することにより、起動元の景観シミュレータに制御を戻すことなく、このダイアログの中で変換を行い、ログファイルを表示し、メタファイルのデバッグを行うことができる(図 3-3-9)。

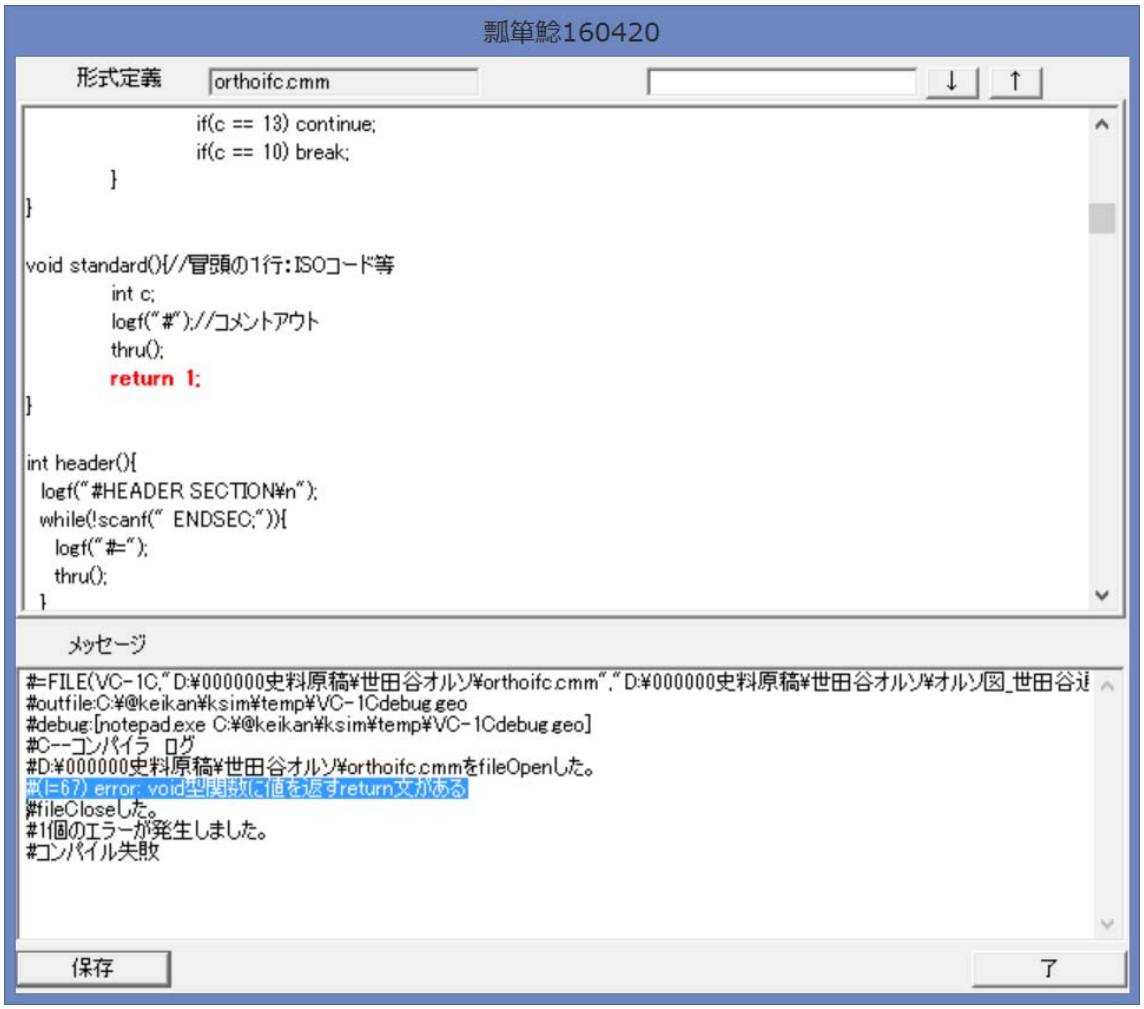


図 3-3-9 VC-1C_D.exe のデバッグ画面

（３）パラメータ設定ダイアログの単独起動

更に、景観シミュレータからの起動ではなく、Windows 上で、このダイアログ部 VC-1C_D.exe を単独のアプリケーションとして起動することも可能である。この場合、変換結果は、固定名称の VC-1CDebug.geo としてテンポラリディレクトリに残されるため、変換結果を検査することができる。また、デバッグ・ボタンを選択することにより、メタファイルを修正する編集画面を開くことができるように改良した。これにより、メタファイルのデバッグ等を行う簡単なツールとして使用することができる（図 3-3-9）。

表 3-3-1　VC-1C の用途と処理内容

用途	メタファイル	データファイル	出力
テスト・デバッグ(固定形状を記述したメタファイルとして実行)	LSS-G ファイル	ダミー	LSS-G ファイル（メタファイルと同じ内容）または、機械語リスト（--code オプション）
簡単なパラメトリック部品の試作	パラメータから形状を生成するプログラム	小数のパラメータを格納した小さなファイル	生成したパラメトリック部品
ファイルコンバータ	コンバータ処理	各種データファイル	LSS-G ファイル
汎用スクリプト言語としての用途	プログラム	パラメータ	printf を使用して各種ファイルを生成

表 3-3-2 VC-1C の起動方法

1. コマンドラインからの直接起動
2. パラメータ設定ダイアログ VC-1C_D.exe からの起動（変換モード、デバッグモード）
3. 景観シミュレータからの起動
3-1.　LSS-G ファイルに埋め込まれたファイルコマンドからの起動
3-2.　形状生成メニューからパラメータ設定画面 VC-1C_D.exe を開く起動
3-3.　生成されたオブジェクトを選択して、パラメータの再設定から VC-1C_D.exe を開く

VC-1C のビルド（VS2005）を表 3-2-3 に、メイクファイルをリスト 4-5-1 に示した。コンパイラ・インタプリタ系のソースコード（原著＋増補）だけによって構成されている。

景観シミュレータのための外部関数の終了コードは、一般に次のように定められている：

表 3-3-3 外部関数エラーコードの一般則

０：未定義のエラー
１：正常終了
２：引数の数が、当該外部関数が必要とする数と合わない場合
３：出力ファイルが開けない場合

外部関数を起動する景観シミュレータ(simexe)ではこの終了コードを見て、1 の場合には生成した形状を入力・表示して終了し、それ以外の場合にはダイアログから起動した場合にはダイアログに戻り、LSS-G ファイル入力処理中の場合には、別途エラー処理を行う。

原著における main 関数は、cci.c に含まれている。main 関数は、通常 4 の引数が与えら

れる｛VC-1C.exe のフルパス、メタファイル名、データファイル名、出力ファイル名｝。実装では、５以上の引数が与えられた場合に、出力ファイルを最後の引数とし、４番目の引数をデバッグなどに保留し、現在は無視している。

従って、VC-4C メタファイル --code データファイル ダミー引数 出力ファイル

のようなコマンドを与えている。

景観シミュレータの外部関数として実装するために、main 関数のリターンコードをリスト 3-3-3 のように定義した。

リスト 3-3-3 main 関数の戻り値の意味

０：エラー １：成功 ２：引数の数が合わない場合 ３：出力ファイルが開けない場合 ４：メタファイルが開けない場合 ５：データファイルが開けない場合 ６：ログファイルが開けない場合 ７：メタファイル・コンパイルエラー ８：実行時システムエラー その他：メタファイルの exit, return 命令により返されたリターンコード

３－４．VC-2V の開発

（１）DML ライブラリを用いたメモリ上のデータ構築

この実装では、ライブラリ関数は、cci_dml.c で記述したライブラリ関数により処理系のメモリ空間上に、OpenGL で表示を行うためのオブジェクトを生成する。このオブジェクトには、頂点座標、面、立体、カラー、法線、テクスチャ、属性等が含まれる。

入力動作は、コンソール・アプリとして一度 LSS-G 形式のファイルを生成してから読み込む VC-1C と同様であるが、処理結果が直接メモリ上に形成され表示に使用されるため、ファイル出力・ファイル入力の処理が不要となり、それだけ処理速度は向上する。

（２）景観シミュレータ用のプラグイン DLL としての実装

具体的な実装例として、景観シミュレータのためのプラグイン.dll の仕様をもつ VC-2V.dll を試作した。この Windows のための dll は、メタファイルとデータファイルの組を入力して表示を行う、という点に関しては、VC-1C と同様であるが、プラグイン DLL であることから現在メモリ空間内にあるオブジェクトにアクセスすることができる。したがって、メタファイルで定義した任意の保存形式で、現在表示されている建物などを指定した名前のファイルとして出力することができる。

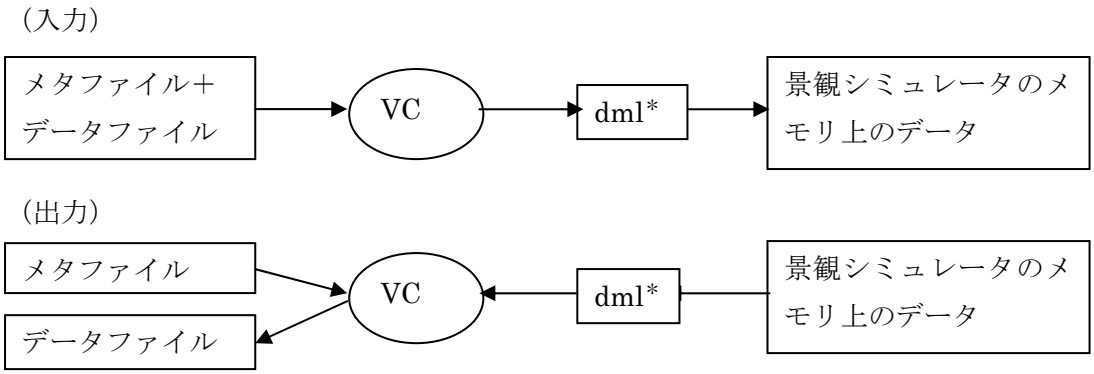


図 3-4-1 VC-2V による処理 (*dml は、cci_dml.c によるライブラリ関数の実装)

ユーザーは、景観シミュレータのプルダウン・メニュー[形状生成][プラグイン]で表示される一覧から、VC-2V を選択することにより、起動する（図 3-4-2)。

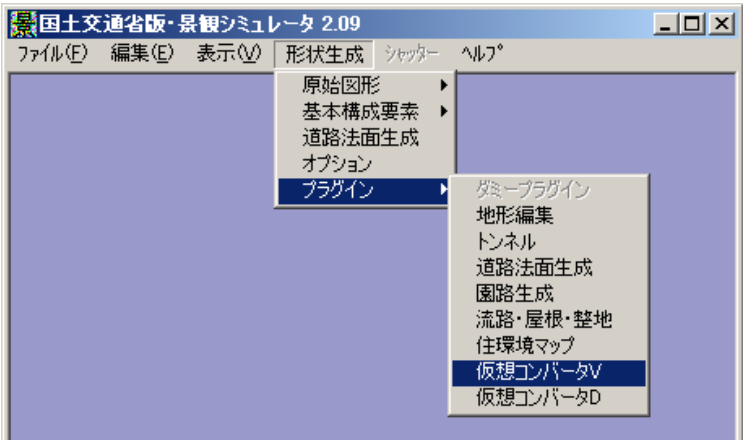


図 3-4-2 VC-2V の起動

メタファイルとデータファイルをファイル選択ダイアログまたは直接キーボード入力により選択し、入力か出力かをボタンで選択する(図 3-4-3)。

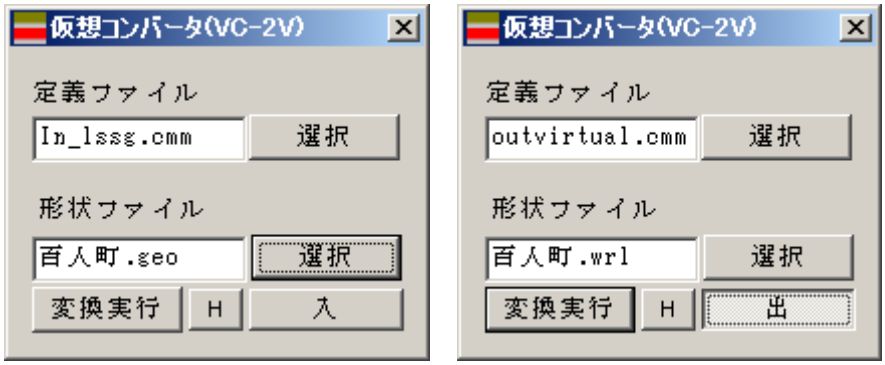


図 3-4-3 VC-2V 画面（左：形状ファイル入力　右：形状ファイル出力）

入力において定義ファイルと形状ファイルを指定して変換実行を行うと、コンパイラにより実行形式が生成され、これを用いたデータファイルが入力され、メイン画面に表示さ

れる(図 3-4-4)。

出力において定義ファイルと形状ファイルを指定して変換実行を行うと、コンパイラにより実行形式が生成され、これが実行されることにより、現在表示されているメモリ上の三次元データが、変換され指定された名称のデータファイルとして出力保存される。このデータファイルは、定義ファイルにより任意の形式とすることができる。

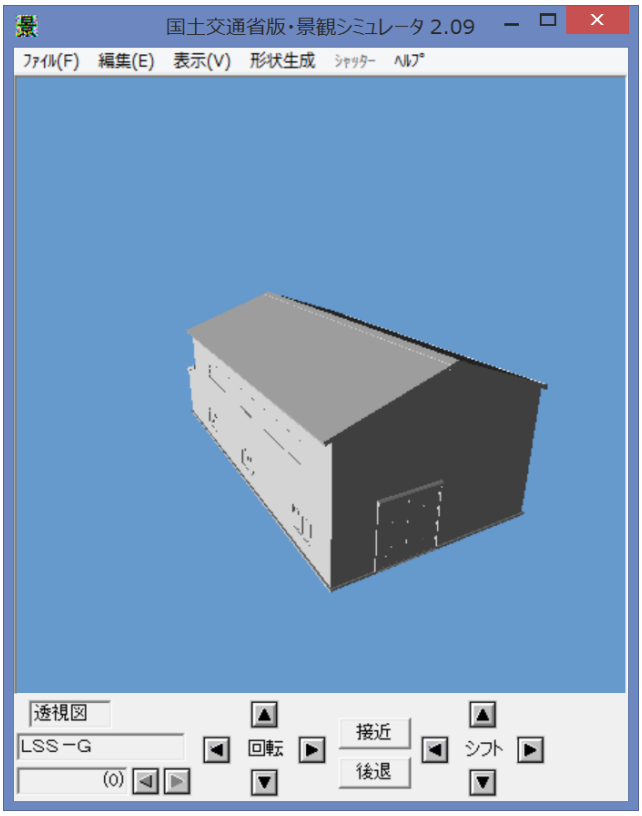


図 3-4-4 データファイル解読結果の表示

（３）出力系関数群の実装

この出力側の変換もまた、利活用形態の一つであり、後述の VC-4D においてまずライブラリ関数の作成と動作のテストを行った。ここで関数使用を定めてから、同じ定義ファイルを用いて出力処理を実行することができるように、VC-2V のライブラリ関数を作成した。なお、(4)VC-3M においても原理的には同じような出力処理を行うことができるが、携帯端末上に生成したデータファイルの実用的な意味が当面ないため、現在はまだファイル名を指定して保存するような操作環境を用意していない。

コンパイラ基幹部分が出力するエラー等に関するメッセージは、景観シミュレーション・システムの一時的ファイル格納用ディレクトリ（デフォルト ksim/temp）に CClog.geo という名称のファイルに保存され、変換処理が終了した時点でテキストエディタを起動して表示する。よって、VC-2V は、メタファイルをデバッグするための作業環境として使用することができる。

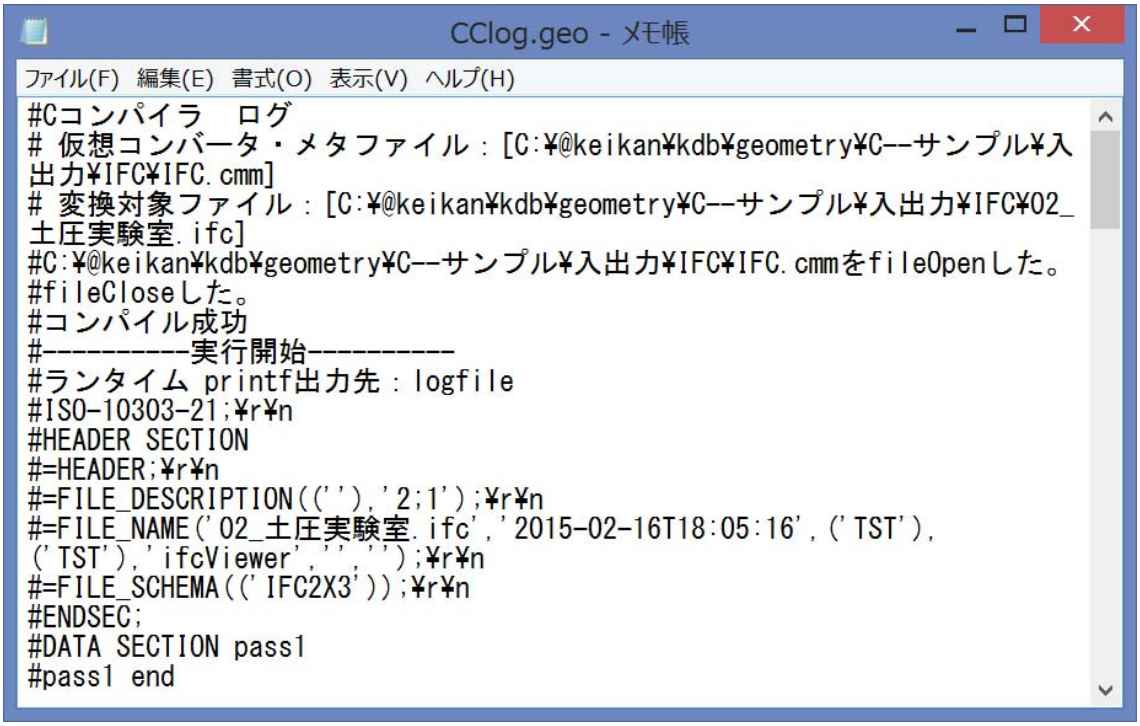


図 3-4-5 テキストエディタによるエラーメッセージ等の表示

データファイルが空欄のまま実行された場合には、コンパイルだけが実行され、生成したコードをテキストエディタで表示して終了する。この動作は、VC-1C において、第二引数として「--code」が入力された場合と同様である(図 3-4-6)。

この機能は、基幹部分のデバッグを行う場合に有用である(図 3-4-7)。

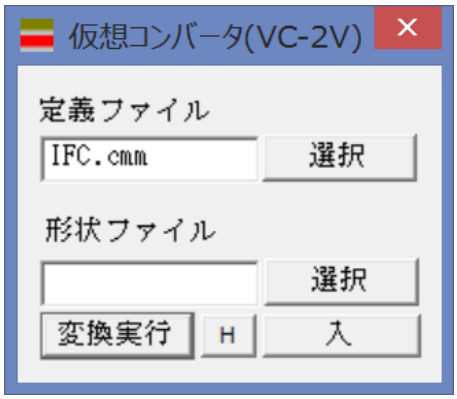


図 3-4-6 機械語表示の指定（形状ファイル入力欄を空欄のまま入力変換実行）

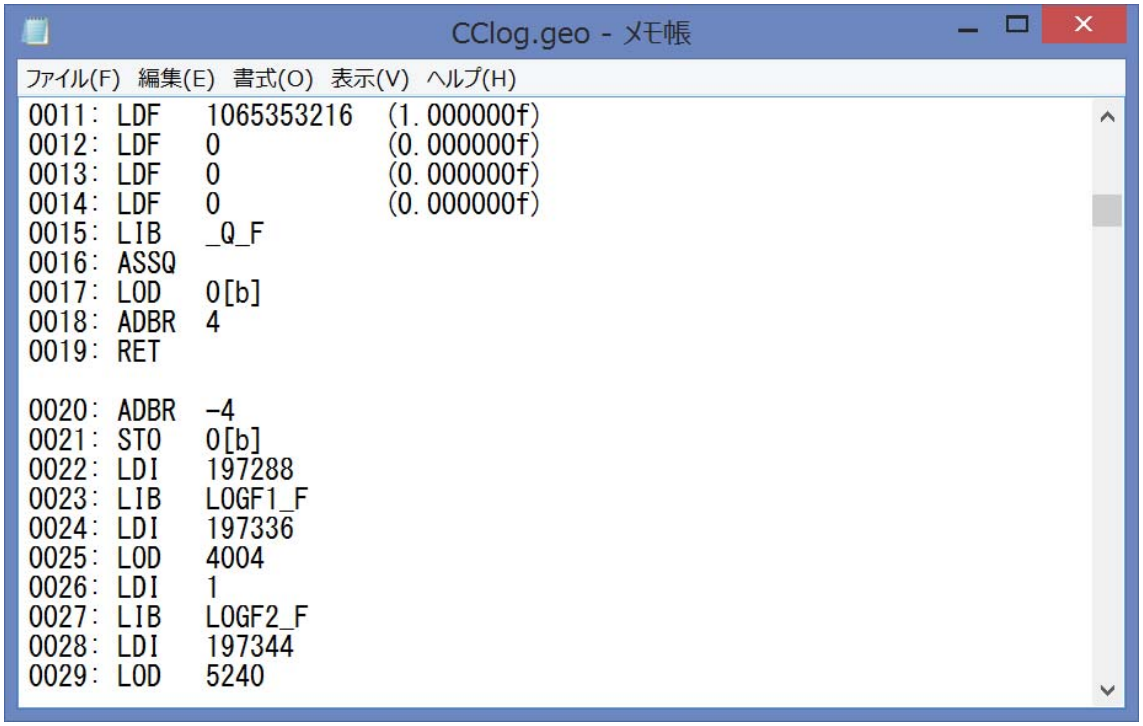


図 3-4-7 テキストエディタによる機械語の表示

実行ボタンが押されると、コンパイル・変換処理のスレッドが開始される。処理が正しく終了すると、入力の場合にはメモリ上にインタープリタがデータファイルを解釈して生成した建物等が画面上に表示される。

表示方法や、ロード後の建物等データの編集操作は、景観シミュレータにおける通常のデータと同様である。偏光液晶パネルを備えた立体視可能なディスプレイが接続されている場合には、立体視を行うこともできる。

VC-2V.dll は、VS2005 を開発環境とし、MFC を用いてプログラミングされている。コンパイラ基幹部分のソースコード群に加えて、DllMain 関数を有し、ウィンドウを構築する VC-2V.cpp、ユーザー操作等のコールバックを処理する 2VWnd.cpp を有する。また、これらに付随したヘッダファイル等を含む。

表 3-4-1 VC-2V のソースコード等

①VC-2V.cpp 景観シミュレータのプラグイン DLL にほぼ共通の基本処理 この DLL の開始と終了の処理
②2Vwnd.cpp 表示するダイアログにおけるユーザーインターフェース
③VC-2V.h ①のクラス等を定義したヘッダファイル
④2Vwnd.h②のクラス等を定義したヘッダファイル
⑤stdafx.cpp 開発環境が自動的に生成する共通ソース
⑥stdafx.h 開発環境が自動的に生成するヘッダファイル

⑦VC-2V.rc ダイアログのレイアウト等を定義したリソースファイル
⑧VC-2V.rc2 同上
⑨VC-2V.dll.ja.xml 日本語のダイアログ文字の定義ファイル
⑩VC-2V.dll.id.xml 外国語のダイアログ文字の定義ファイル（インドネシア語）
⑪VC-2V.ja.txt 日本語のヘルプファイル
⑫VC-2V.id.txt 外国語のヘルプファイル（インドネシア語）

cci_dml から呼び出される、景観シミュレータの DML ライブラリ関数（表示データ構築に関するもの）は、この DLL の呼び出し元であるアプリケーション本体 sim.exe の内部にビルドされた関数を呼び出すことにより実行している（ビルドに sim.lib を加えることにより実現）。よって VC-2V のビルドには、DML 関連のソースコードは含まれていない。

ポイントクラウド等の大きなデータファイルを扱う場合には、数分～数十分の処理時間がかかる場合があり、進捗状況が把握できないと、正常に処理が行われているかどうかの疑念が生じる場合がある。そこで、処理中断の機能、およびプログレス・インジケータの表示機能を追加した。

具体的にはインタープリタの処理系に Debug 関数と、TraverseDebug 関数(cci_code.c)を追加して、インタープリタの execute 関数(cci_code.c)による機械語の実行ステップのループの中で必ず Debug 関数を実行し、戻り値が非ゼロならば終了するようにした。この Debug 関数は、初期状態では何もしないが、TraverseDebug 関数により処理関数が登録されると、その関数を実行する。

VC-2V の操作画面で実行ボタンが押されると、まずデバッグ関数を引数として TraverseDebug 関数を呼び出して、VC-2C のためのデバッグ関数の登録を行う。デバッグ関数の中では、ダイアログ側から、変換データファイルへのアクセス位置をファイル全長で除した割合を、画面に表示する(図 3-4-8)。さらに、実行中にユーザーにより再度実行ボタンが押され、中断が要求された場合には、この寄り道処理の中で処理を中断するようにした。これにより、大規模なファイルのためのメタファイル作成の能率を上げることができる。

デバッグのための関数を登録する TraverseDebug 関数が実行されていない処理系においては、インタープリタの execute 関数は従来通りの動作を行うため、処理速度も従来とほとんど変わらない。

登録されたデバッグ関数を解除するためには、TraverseDebug(NULL)を実行する。

具体的には、ダイアログの実行ボタンが押されたタイミングで、コールバック関数である OnBnClickedOK 関数(2Vwind.cpp)の中で変換処理開始前に&dumy 関数を引数として TraverseDebug を呼び出す。この dumy 関数は、2Vwnd.cpp の中で static int 型として定義されており、データファイルへのアクセス位置をデータファイルの長さで除した値を、データファイル名欄に表示するような動作を行う。また、再度実行ボタンが押されているかをチェックし、もし押されていたら処理を中断して変換を直ちに終了する。

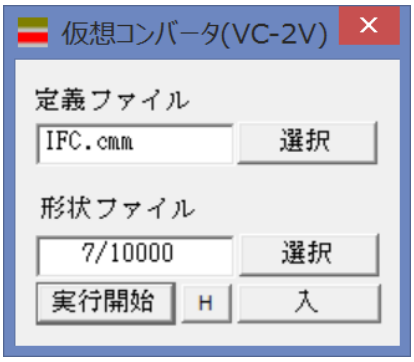


図 3-4-8 プログレス・インジケータ

メタファイルをコンパイルして生成した機械語のプログラムは、cci_code.c の execute 関数の中のループで順次フェッチ（取得）され実行されている。このループの中で Debug 関数が次の機械語の実行に先立って実行されている。

リスト 3-4-1 プログレス・インジケータの処理

```
//cci_code.cで、プログレス・インジケータとブレーク処理に関連する部分

int execute(char *fname) /* プログラム実行 */
{
------------------中略-----------------
  for (;;) {
    if(0 < exe_err_ct) break; //140622
    if(DebugFunction) if(Debug(Pc)) break;//140624
    if (Pc<0 || codeCt<Pc) {
                    fprintf(stberr, "# Pc=%d, codeCt=%d¥n", Pc, codeCt);/* exit(1); */     /* 念のため */
                    exe_err("プログラムカウンタが範囲外");
                    break;
    }
------------------中略-----------------
    機械語のフェッチと実行ループ
------------------中略-----------------
  }//end for
}//end execute
```

この Debug 関数は、DebugFunction が定義されていれば実行し、定義されていなければ（関数へのポインタが NULL)、何もせずに終了する。

リスト 3-4-2 デバッグ関数の動的登録

```
static int (*DebugFunction)() = NULL;//登録デバッグ関数へのポインタ。未登録ならNULL
void TraverseDebug(int (*func)(int pc,int pos, int len))
{//実行時のプログレス・インジケータ情報を受け、ブレークならTRUEを返す関数を登録する
   DebugFunction = func;
}

static int Debug(int pc){
          if(DebugFunction){
                    if(0<FCNT) return (*DebugFunction)(pc,SIORI(),FCNT);//戻り値がTRUEならブレーク
                    else return (*DebugFunction)(pc,0,0); //出力系の場合（とりあえず）
          }else return FALSE; //デバッグ関数登録が無ければブレークなし
}
```

DebugFunction は、プログラムの実行に先立って、TraverseDebug という関数により、Debug 関数を外から与えることができる。VC-2V における実装例として dumy という関数を用意した。この関数は、プログラムカウンタ、データファイルへの現在のアクセス位置、およびデータファイルの長さを取得し、中断が必要ならば１を、継続するなら０を返す。

中断は、実行中に再度実行ボタンが押されると、g_BREAK という大域変数を初期値０から１に変更することによって指示する。dumy 関数は、この値を返すことにより、インタープリタの execute 関数に中断／継続の指示を行う。

リスト 3-4-3 ブレーク処理

```
//2VWND.cppで、実行ボタンが押された時の処理

int g_PC, g_POS,g_BREAK;
#define 実行中g_bRunning
#define 停止中g_BREAK
HANDLE hThread = NULL;
DWORD ThreadId;

// 解読処理実行中にプログレス・インジケータが更新されるように独立したスレッドでメタファイルを実行する
void CreateAndStartThread(){
          g_iCount = 0;
          g_bRunning = true;
          hThread = CreateThread(NULL,0,MyThreadFunc,NULL,0,&ThreadId);
}

void C2vWnd::OnBnClickedOk(){
          // TODO: ここにコントロール通知ハンドラコードを追加します。
          char *path1,*path2;
          int hasil;
          char s[200];

          if(実行中){ //●実行中に再度実行ボタンが押されるとブレークを予約する
                    g_BREAK = 1;
                    GetDlgItem(IDOK)->SetWindowTextA(Kanji("RESET"));
                    return;
          }

          if(停止中){
                    g_BREAK = 0;
                    GetDlgItem(IDOK)->SetWindowTextA(Kanji("START"));
                    SetDlgItemText(IDC_EDIT_GEOFILE,g_geofile);
                    return;
          }

          GetDlgItemText(IDC_EDIT_GEOFILE,g_geofile,80);

          sprintf_s(s,200,"%s¥¥CClog.geo", e3GetEnvParam(E3_TEMP_PATH)->path);
          fopen_s(&stberr,s,"wt");
          fprintf(stberr,"#Cコンパイラ　ログ¥n");

          path1 = metafile.GetBuffer();//メタファイル
          path2 = geofile.GetBuffer();//データファイル

          TraverseDebug(NULL);
          TraverseDebug( &dumy ); //OK.VCにデバッグポートを登録
          SetIndicator(m_hWnd);//インジケートするウィンドウの指定
```

```
        if(b_IO){//ボタンが押されている：出力側
                //メモリ空間上にあるデータに、仮想コンバータがアクセスできるようにする。
                if(!access(path2,0)){//既にある（ないなら－１）
                        if(IDNO == MessageBox("上書き？",Kanji("OUT"),MB_YESNO)){
                                fprintf(stberr,"#既存データファイルへの上書き出力が非選択¥n");
                                fclose(stberr);
                                return;
                        }
                        if(access(path2,2)){//書き込み禁止
                                fprintf(stberr,"#出力ファイル(%s)が書き込み禁止¥n",path2);
                        }else{
                                fopen_s(&outfile,path2,"wt");
                        }
                }else{//まだない、新しいファイルを作成
                        fopen_s(&outfile,path2,"wt");
                }
                if(!outfile){
                        fprintf(stberr,"#出力ファイル(%s)が開かない¥n",path2);
                        MessageBox(path2,"#出力ファイルが開かないため、ログファイルに出力する");
                }else{
                        fprintf(stberr,"#出力ファイル(%s)を開いた¥n",path2);
                }
                //この時、メモリ空間に対して、LSSG系コマンドが実行されると厄介なことに
                //例えば、出力なのにメタファイルがLSSG準拠だとデータが追加されてしまう
                ReadOnly = 1; //cci_parsのグローバル変数
        }else{//入力
                if(access(path2,0)){
                        fprintf(stberr,"入力データファイル(%s)がない",path2);
                        MessageBox(path2,"入力データファイルがない");
                }
            ReadOnly = 0; //cci_parsのグローバル変数
        }
        fprintf(stberr,"# 仮想コンバータ・メタファイル：[%s]¥n", path1);
        fprintf(stberr,"# 変換対象ファイル：[%s]¥n", path2);
        if(path2[0]==0) dump_table(stberr); //140713 untableの中でテーブルをダンプ
        hasil = compile(path1);
        if(!hasil){
                fprintf(stberr,"#コンパイル失敗¥n");
                fclose(stberr);
                if(outfile) fclose(outfile);
                theApp.m_pMain->m_pView->m_drawFrm->RedrawWindow();
                PembantuX(s);
        }else if(path2[0]==0){
                code_dump();
//              dump_table(stberr);
                fclose(stberr);
                PembantuX(s);
        }else{//実行
                d3Group **root;
                int num,rc;
        //dcは、用が済んだらすぐ解放する必要があるため、メモリブロック型で取得
                CClientDC *dc = new CClientDC(theApp.m_pMain->m_pView->m_drawFrm);
                *theApp.m_pMain->m_pView->m_drawFrm->m_HDC = dc->m_hDC;
                wg3AssignDrawarea( (void*)&theApp.m_pMain->m_pView->m_drawFrm->m_HDC );
                rc = g3GetRootGroup(&num, &root);
                wg3AssignDrawarea(NULL);
                delete dc; //このDCを使用するスレッドを起動する前に解放しておく
                if (!rc){
          //            z3Message(5000,"ルートなし");
                        z3Message(5000,Kanji("KANJI_4"));
                        return;
                }
                rootgroup = root[0];
```

```
                    if(b_IO) CreateLGarray(rootgroup);
                    fprintf(stberr,"#コンパイル成功¥n");
                    SetRootGroup(rootgroup);

                    GetDlgItem(IDOK)->SetWindowTextA(Kanji("BREAK"));
                    if(b_IO) g_path2 = NULL;
                    else g_path2 = path2;
                    hwnd = m_hWnd;//マルチスレッド用
                    CreateAndStartThread();
          }
}
```

コンパイルと実行の開始を指示する［実行］ボタンが押された時点で、TraverseDebug 関数の引数に dumy 関数を指定して実行することにより、実行段階でのプログレス・インジケータ表示とブレーク処理を登録する。

リスト 3-4-4 プログレス・インジケータの表示関数

```
//2VWND.cppで登録するプログレス・インジケータ表示と強制ブレークのための関数

static int dumy(int pc, int pos, int len){
          char kata[2000];
          int rate,i,T,H;

          static int prate;//前回の進捗率
          g_PC = pc;//終了時デバッグ出力用
          g_POS = pos;//同

          if(len<1) return g_BREAK;//出力時
          T = len/10 + 1; //ファイルが小さい時、少なくとも１
          H = 1000;
          rate = 0;
          for(i=0;i<4;i++){
            if(!T) break;
            while(T < pos){//千の桁
                    rate += H;
                    pos -= T;
            }
            T = T/10;
            H = H/10;
          }
          if(rate != prate){
                    sprintf(kata,"%4d/10000",rate);
                    SetDlgItemText(laporHWND,IDC_EDIT_GEOFILE,kata);
                    UpdateWindow(laporHWND);
                    prate = rate;
          }
//        return 0; //ブレークしない
//        return 1; //ブレークする
          return g_BREAK;
}
```

最後に、ブレークポイントによる中断である場合には、メッセージを出力する。

リスト 3-4-5 ブレークによる中断の表示

```
//2VWND.cppで、コンパイル・実行を行っているスレッド

DWORD WINAPI MyThreadFunc(LPVOID lpParam){
          int hasil,rc;
```

```
            char s[200];
            fprintf(stberr,"#----------実行開始----------¥n");
            hasil = execute(g_path2); //出力の時はNULL
            fprintf(stberr,"#----------実行終了----------¥n");
            if(g_BREAK){
                        fprintf(stberr,"#ブレークによる強制終了¥n"); //●ブレークによる終了の表示
            }else{
                        fprintf(stberr,"#スレッドが終了¥n");
            }
            fprintf(stberr,"#終了コード：%d¥n", hasil);
            fprintf(stberr,"#終了時PC:%d, SIORI=%d¥n",g_PC, g_POS);
            if(!g_path2) DestroyLGarray();//出力時

    //スレッド起動側で、起動後にDCを解放するため、ここで新たに取得する
            CClientDC dc(theApp.m_pMain->m_pView->m_drawFrm);
            *theApp.m_pMain->m_pView->m_drawFrm->m_HDC = dc.m_hDC;
            rc = wg3AssignDrawarea( (void*)&theApp.m_pMain->m_pView->m_drawFrm->m_HDC );
            if(rc){
                        rc = g3LoadTextureAll();
                        rc = wg3AssignDrawarea(NULL);
            }else{
                        MessageBox(hwnd,"wa3SiignDrawarea失敗","thread",MB_OK);
            }
            SetRootGroup(NULL);//140605

            fclose(stberr);
            if(outfile) fclose(outfile);
            theApp.m_pMain->m_pView->m_drawFrm->RedrawWindow();
            //          PembantuX("c:¥¥@keikan¥¥ksim¥¥temp¥¥CClog.geo");
            sprintf_s(s,200,"%s¥¥CClog.geo", e3GetEnvParam(E3_TEMP_PATH)->path);
            PembantuX(s);

//          GetDlgItem(IDOK)->SetWindowTextA(Kanji("START"));
            if(!g_BREAK){//最後まで実行された場合
                        SetDlgItemText(hwnd,IDOK,Kanji("START"));
            }
//          g_BREAK = 0;
            g_bRunning = false;
            ExitThread(0);
            return 0;
}
```

３－５．ＶＣ－３Ｍの開発

（１）OS を Android とする場合の開発課題

ライブラリ関数は、処理系のメモリ空間上に、OpenGL で表示を行うためのオブジェクトを生成する点は、VC-2V と同様である。また、背面カメラから取得した画像の上に三次元モデルからレンダリングした CG を合成表示する処理は、景観シミュレーションにおける写真合成と同じ処理である。但し、背景画像と写真合成に必要なカメラ（視点座標やカメラアングル）に関する情報が時々刻々、リアルタイムで入力される点が異なる（いわば、「リアルタイム写真合成」とも呼ぶべき処理である）。

この処理系は、Android OS 上で使用する VC-3M.apk として実装し、サンプルデータを現場で、GPS センサで取得した視点位置から、磁気センサおよび加速度センサで取得したカメラアングルで表示するための、三次元データを用いた一種の AR アプリ(Augumented

Reality)である。具体的な配信に際しては、現場毎に、表示するメタファイル+データとアプリをセットとしたパッケージとして提供している。GPS 位置情報の精度を除き、何も手掛かりがないような茫漠とした場所であっても、記録された集落等を表示することができる。

モデルを記述するために用いた座標軸の原点を緯度・経度・標高で記述する。対象物が視界の中に無い場合には、左右あるいは後方のいずれにあるかを画面中央の文字表示でユーザーに指示する。対象物までの距離が大きい場合にも文字表示でユーザーに知らせる。

Windows 版の景観シミュレータからは独立して利用できる処理系であるが、OpenGL 表示、およびメモリ上のデータ構築に関して、景観シミュレータのライブラリ（ソースコード）を多く活用した。

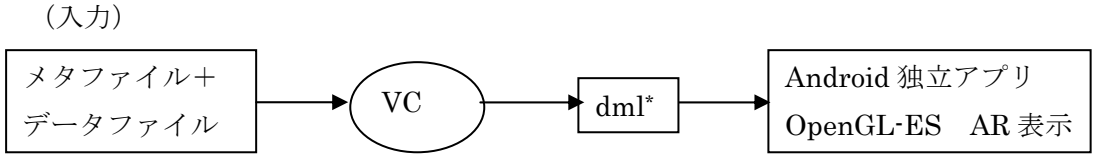


図 3-5-1 VC-3M による処理(*dml は、cci_dml.c によるライブラリ関数の実装)

小さいながら仮想コンバータのコンパイラ機能を搭載しているため、携帯端末だけを用いてメタファイルをプログラミングし、携帯端末上で利用できるテキストエディタを用いてデバッグすることもできる。

①開発環境

開発に際しては、Windows 上で動作する開発環境 Eclipse を使用した。更に、仮想コンバータのソースコード群（C 言語）をコンパイルするために、携帯端末で提供されている DALVIC 仮想マシンのための機械語を生成する ndk を使用し、これを用いて仮想コンバータの機能を実装した DLL（VirtualConverter.so）を作成した。表示速度を高めるために、OpenGL による描画部分も C 言語で記述し、この DLL の中に含めた。

②OpenGL

Android 上で利用した OpenGL ライブラリは OpenGL ES1.0 であり、景観シミュレータ等で使用している Windows 版と比較すると、利用できるコマンドが少なく、制約されている。具体的には、4 以上の頂点を有するポリゴンが出力できず、三角形のみの出力となる。このため、景観シミュレータで描画処理を行っている g3drl ライブラリを活用しつつ、不足する OpenGL 関数に関しては、関数プロトタイプを定義しているヘッダファイルの中に直接処理を書き込む方法で実装した。例えば、多角形の出力が要求された場合には、3 頂点からなる配列を用意しておきバッファリングを行い、多角形の全ての頂点が出力された段階で、分解後の三角形の描画を OpenGL ES ライブラリに対して要求している。

③センサ計測値による視点座標、カメラアングルの取得

一方、ユーザーインターフェース、各種センサの計測値の取得は、マーカーを用いたARアプリの開発実績があるエム・ソフト社に外注し、全体をVC-3M.apkというセットアップファイルに集約した。2012年2月に、東京都内、つくば市内および釜石市内でテストを行った。東京都内では、磁気センサの値が直流電車の影響などを受けることが判明したが、つくば市内のTX沿線や釜石ではこれは認められなかった。

2013年11月には「土木の日体験教室」で国総研旭本館裏の位置に配置した住宅をサンプルとして公開を行った。その後、テクスチャ表示機能、機種の違い（画面サイズ、縦横比など）による互換性への対応を行った上、2014年3月に、岬地区の復原データを用いて、奥尻島でのテストを行った（写真3-5-1）。

この再現表示のためのデータは、図2-3-13のデータを構成する地面、道路、複数の住宅を一つのデータファイルに集約すると共に、メタファイルを付したものである。

このデータファイルとメタファイルに、携帯端末を用いて表示するために必要なアプリケーションの組を合わせたセットアップ一式を、４種類の携帯端末（タブレット３種、スマートフォン１種）で確認した上で、上記サイトからダウンロードできるように試験的に公開開始した(2014.3)。セットアップ方法、操作方法等に関する解説資料を添えた。

但しアプリケーションはまだ、Android携帯端末の多様な機種、OSバージョンにおける互換性の改善の余地があるため、デバッグ認証の段階である。2014年8~9月に、３機種15台のタブレットを用いて性能テストを行い、精度の高かった５台を使用して10月８日に、奥尻島で小学生を対象とした体験教室を開催した(写真2-1,2-2)。

写真3-5-1 奥尻島岬地区での表示テスト（2014.3）

図 3-5-2 FSMFC 起動画面(左：現地調査、右：記録再生)

④Windows 上のモックアップ FSMFC の作成

この処理系の開発に先だって、デバッグ環境の良い PC 上の Windows－VS2005 で、プログラムの全体構成を検討するために、FSMFC.exe というモックアップを作成した。GPS センサ、磁気センサ、加速度センサが利用できないため、これ自体、実用性は無いが、VC-3M の状態遷移、画面構成等を検討するために有用であった(図 3-5-2)。

また、「見たい場所」として一覧を表示する名称、メタファイル、メタファイル等のリストである ModelIndex.txt の準備段階でのテスト等にも有用である。

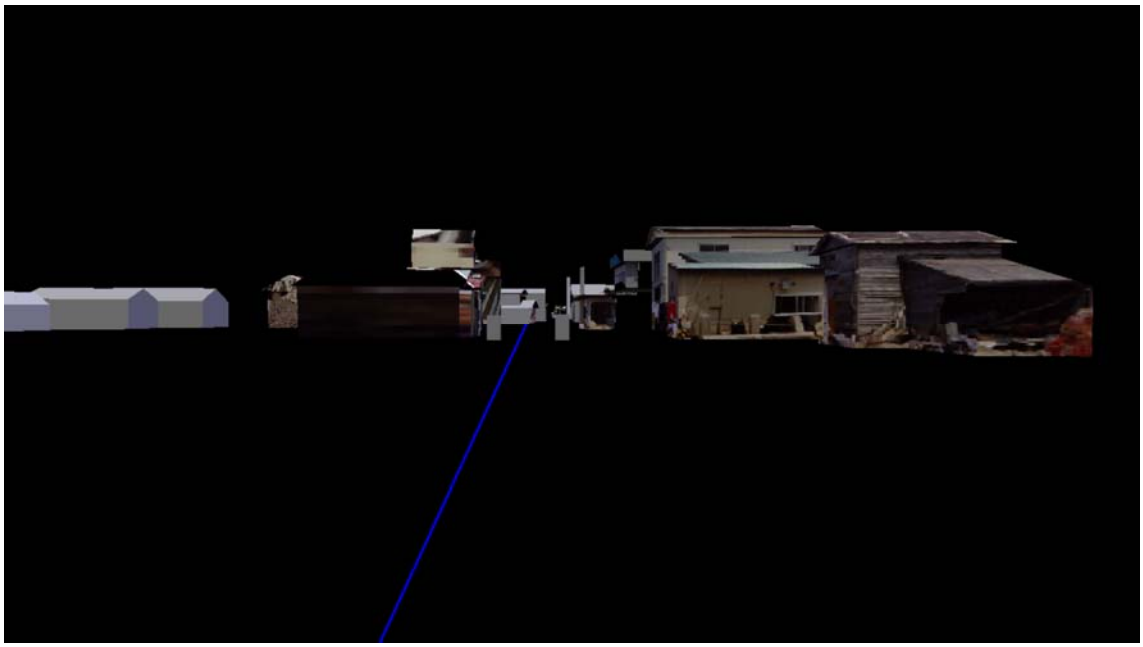


図 3-5-3 FSMFC による表示状態（背景画像はない）

仮想コンバータの処理結果を、景観シミュレータの dml ライブラリを用いてメモリ上に構築し、g3drl ライブラリを用いて描画を行う構成は、VC-2V とほぼ同様である。しかしながら、Android 上で三次元表示に使用できる OpenGL-ES1.0 は、PC 用の OpenGL と比較して、利用できる命令の種類やデータの精度が限られている。利用できない命令に関して

は、利用できる命令を組み合わせて疑似的に構築する必要があった。この疑似的に増補する関数は、OpenGL 関数のプロトタイプを定義しているヘッダファイルの中で直接記述してリンクエラーを回避する方法を採った。

（２）アプリケーションと基幹部分の開発の役割分担とインターフェース関数

Android 系のユーザーインターフェースに関して国総研に開発経験が無かったため、2012 年度の研究予算において、「仮想コンバータの可搬性検証のためのプログラム業務」を一般競争入札によりエム・ソフト社に外注した。同社が作成した部分は、Android SDK の開発環境上で Java 言語を用いて、FSMFC で検討した画面遷移を実現すること、およびユーザーの操作、各種センサの計測値、背面カメラで取得した画像を表示画面の背景として表示することである。プログラムの役割分担を明確化するために、以下のようなインターフェース関数を定義し、エム・ソフト社が作成するのは、インターフェース関数を呼び出すフレームワークとし、インターフェース関数の実行内容は国総研（小林）がプログラムした。このインターフェース関数は全て C 言語で記述し、ndk でビルドし、.so の拡張子を持つ Java アプリのためのダイナミックリンクライブラリとして、フレームワークから起動する方法とした。そのためのインターフェース関数は以下の通りである。

① MOpen(char *logfilename, char *scnfilename)

コンパイラと実行形式の処理結果を記録するログファイルと、ユーザーのシャッター操作を記録するシーンファイルの名称を引数として、システムを初期化する。シーンファイルが存在すれば、それをロードしてセッションの初期条件とする。「歩いた場所」の一覧から、シャッターを押した場所を選択できるようにする。シーンファイルを LSS-S 形式とし、シーンファイルの読み込みにも仮想コンバータを使用することとし、このために、以下のコマンドをライブラリ関数として追加した。

SCN, MODEL, LIGHT, LIGHTGROUP, CAMERA, TIME, EFFECT, EFFECTGROUP

② MInit(int W, int H, int w, int h, double fovy)

表示ディスプレイの横 W 縦 H 寸法、背面カメラが取得する画像の横ｗ縦ｈ寸法、カメラの画角 fovy を引数として、表示エリアを初期化する

③ MLoad(char* metafile, char*datafile, double Ido, double Keido, double Takasa)

メタファイル名、データファイル名、座標原点の緯度経度標高を引数として、データをロードし表示できるようにする。

④ MMove(int Hold, double Ido, double Keido, double Takasa, double R, double P, double Y, double Zoom)

ホールドボタンの状態（キャリブレーションに使用）、GPS で計測された現在の緯度、経度、標高、端末の姿勢を示す３値、ズーム

なお、当初、携帯端末の姿勢を示す R,P,Y にロール（回転）、ピッチ（仰角）、ヨー（方

位角）を用いて、API 関数から得られる値を渡していたが、最新版では、機種による互換性を高めるために、磁気センサの計測値と加速度センサの計測値から直接携帯端末の姿勢を示す四元数を直接計算し、その x,y,z 各成分をこの関数に渡している。

Zoom は背面カメラズームが設定された場合の値を渡すが、本稿時点では簡単のためズームは固定としている(ModelRenderer.java の中で 1.0 に固定)。

2014 年 10 月の「むかしめがね」バージョンでは、GPS がデータを取得するまでの間（Ido, Keido, Takasa が全て 0.0）は、携帯端末の姿勢データだけを用いて、町並全体を対応する向きから画面に対して適切な寸法でプレビューする機能を加えた。

⑤ MShutter(const char *image)

シャッターが押された段階で起動する。その時点の背景画像を API の側で JPEG 形式のファイルに保存し、引数として渡す。関数の中で、現在の視点、表示中のメタファイルとデータファイルの名称などを記録する。この記録は終了時点で、Mobile.ja.scn ファイルに保存される。

⑥ MSelect(const char *imagefilename)

「歩いた場所」を選択する画面において、ユーザーがサムネイル画像の一つを選択して表示を指示した段階で呼び出される。引数で渡された画像ファイルを背景画像として表示し、その上にシャッター時点で記録されたデータファイル、メタファイル、カメラ位置等をもとに、現場で記録した写真合成画面を再生表示する。なおこの記録再生画面は固定であり携帯端末を動かしても変化しない。

⑦ MDelete(const char *imagefilename)

「歩いた場所」を選択する画面で、ユーザーが選択した画像の削除を要求した場合に、その記録を削除する。

⑧ MTerm()

OpenGL の表示画面を終了し、デバイスコンテキストを解放する。

⑨ MClose()

シーンファイル（mobile.ja.scn）を保存し、メモリを解放する。

以上①～⑨の関数をインターフェースとして、役割分担してプログラム作成を進めた。

国総研では、①～⑨の機能を有するトップレベルのソースコード mobile.c を先頭に、これから以下のソースコードにより compile 及び execute 処理を実行すること、及び VC-2V と同様に、景観シミュレータのソースコードから DML ライブラリ及び G3DRL に関するものをビルドに含め、これらを ndk コンパイラを用いて、Android が実行されるマシンの機械語の DLL である、VirtualConverter.so を作成した。

ソースコードの一覧は、メイクファイルである、Android.mk ファイルに列挙されている。

リスト 3-5-1 Android.mk メイクファイル

```
LOCAL_MODULE    := VirtualConverter
LOCAL_SRC_FILES := ¥
```

```
mobile/cci.c¥
mobile/cci_code.c¥
mobile/cci_dml.c¥
mobile/cci_file.c¥
mobile/cci_pars.c¥
mobile/cci_misc.c¥
mobile/cci_tbl.c¥
mobile/cci_tkn.c¥
mobile/cci_face.c¥
mobile/D3dml.c¥
mobile/D3malloc.c¥
mobile/D3mat.c¥
mobile/S3sml.c¥
mobile/S3set.c¥
mobile/S3get.c¥
mobile/S3delete.c¥
mobile/G3drl.c¥
mobile/G3load.c¥
mobile/mobile.c¥
VirtualConverter.c¥
LOCAL_MODULE_FILENAME := libVirtualConverter
LOCAL_LDLIBS := -lGLESv1_CM -ldl -llog
include $(BUILD_SHARED_LIBRARY)
```

これらのビルドは、コンソール(cmd.exe)において、Android.mk ファイルが存在するディレクトリに移動し、

```
ndk-build
```

とタイプすることにより実行される。

なお、この共同作業により一通りの動作を行う処理系を完成した後に、地面の高さの取得、四元数を用いた姿勢制御、テクスチャ付きの画像の表示等を行った。本稿執筆時点のNDK ビルドの構成については、４－５に掲載する。

一方､Java 言語による開発環境 Eclipse 上の AndroidSDK を用いて､ユーザーの操作と、各種センサの値の変化に対応する処理を開発した。

起動から終了に至る遷移に基本的な違いはないが、画面デザイン上の制約から、トップに機能選択メニューを追加し、そこから「現地調査」（アプリ上では、「見たい場所」の選択画面）と、「記録再生」（アプリ上では、「歩いた場所」の選択画面）のどちらを開くかを選択するようにした。更に、歩いた場所については、機械的な文字列（日付時刻等を表す）から選択するのではなく、サムネイル画像を表示してそこから選択できるようにした。

SDK のためのビルドを構成するディレクトリを、リスト 3-5-2 に示す。

リスト 3-5-2 Android SDK のビルドを構成するディレクトリ

```
VC-3M
/.settings ビルドの環境設定等
/assets 空
/bin　VC-3M.apk (ビルドの結果生成するセットアップ)
/gen　BuildConfig.java, R.java
/jni C 言語のソースコード群(.c)
/libs DLL である libVirtualConverter.so
/obj C ソースをコンパイルした結果のオブジェクトファイル群(.o)
/res アイコン画像、操作画面のレイアウト.xml、文字列リソース
/srs この中に、java ソースコードが格納されている
```

java で記述されたソースコードをリスト 3-5-3 に示す。

リスト 3-5-3 java ソースコード一覧

Nativelib.java
VirtualConverterActivity.java
/adapter/GpsValue.java
/adapter/ImageItem.java
/adapter/InfoFileAccessor.java
/adapter/Listitem.java
/adapter/OrientationValue.java
/adapter/SelectImageAdapter/java
/adapter/SelectModelAdapter.java
/base/BaseRenderer.java
/realtime/ModelRenderer.java
/realtime/RealtimeActivity.java
/replay/ReplayActivity.java
/replay/SceneRenderer.java
/selectimage/SelectImageActivity.java
/selectmodel/SelectModelActivity.java
/util/DateUtility.java
/util/DialogUtility.java
/util/FileUtility.java
/util/VirtualConverterErrorDialog.java

前述の ndk でビルドを行い作成した Android のための dll である libVirtualConverter.so を含めたビルドを行い、実機の上にセットアップしてテストする作業を繰り返して完成した。

なお、この java 言語のソースコードに関しても、現在までに姿勢センサで取得した情報を表示系に四元数で渡すための/realtime/quat.java を追加した。

2013 年 1 月に、基本的な動作を確認する所まで到達し、東京都内、およびつくば市内で表示の位置ずれ等の計測を行った。更に、２月には、釜石市内で、表示確認を行った（写真 3-5-2)。

写真 3-5-2：釜石におけるテスト風景(2012.2.17)

表示精度に影響するのは、GPS センサが計測する位置精度、磁気センサが計測する地磁気センサへの局所的な環境条件や携帯端末の帯磁の影響等である。センサの計測値を確認するために、GPS Status という無料アプリ、および受注社が補助的に Android SDK だけで作成し、数値でセンサ計測値を画面に表示する SensorEx.apk という、計測値を直接画面に表示する簡単なアプリを用いて、環境条件の計測を行った。例えば、上野公園内で定点計測すると、直流で送電している地下鉄が通過するだけで、地磁気が大きく変動すること

が判明した。この補助的なアプリは、機能の追加が容易であり、画面の縦横切り替えなどの機種による互換性の向上のためのテストにおいても有効であった。

この改良に際して、携帯端末の姿勢を、SDK の関数が提供するロール・ピッチ・ヨーに加えて、磁気センサと重力センサの生の計測値から VC-3M アプリ側で計算するように修正したが、計測値の履歴をファイルに記録する方法が特に有効であった。

Android 開発環境においても、PC 上の携帯端末エミュレータが利用可能であった。しかし、FSMFC と同様に、屋外で使用する 3 種類のセンサは利用できないため、上記のモックアップとは大差なく、データ転送と起動にも時間がかかるため、本処理系の開発に際しては、あまり有用性は感じられなかった。

機種の互換性の上で問題となったのは以下の点である。

・画面の縦横寸法の取得、現在の縦横の判定、センサの座標軸の判定

・背面カメラから取得した画像と、グラフィックス画面の上下関係

また、センサの物理的な性能を補う上で効果があったのは以下のような点である。

・手振れは、加速度センサの計測値の揺らぎを生じるため、過去の計測値の履歴を反映させたデータを表示用に用いるようにした。具体的には、3 軸の計測値それぞれの値を記憶するスタティック変数を用意しておき、記憶＊0.9＋新値＊0.1 を新たな記憶とし、これを表示に反映させている。このパラメータ 0.1 は、携帯端末の加速度センサの感度や変化検出頻度に依存し、小さすぎると姿勢変化への反応が緩慢となり、大きすぎると手振れに対して過敏となるため、動作テストの結果得られた経験値である。

Eclipse 開発環境においては、ソースコード(java)の修正⇒保管を行うことにより、自動的にリビルドが行われる、とされている。しかしながら、C 言語部分（コンパイラ・インタプリタ処理系、あるいは OpenGL 描画系）の修正を行い、ndk コンパイラによるリビルドを実行した場合には、更新されるのは、DLL である libVirtualConverter.so のみであり、その場合にはアプリケーション全体(.apk ファイル)のリビルドは行われない。そこで、リソースのストリングスにヴァージョンを記入することとし、DLL を更新した場合にはこの文字列を修正することにより、リビルドをトリガーした。

コンパイラ・インタプリタ系の C 言語ソースコード（3－2 で解説した基幹部分）には、コンパイルエラーメッセージなどが直接日本語の文字列で埋め込まれている。ビルドに際して、ソースコードを Shift-JIS 形式で保存すると、Android 系の出力ではエラーメッセージやログファイルが正しく表示されない場合がある。コード変換処理を組み込むことも検討したが、現段階ではコンパイラ・インタプリタ系のソースコードを utf-8 形式で保存すると共に、他の Windows 系利活用処理系においてもこのソースコードを共用することとした。

VS2005 開発環境にあっては、ビルドの設定により、たとえソースコードが utf-8 形式であっても、プログラム中に埋め込まれた UNICODE 形式の日本語のリテラル文字列を、fprintf 等により Shift-JIS 形式で出力する。

一方、メタファイルの中にリテラル文字列として埋め込まれた日本語の文字列の処理は、

メタファイルを保存する形式(Shift-JIS / utf-8)に依存する。PC、タブレット、サーバの間で互換性を維持するために当面、メタファイルは utf-8 形式で保存すると共に、Windows 系 PC の処理系において、Shift-JIS 形式に変換してログファイルに出力するような処理を行っている。

国総研における仮想コンバータのプログラミングと、外注先の業務の役割分担リスト 3-5-4 のように定めた。

リスト 3-5-4 VC-3M プログラム作成における役割分担を定めた開発手順

1　　業務目的

本業務は、国土技術政策総合研究所が開発した、三次元住宅情報の永久保存のために使用する仮想コンバータに関して、その可搬性を検証するために、異なる実行環境に移植するものである。

2　　業務構成

(1)　プログラム開発環境及び実行環境の構築
(2)　表示機能、操作ダイアログの開発
(3)　仮想コンバータの移植
(4)　サンプルデータを用いた動作テスト
(5)　セットアップ一式の作成
(6)　報告書の作成

3　　業務内容

(1)　プログラム開発環境及び実行環境の構築

実行環境は携帯端末であって、以下の性能を有するものとする。

ａ．三次元表示を行うに十分な解像度とカラー数のあるディスプレイを有する
ｂ．三次元モデルを表示するために必要な視点、注視点等に関するデータを、ＧＰＳ装置、地磁気センサー、加速度センサー等により自ら取得できる
ｃ．三次元モデルと合成表示する背景画像を、現場においてリアルタイムで取得するＣＣＤカメラを内蔵している
ｄ．三次元モデルを適切な視点・注視点等の条件設定に従って、立体的に表示するために OpenGL ES ライブラリが利用できる
ｅ．WEB 経由でサーバからファイルをダウンロードできる

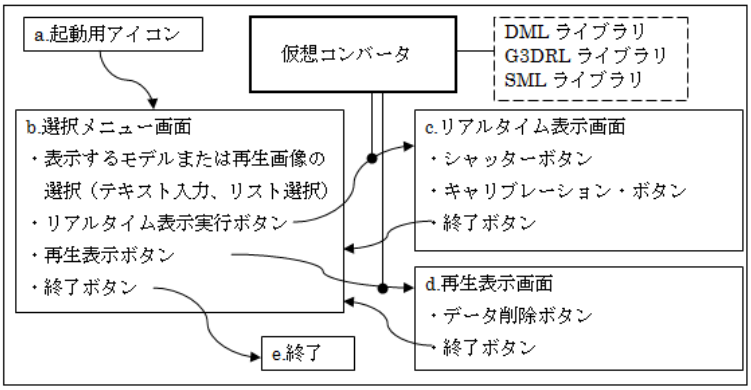
プログラムの作成に使用する開発環境は、無償で公開されているコンパイラ、リンカ、携帯端末エミュレータ等を用いて構成する。この開発環境は、甲においても容易に取得し利用できるものでなければならない。この開発環境は、ユーザーが動作を指示するためのインターフェースおよび端末が有する各種センサーのデータを取得するためのインターフェースにおいては JAVA 言語を、また仮想コンバータの移植のためには、C 言語を使用し、これらを合わせて一体として動作する実行形式を生成する機能を必要とする。

開発環境には、甲が実行形式を無償で自由に配布するライセンスを制限するような有償ライブラリ等を含めてはならない。

なお、動作テストに用いる携帯端末機器は、乙において用意することとし、(4)の動作テストが終了した後、納品することを要しない。

(2)　表示機能、操作ダイアログの開発

システムの全体は、ダイアグラム１に示すような状態遷移を有する。



a. 起動手段

まず、起動用アイコン(a)のタップ操作により選択メニュー画面(b)を表示する。

b. 選択メニュー画面

選択メニュー画面においては、ユーザーが｛①表示すべき住宅等の指定を行う　②リアルタイム表示(c)を指示する　③再生表示(d)を指示する　④終了する(e)｝のいずれかの機能選択を行う。

リアルタイム表示(c)、再生表示(d)が指示された場合、仮想コンバータ(3)に表示すべき三次元データ（ストリーム）と、そのデータ形式を定義するメタファイルを渡し変換を行う。WEB 上のデータとメタファイルが URL で指定された場合には、ダウンロード処理を行った上で、取得したローカル・ファイルを同上の処理対象とする。変換したデータを表示部(c.または d.)に渡し、制御を移す。

c. リアルタイム表示画面

リアルタイム表示画面においては、仮想コンバータの処理結果であるメモリ上の三次元モデルに関して、端末が取得する位置情報を用いて視点・注視点からの透視図を作成し、端末の CCD カメラが取得する背景画像の上に合成表示する。携帯端末が移動し、あるいは回転することにより視点・注視点の変化が検出された場合には、新たな背景画像及び視点・注視点を反映させた画像に更新表示を行う。視点の変更などは、後述の G3DRL ライブラリ関数が機能を提供している。

モデルが、携帯端末の表示画面の視野範囲の外となって表示されない場合には、モデルの中心位置が表示画面のどちら側にあるかを示す矢印等のアイコンを画面に表示し、ユーザーがモデルの位置を確認する作業を支援する。

シャッターボタンが操作された場合には、背景画像を携帯内部のファイルとして保存すると共に、その時点における視点・注視点、時刻情報、表示していたモデル、保存した背景画像のファイル名等を、ＳＭＬライブラリのシーン情報として保存する。この保存は、電源 OFF や OS の再起動により消失しない方法で行う。

キャリブレーション･ボタンは､背景画像の中の目印となる物件との比較対照においてモデルの表示位置が、GPS 位置情報の誤差等の理由により、正しい位置から継続的に一定値だけずれているような場合の補正のために使用する。このような場合に、キャリブレーション・ボタンが押されている間は、GPS 等のセンサーが検出した新たな位置情報をモデルの表示に適用することを停止し、モデルを画面に対して固定的に表示する。ユーザーはこの間に携帯端末の角度を変化させ、あるいは立ち位置を移動することにより、背景画像上でモデルが正しい位置に表示される状態を見出す。正しく表示された状態でキャリブレーション・ボタンを離すと、その時点での視点・注視点と、ホールドされていた視点・注視点との差分を補正データとして取得し、これを以後の視点移動に伴う表示更新に適用する。この差分から補正データを作成し、以後の表示に反映させるロジックは、建築技術的要素を含むため、甲が作成する。乙は、処理に必要な、空の関数、もしくは簡単な処理を行う関数と、必要なスタティック変数等を定義するだけで良い。

終了ボタンが押された場合には、表示を終了し、選択メニュー画面に戻る。

d. 再生表示画面

再生表示画面においては、リアルタイム表示画面(c)におけるシャッター操作により保存されているデータの内、選択画面(a)で指定されたものを、背景とモデルの合成表示として、シャッター操作時の表示画面と同一の静止画像として表示する。

データ削除ボタンが押された場合には、この背景画像ファイルと対応するシーンデータを削除する。

終了ボタンが押された場合には、表示を終了し、選択メニュー画面に戻る。

(3)　仮想コンバータの移植

移植する仮想コンバータのソースコードは C 言語で記述されており、その全体は別添１の通りである。この内、cci_dml.c で定義された組み込み関数群を変換結果出力手段として使用する。更に、別添２に示す DML ライブラリを構成するソースコード（C 言語）をコンパイル・リンクすることにより、変換結果をメモリ上のデータ構造として構築する。次に、G3DRL ライブラリを構成するソースコード（C 言語）をコンパイル・リンクすることにより、OpenGL でモデルを画面に出力することができる。

但し、携帯端末においては、OpenGL ライブラリの機能が一部制約された OpenGL ES を使用することが一般的である。このために必要な対応は甲が行うこととし、乙が G3DRL ライブラリを修正する必要はない。G3DRL ライブラリの多くの機能は、本件の移植には必要でないため、ES にない OpenGL 関数の参照に関してはダミー関数を作成しリンクすることで対応し、処理が必要な場合には本件専用の OpenGL 関数を作成する。可能な限り

G3DRL の変更は行わない。乙が作成する範囲は、OpenGL ES が描画するモデルのパースを、背景画像と合成表示（オーバーレイ）する機能である。

次に、表示画面においてシャッターボタンの操作が行われた場合の処理を行うために、SML ライブラリを構成するソースコード（C 言語）をコンパイル・リンクする。シャッターボタンのコールバックの中で、必要な関数を使用する。

移植に使用する仮想コンバータ、DML ライブラリ、G3DRL ライブラリ、および SML ライブラリのソースコード、および OpenGL ES に関連して追加で必要となる関数等のソースコードは、CD-R の形で契約の後、作業に先立って甲が乙に貸与する。

(4) サンプルデータを用いた動作テスト

動作テストに使用する建物等のサンプルデータは、甲が２種類提供する。それぞれのデータが対象とする建物の所在地は、茨城県つくば市内、及び乙の所在地に近い地区であって、GPS 電波等が良好に受信できるような場所に設定する。サンプルデータは、仮想コンバータが処理することのできる形で、地球座標（ＧＰＳ座標）に関連づけるために十分な情報を含むものとする。

動作確認においては、このサンプルデータの所在地の近傍において、携帯端末によるモデルの表示を行い、正しい位置からの表示のずれ等を計測する。

この動作確認は、GPS の誤差に影響する大気の状態が異なる環境下で少なくとも３回実行する。

動作テストに使用するサンプルデータは、データ形式を記述したメタファイルと共に、CD-R 等の形で契約の後に貸与する。また、動作テストのために、同じファイルを甲が管理するサーバから WEB 経由でダウンロードできるように甲が準備する。

(5) セットアップ一式の作成

作成したプログラムの実行形式を、携帯端末に新たに導入するためのセットアップ手段を作成する。このセットアップ手段の操作方法は必ずしも簡潔である必要はない。むしろ、実行形式がデバッグされ、あるいは改良された場合に、セットアップの再作成が容易であることが望ましい。その手段の実現方法・形態は随意とする。

以上

別添１：仮想コンバータのソースコードのファイル構成（Ver.1.0）

cci.c (43)	コンバータの動作制御（組み込み用には不要）
cci_tkn.c (341)	メタファイルのトークン解析
cci_pars.c (1652)	メタファイルのパーサ
cci_tbl.c (121)	変数テーブル管理
cci_file.c (90)	データファイル入力関数
cci_ip.c (545)	ファイル出力関数（組み込み用には不要）
cci_misc.c (119)	エラー処理等
cci_dml.c (840)	変換結果を DML ライブラリに出力する（本業務で使用）
cci_code.c (810)	仮想マシン上でのコンバータ実行形式の生成と実行
cci_quat.c(391)	クォータニオン演算ライブラリ、地理空間座標変換関数

(　)内はステップ数

別添１は新規に開発した成果である。本業務の成果などを反映して、一般性・移植性を高めるために更に改良する予定である。

別添２：リンクすべきライブラリとソースコード

①DML ライブラリ　メモリ上の三次元データの管理
　d3dml.c(1442)　　各種表示オブジェクトの生成・削除
　d3malloc.c(983)　メモリ管理
　d3mat.c(385)　　座標に関する同次行列演算
②G3DRL ライブラリ　三次元データの表示出力

g3drl.c(5866)	各種オブジェクトの OpenGL ライブラリへの送出・描画
③SML ライブラリ	シーンに関するデータ管理
s3sml.c(681)	初期化・終了
s3set.c(185)	データの作成・設定
s3get.c(140)	データの取得
s3delete.c(259)	データの削除

( )内はステップ数

別添２については、国土技術政策総合研究所　研究報告第 42 号に解説している。同報告付録 CD-ROM に採録している景観シミュレーション・システムのソースコードの一部に含まれるものである。なお、同研究報告及び付録 CD-ROM の内容は、下記の WEB サイトからダウンロードすることができる。

http://www.nilim.go.jp/lab/bcg/siryou/rpn/rpn0042.htm　（本文のみ）

http://sim.nilim.go.jp/MCS/REPORT42/REPORT42.asp　（本文＋付録 CD-ROM）

http://sim.nilim.go.jp/ksim/program/SIM209/SRCNT/LIBRARY/

（３）結果の現場検証

動作を確認するために、東京、筑波および釜石市で動作テストを行った。直流電車（地下鉄含む）の近傍で、磁気が安定しないことがわかっていたため、東京（上野付近）でのテストの結果は芳しくなかった。これに対して、つくば市は、付近を通過する常磐線やつくばエキスプレスが、八郷の地磁気観測所への影響を避けるために交流給電を行っているため、TX の直下でも磁場は安定していた。更に、釜石市においては、ローカル鉄道は電化されておらずかつ運休状態にあり、磁場を発生させるようなアクティビティも低下していたため、姿勢センサの計測値は最も安定していた。

センサ計測値は、シャッターボタンを操作することにより記録されており、報告書にも掲載しているが、この時点では携帯端末の精度が低く、本稿では割愛する。

３－６．ＶＣ－４Ｄの開発

VC-2V および VC-3M においては、仮想コンバータがメタファイルとデータファイルから取得するデータは全てメモリ上に蓄積される。従って、メモリ容量を超えた大きなデータを扱うことはできない。また、電源を喪失した場合には、メモリ上のデータは失われる。

一方、MSSQL, Oracle などのデータベースでは、テーブルの形で登録されたデータは、ハードディスク上のファイルとして記録されており、処理途中での電源障害等に対して強靭な仕組みが用意されている。

メモリ上の三次元データは、基本的には、複数の数値を一定の規則で束ねた構造体の、可変長の配列として組み立てられているため、データベース上のテーブルに変換することが可能である。これにより、電源の問題と、データ規模の問題を解決することができる。個別のデータベースが実体化されたファイル（MSSQL の場合には、mdf という拡張子を有する）のサイズを計測してみると、圧縮アルゴリズムが用いられている。

市販の CAD ソフトや、GIS ソフトの多くは、操作しているユーザーへの応答性の観点から、基本的にはメモリ上に編集中のデータを展開して表示している。しかし、処理速度が求められない長期保存やデータ形式の変換の用途のためには、リクエストを受けてからバッチ処理でデータの分解と、別形式への再構築をバックグランドで実行し、処理が終了してからクライアントに結果を返すような堅牢で確実な処理系ないしサービスが存在すれば有用であると着想し、試作を行った。

WEB ベースで受付と処理結果の納品を行うようなサーバ上のアプリケーションで、以下のような機能を有するものとする。

・アップロード受付画面で、任意形式のメタファイルとデータファイルを受け付ける（アップロード)。
⇒サーバ側では、このペアを仮想コンバータで処理し、データファイルを完全に要素に分解した上で、データベース上のテーブルを構築する。

・ダウンロード受付画面で、データベースの名称と、必要とする（別の）形式のメタファイルを登録する。
⇒サーバ側では、データベース上のテーブルから、メタファイルの指示に基づいて、新たなデータファイルを構築して、クライアントに送付する。

このような処理を実現するためには、メタファイルをコンパイルした実行形式が起動するライブラリ関数が SQL コマンドを生成し、データベース上に頂点座標、面、立体、法線、テクスチャ、属性等のテーブルを構築する。また、構築されたテーブルを参照し、異なるデータ形式を定義したメタファイルにより、異なるデータを出力することができる。

Apache を介して、WEB サーバにおいて動作するアプリから起動し、WEB を介してアクセスしたユーザーに対して、サービスを提供する。

VC-2V、VC-3M と比較すると、読み込みの速度に時間がかかることから、ユーザーからのアップロード要求、ダウンロード要求を受け付けた時点で進行状況を表示するページを立て、ユーザーの進捗状況を伝えるような処理とし 2013 年 3 月に初版が稼働した。

この SQL データベースは、建物や町並などの物件毎のファイルを作成するシステムであるため、メモリ容量による制約を受けることはなく、また処理中の電源障害等に対して強靭である。この処理系は dml ライブラリや g3drl 等、景観シミュレーション関連ソフトからは完全に独立して動作する。このことにより、仮想コンバータの処理系の可搬性を、より高いレベルで検証したことになるであろう。

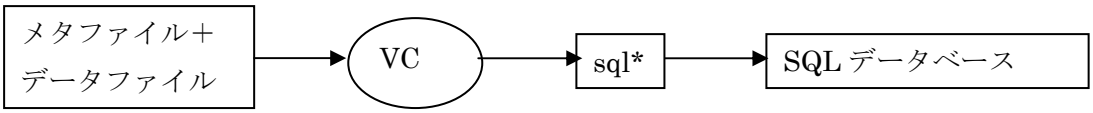
現在は、政府統合サーバへの移行（仮想化）に伴い、OS(Windows2003→2012Server)と SQL データベースの変更（MSSQL→PostgreSQL）の要求に対応して、更に研究開発を続けている。

経時的な使用方法は社会的ニーズに依存するが、基本形は、過去に保存されたデータファイルとメタファイルのペアを、利活用時点で本処理系にアップロードし、直ちに別形式のデータファイルでダウンロードする、データ形式変換サービスである。このようなサービスにより、例えば過去に記録された町並のデータを用いて、避難シミュレーションや延焼シミュレーションを実行可能なデータ形式への変換を行うようなニーズが存在するであろう。

更に、データベース（その実体はサーバー上のファイル）の継続性が制度的に保証されるならば、データの保存方法の一形態として、アップロードからダウンロードまでの一定

期間の保管を行うサービスとしてのニーズも存在する可能性がある。

（入力）



（出力）

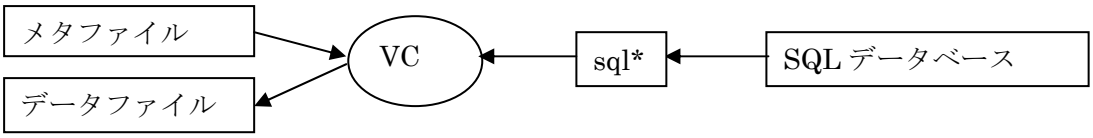


図 3-6-1 VC-4D による処理(*sql は、cci_sql.c によるライブラリ関数の実装)

最終的に作成した WEB アプリは、二つの操作画面を持つ。一つは、データファイルとメタファイルを選択してアップロードし、SQL サーバ上にデータを展開する操作画面である。今一つは、SQL サーバから物件と任意形式のデータ形式のためのメタファイルを指定して、要求した形式のデータのダウンロードを行うための操作画面である。

いずれも、ローカルな端末において選択・指定したファイルがサーバに届けられた段階で、制御が戻る。以後、サーバ側ではバッチ処理としてデータの展開や再構成を行い、終了した段階でユーザーによる結果のダウンロードが可能になる機構となっている。

この処理系を実現するための通過点として、以下のような試作プログラムを段階的に作成した。

（１）VC-4D(dll)（2012.10.11）

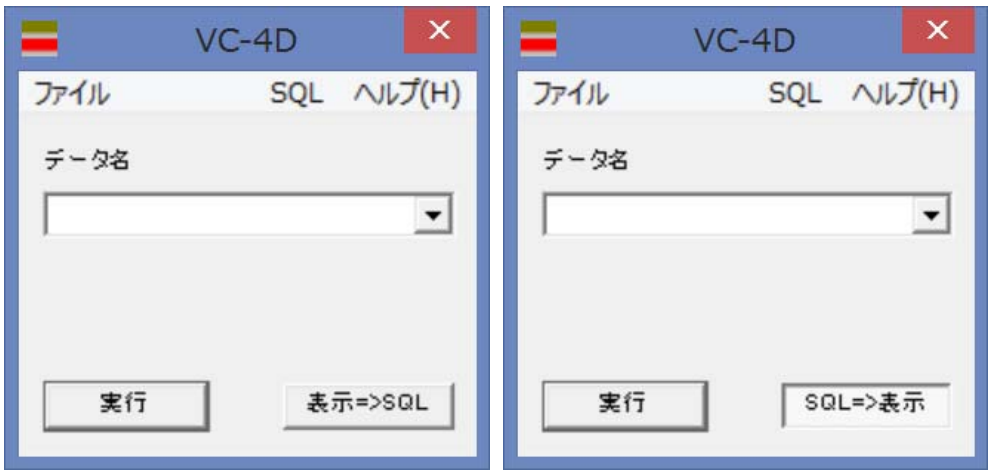


図 3-6-2 VC-4D の画面（SQL データベース入出力のプログラムに使用

表示→SQL では、主画面に現在表示されている建物・町並を、任意名称の一つのデータベースとして記録するとともに、名称を「VC4D」というデータベースの master テーブルに追加登録する（表示→SQL）。

SQL→表示では、登録された名称一覧から選択したデータベースに記録されている建物・町並を取り出して、現在表示されている建物・町並に追加し、画面表示する。

メニューの SQL の下に、以下のメニューを用意した

－ログ表示・・・データベースへの接続、コマンド発行、結果などの履歴を記録
－ログ削除・・・ログをクリア
－データベース設定管理：以下のサブメニューをもつ
　－サーバ設定・・・サーバ、ユーザ、ディレクトリ、ログファイル名を設定する
　－マスタ作成・・・データベースを一覧するテーブルを作成する
　－個別データベース削除・・・選択したデータベースを削除する
　－全データベース削除・・・全てのデータベースと一覧テーブルを削除する
　－データベース集計・・・選択したデータベースの各テーブルのサイズを表示する

アプリの表題は、VC-4D となっているが、仮想コンバータのコンパイラ・インタプリタ処理系は使用せずに、PC 上で SQL の基本的な入出力に関する動作確認を行う関数群を作成するために作成した。メモリ上に展開され表示されているオブジェクトを、SQL テーブルに保管し、あるいは SQL テーブルからメモリ上に取得し表示する。後の段階では、仮想コンバータを組み込んだ処理系が作成した SQL テーブルの内容を表示確認するためのツールとして使用した。

データベース関連の設定を行うダイアログが、メニューの[SQL][データベース設定管理][サーバ設定]から開く、図 3-6-3 に示した SQL サーバ設定画面である。

図 3-6-3 SQL サーバ設定

ソースコードに含まれる SQLDB.cpp の中に、データベースへの接続、SQL コマンドの発行、エラー処理、処理速度の計測実験などを行う関数群を作成した。

データベースとしては、MSSQL または SQLEXPRESS を使用した。

sql サーバにコマンドラインでアクセスするためには、コンソールから、

>sqlcmd –S (local)¥sqlexpress –E

>sqlcmd –S .¥sqlexpress –E

と入力する。接続に成功すれば

1>

という状態で SQL コマンドを受け付ける状態となる。

ここで「exit」と入力すれば、終了する。

>sqlcmd –S .¥sqlexpress –U sa

というオプションでログインすると、パスワードを尋ねてくる。

Windows の C++プログラムからデータベースにアクセスする方法は複数存在する(４－５)。本処理系では、ADO (ActiveX Data Objects) を使用した。接続方法をリスト 3-6-1 に示す。

リスト 3-6-1 VC-4D.exe におけるデータベースの接続

```
１．基本的な変数
_ConnectionPtr m_con;
_RecordsetPtr m_rs;
_variant_t t;

２．開始
CoInitialize(NULL);
m_con.CreateInstance(__uuidof(Connection));
m_con->Open(str,"ユーザ名","パスワード",adConnectionUnspecified);

３．各種処理：SQL コマンド str を以下のように実行する
m_rs = m_con->Execute(str,NULL,adCmdUnknown);
GetFieldValue(m_rs,t,フィールド名);
(t に、データ型に応じた値が格納されているので処理系で利用する)
  データ型は、t.vt の数値で知ることができる。

４．終了処理
m_con->Close();
m_con.Release();
m_con = NULL;
CoUninitialize();
```

この手順は、Windows 上の MSSQL を使用するための特殊な方法である。そこで処理を一般化するために、SQL コマンドを能率的に発行するための関数

pRs Q(const char *format, …);

を用意した。printf(書式文字列,引数リスト)と同様の書法で SQL コマンドを文字列として発行している。戻り値はレコードセット形式であり、これを用いて個々のデータを参照することができる。

以上のような、Windows＋MSSQL に固有の処理を sqllib.cpp に集約し、SQL 文の生成を主な内容とするライブラリ関数の実行を cci_sql.c に、また main 関数を含むソースコードを vc-4d_exe.cpp に分離する方法で、仮想コンバータを構成するソースコードの可搬性に配慮した。

三次元図形を記述する新たなデータベースを作成する際には、createdb 関数を呼び出す。

この関数は以下のような処理を行う。

・必ず存在する master データベースを指定して SQL サーバを開く。

・dbname 変数で指定した名称の新たなデータベースを作成する。

・vc4d データベースの master テーブルに、この新たなデータベース名を追記する。

・dbname 変数で指定した名称の新たなデータベースに、COORD, COLOR, NORMAL, TCOORD, VERTEX, FACE, GRUPPE, LINK のテーブル群を作成する(リスト 3-22)。

・SQL サーバを閉じる。

リスト 3-6-2 データベース作成処理

```
int createdb(char *dbname){//vc4d_exe.cppから参照
          if(SQL_ErrCt) return 0;
          if(SQL_D) path = SQL_D;
          else path = "c:¥¥@keikan¥¥kdb¥¥snapshot";

          DB("master");
          Q("CREATE DATABASE %s ON(NAME=%s_dat, FILENAME='%s¥¥%s.mdf')"
           " LOG ON(NAME=%s_log,FILENAME='%s¥¥%s.ldf')",dbname,dbname,path,dbname,dbname,path,dbname);
          Q("USE vc4d");
          Q("INSERT master VALUES('%s')",dbname);
          Q("USE %s",dbname);
          Q("CREATE TABLE COORD (id int IDENTITY(1,1), X float, Y float, Z float)");
          Q("CREATE TABLE COLOR (id int IDENTITY(1,1), R float, G float, B float, A float)");
          Q("CREATE TABLE NORMAL (id int IDENTITY(1,1), X float, Y float, Z float)");
          Q("CREATE TABLE TCOORD (id int IDENTITY(1,1), X float, Y float)");
          Q("CREATE TABLE VERTEX (Gid int, Fid int, Ix int, Icoord int, Inormal int, Icolor int, Itcoord int,
            Ivirtual int)");
          Q("CREATE TABLE FACE (Gid int, Fid int, idcolor int, idnormal int, idmaterial int, idtexture int,
            SHP int)");
          Q("CREATE TABLE GRUPPE (Gid int, idmaterial int, idtexture int, Attribute varchar(max))");
          Q("CREATE TABLE LINK (id int IDENTITY(1,1), GPid int, GCid int,"
                              "mat0 float, mat1 float, mat2 float, mat3 float,"
                              "mat4 float, mat5 float, mat6 float, mat7 float,"
                              "mat8 float, mat9 float, mat10 float, mat11 float,"
                              "mat12 float, mat13 float, mat14 float, mat15 float)");
          DB(NULL);
          return 1;
}
```

COORDテーブルは、座標値のidと 三次元座標値(浮動小数)を格納する。

COLORテーブルは、カラーのidとRGBAの４数値を格納する。

R(赤)G(緑)B(青)A(不透明度)の各値は、0.0～1.0の間の浮動小数である。

NORMALテーブルは、法線ベクトルのidとXYZの３成分値(浮動小数)を格納する。

TCOORDテーブルは、テクスチャ座標のidとUVの２値(浮動小数)を格納する。

VERTEXテーブルは頂点を登録し、帰属するグループのid、面のid、頂点番号Ix に続き、頂点の属性を座標値Icoord、 法線ベクトルInormal、カラーIcolor、テクスチャ座標Itcoord、仮想線フラグIvirtualの順に各テーブルの参照idで示す。

FACEテーブルは、面を登録し、帰属するグループのid、面の通し番号に続き、面の属性をカラーidcolor、法線ベクトルidnormal、材料idmaterial、テクスチャidtexture、形状（面、折れ線、点群）を示すコードSHPで示す。

面の通し番号は、１から始まり、帰属するグループの中で連続している。

従って、一つの面が複数のグループに帰属することはない。

GRUPPEテーブルは、意味あるまとまり（形状の単位）を記述する。

このGidが、面や頂点の帰属先を示すために参照される。

Gidは開始番号が不定であり、連番で登録されている必要はない。

＊なお、本来であればライブラリ関数名に対応した「GROUP」をテーブル名とするのが明快であるが、SQLサーバにおける予約語として使用不可であったため、方言的名称とした。

LINKテーブルは、複数の意味あるまとまりの親子関係を記述する。

関係のid、親グループのid、子グループのid、相対的な位置関係を記述する同次行列**の16成分(浮動小数)を登録する。

**同次変換行列、同次座標変換行列、斉次行列、homogeneous transformation matrix

この親－子のペアは、例えば団地と住宅、建物と部材等の関係である。

上記のように、VC-4Dにおいては、例えば頂点を格納するVERTEXテーブルを面毎に作成するのではなく、一つの記録に対応するデータベースに属する全ての頂点を一つのテーブルに格納するとともに、個々の頂点にはそれが帰属するグループと面のIDを登録している。ある面に属する頂点を取り出すためには、SQLコマンドのSELECT文で立体と面のidを絞り込む。

リスト3-6-3 データベース削除処理

```
void deletedb(char *dbname){
        DB("master");
        Q("DROP DATABASE %s",dbname);
        Q("USE vc4d DELETE FROM master WHERE name='%s'", dbname);
        DB(NULL);
}
```

データベースを削除する場合には、deletedb 関数を呼び出す。この関数は、dbname で指定した名称のデータベースを削除すると共に、vc4d データベースの master テーブルからこのデータベース名を削除する。

仮想コンバータの機械語の実行時にライブラリ関数が呼び出されると、cci_sql.cで定義した処理が実行される。例えばCOORD関数が実行されるとVC-4Dにおいては、リスト3-24に示したようなcoord関数を起動する。この実装では、データベースのcoordテーブルから、引数のXYZ座標値が一致するレコードを探し、存在すればそのidを返す。存在しなければ、新たなレコードを追加しそのidを返す。

リスト 3-6-4 VC-4D における COORD 関数による頂点座標の登録処理

```
int coord(double v[3]){//一致するものがあればそのIDを、なければ追記し新しいIDを、返す（目標）
        int susun;
        Sel(hS);
        susun = TABLESIZE("coord WHERE X = %lf and Y = %lf and Z = %lf",v[0],v[1],v[2]);
        if(0 < susun){
                Q("SELECT id FROM coord WHERE X = %lf and Y = %lf and Z = %lf",v[0],v[1],v[2]);
                return Vint("id");
        }
        Q("INSERT coord VALUES(%lf,%lf,%lf)",v[0],v[1],v[2]);//
        Q("SELECT IDENT_CURRENT('coord') as hasil");
        return Vint("hasil");
}
```

（２）VC-4D(win)（2012.10.24）

上記(1)のように景観シミュレータに依存せずに、Windows 上で単独でダイアログを表示し動作するスタンドアロンの実行形式である。初期の開発段階では、この実行形式の中で直接データベースの作成や読み出しを実行していたが、最終段階では、③で解説するサーバー用のコンソール・アプリを、この処理系から別プロセスとして起動する方法とし、③の開発におけるテスト・デバッグのためのフレームとして使用した。

図 3-6-4 VC-4Dwin の初期画面

この試作プログラムは、既存の形状記述ファイル（データファイル）と形式定義ファイル（メタファイル）をそれぞれファイル・ダイアログで選択し、任意の新しいデータベース名を記入してアップロードボタンを押すと、データベース名の SQL データベースを構築する Windows アプリケーションであり、単独で使用することができる。

既存のデータベース名をドロップボックスで選択し、形式定義ファイル（メタファイル）を選択ボタンから開くファイル・ダイアログで選択し、形状記述ファイルに任意の新しいファイル名を記入してダウンロードボタンを押すと、メタファイルで指定した形式で、形状記述ファイル欄に指定した名称のファイルを生成する。

左上のアイコンをクリックすると、バージョン情報のダイアログが開く。このダイアログの中で、図 3-20 とほぼ同様に、使用する SQL サーバ名、ユーザー名、パスワード、ディレクトリ、実行形式、ログファイル名を指定する。

パスワードは、暗号化されたバイナリファイルに記録する。このファイルは、サーバー上にセットアップする際に使用することができる。

ディレクトリは、データベースを新たに作成する際にファイル(.mdf, .ldf)を作成するディレクトリである。

実行形式は、データベースの入出力のために起動する別プロセスの実行形式④である。

（３）VC-4D.exe(2012.12.26)

サーバー上で動作するコンソールアプリであり、引数としてデータファイル、メタファ

イル、SQL 名称を受け取り、SQL データベースを構築する。

また、引数として指定されたメタファイルと SQL 名称から、指定した名称のデータファイルを再構築する。入力か出力かは引数として渡されるフラグで識別する。

サーバー側の WEB アプリケーションの処理ロジックは java でプログラムされており、ユーザーから必要な情報が取得された後に、実行形式 vc-4d.exe が起動されて、メタファイルのコンパイルを行い、SQL データベースの入出力を行う。

main 関数をリスト 3-6-5 に示す。

リスト 3-6-5 VC-4D.exe の main 関数

```
int main(int argc, _TCHAR* argv[], char*env[])
{
	char *metafile;
	char *datafile;
	char *sqlname;
	char *logfile;
	char *p;
	int IO;
	int hasil;

	showenv(env);
	if(argc<5) return 2;
	metafile = q(argv[1]);
	datafile = q(argv[2]);
	sqlname = q(argv[3]);
	IO = atoi(q(argv[4]));
	if(IO<0) return 2;
	if(2<IO) return 2;
	/*ログファイル名称の指定、無ければデフォルト名を使用*/
	if(5<argc) logfile = q(argv[5]);
	else logfile = q("Slog.geo");
	p = strrchr(logfile,'¥¥');
	if(p) p++;
	else p = logfile;
	if(!strcmp(p,"result.txt"))
		strcpy(p,"tluser.txt");/*ログ名result.txtが要求*/
	/*以下の戻り値は、それぞれの処理の中で決定(0-)*/
	if(IO==1) hasil = upload(metafile,datafile,sqlname,logfile);
	else hasil = download(metafile,datafile,sqlname,logfile);
	summary(logfile,hasil);
	return hasil;
}
```

アップロードとダウンロードは同じ実行形式が行っている。入出力は、第 4 引数で区別し、upload 関数(リスト 3-6-6) か download 関数(リスト 3-6-7)に制御を渡している。

リスト 3-6-6 VC-4D.exe の upload 関数

```
int upload(const char*m_MetaFile, const char*m_DataFile, const char*dbname, const char *logfile){
	int i,hasil;
	char debugcommand[240];

	ReadOnly = 0;
	SQL_ErrCt = 0;
//	fopen_s(&stberr,"Slog.geo","wt");
	fopen_s(&stberr,logfile,"wt");
```

```
            if(!stberr) return 2;//ログファイルが開かない
            fprintf(stberr,"#upload logfile:%s¥n",logfile);
            fprintf(stberr,"#metafile:%s¥n",m_MetaFile);
            fprintf(stberr,"#datafile:%s¥n",m_DataFile);
            fprintf(stberr,"#database:%s¥n",dbname);
            sprintf_s(debugcommand,240,"notepad.exe %s",logfile);
            errorlog = stberr;

            i3UserInfo *ui;
            char Usedb[80];

            ui = Makelist();
            for(i=0;i<ui->count;i++){
                        if(!strcmp(ui->keywords[i],dbname)) break;
            }
            if(i < ui->count){
                        fprintf(stberr,"#既存のDB(%s)に上書きupload¥n",dbname);
                        deletedb((char*)dbname);
            }
            if(!createdb((char*)dbname)){
                        fprintf(stberr,"#DB(%s)作成失敗。終了する",dbname);
                        goto end;
            }
            DB((char*)dbname);
            sprintf_s(Usedb,80,"USE %s",dbname);
            usedb = Usedb;

            InitProgressFILE();//インジケータ初期化(0を初期値とする)

            fprintf(stberr,"#C--コンパイラ　ログ¥n");
            SetRootGroup(0);//cci_sql の面、頂点リストに係るメモリブロック、カウンタ等初期化
            HAJIME();//cci 初期化

            if (compile((char*)m_MetaFile)) {//argv[1]はメタファイル
              fprintf(stberr,"#コンパイル成功(%d行=>%d行)¥n",srcLineno,codeCt);
              if (strncmp(m_DataFile,"--code",6)==0){//入力ファイル名の冒頭６文字
                          code_dump();
              }else{
                          fprintf(stberr,"#---------------実行開始--------------¥n");
                        hasil = execute((char*)m_DataFile);
                        untable();
                        fprintf(stberr,"#---------------実行終了--------------¥n");
                        fprintf(stberr,"exe_ret_code=%d¥n",hasil);
                        reset();//メモリ解放
              }
            }else{
              fprintf(stberr,"#コンパイル失敗¥n");
            }
end:
            fprintf(stberr,"SQL_ErrCt=%d¥n",SQL_ErrCt);
            fprintf(stberr,"err_ct=%d¥n",err_ct);
            fprintf(stberr,"exe_err_ct=%d¥n",exe_err_ct);
            DB(NULL);
            fclose(stberr);
            reset();//cci_sql メモリブロック等の解放
            Freelist(ui);
//リターンコード生成
            if(0 < SQL_ErrCt) return 3;//SQLエラー
            if(0 < err_ct) return 4;//コンパイル・エラー
            if(0 < exe_err_ct) return 5;//ランタイム・エラー
            switch(hasil){
                        case 0:
                        case 1:
```

```
                                return hasil;
                        default:
                                return 6;
        }
}
```

リスト3-6-7 VC-4D.exeのdownload関数

```
int download(const char*m_MetaFile, const char*m_DataFile, const char*dbname, const char *logfile){
        int hasil;
        char debugcommand[240];
        char Usedb[80];

        ReadOnly = 1;
        fopen_s(&stberr,logfile,"wt");
        if(!stberr) return 2;//ログファイルが開かない
        fprintf(stberr,"#download logfile:%s¥n",logfile);
        sprintf_s(debugcommand,240,"notepad.exe %s",logfile);
        errorlog = stberr;
        if(!strncmp(m_DataFile,"--code",6)) goto dump;
        fopen_s(&outfile,m_DataFile,"wt");
        if(!outfile){
                fprintf(stberr,"#outfile(%s)が開けない¥n",m_DataFile);
                hasil = 2;
                goto end;
        }

        SQL_ErrCt = 0;
        DBname = NULL;
        DB((char*)dbname);
        sprintf_s(Usedb,80,"USE %s",dbname);
        usedb = Usedb;
        if(SQL_ErrCt){
                hasil = 3;
                goto end;
        }
        InitProgressSQL();//インジケータ初期化(面の総数を初期値とする)

        fprintf(stberr,"#C--コンパイラ　ログ¥n");
        SetRootGroup(0);//cci_sql の面、頂点リストに係るメモリブロック、カウンタ等初期化
        HAJIME();//cci 初期化
dump:
        if (compile((char*)m_MetaFile)) {//argv[1]はメタファイル
          fprintf(stberr,"#コンパイル成功(%d行=>%d行)¥n",srcLineno,codeCt);
          if (strncmp(m_DataFile,"--code",6)==0){//入力ファイル名の冒頭６文字
                  code_dump();
          }else{
                fprintf(stberr,"#---------------実行開始--------------¥n");
                fprintf(stberr,"#DB(%s)からoutfile[%s] に出力する¥n",dbname,m_DataFile);
                hasil = execute(NULL);//入力ファイルはなし
                untable();
                fprintf(stberr,"#---------------実行終了--------------¥n");
                fprintf(stberr,"exe_ret_code=%d¥n",hasil);
                reset();//メモリ解放
          }
        }else{
          fprintf(stberr,"#コンパイル失敗¥n");
        }
end:
        fprintf(stberr,"SQL_ErrCt=%d¥n",SQL_ErrCt);
        fprintf(stberr,"err_ct=%d¥n",err_ct);
        fprintf(stberr,"exe_err_ct=%d¥n",exe_err_ct);
        DB(NULL);
```

```
        if(outfile) fclose(outfile);
        if(stberr) fclose(stberr);
        reset();//cci_sql メモリブロック等の解放

//リターンコード
        if(0 < SQL_ErrCt) return 3;//SQLエラー
        if(0 < err_ct) return 4;//コンパイル・エラー
        if(0 < exe_err_ct) return 5;//ランタイム・エラー
        switch(hasil){
                case 0:
                case 1:
                        return hasil;
                default:
                        return 6;/*メタファイルによる異常終了報告：コードは、ログファイル参照*/
        }
}
```

download 処理においては、出力形式を定義するメタファイルに従って、データファイルとして名称が指定されたファイルに書き込みを行う。このとき、保存データファイルは既に解読されて SQL サーバのテーブルとして展開され蓄積されている。メタファイルは、このテーブル群にアクセスしてデータを取り出し、これを用いて別形式のファイルを出力する。

この処理を行うために、テーブル群にアクセスするための出力系ライブラリ関数群を増補した。これらは、保存ファイルに添付するメタファイルにおいて利用されることはない。従って、利活用段階でこれらの出力系ライブラリ関数を更に増補し、あるいは目的に応じて動作を変更しても、保存ファイルの解読処理に影響を及ぼすことはない。保存時点から年数が経過した後の、利活用時点における処理系に属する機能である。これらの内部処理に関しては３－７で、また別形式での出力用に作成する第二のメタファイルにおける参照方法に関しては４－２(3)で解説する。

（４）WEB アプリケーション「三次元データ保管庫」

以上の段階で開発した、メタファイルの記述に従いデータファイルを処理し、SQL サーバへの入出力を用いる実行形式（エンジン部分）を、ユーザーからのリクエストに基づき起動し、得られた結果を返送する WEB アプリケーションを「三次元データ保管庫」としてとりまとめた（２－７）。この処理系においては、Java で記述され稼働しているサーバ側のアプリから、必要に応じて VC-4D.exe が起動され、引数として渡されたデータファイル名、メタファイル名、およびログファイル名等の情報に基づき処理を行う。終了後にサーバーアプリに制御が戻される。このような前記 VC-3M と同様の国総研と受注者マックスネットコンサルティング株式会社との間での業務の切り分けにより、インターフェース境界をめぐる支障を避けつつ、同時平行的にプログラム、デバッグの作業を進めることができた。

３－７．出力系のライブラリ関数の開発と出力系メタファイルの作成

VC-4D の特徴は、一度 SQL サーバー上に構築された三次元データを系統的に取り出し、ユーザーがリクエストした形式に再編成してデータファイルに出力する工程のための機能である。このために、データベース上のオブジェクトにアクセスするためのライブラリ関

数群を用意した。ダウンロードのリクエストと共に添付した第二のメタファイルは、これらのライブラリ関数を呼び出し、取得した数値などを printf 等の組込関数に渡して、求められたデータファイルの出力を行う。

これらの関数は、①VC-1C においては、全て何もしない関数として、cci_ip の中でダミー定義している。②VC-2V においては、同じ仕様で dml メモリ空間から形状や属性を取り出す処理を記述した。

これらの出力系ライブラリ関数を用いたファイル出力の形式定義は、利活用局面において使用するものであるため、データファイルの長期保存に際して添付するメタファイルの中で必ずしも使用しなくともよい。

int G(); 全てのグループを順にアクセスし、ID を返す。終了時はゼロを返す。
int Gattribute(); アクセスされているグループの文字列を登録し、メモリアドレスを返す。

リスト 3-7-1 全てのグループへの順次アクセス

```
int gid,fid;
while(gid=G()){//全てのグループに順次アクセスする
  //ここに、選択されているグループに関する処理を記述する
  d = Gattribute();//グループに設定されている文字列をメモリ上に取得しアドレスを返す
  //ここに、属性を用いた出力処理を記述する
  while(fid=F()){//選択されたグループに属する全ての面に順次アクセスする
    //ここに、選択されている面に関する処理を記述する
    while(vid=V()){//選択された面に属する全ての頂点に順次アクセスする
    }
  }
}
```

仮想現実を記述する VRML 形式や、建築物や機械部品の CAD データを保存する DXF 形式において、予め定義された部品やシンボルが繰り返し位置や向きを変えて配置されているシーングラフ型のデータ構造の場合には、リスト 3-7-2 のようにアクセスする。

リスト 3-7-2 全ての表示グループへの順次アクセス

```
int did;
while(did=D()){
  //ここに、選択されている表示グループに関する処理を記述する
}
```

例えば、２０本の同じネジを使った家具が、6 個配置されているとすると、ネジは全体で 120 本存在する。この時、ネジの形状を定義したグループは一つで足りる。このような空間構成に対して、D()関数を使いアクセスすることにより、120 回配置されている同一の

ネジのそれぞれの位置と向きを取得することができる。

この時、親子関係を示すリンクテーブルには、２６の位置関係が定義されている。そのうち６は、６の家具の位置に関する情報であり、２０は、一つの家具の中に用いられているネジの位置に関する情報である。このような位置情報にアクセスするためには、L()関数を使用する。

また、この時、グループは二つだけ存在し、一つは家具の形状を示すグループ、もう一つはネジの形状を示すグループである。上記の G()関数はこの二つのグループを取得する。

D（）関数を用いることにより、表示されている全てのオブジェクトを再構成することができ、STL 形式のような模型加工のデータを生成することができる。G（）関数と L（）関数を用いることにより、冗長性を避けた最小限の形状定義によるデータを再構成することができる。

データアクセスに関する関数は、記録保存段階では必要ではなく、利活用段階において必要となる機能である。しかしながら、記録保存段階で、記録保存方法が有効であることを検証（ベリファイ）する手段としては有用である。

ここでは、出力系のライブラリ関数の仕様と内部処理について解説し、メタファイルからの呼び出し等の用法については、４－２（３）で解説する。

（１）出力系のライブラリ関数

出力系のライブラリ関数はまず VC-4D において、構築済のデータベースのテーブル要素を読み出す機能として実装した。内部処理においては、SQL 文を発行し得られたレコードセットに順次アクセスするような構成となっている。

具体的には、それぞれの階層のオブジェクト（立体、面、頂点等）が複数存在する場合に、そのオブジェクト群（データベースにおけるレコードセットの概念に相当する）の中のオブジェクトを先頭から順次選択し、残りのオブジェクト数を戻り値として返す整数型のオブジェクト関数を基本形とした。更に、現在選択されているオブジェクト（例えば立体）にカーソルが置かれた状態で、そのオブジェクトの構成要素（例えば面）を、別の関数を用いて同じような手順で順次アクセスする。

さらに、現在選択されているオブジェクト（例えば頂点）の様々なパラメータ（例えば座標値や色彩）を数値として取り出すライブラリ関数を用意した。

これらのライブラリ関数が、SQL データベースに保存データの解読結果を蓄積する VC-4D の上で正常に動作し、ファイル出力処理を記述するために有用であることを確認した上で、出力系のライブラリ関数が同じようにメモリ上に保存データの解読結果を蓄積し３D 表示に使用する VC-2V においても同様の手順でデータを読み出すように cci_dml においても実装した。その際には、オブジェクトへのアクセス順序戻り値の定義に関して一貫性を保つための検証と調整を行い、互換性を確保した。これにより、同じメタファイルを使用して、二つの経路で保存データを対象としたファイル変換を行うことができる。

保存データ＋メタファイル→VC-4D による解読（DB）→別形式の出力

保存データ＋メタファイル→VC-2V による解読（表示）→別形式の出力

これにより、同じ保存データとメタファイルから出発し、この別形式の出力に際して、同じ第二の出力用メタファイルが二種類の異なる処理系である VC-4D 上と VC-2V 上で同じファイルを出力するような結果が得られた。以下、各関数に関して解説する。

① G()関数

全ての立体に順次アクセスするために使用する。

初回：立体の総数 nG をカウントし、最初の立体にカーソルを当てる。 次回：次の立体にカーソルを移動する。 終了：次の立体が無い場合、初期化する。 戻り値：選択された立体を含む残りの立体数（初回は総数)。

これにより、出力系メタファイルにおいて、リスト 3-7-1 に示したようなループ処理が簡単に書ける。

派生的な関数として、選択された（カーソルの当たっている）立体の ID 値を取り出す Gid （）関数、属性を取り出す Gattribute()関数を用意した。

② F()関数

選択された立体に属する全ての面に順次アクセスする

初回：面の総数ｎF をカウントし、最初の面にカーソルを当てる。 次回：次の面にカーソルを移動する。 終了：次の面が無い場合、初期化する。 戻り値：面の ID（1～N)またはゼロ（終了）

これにより、出力系メタファイルにおいて

```
while(F()){
   処理
}
```

というループが簡単に書ける。

選択されている面の属性を取得する関数として、Fshp(), Ftexture(), Fmaterial(), Fcolor(), Fnormal()を用意した

③ V()関数

カーソル面に属する全ての頂点に順次アクセスする

初回：頂点の総数ｎF をカウントし、最初の面にカーソルを当てる。 次回：次の頂点にカーソルを移動する。 終了：次の頂点が無い場合、初期化する。 戻り値：面の ID（1～N)またはゼロ（終了）

これにより、出力系メタファイルにおいて

```
while(V()){
  処理
}
```

というループが簡単に書ける。

派生的な関数として、Vcoord,Vcolor,Vnormal,Vtexture を用意した

更にその下の派生関数として、Sx,Sy,Sz,Sq,Cr,Cg,Cb,Ca,Cq,Nx,Ny,Nz,Nq,Tx,Ty,Tq を用意した。

④ F3()関数

カーソル面を三角形分割し順次アクセスする。

三角形分割された後の面の総数を nF3 に保存する。

三角形分割が行われた場合(0<nF3)、V_()コマンドは、分割三角形の 3 頂点に順次アクセスする。

⑤ S_()コマンド

全ての頂点座標に順次アクセスする。オブジェクト毎に頂点座標や面を出力するのではなく、予め全ての頂点座標を定義しておいて、それに基づいて面や立体を構成するようなファイル形式での出力を行う処理のために用意した。

⑥ N_()コマンド

全ての法線ベクトルに順次アクセスする。

⑦ T_()コマンド

全てのテクスチャ座標に順次アクセスする

⑧ C_()コマンド

全てのカラーに順次アクセスする

⑤～⑧のライブラリ関数は、立体－面－頂点の構造や選択状況に関わりなく、各テーブルに登録された全てのレコードに順次アクセスする。これらの処理は、SQL データベース上に展開されたデータに関しては、一つへのテーブルへの順次アクセスとして実装が容易であるが、cci_dml において同一機能を実現するためには工夫が必要となった。具体的には(時間はかかるが) 初回アクセスで、全空間の例えば⑧の場合、カラーを検索しリスト化し、先頭にカーソルを置く。二度目からはリストの次の要素にアクセスし、終端に到達した段階で、リストを解放する。

このリストとしては、例えば⑧の場合、データファイルの読み込み段階でライブラリ関数 COLOR(・・・);が実行される度に成長するリストを再利用することができる。しかし、VC-2V 処理系においては、読み込み処理が終了した段階で構築された配列は一度解放されているため、C()関数が呼び出される出力処理段階では、初回アクセスで、メモリ上の全ての面や頂点に定義されたカラーからこの配列を再構築している。その際にソーティングを行っている。

（２） 凹ポリゴン

面が凹ポリゴンである場合、面の種別は通常のポリゴン(D3_SHP_FACE=0)ではなく、凹ポリゴン(D3_SHP_CONCAVE=2)の属性が SHP メンバとして付与される。

面の SHP メンバは､IP の FACE コマンドの中で明示的に面毎に指示されることはなく、CONCAVE(ON); CONCAVE(OFF); のスイッチにより、FACE コマンドが発行された時点でのコンテキストとして設定される。

DML,DRL においては、F->shp の属性を見て、図形演算や描画の方法を決定している。

従って、入力に際しては、FACE コマンドを実行する前に、その面の凹凸を判定して、必要な CONCAVE コマンドを直前に発行する。

LSSG 出力に際しては、FACE コマンドを発行する前に凹ポリゴンであれば、その面の凹凸を判定して、CONCAVE コマンドを直前に発行する。

SQL においては、FACE テーブルの第七列として、SHP 属性を保存している。

因みに、凸面(0)と凹面(2)以外に、線(1)と点群(3)の属性があり、これらは同じく面のテーブルに保存される。また、（４）で解説する穴あきポリゴンの場合には、必ず凹(2)となる。

（３） VERTEX コマンドとポイントの扱い

メタファイルにおけるライブラリ関数である VERTEX 関数は、LSSG 形式における VERTEX コマンドとの互換性を保つために､以下の特殊な引数の与え方を可能としている。

・フルスペック：５個の引数

VERTEX(V,N,T,C,Va);

V：頂点名称

N：法線名称

T: テクスチャ座標名称

C：カラーを定義名称

Va：仮想線を定義する頂点名称

ここで言う名称とは、仮想コンバータにおいてはメタファイル中の変数名である。

実行時の VERTEX 関数の内部処理においては､この変数に代入された ID が使用される。

データファイルに使用された変数名を変数として使用したい場合には、

scanf 関数等で、データファイルの変数を記号表に登録し、そのアドレスをメタファイルの変数に取得する。VERTEX 関数の引数には、「＊メタファイル変数」の表現により、

データファイルの変数に代入されている値を使用する。

なお、景観シミュレータの歴史的経緯により、内部処理における D3_VA_XXX の定義や d3Vertex 構造体では、以下のような順が用いられており、外部ファイルの VERTEX の引数順とは異なっている点には公開ソースコードのライブラリ関数を応用する際に注意を要する。外部ファイルの順序を変えると、これまで作成した大量のファイルが読めなくなる。また､内部構造体の順序を変えると､プラグイン.dll 等の処理に不具合が生じる恐れがある。

リスト 3-7-3 景観シミュレータの dml ライブラリにおける VERTEX 構造体定義

```
 (d3dml.h)
#define D3_VA_NORMAL        0x0001
#define D3_VA_COLOR         0x0002 /*何故か、IP,struct と異なり、texture の前にある*/
#define D3_VA_TEXTURE       0x0004
#define D3_VA_VIRTUAL       0x0008 /*990115 DR.H.K. sifat pulau lobang*/

struct _d3Vertex {
        unsigned long va;
        double v[3];
        float n[3];
        float t[2];
        float c[4]; /*この順序は外部関数と同様である。*/
};
```

（省略形について）

頂点座標は必須であるが、それ以外の要素は省略可能である。

その場合、省略された項目は空欄とすることができる。

[例]VERTEX(P1,,,P2);

更に、ある項目以降が全て空欄である場合には、省略することができる。

[例]VERTEX(P);　VERTEX(P,,T);

内部処理においては、空欄が入力された項目は-1 として処理されている。

（４）穴あきポリゴンと仮想線

景観シミュレータ(1996-)においては、穴あき図形は仮想線により表現した。

この方法の長所は、OpenGL を用いた三次元画像表示において、通常の凹ポリゴンと同じ方法で穴のあいた面がそのまま高速描画できる点にある。

つまり、三角形の中に三角形の穴が抜けた図形は、２辺が重なる８角形と等価である。

但し、頂点や辺を集計する場合には、幾何学的に意味のない便宜的に設定された仮想線は除外する必要がある。またワイヤーフレームの表示を行う場合には、仮想線は表示しない方が明快である。このような場合に明示的な違いが生じる。

穴あきポリゴンの表現方法は二次元システムにも共通する問題である。GIS 系のデータ形式においては、外側を親、中の穴を子とする二層のデータ構造を有する場合が多い。漢字等の形状を記録するフォントファイルも二層のデータ構造となっている。その場合には、仮想線による表現に変換するか、三角形分割しなければ OpenGL 等のグラフィックライブラリで表示できない。

この変換ないし分割の方法は一意的には定まらない。また、担当エリアの入力オペレータの操作を反映して、複数の独立した領域が、それぞれ独立した図形となっていたり連結した図形となっていたり、回る向きが反転していたり一貫しない場合も見受けられる。このような幾何的に不正なデータであっても、地図などの二次元の線画として表示する場合には同じ表示結果が一応得られる。

これに対して逆に、仮想線による表現を二層構造（外周＋穴）に変換する処理は、一意的に行うことができる。

△P0-P1-P2 と、穴△P3-P4-P5 の間を、仮想線 P0-P3 が繋いでいる場合、

P0 の第五引数に P3、P3 の第五引数に P0 を設定する。

面は、頂点列 P0-P1-P2-P0-P3-P4-P5-P3 として定義する。

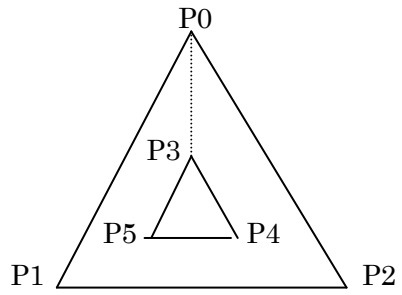


図 3-7-1 穴あきポリゴンの表現

面の描画においては、通常の凹ポリゴンとして P0-P1-P2-P0-P3-P4-P5 を描く
ワイヤーフレーム描画においては、P0-P3 と、P3-P0 のラインの描画を省略する。
dml 空間においては、面には長さ８の d3Vertex 構造体の配列が割り付けられ、次の頂点との間が仮想線である場合、配列 d3Vertex の va メンバにビットフラグが建っている。

リスト 3-7-4 インタープリタによる仮想線をもつ面の出力処理

```
#define D3_VA_NORMAL        0x0001
#define D3_VA_COLOR         0x0002
#define D3_VA_TEXTURE       0x0004
#define D3_VA_VIRTUAL       0x0008
```

LSS-G 形式の外部ファイルとの入出力を行うインタープリタ上の ip 空間においては、DML 表示用８頂点の内、重複している２頂点の出力を１頂点名称に統一した上で６頂点のみを出力し、頂点の属性として仮想線の先の頂点を属性として表示する。

リスト 3-7-5 仮想線のある面の出力結果

```
P00 = VERTEX(V00, , , C00, V03);
P01 = VERTEX(V01, , , C01);
P02 = VERTEX(V02, , , C02);
P03 = VERTEX(V03, , , C04, V00);
P04 = VERTEX(V04, , , C05);
P05 = VERTEX(V05, , , C06);
F00 = FACE(P00, P01, P02, P00, P03, P04, P05, P03);
```

言い換えると、６の頂点を有する穴あき面の頂点列

（外形）P0-P1-P2（穴）P3-P4-P5 を、

P0(P3)-P1-P2-P0(P3)-P3(P0)-P4-P5-P3(P0)

と表現する。(　)は、ワイヤーフレーム表示や線・頂点の集計において省略する同一頂点から発する仮想線の先の頂点を示している。

面を構築する FACE 関数では、P0(P3)-P1 は通常のワイヤーとして扱い、P0(P3)-P3(P0) は、ワイヤーを省略するように面を構成する。

面を構成する頂点を順次取得しファイル出力する、景観シミュレータの i3_outputFace 関数(i3ipoutput.c)では、８の表示用頂点と、６の座標値を出力するために、座標値の比較

を行っていた（リスト 4-5-15）。本処理系では、座標値を登録する段階でソーティングが行われているため、V 関数で８の表示用頂点が順次取得されるが、それらが参照する頂点座標の数は６である。

リスト 3-7-6 景観シミュレータのインタープリタにおける出力部分

（頂点座標関連以外の、カラーや法線等の処理を省略する）

```
int i3_outputFace(FILE *fp,d3Face *f,int type)
{
  int i,j,*nomor,nC,nN,*nomC,*nomN;
  d3Vertex *vl,*vlasli;

  nomor = d3Malloc(f->vnum * sizeof(int) );
 if(OPTIM & I3_OPT_N) nomN = d3Malloc(f->vnum * sizeof(int) );
 if(OPTIM & I3_OPT_C) nomC = d3Malloc(f->vnum * sizeof(int) );
  vlasli = d3GetFaceVertex(f);

  if( (7 <= f->vnum) ){//仮想頂点の抽出
        if(!vlasli){
              d3Free(nomor);
              return(FALSE);
        }
        vl = d3Malloc(f->vnum * sizeof(d3Vertex));  //作業用コピー
        if(!vl){
          fprintf(fp,"#メモリ不足のため、頂点照合省略¥n");
          vl = vlasli;
        }else{
          int mask;
          memcpy(vl,vlasli,f->vnum * sizeof(d3Vertex));
          mask = 15 - D3_VA_VIRTUAL;
          for(i=0;i<f->vnum;i++){ //仮想頂点以外の属性を除く
                if(15<vl[i].va) z3Message(3413);
                vl[i].va &= mask;
          }
        }
  }else{//頂点が 7 未満で仮想なき場合
          vl = vlasli;
  }
  //COORD,NORMAL,COLOR,TEXTURE の出力
  for(i=0;i<f->vnum;i++){
        if(vl != vlasli){//仮想橋の処理
              for(j=0;j<i;j++) if(!memcmp(vl+j,vl+i,sizeof(d3Vertex))) break;
              if(j<i) continue; /*dsb*/
        }
    if(bu){//頂点座標のソートを行う (#define bu 1)
              nomor[i]=buCOORD(fp,vl[i].v);
        }else{
              fprintf(fp,"¥tV%.2d = COORD(%0.15g, %0.15g, %0.15g);¥n",
                    i,vl[i].v[0],vl[i].v[1],vl[i].v[2]);
        }
    if(f->va_v & vl[i].va & D3_VA_NORMAL){/*処理略*/}
    if(f->va_v & vl[i].va & D3_VA_COLOR){/*処理略*/}
    if(vl[i].va & D3_VA_TEXTURE){/*処理略*/}
  }

  //VERTEX の出力
  for(i=0;i<f->vnum;i++){
        if(vl != vlasli){
              for(j=0;j<i;j++) if(!memcmp(vl+j,vl+i,sizeof(d3Vertex))) break;
              if(j<i) continue; /*出力済の位置にある重複頂点は省略する*/
        }
        fprintf(fp,"¥tP%.2d = VERTEX(V%.2d, ",nomor[i],nomor[i]);
    if(f->va_v & vl[i].va & D3_VA_NORMAL){
```

```
				if(OPTIM & I3_OPT_N){
					fprintf(fp,"N%.2d",nomN[i]);
				}else{
					fprintf(fp,"N%.2d",i);
				}
		}
		fprintf(fp,", ");
		if(vl[i].va & D3_VA_TEXTURE){
			fprintf(fp,"T%.2d",i);
		}
		fprintf(fp,", ");
//		if(vl[i].va & D3_VA_COLOR){
		if(f->va_v & vl[i].va & D3_VA_COLOR){
				if(OPTIM & I3_OPT_C){
					fprintf(fp,"C%.2d",nomC[i]);
				}else{
					fprintf(fp,"C%.2d",i);
				}
		}
		//第 5 引数として
			if(vl != vlasli){//100129 同一頂点は既にスキップしているので、ここに来るとは限らない
				int k;
				for(k=0;k<f->vnum;k++)		if(vlasli[k].va		&		D3_VA_VIRTUAL)
if(!memcmp(vl+i,vl+k,sizeof(d3Vertex))) break;
				if(k<f->vnum){//同一位置の頂点で仮想あり
					if(k+1 < f->vnum) k++; else k=0;//仮想の先
					for(j=0;j<k;j++) if(!memcmp(vl+j,vl+k,sizeof(d3Vertex))) break;//仮想の先の前出
					fprintf(fp,", V%.2d",nomor[j]);
				}
			}
		fprintf(fp,");¥n");
	}
	//FACE の出力
	if(type == I3_EXT_FACE){
		fprintf(fp,"¥tF%.2d = FACE(",FaceNumber);
	}else{	/* LINE */
		fprintf(fp,"¥tL%.2d = LINE(",LineNumber);
	}
	for(i=0;i<f->vnum;i++){
		//長い行の折り返し処理
		if(i != 0){
			if(i % 6 == 0){
				fprintf(fp,"¥n¥t¥t");
			}
			fprintf(fp,", ");
		}
/*		fprintf(fp,"P%.2d",i); */
			if(vl != vlasli){//出力済の頂点をスキップする
				for(j=0;j<i;j++) if(!memcmp(vl+j,vl+i,sizeof(d3Vertex))) break;
				fprintf(fp,"P%.2d",nomor[j]);/*dsb.*/
			}else{
				fprintf(fp,"P%.2d",nomor[i]);
			}
	}
	fprintf(fp,");¥n");

//面の属性（法線、カラー）の出力
	if(f->va_f & D3_VA_NORMAL){
			if(type == I3_EXT_LINE){
				if(OPTIM & I3_OPT_N){
					nN = buNORM(fp,f->n);
					fprintf(fp,"¥tLINE_NORMAL(L%.2d, N%.2d);¥n",LineNumber,nN);
				}else{
					fprintf(fp,"¥tN%.2d									=
NORMAL(%g, %g, %g);¥n",f->vnum,f->n[0],f->n[1],f->n[2]);
					fprintf(fp,"¥tLINE_NORMAL(L%.2d, N%.2d);¥n",LineNumber,f->vnum);
				}
```

```
                }else{//FACE
#if 1
                        if( OPTIM & I3_OPT_N ){
                                d3Face F;
                                d3FindNormalf(f,F.n);
                                //if( *f->n == *F.n )
                                if(!memcmp( f->n, F.n, 3*sizeof(float) ) )
                                        goto NORMALEND;
                        }
//#else
                        if( OPTIM & I3_OPT_N ){
                                nN = buNORM(fp,f->n);
                                fprintf(fp,"¥tFACE_NORMAL(F%.2d, N%.2d);¥n",FaceNumber,nN);
                        }else{
                                fprintf(fp,"¥tN%.2d                                                                     =
NORMAL(%g, %g, %g);¥n",f->vnum,f->n[0],f->n[1],f->n[2]);
                                fprintf(fp,"¥tFACE_NORMAL(F%.2d, N%.2d);¥n",FaceNumber,f->vnum);
                        }
#endif
                }
  }
NORMALEND:
  /*040625*/
            if(f->va_f & D3_VA_COLOR){
              if( OPTIM & I3_OPT_C ){
#if 0
                        fprintf(fp, "#FACE_COLOR とりあえず省略¥n");
#else
                        nC = buCOLOR(fp,f->c);
                        if(type == I3_EXT_FACE){
                                fprintf(fp,"¥tFACE_COLOR(F%.2d, C%.2d);¥n",FaceNumber,nC);
                        }else{
                                fprintf(fp,"¥tLINE_COLOR(L%.2d, C%.2d);¥n",LineNumber,nC);
                        }
#endif
                }else{
                                fprintf(fp,"¥tC%.2d                                                                     =
COLOR(%g, %g, %g, %g);¥n",f->vnum,f->c[0],f->c[1],f->c[2],f->c[3]);
                                if(type == I3_EXT_FACE){
                                  fprintf(fp,"¥tFACE_COLOR(F%.2d, C%.2d);¥n",FaceNumber,f->vnum);
                                }else{
                                  fprintf(fp,"¥tLINE_COLOR(L%.2d, C%.2d);¥n",LineNumber,f->vnum);
                                }
                }
            }

    if(f->material){
                        if( OPTIM & I3_OPT_M ) CountMaterialOpt++;
                        else
                                fprintf(fp,"¥tM%.2d                                                                     =
MATERIAL(%s);¥n",f->material,d3GetMaterialName(f->material));
                        if(type == I3_EXT_FACE){/*040608 DR.H.K.*/
                                fprintf(fp,"¥tFACE_MATERIAL(F%.2d, M%.2d);¥n",FaceNumber,f->material);
                        }else{
                                fprintf(fp,"¥tLINE_MATERIAL(L%.2d, M%.2d);¥n",LineNumber,f->material);
                        }
    }
    if(f->texture){
                        if( OPTIM & I3_OPT_T ) CountTextureOpt++;//テクスチャ宣言は冒頭で一括済み
                        else{
                    fprintf(fp,"¥tI%.2d = TEXTURE(¥"%s¥");¥n",f->texture,d3GetTextureName(f->texture));
                        }
                        if(type == I3_EXT_FACE){
                                fprintf(fp,"¥tFACE_TEXTURE(F%.2d, I%.2d);¥n",FaceNumber,f->texture);
                        }else{
                                fprintf(fp,"¥tLINE_TEXTURE(L%.2d, I%.2d);¥n",LineNumber,f->texture);
                        }
    }
```

```
    if(type == I3_EXT_FACE){
      FaceNumber++;
    }else{
      LineNumber++;
    }
    if(vl != vlasli) d3Free(vl);
    d3Free(nomor);
    if(OPTIM & I3_OPT_N) d3Free(nomN);
    if(OPTIM & I3_OPT_C) d3Free(nomC);
    return(TRUE);
}
```

一方、本処理系において、データファイルとメタファイルから取得され SQL データ部宇に蓄積されたたデータから LSSG 形式に出力する処理を定義するメタファイルにおいては、面の出力部分は以下のように記述される。

リスト 3-7-7 LSSG 形式を出力するメタファイルにおける仮想線の扱い

```
（一つのオブジェクトに属する面を出力するループの部分）
        while(j=F()){
            for(nv=k=V();k;k=V()){
                printf("P%d=COORD(",Vcoord());
                printf("%f,",Sx());
                printf("%f,",Sy());
                printf("%f);¥n",Sz());
            }

            for(nv=k=V();k;k=V()){
                iN = Vnormal();
                iC = Vcolor();
                iT = Vtcoord();
                iV = Vvirtual();
                printf("V%d=VERTEX(",nv-k+1);
                printf("P%d",Vcoord());
                if(iN+iC+iT+iV == 0){
                    printf(");¥n");
                    continue;
                }else printf(",");

                if(iN){
                    printf("N%d",iN);
                }
                if(iT+iC+iV== 0){
                    printf(");¥n");
                    continue;
                }else printf(",");

                if(iT) printf("T%d",iT);
                if(iC+iV == 0){
                    printf(");¥n");
                    continue;
                }else{
                    printf(",");
                }
                if(iC) printf("C%d",iC);
                if(iV == 0){
                    printf(");¥n");
                    continue;
                }else{
                    printf(",P%d);¥n",iV);
                }
            }
        }
```

リスト 3-7-8 景観シミュレータから保存した LSSG 形式のファイル

```
# (ip ver.2.09) 国土交通省版・景観シミュレータ 2.09
    GRP2 = GROUP();
    CONCAVE(ON);
    V00 = COORD(1,1.4,0);
    C00 = COLOR(0, 0, 1, 1);
    V01 = COORD(0,0,0);
    C01 = COLOR(1, 0, 0, 1);
    V02 = COORD(2,0,0);
    C02 = COLOR(0, 1, 0, 1);
    V03 = COORD(1.106,0.679,0);
    C04 = COLOR(0.2045, 0.3105, 0.485, 1);
    V04 = COORD(1.21,0.33,0);
    C05 = COLOR(0.277143, 0.487143, 0.235714, 1);
    V05 = COORD(0.733,0.306,0);
    C06 = COLOR(0.524214, 0.257214, 0.218571, 1);
    P00 = VERTEX(V00, , , C00, V03);
    P01 = VERTEX(V01, , , C01);
    P02 = VERTEX(V02, , , C02);
    P03 = VERTEX(V03, , , C04, V00);
    P04 = VERTEX(V04, , , C05);
    P05 = VERTEX(V05, , , C06);
    F00 = FACE(P00, P01, P02, P00, P03, P04
        , P05, P03);
    N08 = NORMAL(0, 0, 1);
    FACE_NORMAL(F00, N08);
    GROUP_FACE(GRP2, F00);
```

リスト 3-7-9 LSSG 形式を定義したメタファイルによる出力結果

```
#入力ファイルは指定されていないことを確認
CONCAVE(ON);
#outvirtual.cmm(140531)
N1=NORMAL(0.000000,0.000000,1.000000);
C1=COLOR(0.000000,0.000000,1.000000,1.000000);
C2=COLOR(1.000000,0.000000,0.000000,1.000000);
C3=COLOR(0.000000,1.000000,0.000000,1.000000);
C4=COLOR(0.204500,0.310500,0.485000,1.000000);
C5=COLOR(0.277143,0.487143,0.235714,1.000000);
C6=COLOR(0.524214,0.257214,0.218571,1.000000);
P1=COORD(1.000000,1.400000,0.000000);
P2=COORD(0.000000,0.000000,0.000000);
P3=COORD(2.000000,0.000000,0.000000);
P4=COORD(1.106000,0.679000,0.000000);
P5=COORD(1.210000,0.330000,0.000000);
P6=COORD(0.733000,0.306000,0.000000);
G0=GROUP();
G1=GROUP();
V1=VERTEX(P1,,,C1);
V2=VERTEX(P2,,,C2);
V3=VERTEX(P3,,,C3);
V4=VERTEX(P1,,,C1,P4);
V5=VERTEX(P4,,,C4);
V6=VERTEX(P5,,,C5);
V7=VERTEX(P6,,,C6);
V8=VERTEX(P4,,,C4,P1);
F1=FACE(V1,V2,V3,V4,V5,V6,V7,V8);
FACE_NORMAL(F1,N1);
GROUP_FACE(G1,F1);
L1=LINK(G0,G1);
LINK_XFORM(L1,LOAD,MATRIX,1.000000,0.000000,0.000000,0.000000,0.000000,1.000000,0.000000,0.000000,0.
000000,0.000000,1.000000,0.000000,0.000000,0.000000,0.000000,1.000000);
CONCAVE(OFF);
```

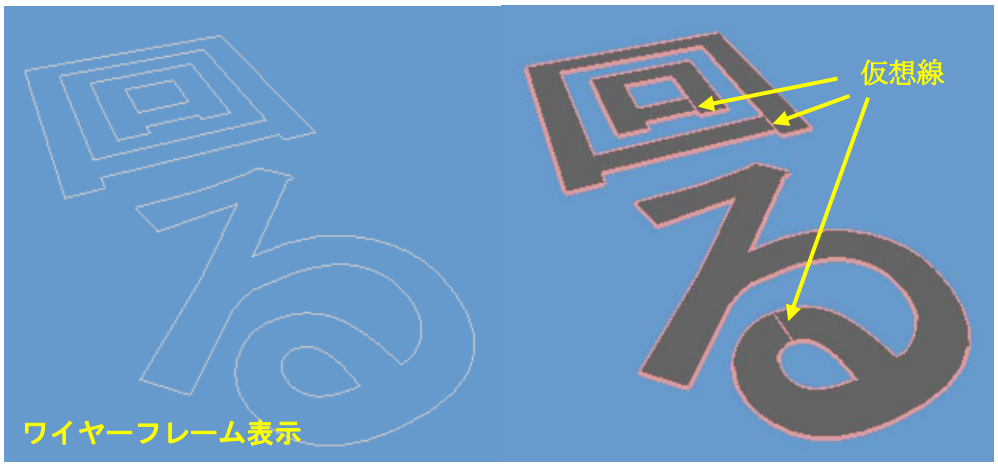


図 3-7-2：フォントファイルから生成した穴あき図形

（５） LINK の出力処理のためのライブラリ関数

① 関数の仕様と用法 cci_sqo, cci_dml における処理

シーングラフ型のデータにおいては、一つのオブジェクト（シンボル等とも呼ばれる）を、位置やスケールを変えて複数回使用することができる。例えば、４本の足と天板から成るテーブルを３０脚配置された教室を例に取ると、実際に形状が定義されているオブジェクトは天板と足の２個である。リンクの数は、教室に３０脚を配置する３０と、一つのテーブルを定義する５、合計３５である。表示されるオブジェクトは、天板３０と、足１２０の合計１５０である。リンクは、３５のリンクに順次アクセスし、カーソル・リンクのそれぞれに定義された位置情報等を取り出すことが目的である。

（メタファイルの書法については４－２で解説した）。

L_()以下の実装は、以下のような呼び出しを想定して行った。

まず、L_()関数を用いて、全てのリンクにアクセスする。

int Lp()関数で親グループの Gid を得る

int Lc()関数で子グループの Gid を得る

② 移動・スケール・回転の取得

カーソル・リンクに関して、位置情報を取得するために、以下の関数を用意した。

float Lm()関数を 16 回呼び出して、マトリクスの成分にアクセスする

int Li() 単位マトリクスなら１、それ以外ならゼロ

quat Lt() 平行移動成分を返す

quat Lr() 回転成分を返す（引数に PRE または POST を指定すると、前または後の回転を、また引数がない場合には二つの回転の合成を返す）

quat Ls() スケール成分を返す。

景観シミュレータ(1996)において未解決の問題であったが、仮想コンバータにおける四元数とテンソルの数学的な検討(4-4)に基づいて、任意の回転マトリクスを二つの回転と一つ

のスケールに分解し M(Qs,Qr,Qt)と表現する計算処理を新たに実装した。

（６）ディスプレイ・リストの取得

・ルートグループ（上方リンクがない）も表示の対象とする

・下方リンクを N 個有するグループが M 個の上方リンクを有する場合、
リンクリストでは高々M+N 個のリンクを処理すれば良いが、
ディスプレイ・リストでは少なくとも M×N 個の処理を行う必要がある。

int D_() ディスプレイ・リストを作成し、表示グループの ID を順次返す
表示グループにカーソルを当てる
終了するとゼロを返す

int Dlevel() 現在の選択グループの階層の深さを表す
ルートグループなら 0
カーソルグループがなければ-1

int Dfirst() 表示グループに対応する実体グループに最初にカーソルが当たった時 1
実体グループの内容を出力する必要がある場合に使用する

D_()ループが実行されている間は、頂点座標と法線ベクトルの意味が異なる
Sx(),Sy(),Sz(),Nx(),Ny(),Nz()は、グローバル座標における値とする
DL 配列が存在している時、Vcoord 関数が実行された時点でグローバル座標をスタティック変数 Gx 等に作成
Lm()については未定義

３－８．仮想コンバータのセキュリティ

開発段階において、仮想コンバータが正常に動作するところまで実現すれば、データファイルの長期保存という当初の目的は達成されたことになる。しかしながら、本章を執筆する 2015 年時点では、情報システムのセキュリティが社会問題となっている。

本研究を実施していた 2010~15 年当時、PC をネットワークに常時接続する使用環境が普及し、悪意あるソフトウェアからシステムやデータを保護するためのセキュリティ対策の重要性が認識され、対策が工夫されていた。

（１）セキュリティ対応のための C 関数の仕様変更への対応

C 言語で頻繁に使用する固定長配列の限界を超えたアクセスは、例えば想定外の長いファイル名を有するファイルや、ファイル名の一部にプログラムを含むようなファイルによる攻撃の侵入経路となりうることから、従来広く使われていた C 言語の入出力関数や文字列処理関数の脆弱性が指摘され、VS2005 からセキュリティを高めた仕様の関数を使用することが勧奨され、従来の標準 C の関数の使用に対して警告が表示された。一例を挙げると、

```
char buffer[30];
fscanf(fp,"%s",buffer);
```

というコードは、データファイル中に長い行が存在すると簡単にバッファ・オーバーフロ

ーを起こす。これに対処して、

fscanf_s(fp,"%s",buffer,30);

という代替関数を用い、固定長配列の限界を超えた入力をブロックする方法である。この場合、エラー終了となる。

しかしながら、このような代替関数への変更を行ったソースコードは、異なる開発環境（例えば、後述の Android アプリのための C コンパイラ NDK）ではリンクエラーを起こす。移植のためには、再び脆弱な fscanf 関数に戻さなければならない。

このため、メタファイルとデータファイルを扱う仮想コンバータの基幹部分においては、fscanf_s という代替関数を使用せず、例えば上記のような場合には、

fscanf(fp,"%30s",buffer);

のようにしてバッファ・オーバーフローを予防しつつ可搬性を確保する。

一方、メタファイル作成のために用意した scanf 関数においては、固定長のバッファに可変長のデータを受け取るような処理は記述できないようになっている。

（2）メタファイルが呼び出すライブラリ関数のセキュリティ

例えば、保存されたデータファイルとメタファイルを利活用する時点で、これを WEB や媒体を通じて配布するような状況が考えられる。その時に、悪意のある偽のメタファイルやデータファイルが配布された場合に、これがユーザの形態端末や WEB サーバから個人情報等を引き出して他者に送信するような動作が行えないようにあらかじめ機能を制約しておく、ということも重要な配慮事項となる。

本処理系においては、以下のような配慮を行っている。

1）コンパイルされた実行形式を格納する領域は、処理系が固定長配列としてメモリ上に確保した領域のみとし、長いメタファイルを処理した結果がこれを超える場合にはエラーとした。

2）メタファイルのコマンド（ライブラリ関数）でアクセス可能なメモリ空間は、インタープリタ処理系が固定長配列として用意したメモリ空間のみとし、OS や、その他の別の処理系が使用中のメモリにはアクセスできないようにした。メタファイルの中で定義する変数や配列は全て、この固定長配列の一部として割り当てる。

3）メタファイルのコマンド（ライブラリ関数）では、メモリブロックを取得するための C 言語における malloc-free 系のコマンドは提供しない。メタファイルで宣言する配列は全て固定長として宣言する。メタファイルが添付するデータファイルが特定であることから、自由に伸長可能な配列をメタファイルの記述の中で用意する必要はなく、十分な長さの固定長配列を使用することにより対応できる。

4）メタファイルが入力のためにアクセスするファイルは、データファイルのみとし、メタファイルをコンパイルして生成した実行形式をインタープリタが処理開始する時点では既に固定的なアクセス先としてオープンされているようにした。メタファイルのコマンド（ライブラリ関数）が新たな任意名称のファイルをオープンしてアクセスすることはでき

ない。
５）メタファイルが出力先とするファイルは、利活用形態別にあらかじめ定められた出力ファイルとログファイルのみとし、メタファイルのコマンド（ライブラリ関数）で新たな任意名称のファイルをオープンして書き込み動作を行うことはできない。
６）サーバ上で動作する VC-4D においては、悪意のある攻撃的なメタファイルを添付したデータファイルが投稿されることが考えられる。例えば、意味のない無限に多くの頂点や面を有する図形データを、エンドレスループで定義するような処理が定義されている場合、データベース上に巨大なテーブルを生成することも可能である。また同様に、同じ文字列を反復するだけの無限に長いデータファイルやログファイルを出力するような処理も記述可能である。また、単に終わりのないエンドレスループを定義するだけで、処理系は無限の時間を浪費する可能性がある。このような場合、データファイルへのアクセスポイントが一定時間経過しても変化しない場合に処理を打ち切るような処理を行っているが、まだ完璧とは言えず、今後工夫する余地がある。

（３）ライブラリ関数の、利活用処理系における実装

個別の処理系に関しては以下の通りである。

１）入出力

VC-1C において、purintf, logf の出力が行われた場合：ログファイルに追記

S_()などのアクセス系コマンドが実行された場合：ダミー関数で何もしない

VC-2V において、

ファイル出力時にデータ構築系のコマンド GROUP()等が実行された場合

→①メモリ空間にデータが構築されてしまう

②出力ファイルはオープンされるので、空のファイルが生成する

ファイル入力時にファイル出力系のコマンド(printf 等）が実行された場合

→ログファイルに出力が行われ、入力データファイルに影響はない

VC-4D において、

ファイル入力時にファイル出力系のコマンド(printf 等）が実行された場合→ログファイルに出力を行う。

ファイル出力時にデータ構築系のコマンド GROUP()等が実行された場合

→現段階（2015 年 10 月）では、実行されてしまう。

今後の対策として、コンパイラのエラーチェックを強化し、LSSG 系コマンドを、ファイル出力時にはコンパイル・エラーとする方法が考えられる。

２）入出力の選択をコンパイラに伝えるための方法

２－１）VC-2V.dll の場合、

2Vwnd.cpp の中の、OnBnClickedOk()関数において、

入出力選択ボタン（ボタンの外見を有するチェックボックス）の状態を b_IO フラグで見る。

b_IO が０なら、入力処理を行い、データファイルから読み出して DML に出力する。
b_IO が１なら、出力処理を行い、DML から読み出してデータファイルに出力する。

２－２）VC-4D.exe の場合、コマンドラインの第三引数で,VC-4D_exe.cpp の main 関数が、第 4 引数を IO として受け取る。upload 関数か download 関数かを選択する。
IO が０なら、ダウンロード処理を行い、SQL から読み出してデータファイルに書き込む。
IO が１なら、アップロード処理を行い、データファイルから読み出して SQL に書き込む。

３）compile 関数に対して、入出力を伝えるための方法

データベースを改変する処理、具体的には記憶系のライブラリ関数が、利活用のためのメタファイルから呼び出されると、不適切な結果となる。このようなチェックは、メタファイルの実行段階でブロックすることができるが、コンパイル段階で事前にエラーとして処理し、実行段階に移行しない方が確実である。

cci_pars.c に、ReadOnly グローバル変数を用意し、値が非ゼロにおいて記憶系のライブラリ関数が使用されるとコンパイル・エラーとした（statement 関数および factor 関数の内部での処理)。

４）サービスによる処理の区分

VC-4D のようにアップロードとダウンロードのサービスを提供する処理系においては、ユーザが保存データとメタファイルをアップロードする場合と、新たな形式を記述したメタファイルを添付してダウンロードする場合で、以下のように処理を区分している。

表 3-8-1 セキュリティ対策

処理内容	アップロード時	ダウンロード時
printf 関数	ログファイルに出力	データファイルに出力
logf 関数	ログファイルに出力	ログファイルに出力
scanf 関数	データファイルから入力	エラー
COORD 関数等	SQL データベースに入力	コンパイル・エラー
Sq()関数等	無意味な結果	SQL データベースから取得

これにより、不本意に保存データが上書きされ損傷されるリスクは回避されていると考えられるが、メタファイルの実行時に長時間応答がなくなる場合や、無意味なファイルが作成され返送される可能性が残されている。

（４）書式文字列の検査

printf、logf、scanf 関数においては、書式文字列と引数が不整合のまま実行されると、スタックを介して不正なコードが実行される可能性がある。

本処理系においては、プログラム領域とメモリ領域とスタック領域を分離し、プログラム領域以外をプログラム・カウンタが参照できないようにしている。

引数の数は０か１としているため、書式文字列の中で「%数値型」により参照する引数

は高々一つである。複数の引数が参照された場合には、コンパイル・エラーとして処理した。

標準 C 言語においては、printf(“%*d”,width,value); の形で引数により幅を指定するが、本処理系ではこの形を許容していない。

標準 C 言語において、printf(“・・・%n・・・”, &location); の形で printf の引数（アドレス）に値（出力済文字数）を代入する脆弱な方法（想定外の書式文字列により読み出し用に用意したデータが書き換えられる）は、printf(“・・・%n・・・”, location);の表記法で利用可能とした。

書式文字列としては、標準 C 言語においてはポインタ渡しで任意のものを使用可能であるが、本処理系においてはメタファイルの中でリテラル文字列として固定的に定義した文字列のみを使用している。

（５）サーバのセキュリティ

サーバのセキュリティが社会的な問題になっている。様々な攻撃パターンに対する一般的な対策の勧奨と、実施状況の報告などが求められている。これらの報告が簡単なメール添付で行われていることが多く、サーバの構成やセキュリティ対策を記載した報告様式のデータにパスワードを付すような対策を行っている。

一方、悪意のある攻撃の中には、無意味なアクセスを集中させるようなパターンがある。これは、感染マシンからのプログラムによる攻撃である場合が多いため、画面に表示した簡単なパスワードでガードするだけでもある程度は防ぐことができる。重要な点は、攻撃を検知した時に、サーバ側に生じている現象を分析し、個別に対策を考える努力を行うことではないかと考えている。

３－９．まとめ

以上解説したように、新しい処理系を作成に当たり、以下のような点を考慮しながら作業を進めることは効果があったと考える。

（１）段階的な前進

①既存のデバッグの進んだライブラリ（ソースコード）の活用

a コンパイラ処理系を、シンプルな教材用処理系から拡充して開発

b 景観シミュレータの外部関数、プラグイン DLL として実装することにより、基幹部分動作確認を既存の処理系を用いて実施

c 景観シミュレータの既存ライブラリ（DML、g3DRL）のデバッグが進んだソースコードを利活用処理系に最大限活用した

②異なるプラットフォーム上での開発における技術要素の分解

新しい処理系を実現する上で超えなければならないハードルを、異なる要素に分解し、仮設的な処理系を試作しながら一つずつ段階的に解決すること

a VC-3M のためのエミュレータ

Windows システム上でのエミュレータを開発して動作検証した上で、JAVA 言語による

Android 上の API をアウトソースにより開発した。

b VC-4D のためのエミュレータ

基幹部分に加えて、WEB サーバや SQL データベース等の複数技術を併用しているため、SQL 上での検証の上、WEB サーバへの実装を分けてアウトソースにより開発した。

③役割の終わった仮設的処理系の活用

通過点として作成した仮設的プログラムも、データ検証ツール等としての用途を見出すことができる。

a VC-1C、VC-2V は、景観シミュレーション・システムの機能拡張として活用

b VC-4D は、SQL サーバの検証ツールとして活用

（２）イノベーション

①VC-1C

a プラットフォームに依存しないコンバータ

従来の、景観シミュレータにおける各種ファイル形式のデータを入力するための外部関数は、C 言語のソースコードとして記述され、Windows 上の開発環境によりビルドされて Windows 上の実行形式として提供されていた。この実行形式は、UNIX 等の別の処理系の上では実行することができず、また修正を行うためには、開発環境に戻らなければならなかった。

b コンバータのソースコードとしてのメタファイル

VC-1C は、実行時に、まずメタファイルをコンパイルし、生成した実行形式をインタープリタが実行することにより、入力するデータファイルを変換する。したがって、メタファイルは、様々な処理系に共通で使用することができ、テキストエディタが使用できる環境においては、修正したり増補したりできる。

②VC-2V

a 中間ファイルなどを経由しないライブラリ関数の実装形態

実行時にライブラリ関数からメモリ上の構造体を構築する DML ライブラリ関数を直接実行し、直ちに表示を行う。外部ファイルを経由しない

b 出力系のライブラリ関数の実現

DML ライブラリ関数を通じ、メモリ空間から形状構成要素を取り出して、異なるファイル形式でのファイル出力処理を行うことができる。

③VC-3M

a Android 上での基幹部分の可搬性の検証

b 背面カメラ、GPS、磁気、加速度センサを用いたリアルタイム写真合成

c 簡単な操作による三次元的なキャリブレーション（「足によるキャリブレーション」）

d シャッター操作による現場活動の中での位置情報と画像の記録保存

④VC-4D

a サーバ OS 上での基幹部分の可搬性の検証

b SQL データベースへのデータの完全な展開（電源を切っても消えない）
c SQL データベースからのデータの取り出しと、任意形式のデータの再構成・出力

数年間の継続期間をもって運用されるサービスとして見た場合、処理中のデータをデータベースが管理するファイルとして保管しているこのシステムは停電等に対して有効である。また、バックアップ等の様々なデータベース関連の機能を活用することができる。

しかしながら、百年単位での長期保存という観点から見ると、WEB サーバの OS のバージョンアップへの対応を継続的に求められる点や、物理的にデータを保管しているハードディスク装置（RAID により、クラッシュに対しては対応できるが、その場合にも交換する体制を維持する必要がある）のメンテナンスが必要であり、制度的な裏付けがなければ、長期保存の目的を実現することはできない。

このような意味で、VC-4D の位置づけは、「長期保存の手段」ではなく、「利活用の手段」の一形態であり、保存されたデータをユーザーがメタファイルで指定した任意形式のファイルに変換するサービスとして位置付けることができる。