User's Guide of Polylib

Polygon Management Library

Ver. 3.0.0

Advanced Visualization Research Team
Advanced Institute for Computational Science
RIKEN

7-1-26, Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo, 650-0047, Japan

http://www.aics.riken.jp/

September 2013

Release

Edition	3.0.0	2013-09-16
	2.6.8	2013-07-20
	2.6	2013-06-24
	2.2	2012-11-27
	2.1	2012-04-31
	2.0.0	2010-06-30
	1.0.0	2010-02-26

COPYRIGHT

目次

第 1 章

Polylib の概要

本ユーザーガイドでは，ポリゴン要素を管理するライブラリについて，その機能と利用方法を説明します．

1.1　概要

Polygon Management Library (以下，Polylib) は，ポリゴンデータを保持・管理するためのクラスライブラリです．
クラスライブラリの詳細については，「リファレンスマニュアル」を参照してください．

1.2　Polylib の機能

Polylib の主な機能を以下に列挙します．

- 初期化ファイルを利用した STL ファイルの読み込み (Ver.2.0.0 追加機能)
  - 初期化ファイルに記述されたポリゴングループ階層構造，および STL ファイルを読み込み，オンメモリに管理します．
- ポリゴンデータのグルーピング
  - 読み込んだポリゴンデータを STL ファイル単位にグルーピングして管理します．複数のポリゴングループをまとめたグループを作成するなどの，階層的なグループ管理が可能です．グルーピングの設定は初期化ファイルに記述します．
- ポリゴンデータの検索
  - 読み込み済のポリゴンデータについて，指定された領域内に含まれるポリゴンを検索します．検索対象のポリゴンデータは，Polylib 管理下のポリゴン全体や，任意のポリゴングループなどの指定が可能です．
- 並列計算環境下でのポリゴンデータの分散
  - マスターランクで読み込んだポリゴンデータを，領域分割情報に基づき各ランクに配信します．
- ポリゴンデータの移動 (Ver.2.0.0 追加機能)
  - 時間発展計算実行中に，ユーザプログラム側で定義されたポリゴン頂点座標移動関数に基づきポリゴンデータの移動を行います．
  - 並列計算環境下では，隣接ランク領域へ移動したポリゴン情報をランク間でやりとりします．
- ポリゴンデータの一時保存 (Ver.2.0.0 追加機能)
  - 一時保存処理を呼び出した時点のグループ階層構造情報，およびポリゴンデータを，ファイルに保存します．
  - 並列計算環境下でのファイル保存は，マスターノードへの集約保存と，各ランクでの分割保存が選択できます．
- ポリゴンデータの再読み込み (Ver.2.0.0 追加機能)
  - 一時保存処理により保存されたファイルを再読み込みします．
  - 並列計算環境下での再読み込みは，マスターノードでの集約読み込みと，各ランクでの分割読み込みが選択できます．
- obj ファイルの読み込み/書き出し (Ver.3.0.0 追加機能)
  - STL ファイルと同様に, obj 形式のファイルを読み込み/書き出します．
- ポリゴンデータの省メモリ化 (Ver.3.0.0 追加機能)
  - ポリゴンデータの内部構造を変更し, 頂点座標リストと三角形のリストからなる構造に変更しました．
  - 構造変更に伴い, 頂点座標リストを登録する際, 同一頂点かどうかを判断して登録します．
- ポリゴンデータの実数型の選択 (Ver.3.0.0 追加機能)

  - C++ のテンプレートの機能により, ポリゴンデータの実数型が, 選択できます.

- autoconf & automake 対応　(Ver.3.0.0 追加機能)
  - configure スクリプト作成に autotools (autoconf automake) を用いました.

1.3　動作環境

以下の環境下で動作確認済です.

- 開発 OS：CentOS5.4 (64bit)
- 開発言語：C++
- 開発コンパイラ：g++ 4.1.2, Intel C++ Compiler 11.1
- 並列ライブラリ：OpenMPI 1.3.2
- TextParser ライブラリ：TextParser Version1.1

1.4　ライセンス

Polylib は，バージョンにより以下の 2 つのライセンスの適用となります.

- Version 1.0 ~ 2.x
  理化学研究所 VCAD ライセンス `http://vcad-hpsv.riken.jp/`
- Version 3.0 以降
  修正 BSD ライセンス（２条項）

1.5　リポジトリ

公開リポジトリは以下になります.

`https://github.com/avr-aics-riken/Cutlib`

第２章

プログラム構造

本章では，Polylib プログラムのクラス構成，データ構造，入出力ファイルなどについて説明します．

2.1　クラス構成

以下に Polylib のクラス図概要を示します．

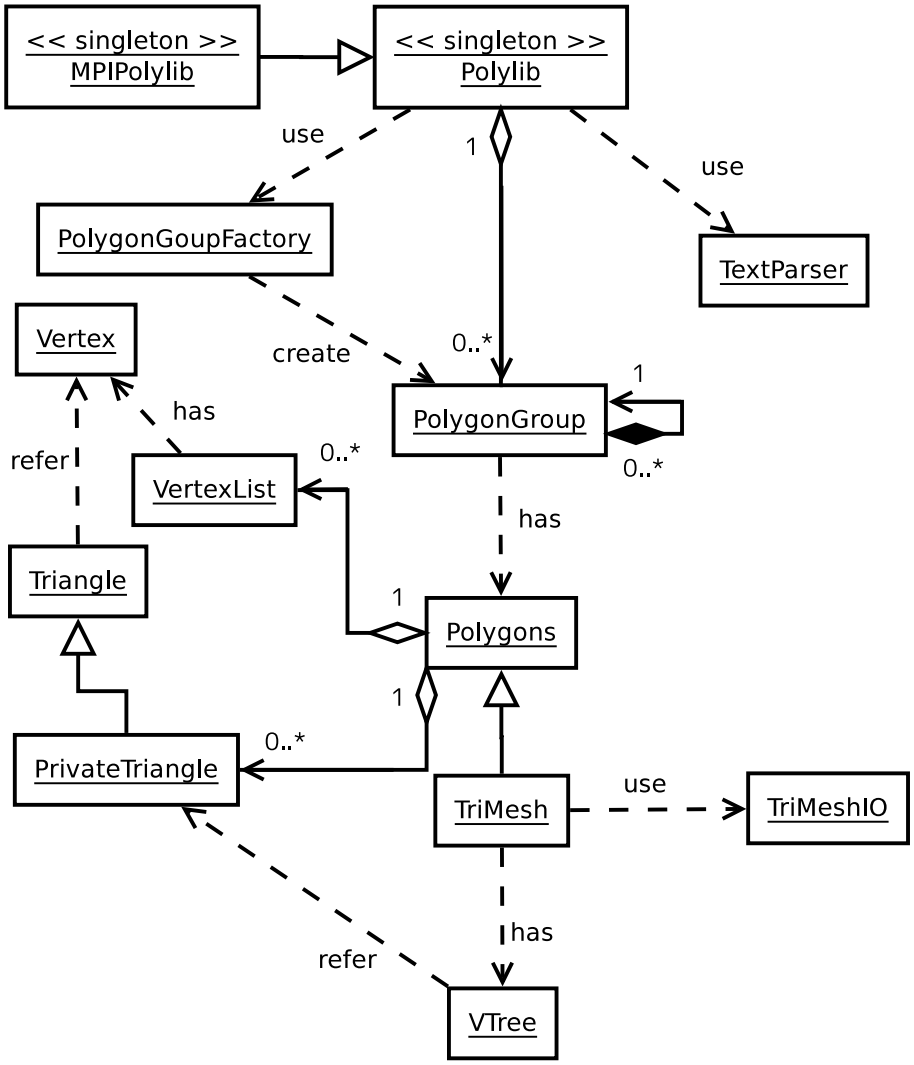


図 2.1　主要クラス図

各クラスの説明は表 2.1 の通りです．

クラス名	概要
Polylib	単一プロセス版の Polylib 本体です．本クラス内部にポリゴングループ階層構造，三角形ポリゴン情報を保持します． 本クラスは singleton クラスであり，１プロセス内に１インスタンスのみ存在します．
MPIPolylib	MPI 版の Polylib 本体です．Polylib クラスを継承しており，各メソッドは並列処理を前提とした内容にオーバーライドされています．
TextParser	Polylib 初期化ファイルを load/save するためのユーティリティクラスです．
PolygonGroup	三角形ポリゴン集合をグルーピングして管理するためのクラスです．ポリゴングループ同士の階層的な包含関係も表現します．STL ファイルから読み込んだ三角形ポリゴン集合を本クラスの属性である Polygons クラスインスタンスで管理します．
PolygonGroupFactory	Polylib 初期化ファイルに記述されたポリゴングループ階層構造情報を元に PolygonGroup クラスとその継承クラスインスタンスを生成する処理を行うクラスです．
Polygons	三角形ポリゴン情報を保持するコンテナクラスであり，ポリゴン検索のための検索メソッドを定義した抽象クラスです．
TriMesh	KD 木による三角形ポリゴン検索アルゴリズムを実装した Polygons クラスの導出クラスです．
TriMeshIO	STL ファイルを load/save するためのユーティリティクラスです．
Vtree	KD 木データ構造クラスです．
Triangle	三角形ポリゴンクラスです． 3 頂点座標へのポインタ，法線ベクトル，面積を保持します．
PrivateTriangle	Polylib 内部で利用する三角形 id などを追加した Triangle クラスの導出クラスです．
Vertex	Polylib 内部で利用する頂点座標のクラスです．Triangle クラスの頂点座標へのポインタは，このクラスのインスタンスへのポインタです．
VertexList	Polylib 内部で利用する頂点座標 Vertex クラスインスタンスを保持するクラスです． 頂点を追加しながら，KD 木を作成します．
VertKDT	Polylib 内部で VertexList クラスの中身の KD 木クラス．

表 2.1　主要クラス一覧

2.2　データ構造

2.2.1　ポリゴングループの管理構造

ポリゴングループの保持・管理は，Polylib クラスのメンバ変数である，std::vector<PolygonGroup> m_pg_list で行います．この vector コンテナはポリゴングループ階層構造の最上位の PolygonGroup インスタンスを保持します．PolygonGroup クラスは，メンバ変数 std::vector<PolygonGroup*> m_children により，PolygonGroup インスタンス同士の階層構造を保持します．

PolygonGroup は複数の子要素を持つことができますが，親要素は最大で１つです．（親要素数がゼロならば最上位の PolygonGroup です）また，階層構造最下位の PolygonGroup のみ，STL/OBJ ファイルから読み込んだ三角形ポリゴン情報を保持します．

これらの階層構造については，ユーザが作成する Polylib 初期化ファイルに記述されており，Polylib は初期化処理時にこのファイルを読み込むことで，グループ階層構造，およびポリゴンデータをオンメモリに構築し，管理します．

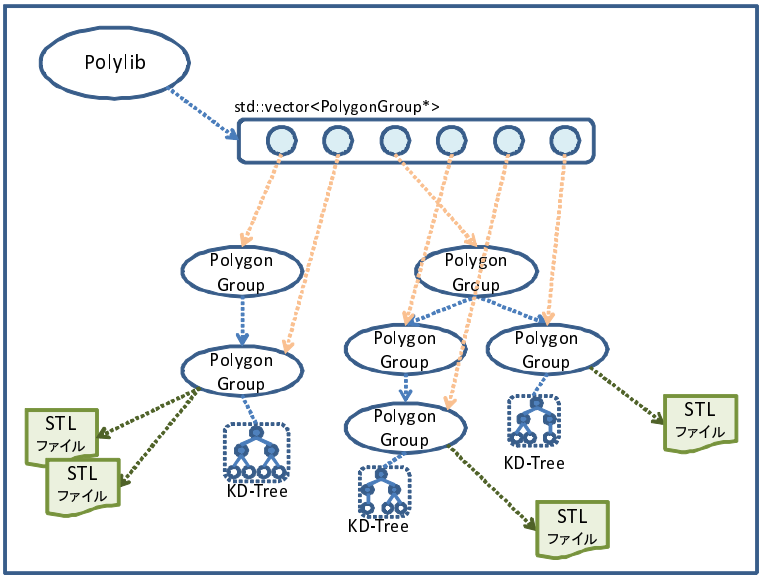


図 2.2　ポリゴングループの管理構造

2.2.2　ポリゴンデータの管理構造

三角形ポリゴンの保持・管理は，Polygons クラスのメンバ変数である，std::vector<PrivateTriangle*> *m_tri_list と VertexList* m_vertex_list で行います．この vector コンテナには，STL/OBJ ファイルから読み込んだ三角形ポリゴン情報を PrivateTriangle クラスと Vertex クラスのインスタンスとして生成して登録します．

また，三角形ポリゴンを高速に検索するために，三角形ポリゴンのバウンディングボックスベースで包含判定を行う KD-Tree 構造を実装した TriMesh クラスを Polygons クラスの導出クラスとして利用しています．KD 木のリーフ要素である三角形ポリゴン情報は，Polygons::m_tri_list コンテナに格納された各 PrivateTriangle インスタンスへのポインタです．

Polylib::search() などのポリゴン検索メソッドは，この KD 木を検索して得られた PrivateTriangle 型ポインタの集合を Triangle 型ポインタに cast して std::vector に詰めて返却します．検索メソッド呼び出し側では，返却結果の std::vector インスタンスを使用後に delete する必要がありますが，その要素である Triangle 型ポインタが指し示すインスタンスを消去してはいけません．

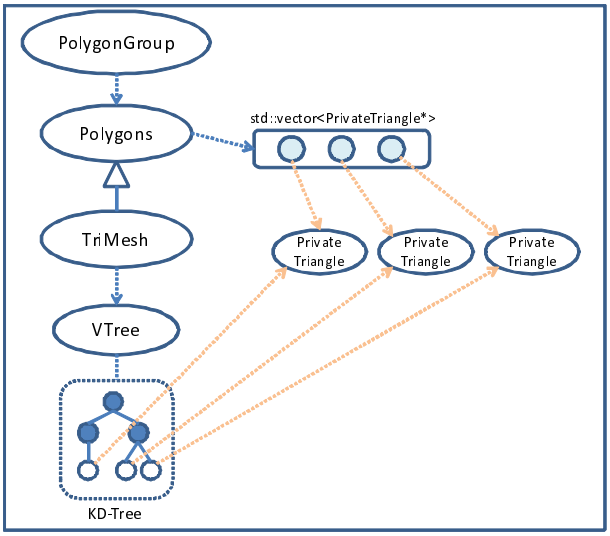


図 2.3　ポリゴンデータの管理構造

2.3　入力ファイル

2.3.1　Polylib 初期化ファイル

Polylib 初期化ファイルは，TextParser ライブラリでパース可能な形式のテキストファイルです．TextPaser 記述方式のについては TextParser ライブラリのマニュアルを参照してください．

Polylib 初期化ファイルは，デフォルトではカレントディレクトリに存在する polylib_config.tpp を読み込みますが，任意のファイル名をデータロード API 引数で指定可能です．

記述例を以下に示します．この例では，STL ファイルをロードしていますが，Obj ファイルも同様に記述します．

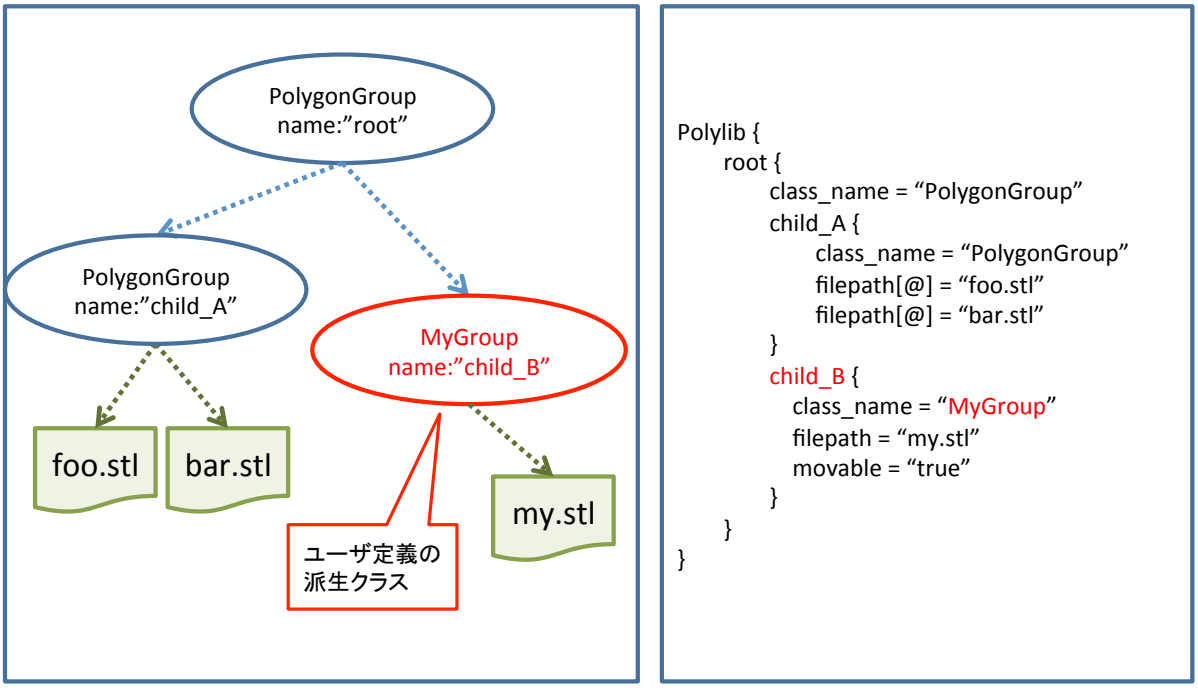


図 2.4　初期化ファイルの例

要素を {} で囲むグループラベルについて以下に説明します．

ラベル名	説明
Polylib	Polylib 初期化ファイルにおけるルートラベルです．
polygon_group_name	ポリゴングループ名称を記述します．子要素としてポリゴングループを階層的に持つことができます．名称は同じ階層内の同一レベルにおいて一意でなければなりません．

表 2.2　グループラベルの説明

ラベル=値形式のラベルについて以下に説明します.

ラベル	説明
class_name	生成するポリゴングループのクラス名．PolygonGroup クラスインスタンスを生成する場合は，”PolygonGroup” を指定．ユーザ定義の派生クラスインスタンスを生成する場合は，その派生クラスでオーバーライドしたメソッド get_class_name() が返す名称を指定する.
filepath	当該ポリゴングループにひもづく STL/OBJ ファイルのパス．カレントディレクトリからの相対パスまたは絶対パスで指定する．階層最下位のポリゴングループでのみ指定が可能.
movable	当該ポリゴングループが move() メソッドにより移動するかどうかを指定する．value には”ture” もしくは”false” を指定する．階層最下位のポリゴングループかつ，当該ポリゴングループが move() メソッド実装済みの PolygonGroup 派生クラスインスタンスの場合のみ指定が有効.
id	ポリゴングループに付与可能な ID 番号．ID 番号の重複は許される．省略可能で省略時は 0 が設定される．境界条件 ID などに利用されることを想定しており, この値が設定されている場合, 当該ポリゴングループ配下の全 Triangle インスタンスのメンバ変数 m_exid に同値が設定される.
label	任意の文字列を指定可能．媒質指定などに利用することを想定.
type	任意の文字列を指定可能．境界条件名の指定などに利用することを想定.

表 2.3　ラベルの説明

上記のラベル=値形式以外にも，ユーザ定義のラベル=値が指定可能です．ユーザ定義ラベルの指定方法については，後述のサンプルモデルによるチュートリアルを参照してください.

2.3.2　STL ファイル

初期化ファイルで指定されたファイル名の STL ファイルを読み込みます．入力となる STL ファイルはアスキー形式，バイナリ形式いずれでもかまいません．拡張子は”stl”であれば形式を自動判別して読み込みます.

バイナリ形式 STL ファイルの場合，各三角形毎に存在する 2 バイト未使用領域を使ってユーザ定義値を設定することが可能です．値は int Triangle::m_exid に設定されます.

2.3.3　OBJ ファイル

OBJ 形式のファイルを読み込みます．OBJ 形式は本来アスキー形式のみです[*1]が，本ライブラリーでは大規模な形状モデルを扱うので，独自に定めたバイナリ形式も利用できます．出力については，3.1.5 を参照してください.

2.4　出力ファイル

本節では，Polylib::save() などのデータセーブ系 API で出力されたデータファイルについて説明します.

2.4.1　Polylib 初期化ファイル

データセーブ時のポリゴングループの階層構造と，保存した STL ファイル名を polylib 初期化ファイル形式で保存します．保存時のファイル命名規則は以下の通りです.

[*1] バイナリ形式もあるようですが，一般的ではありません.

```
polylib_config_{ユーザ指定文字列}.tpp
```

ユーザ指定文字列はデータセーブ系 API 引数で指定可能です．無指定の場合，保存時のタイムスタンプを yyyymmddHHmmss 形式で設定します．

2.4.2　ポリゴンファイル

データセーブ時のポリゴン情報を保存します．保存時のファイル命名規則は以下の通りです．

```
{ポリゴングループ名称フルパス}_{ユーザ指定文字列}.{stl|obj}
```

ポリゴングループ名称フルパスとは，階層最上位のポリゴングループ名称から，当該グループ名称までを'_' でつなげたものです．ユーザ指定文字列はデータセーブ系 API 引数で指定可能です．無指定の場合，保存時のタイムスタンプを yyyymmddHHmmss 形式で設定します．

ファイル拡張子はデータセーブ系 API で指定した保存ファイル形式に基づき設定されます．

MPIPolylib の場合，当該ポリゴングループについて自ランク内に保存すべき三角形ポリゴン情報が存在しない場合は STL ファイルは出力されません．

なお，データロード時に指定した polylib 初期化ファイルにおいて，複数の STL/OBJ ファイルを指定したポリゴングループについてデータセーブを行うと，ポリゴン情報は１つの STL/OBJ ファイルに纏めて出力されます．

2.4.3　三角形 ID ファイル

MPIPolylib::save_parallel() を利用したデータセーブ時に STL ファイル内の三角形ポリゴン並びに対応した三角形 ID を保存します．

三角形 ID とは MPIPolylib が管理する全三角形ポリゴン情報に対し，その三角形が所属するポリゴングループ内で一意となるような int 型の ID 番号で，MPIPolylib 内部でデータロード時に自動的に付与されます．

並列動作時に各ランク毎にデータセーブする場合，各計算領域ガイドセル部分の三角形情報は重複して保存されるため，データを再ロードした際に重複三角形を判別するために三角形 ID 情報ファイルが必要となります．

保存時のファイル命名規則は以下の通りです．

```
{ポリゴングループ名称フルパス}_{ユーザ指定文字列}.id
```

ポリゴングループ名称フルパスとは，階層最上位のポリゴングループ名称から，当該グループ名称までを'_' でつなげたものです．

ユーザ指定文字列はデータセーブ系 API 引数で指定可能です．無指定の場合，保存時のタイムスタンプを yyyymmddHHmmss 形式で設定します．

なお，当該ポリゴングループについて自ランク内に保存すべき三角形ポリゴン情報が存在しない場合は三角形 ID ファイルは出力されません．

第 3 章

API 利用方法

本章では，Polylib の主な API の利用方法を説明します．

3.1　単一プロセス版の主な API の利用方法

本節では単一プロセス版 Polylib の主な API 利用方法を，API 呼び出し順に沿って説明します．
単一プロセス版 Polylib の API を利用する手順は下図の通りです．

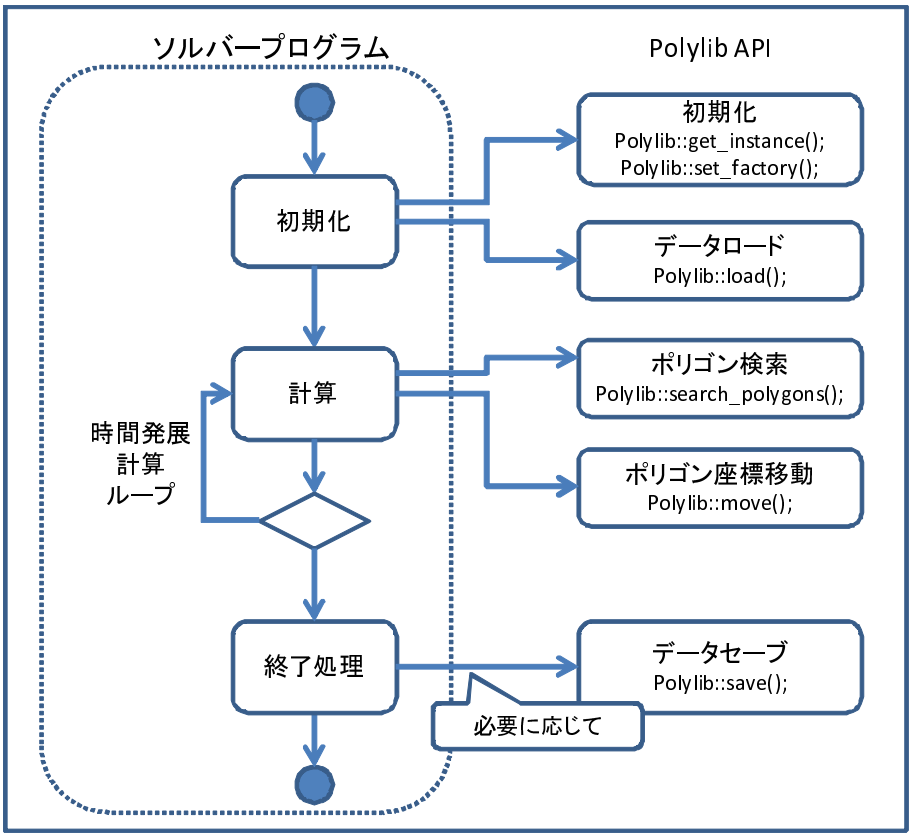


図 3.1　API 利用手順 (単一プロセス版)

3.1.1　初期化 API

Polylib インスタンスの生成

```
static Polylib<PL_REAL>* Polylib<PL_REAL>::get_instance();
```

Polylib は singleton クラスであるため，ユーザプログラム中から Polylib インスタンスを明示的に生成する必要はありません．Static メソッドである Polylib::get_instance() を呼び出すことで，プロセス内唯一の Polylib インスタンスが返却されます．

また，Polylib::get_instance() により返却されたインスタンスをユーザプログラムで明示的に消去する必要はありません．インスタンスはプロセス終了時に自動的に消去されます．

Polylib 全体で用いる実数の型を PL_REAL で指定します. 以下の説明でも PL_REAL は, その実数型を示すものとします.

PolygonGroup 派生クラスインスタンス生成ファクトリークラスの設定

```
void Polylib<PL_REAL>::set_factory(
        PolygonGroupFactory<PL_REAL> *factory
);
```

ユーザが定義する PolylgonGroup 派生クラスを利用する場合，その派生クラスインスタンスの生成方法を記述した PolygonGroupFactory 派生クラスインスタンスを，本メソッドを利用して Polylib に登録する必要があります.

PolylgonGroup 派生クラスを利用しないのであれば，本 API を呼び出す必要はありません.

PolygonGroup 派生クラスの具体的な利用法については後述のチュートリアルを参照してください.

3.1.2　データロード API

```
POLYLIB_STAT Polylib<PL_REAL>::load(
        std::string config_name = "polylib_config.tpp"
);
```

引数 config_name で指定された Polylib 初期化ファイルを読み込み，そこに記述された内容に基づきポリゴングループ階層構造をオンメモリに生成します．そして最下層ポリゴングループに指定された STL ファイルを読み込み，三角形ポリゴン情報のインスタンスの生成と，ポリゴン検索用 KD 木の生成を行います.

引数 config_name が指定されなかった場合，デフォルト初期化ファイル名である”polylib_config.tp”をカレントディレクトリから読み込みます.

3.1.3　検索 API

```
std::vector<Triangle<PL_REAL>*>* Polylib<PL_REAL>::search_polygons(
        std::string        group_name,
        Vec3<PL_REAL>                min_pos,
        Vec3<PL_REAL>                max_pos,
        Bool                 every
)const;
```

ポリゴングループ名 group_name で指定されたグループ階層構造下から，位置ベクトル min_pos と max_pos により指定される矩形領域に含まれる三角形ポリゴンを検索します.

引数 group_name はポリゴングループ名称フルパスで指定します．ポリゴングループ名称フルパスとは，階層最上位のポリゴングループ名称から，当該グループ名称までを’/’ でつなげたものです．たとえば，階層最上位のポリゴングループ名が”group_A”でその直下にある”group_B”内のポリゴンを検索する場合，引数 group_name に指定する文字列は以下の通りです.

```
"group_A/group_B"
```

引数 every の指定方法は以下の通りです.

- ture: 3 頂点が全て指定領域内に含まれる三角形を検索
- false: 一部でも指定領域と交差する三角形を検索

返却された std::vector インスタンスは，呼び出し側で消去する必要がありますが，その要素である Triangle 型ポインタの指し示す Triangle インスタンスを消去してはいけません.

指定点に最も近い三角形ポリゴンの検索

```
const Triangle<PL_REAL>* Polylib<PL_REAL>::search_nearest_polygon(
        std::string        group_name,
        Vec3f                pos
)const;
```

ポリゴングループ名 group_name で指定されたグループ階層構造下から，位置ベクトル pos により指定される点に最も近い三角形ポリゴンを検索します.

3.1.4　ポリゴン座標移動 API

```
POLYLIB_STAT Polylib<PL_REAL>::move(PolylibMoveParams& param);
```

Polylib 管理下にある move メソッド実装済の PolygonGroup 派生クラスに所属する三角形ポリゴンの座標を，move メソッドに実装された座標移動関数に基づきその頂点座標を変更します.

引数 param は，PolygonGroup 派生クラス move メソッドの引数に渡されます．PolylibMoveParams クラス定義は以下の通りです.

```
class PolylibMoveParams {
public:
        int m_current_step;         /*現在の計算ステップ番号*/
        int m_next_step;          /*移動後の計算ステップ番号*/
        double m_delta_t;          /*１計算ステップあたりの時間変異*/
};
```

move メソッドの実装方法の実際については，後述のチュートリアルを参照してください.

3.1.5　データセーブ API

```
POLYLIB_STAT Polylib<PL_REAL>::save(
        std::string        *p_fname,
```

```
                std::string         format,
                std::string         extend = ""
        );
```

本 API 呼び出し時点でのグループ階層構造を Polylib 初期化ファイル形式に，三角形ポリゴン情報を STL ファイルに出力します．

引数 p_fname は出力引数で，p_fname の指し示す std::string インスタンスに保存された Polylib 初期化ファイル名が設定されます．

引数 format は，次のように保存する STL/OBJ ファイルの形式を指定します．

”stl_a”の場合，STL のアスキー形式で保存します．
”stl_b”の場合，STL のバイナリ形式で保存します．
”obj_a” の場合, OBJ 形式で保存します．
”obj_b” の場合, OBJ_BIN 形式で保存します. (法線なし)
”obj_bb” の場合, OBJ_BIN 形式で保存します. (法線あり)

引数 extend は，保存するファイル名に任意の文字列を付加します．ファイル名の書式については，2.4 を参照してください．

3.2　MPI 版の主な API の利用方法

本節では MPI 版 Polylib の主な API 利用方法を，API 呼び出し順に沿って説明します．

MPI 版 Polylib の API を利用する手順は下図の通りです．

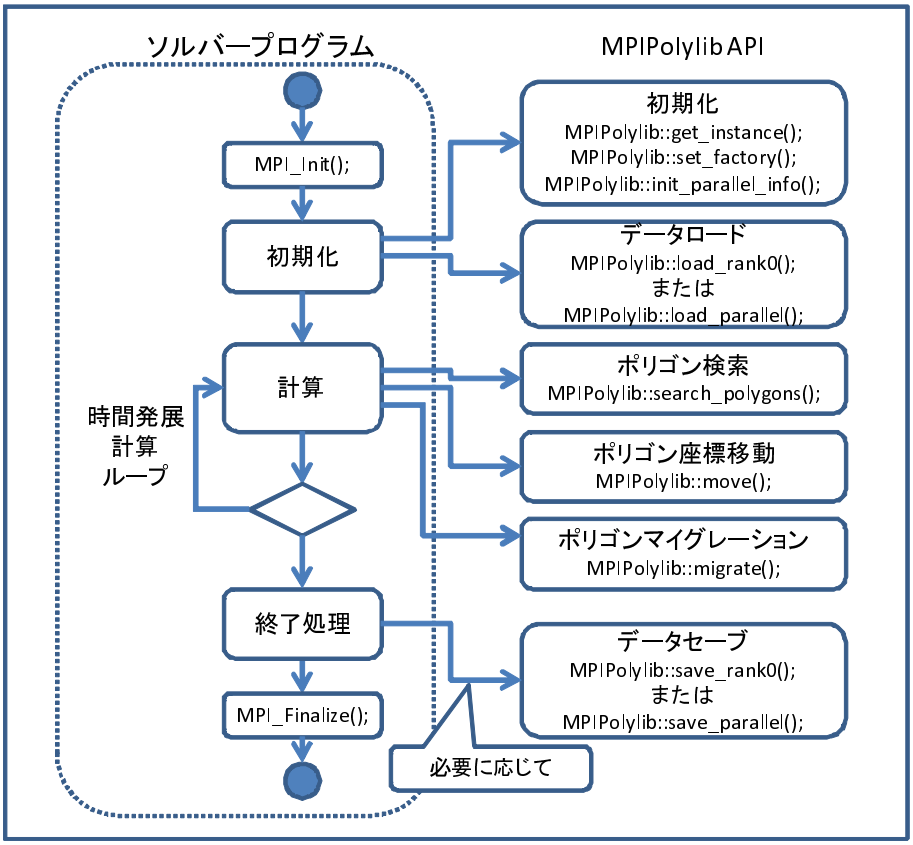


図 3.2　API 利用手順 (MPI 版)

3.2.1　初期化 API

MPIPolylib インスタンスの生成

```
static MPIPolylib<PL_REAL>* MPIPolylib<PL_REAL>::get_instance();
```

MPIPolylib は Polylib クラスを継承し，MPI 版機能を組み込んだクラスです．

Polylib クラス同様に singleton クラスであるため，ユーザプログラム中から MPIPolylib インスタンスを明示的に生成する必要はありません．Static メソッドである MPIPolylib¡PL_REAL¿::get_instance() を呼び出すことで，プロセス内唯一の MPIPolylib インスタンスが返却されます．

また，MPIPolylib¡PL_REAL¿::get_instance() により返却されたインスタンスをユーザプログラムで明示的に消去する必要はありません．インスタンスはプロセス終了時に自動的に消去されます．

MPI 版を利用する場合は PL_REAL には, float か double を指定してください．

PolygonGroup 派生ラスインスタンス生成ファクトリークラスの設定

```
void MPIPolylib<PL_REAL>::set_factory(
        PolygonGroupFactory<PL_REAL> *factory
);
```

ユーザが定義する PolylgonGroup 派生クラスを利用する場合，その派生クラスインスタンスの生成方法を記述した PolygonGroupFactory 派生クラスインスタンスを，本メソッドを利用して Polylib に登録する必要があります．

PolylgonGroup 派生クラスを利用しないのであれば，本 API を呼び出す必要はありません．

PolygonGroup 派生クラスの具体的な利用法については後述のチュートリアルを参照してください．

並列計算情報の設定

```
POLYLIB_STAT MPIPolylib<PL_REAL>::init_parallel_info(
        MPI_Comm     comm,
        PL_REAL        bpos[3],
        unsigned int bbsize[3],
        unsigned int gcsize[3],
        PL_REAL        dx[3]
);
```

並列計算特有の値を MPIPolylib に設定します．各引数の意味は以下の通りです．

- comm MPI コミュニケータ
- bpos[3] 自ランクが受け持つ計算領域の基準座標
- bbsize[3] 自ランクが受け持つ計算領域のボクセル数
- gcsize[3] 自ランクが受け持つガイドセルのボクセル数
- dx[3] ボクセル 1 辺の長さ

引数で設定された自ランク計算領域情報は，本 API 内部処理により MPI 通信により全ランクへ配信されます．
計算領域に関する各値の意味を下図に示します．

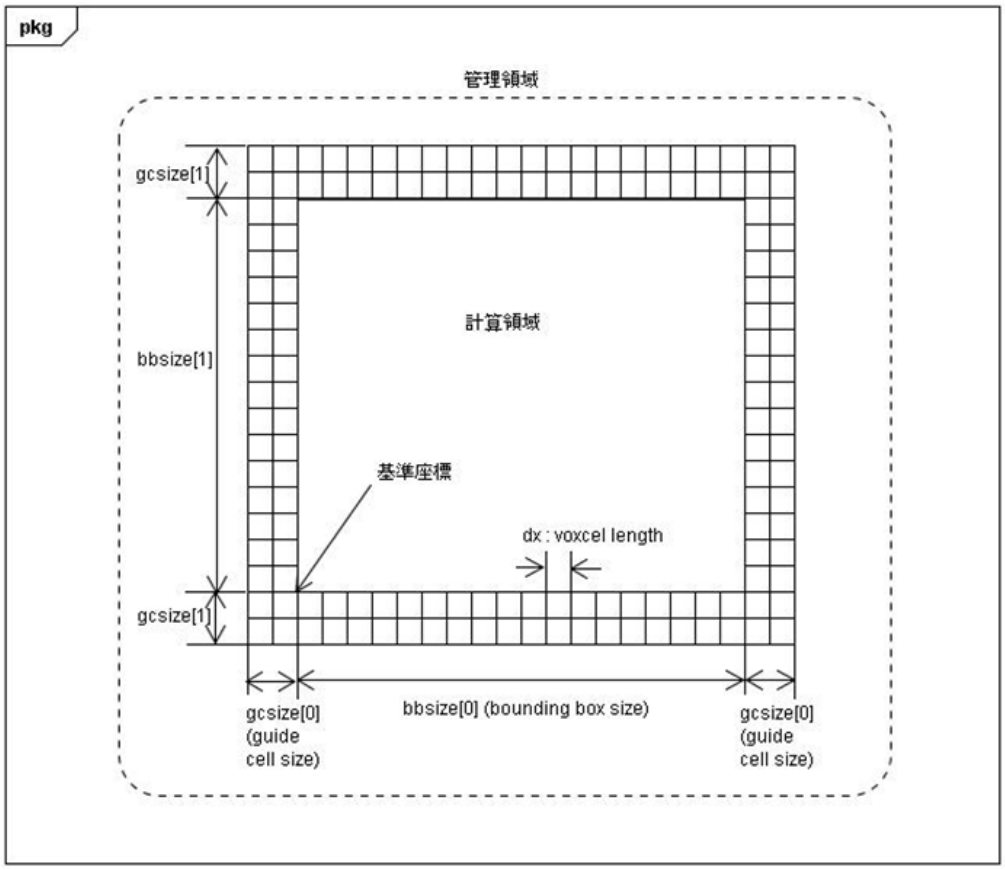


図 3.3　MPIPolylib::init_parallel_info() 各引数の意味

3.2.2　データロード API

```
POLYLIB_STAT MPIPolylib<PL_REAL>::load_rank0(
        std::string config_name = ""
);

POLYLIB_STAT MPIPolylib<PL_REAL>::load_parallel(
        std::string config_name = ""
        ID_FORMAT   id_format   = ID_BIN
);
```

MPIPolylib では 2 種類のデータロード系 API があります.

MPIPolylib::load_rank0 は，rank0 にてファイルを読み込み，他全ランクに対し，グループ階層情報と，各ランクの担当領域ぶんのポリゴン情報を MPI 通信にて配信します.

MPIPolylib::load_paralell は，各ランクにてファイルを読み込みます．MPIPolylib::save_parallel と組み合わせて使用することで時間発展計算の中断/再開に対応しています.

いずれの API も，引数 config_name で指定された Polylib 初期化ファイルを読み込み，そこに記述された内容に基づきポリゴングループ階層構造をオンメモリに生成します．そして最下層ポリゴングループに指定された STL ファイルを読み込み，三角形ポリゴン情報のインスタンスの生成と，ポリゴン検索用 KD 木の生成を行います.

引数 config_name が指定されなかった場合，デフォルト初期化ファイル名である”polylib_config.xml” をカレントディレクトリから読み込みます.

MPIPolylib::load_paralell の引数 id_format は，ロードする三角形 ID ファイルの形式を指定します．未指定の場合，バイナリ形式の三角形 ID ファイルが存在するものとしてロード処理を行います．

3.2.3 検索 API

```
std::vector<Triangle<PL_REAL>*>* Polylib<PL_REAL>::search_polygons(
        std::string        group_name,
        Vec3f                min_pos,
        Vec3f                max_pos,
        Bool                 every
)const;
```

```
const Triangle<PL_REAL>* Polylib<PL_REAL>::search_nearest_polygon(
        std::string        group_name,
        Vec3f                pos
)const;
```

MPIPolylib においても，検索 API は基底クラス Polylib のメソッド Polylib¡PL_REAL¿::search_polygons を利用します．

引数の説明は 3.1.3 を参照してください．

3.2.4 データセーブ API

```
POLYLIB_STAT MPIPolylib<PL_REAL>::save_rank0(
        std::string        *p_fname,
        std::string        format,
        std::string        extend = ""
);

POLYLIB_STAT MPIPolylib:<PL_REAL>:save_parallel(
        std::string        *p_fname,
        std::string        format,
        std::string        extend = "",
        ID_FORMAT   id_format = ID_BIN
);
```

MPIPolylib には 2 種類のデータセーブ系 API があります．

MPIPolylib::save_rank0 は，本 API 呼び出し時点でのグループ階層情報とポリゴン情報を MPI 通信により rank0 に集計し，ファイルに出力します．

MPIPolylib::save_parallel は，本 API 呼び出し時点でのグループ階層情報とポリゴン情報を各ランクにてファイル出力します．

MPIPolylib::save_paralell の引数 id_format は，セーブする三角形 ID ファイルの形式を指定します．未指定の場合，バイナリ形式でセーブ処理を行います．

各引数の意味については，3.1.5 を参照してください．

3.2.5　ポリゴン座標移動 API

```
POLYLIB_STAT MPIPolylib::move(PolylibMoveParams& param);
```

MPIPolylib 管理下にある move メソッド実装済の PolygonGroup 派生クラスに所属する三角形ポリゴンの座標を，move メソッドに実装された座標移動関数に基づきその頂点座標を変更します．

また，後述のメソッド MPIPolylib::migrate() の実行の前処理として，隣接ランク計算領域間をマイグレートしそうな三角形ポリゴンにフラグを立てるなどの処理を行います．

move メソッドを実装した PolygonGroup 派生クラスインスタンスが MPIPolylib 管理下にある場合，本メソッド実行後に MPIPolylib::migrate() メソッドを実行する必要があります．

3.2.6　ポリゴンマイグレーション API

```
POLYLIB_STAT MPIPolylib::migrate();
```

前述のメソッド MPIPolylib::move により隣接ランク計算領域に移動した三角形ポリゴン情報を，隣接ランク同士で MPI 通信により送受信します．

MPI 版 PolylibAPI 利用の注意点

- MPIPolylib の内部では，MPI の初期化・終了処理は行いません．ソルバー側で MPI_Init() および MPI_Finalize() を呼び出す必要があります．
- MPIPolylib::load_parallel() は，MPIPolylib::save_parallel() で保存されたファイル読み込みを前提としています．すなわち，読み込むべき Polylib 初期化ファイルに記述されたグループ階層構造は全 rank で一致しており，STL ファイルは各ランク計算領域部分のポリゴン情報のみを保持しているものとします．また，三角形 ID ファイル形式についても，セーブ形式と同じ形式を指定してロードするものとします．
- MPIPolylib の API のうち，MPI 通信を伴う API は全ランクで同時実行される必要があります．MPI 通信を伴う API は以下の通りです．
  - MPIPolylib::init_parallel_info();
  - MPIPolylib::load_rank0();
  - MPIPolylib::save_rank0();
  - MPIPolylib::migrate();

3.3　C 言語用 Polylib 主な API 利用方法（単一プロセス版）

本節では C 言語用の単一プロセス版 Polylib の主な API 利用方法を，API 呼び出し順に沿って説明します．

C 言語用単一プロセス版 Polylib の API を利用する手順は下図の通りです．

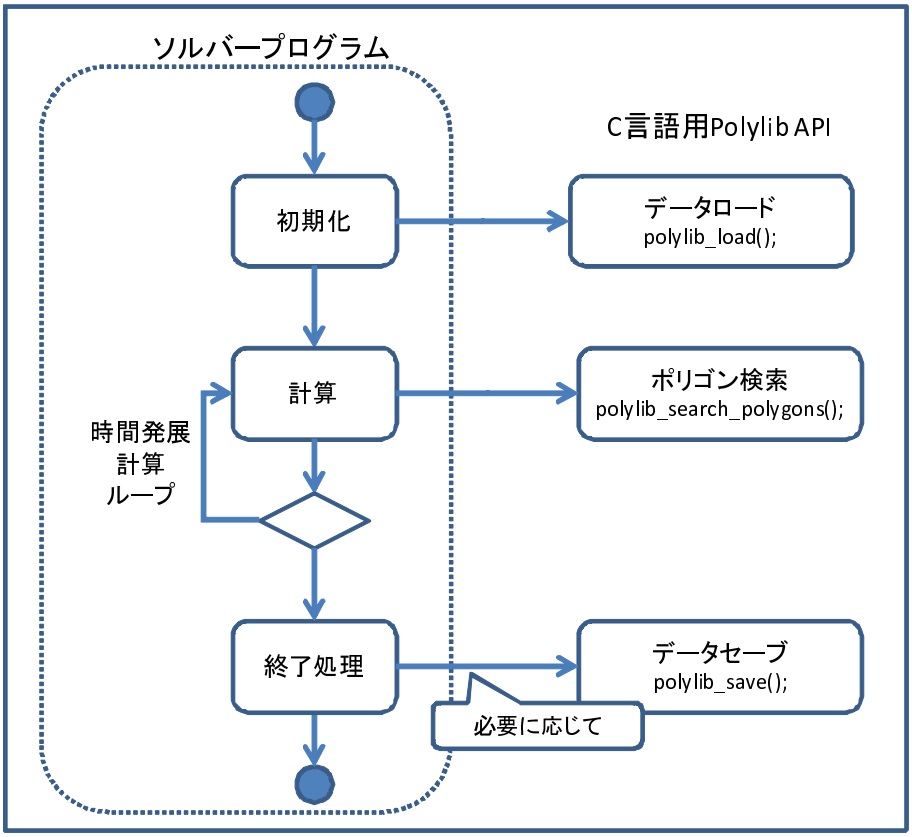


図 3.4　API 利用手順 (C 言語 単一プロセス版)

C 言語用 Polylib の各 API は基本的に C++ 版 Polylib の各メソッドのラッピング関数として実現されています．各 API の引数・返却値等は C 言語向けに型変換されていること以外に違いはありません．

C 言語用 Polylib には，Polylib::move() 相当の機能はありません．これは，Polylib::move() がクラスの継承やメソッドの動的束縛など，オブジェクト指向プログラミングの仕組みを利用して実現されているためです．

3.4　C 言語用 Polylib 主な API 利用方法（MPI 版）

本節では C 言語用の MPI 版 Polylib の主な API 利用方法を，API 呼び出し順に沿って説明します．

C 言語用 MPI 版 Polylib の API を利用する手順は下図の通りです．

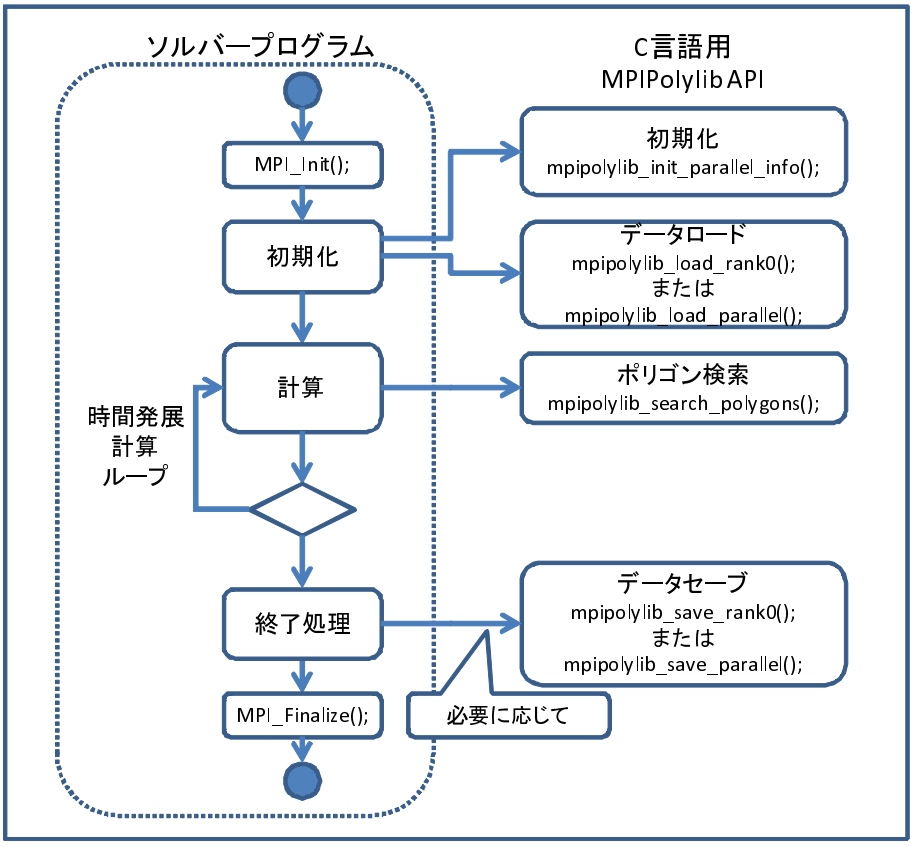


図 3.5　API 利用手順 (C 言語 MPI 版)

C 言語用 MPIPolylib の各 API は基本的に C++ 版 MPIPolylib の各メソッドのラッピング関数として実現されています．各 API の引数・返却値等は C 言語向けに型変換されていること以外に違いはありません．

C 言語用 MPIPolylib には，MPIPolylib::move() や MPIPolylib::migrate() 相当の機能はありません．これは，これらの機能がクラスの継承やメソッドの動的束縛など，オブジェクト指向プログラミングの仕組みを利用して実現されているためです．

3.5　動作確認用 API

本節では Polylib および MPIPolylib の動作確認用 API について説明します．これらの API はソルバー開発時に利用するものであり，動作速度やメモリー消費量の点において非効率的ですので，通常のソルバー実行時には利用しないでください．

3.5.1　ポリゴングループ階層構造確認用 API

```
void Polylib::show_group_hierarchy(
        FILE        *fp = NULL
);
```

Polylib 管理下の全てのポリゴングループについて，その名称を階層レベルに従ったインデントをつけて引数 fp で指定されたファイルへ出力します．fp が未指定の場合は標準出力に出力します．

3.5.2　ポリゴングループ情報確認用 API

```
void Polylib::show_group_info(std::string group_name);
```

指定された名称のポリゴングループについて，グループの情報と配下の三角形ポリゴン情報を標準出力に出力します．

出力内容は以下の通り．

- 親グループ名称
- 自身の名称
- STL ファイル名
- 登録三角形数
- 各三角形の 3 頂点ベクトルの座標
- 法線ベクトルの座標
- 面積

3.5.3　ポリゴン座標移動距離確認用 API

```
POLYLIB_STAT PolygonGroup::init_check_leaped();

POLYLIB_STAT PolygonGroup::check_leaped(
        Vec3f       origin,
        Vec3f       cell_size
);
```

PolygonGroup 継承クラスでオーバーライドした move() メソッドにおいて，三角形ポリゴンの頂点座標が隣接ボクセルより遠方へ移動したか否かをチェックするために利用する API です．

PolygonGroup::init_check_leaped() は，チェック処理の初期化関数です．move() メソッド内で，実際に頂点移動処理を行う前に呼び出します．移動前の頂点座標を一時的に保存しますので，当該ポリゴングループの三角形ポリゴン数に応じてメモリを消費します．

PolygonGroup::check_leaped() は，移動前後の頂点座標の距離を確認する関数です．move() メソッド内で，頂点移動処理実行後に呼び出します．隣接ボクセルより遠方に移動した頂点については，標準エラー出力にその三角形 ID，移動前後の頂点座標情報を出力します．

出力例を以下に示します．

```
PolygonGroup::check_leaped() Leaped Vertex Detected. GroupID:0 TriaID:2355
 before:(-16.0798 1093.99 -605.148) after:(-16.0798 1147.59 -496.047)
```

並列環境下で本 API を利用する場合，各ランクにおける check_leaped() の引数 origin，cell_size は，MPIPolylib::get_myproc() で取得できる ParallelInfo 構造体のメンバ変数 m_area から取得することが可能です．

なお，PolygonGroup::init_check_leaped() で確保された一時的メモリ領域は，PolygonGroup::check_leaped() を呼び出すと解放されます．

3.5.4　メモリ消費量確認用 API

```
unsigned int Polylib::used_memory_size();

unsigned int MPIPolylib::used_memory_size();
```

Polylib および MPIPolylib が確保しているメモリ量を byte 単位で返却します．

MPIPolylib の場合，本メソッドを呼び出したランクにおけるメモリ量が返されます．

報告されるメモリ消費量は概算です．ユーザ定義された PolygonGroup 派生クラスの拡張属性等の Polylib フレームワーク外の消費メモリ利用については含まれません．

3.6　エラーコード

Polylib 内部でエラーが発生した場合に返却されるエラーコード POLYLIB_STAT 型は，include/common/PolylibStat.h に定義されています．エラーコードの一覧を下表に示します．

エラーコード	意味
PLSTAT_OK	処理成功
PLSTAT_NG	一般的なエラー
PLSTAT_INSTANCE_EXISTED	Polylib インスタンスがすでに存在している
PLSTAT_INSTANCE_NOT_EXIST	Polylib インスタンスが存在しない
PLSTAT_STL_IO_ERROR	STL ファイル IO エラー
PLSTAT_UNKNOWN_STL_FORMAT	ファイル拡張子が.stla，.stlb，.stl 以外
PLSTAT_FILE_NOT_SET	リーフグループにファイル名が未設定
PLSTAT_CONFIG_ERROR	定義ファイルでエラー発生
PLSTAT_GROUP_NOT_FOUND	グループ名が Polylib に未登録
PLSTAT_GROUP_NAME_EMPTY	グループ名が空である
PLSTAT_GROUP_NAME_DUP	グループ名が重複している
PLSTAT_MEMORY_NOT_ALLOC	メモリ確保に失敗した
PLSTAT_POLYGON_NOT_EXIST	PolygonGroup に Polygons が未設定
PLSTAT_TRIANGLE_NOT_EXIST	Polygons に三角形リストが未設定
PLSTAT_NODE_NOT_FIND	KD 木生成時に検索点が見つからなかった
PLSTAT_ROOT_NODE_NOT_EXIST	KD 木のルートノードが存在しない
PLSTAT_ARGUMENT_NULL	引数のメモリ確保が行われていない
PLSTAT_MPI_ERROR	MPI 関数がエラーを戻した

表 3.1　エラーコード一覧

第 4 章

テストコード

API を網羅的に評価するサンプルプログラムについて，説明します．

4.1　サンプルプログラム

@todo サンプルプログラムの詳細について記述

第 5 章

チュートリアル

本章では，例題を用いた Polylib の使用法について説明します．

5.1　サンプルモデルによるチュートリアル

Polylib の利用方法を説明するために，以下のサンプルモデルを考えます．

各物体形状にはそれぞれ名前が付いており，時間発展計算の時刻ステップに応じて物体形状が移動する物体についてはその移動計算式が分かっているものとします．

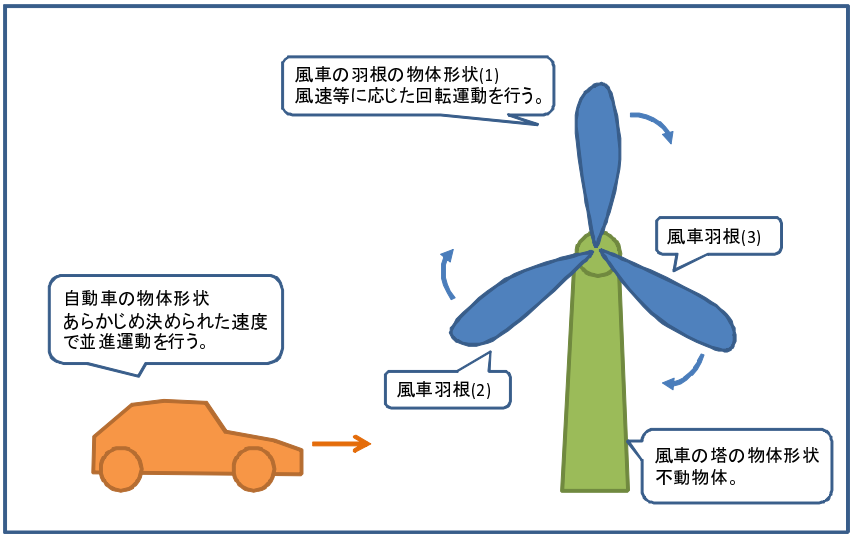


図 5.1　サンプルモデル

5.1.1　物体形状のグルーピング

Polylib では物体形状ごとに，PolylgonGroup クラスインスタンスとして管理します．

PolylgonGroup クラスは，PolylgonGroup クラス同士による階層構造が表現可能です．

ライブラリユーザは，複数の PolygonGroup インスタンスを纏めて，概念的なグループを表現することができます．

サンプルモデルについて，以下のようなグループ階層構造を定義することとします．

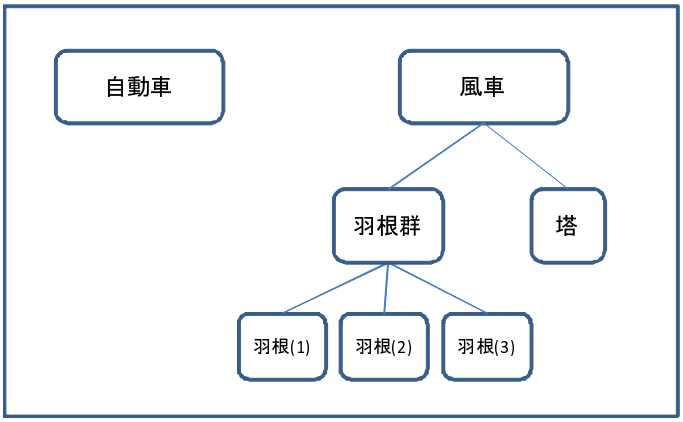


図 5.2　サンプルモデルのグループ階層構造

概念的なグループは，PolygonGroup クラスで表現することができます．特有の属性値を持つグループや，物体形状が移動するようなグループは，PolygonGroup を派生させたユーザ定義クラスで表現します．

サンプルモデルの風車の塔のような不動物体については，特に追加の属性などがなければ，PolygonGroup クラスで表現します．

上記のグループ階層構造図を PolygonGroup クラスとユーザ定義の派生クラスを割り当てたものを下図に示します．（グループ名称も ascii 文字表現としました）

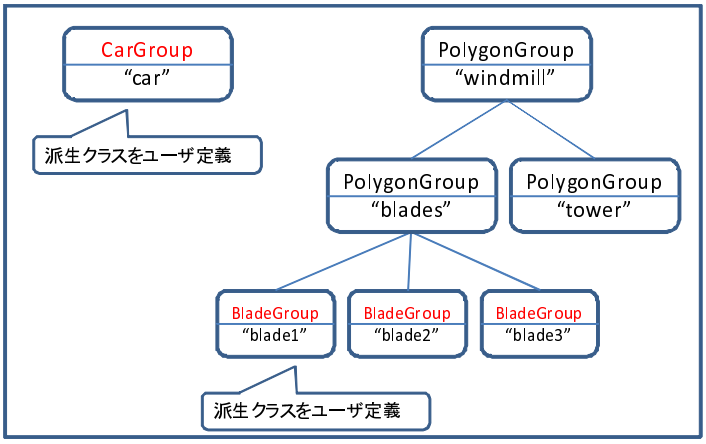


図 5.3　グループ階層構造図

5.1.2　PolygonGroup 派生クラスの定義

ライブラリユーザは，PolygonGroup クラスを派生させた継承クラスを定義することができます．
派生させることの意義は以下 2 点です．

- 当該のポリゴングループに任意の属性値，メソッドを定義したいとき
- 当該のポリゴングループの座標移動計算式を定義したいとき

サンプルモデルでは，”car”が並行移動，”blade1”,”blade2”,”blade3”がそれぞれ回転移動しますので，継承クラスを定義します．

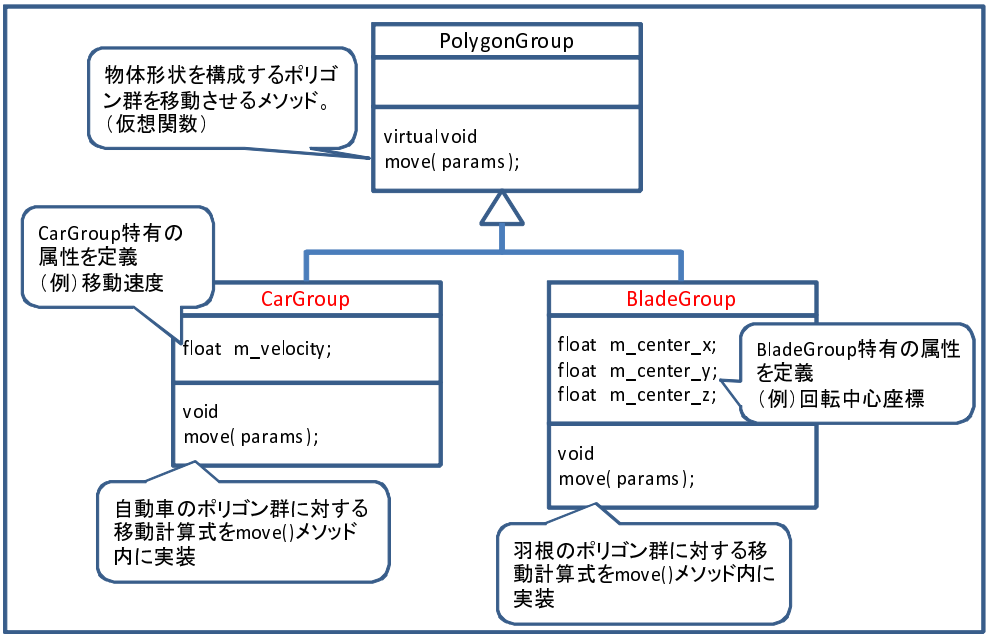


図 5.4　PolygonGroup と派生クラス

派生クラスで追加した属性値は，整数，実数，文字列のいずれかであれば，グループ階層構造構築メソッド PolygonGroup::build_group_tree() をオーバライドすることで，初期化ファイルから値を取得することが可能です．CarGroup の追加属性 float m_velocity の値を取得する CarGroup::build_group_tree() は以下のようになります．

```
POLYLIB_STAT CarGroup<PL_REAL>::build_group_tree(
        Polylib<PL_REAL>* polylib,
        PolygonGroup<PL_REAL>* parent,
        TextParser* tp
)
{
        TextParserError status = TP_NO_ERROR;
        m_velocity = 0;

        // CarGroup クラスで追加定義した変数 m_velocity に値を設定
        vector<string> leaves;
         tp->getLabels(leaves);
        vector<string>::iterator leaf_iter=find(leaves.begin(),leaves.end(),"velocity");
        if(leaf_iter!=leaves.end()){
                string value;
                status=tp->getValue(*leaf_iter,value);
                if(status!=TP_NO_ERROR){
                        tp->TextParserErrorHandler(status," can not read velocity.");
                        return PLSTAT_CONFIG_ERROR;
                }else{
                        int error;
                        m_velocity = tp->convertDouble(value,&error);
                }
        }

        // 基底クラスの属性値読み込み
        POLYLIB_STAT stat = PolygonGroup::build_group_tree(polylib,parent,tp);
        return stat;
}
```

また, 追加属性をデータセーブ時に Polylib 初期化ファイルに書き込むために, PolygonGroup<PL_REAL>::mk_param_tag() メソッドをオーバーライドする必要があります．CarGroup の追加属性 float m_velocity の値を Polylib 初期化ファイルに保存する CarGroup:: mk_param_tag() は以下のようになります．

```
POLYLIB_STAT CarGroup<PL_REAL>::mk_param_tag(
         TextParser* tp,
        string rank_no,
        string extend,
        string format
) {
        stringstream ss;
        POLYLIB_STAT  stat;
         tp->changeNode("/");
        tp->changeNode(acq_fullpath());
        ss<<m_velocity;
        string value;
        ss>>value;

        // velocity タグ作成
        tp->updateValue("velocity",value);

        // 基底クラスが管理するタグ作成
        return PolygonGroup<PL_REAL>::mk_param_tag(tp,rank_no,extend,format);
}
```

ポリゴン群の座標移動計算式は，PolygonGroup<PL_REAL>::move() メソッドをオーバーライドして実装します．た

とえば並行移動を行う CarGroup<PL_REAL>::move() メソッドは以下のようになるでしょう．

```
POLYLIB_STAT
CarGroup<PL_REAL>::move(PolylibMoveParams&   params)
{
    // 引数チェック
    if (params.m_current_step == params.m_next_step)  return PLSTAT_OK;
    if (params.m_current_step > params.m_next_step)   return PLSTAT_NG;
    if (params.m_delta_t <= 0.0)                       return PLSTAT_NG;

    // 移動量
    // X 軸方向に 1step あたり params.m_delta_t × m_velocity だけ移動するものとする．
    PL_REAL move_pos = (params.m_next_step - params.m_current_step) *
                                        this->m_velocity * params.m_delta_t;

    // 三角形リストを取得
    std::vector<PrivateTriangle<PL_REAL>*>* tria_list = m_polygons->get_tri_list();
    std::vector<PrivateTriangle<PL_REAL>*>::iterator it;

    // 三角形リスト内の全ての三角形について頂点座標を更新
    for (it=tria_list->begin(); it!=tria_list->end(); it++) {
        PrivateTriangle<PL_REAL> *tria = (*it);
        Vec3<PL_REAL>             *last_vtx = tria->get_vertex();
        Vec3<PL_REAL>              moved_vtx[3];

        // X 座標 (t[0]) のみ更新
        moved_vtx[0].t[0] = last_vtx[0].t[0] + move_pos;
        moved_vtx[1].t[0] = last_vtx[1].t[0] + move_pos;
        moved_vtx[2].t[0] = last_vtx[2].t[0] + move_pos;
        moved_vtx[0].t[1] = last_vtx[0].t[1];
        moved_vtx[1].t[1] = last_vtx[1].t[1];
        moved_vtx[2].t[1] = last_vtx[2].t[1];
        moved_vtx[0].t[2] = last_vtx[0].t[2];
        moved_vtx[1].t[2] = last_vtx[1].t[2];
        moved_vtx[2].t[2] = last_vtx[2].t[2];

        // 移動後の頂点座標を設定．法線ベクトルも再計算
        tria->set_vertexes( moved_vtx, true, false );
    }

    // 頂点座標が移動したことにより，KD 木の再構築が必要．
    // 要再構築フラグを立てる．
    m_need_rebuild = true;

    return PLSTAT_OK;
}
```

実装した move() メソッドで実際に三角形ポリゴン頂点座標が移動した場合，move() メソッド内で，PolygonGroup::m_need_rebuild フラグを true にセットしなければならないことに注意してください．このフラグを ture にしないと KD 木の再構築が行われないため，Polylib::search_polygons() メソッドが正しい検索結果を返せなくなります．

また，PolygonGroup 派生クラスでは，PolygonGroup::get_class_name() メソッドおよび PolygonGroup::whoami() メソッドをオーバライドして実装し，システム一意なクラス名称を返却するようにします．これは初期化ファイルに記述するクラス名称と一致させる必要があります．以下に CarGroup の場合の例を示します．

```
static std::string CarGroup<PL_REAL>::get_class_name()
{
        return "CarGroup";
}

virtual std::string CarGroup<PL_REAL>::whoami()
{
        return get_class_name();
```

```
};
```

5.1.3　PolygonGroupFactory 派生クラスの定義

サンプルモデルでは，CarGroup と BladeGroup の 2 種類の派生クラスがユーザ定義されました．

Polylib フレームワーク内部でこれら派生クラスインスタンスを生成するためには，生成方法を記述した，PolygonGroupFactory 派生クラスを定義する必要があります．

ここでは，MyGroupFactory という名前の派生クラスを定義することとします．

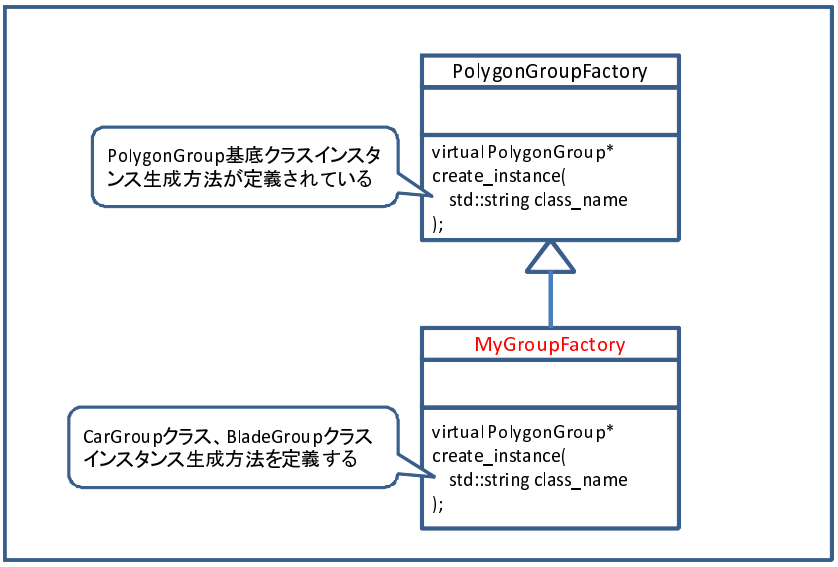


図 5.5　PolygonGroupFactory と派生クラス

Polylib には 1 つの PolygonGroupFactory 派生クラスインスタンスしか登録できません．今回のようにユーザ定義した PolygonGroup 派生クラスが複数ある場合は，その全てのインスタンス生成方法を MyGroupFactory::create_instance() で実装します．以下にそのコード例を示します．

```
PolygonGroup*
MyGroupFactory<PL_REAL>::create_instance( std::string  class_name )
{
    //---- class_name に応じたインスタンスを生成
    // "CarGroup"の生成
    if (class_name == CarGroup<PL_REAL>::get_class_name()) {
         return new CarGroup<PL_REAL>();
    }
    // "BladeGroup"の生成
    if (class_name == BladeGroup<PL_REAL>::get_class_name()) {
         return new BladeGroup<PL_REAL>();
    }
    else {
         // 基底クラスの create_instance に移譲
         return PolygonGroupFactory<PL_REAL>::create_instance(class_name);
    }
}
```

5.1.4 初期化ファイル

サンプルモデルの初期化ファイルの内容を以下に示します.

```
Polylib {
        car {
                class_name = "CarGroup"
                filepath = "/foo/bar/car.stl"
                movable = "true"
                velociry = 0.50
        }
        windmill {
                class_name = "PolygonGroup"
                blades {
                        class_name = "PolygonGroup"
                        blade1 {
                                class_name = "BladeGroup"
                                file path = "./blade1.stl"
                                movable = "true"
                                center_x = 0.0
                                center_y = 123.45
                                center_z = 345.67
                        }
                        blade2 {
                                class_name = "BladeGroup"
                                file path = "./blade2.stl"
                                movable = "true"
                                center_x = 0.0
                                center_y = 123.45
                                center_z = 345.67
                        }
                        blade3 {
                                class_name = "BladeGroup"
                                file path = "./blade3.stl"
                                movable = "true"
                                center_x = 0.0
                                center_y = 123.45
                                center_z = 345.67
                        }
                }
                tower {
                        class_name = "PolygonGroup"
                        file path = "./tower.stl"
                }
        }
}
```

5.1.5　メインプログラム

以上のサンプルモデルを前提とした MPIPolylib を利用するメインプログラム例を以下に示します.

```
#include <stdio>
#include "mpi.h"
#include "MPIPolylib.h"
#include "CarGroup.h"
#include "BladeGroup.h"
#include "MyGroupFactory.h"

using namespace std;
using namespace PolylibNS;

#define PL_REAL double

// 領域分割情報構造体定義
struct MyParallelInfo {
    PL_REAL          bpos[3];  // 基準座標
    unsigned int bbsize[3];  // 計算領域ボクセル数
    unsigned int gcsize[3];  // ガイドセル領域ボクセル数
    PL_REAL            dx[3];  // ボクセルサイズ
};

// ４並列を前提とした領域分割データ
static MyParallelInfo myParaInfos[4] = {
    {{-1100, -1800, -1800}, {18, 18, 18}, {1, 1, 1}, {100, 100, 100} },
    {{-1100,     0, -1800}, {18, 18, 18}, {1, 1, 1}, {100, 100, 100} },
    {{-1100, -1800,     0}, {18, 18, 18}, {1, 1, 1}, {100, 100, 100} },
    {{-1100,     0,     0}, {18, 18, 18}, {1, 1, 1}, {100, 100, 100} }
};

int main( int argc, char** argv )
{
    int             rank;
    unsigned int  step;
    POLYLIB_STAT  stat;
    PolylibMoveParams params;

    // MPI 初期化
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    cout << "Starting program on rank:" << rank << endl;

    //---- MPIPolylib の初期化処理 -----
    // MPIPolylib インスタンス取得
    MPIPolylib<PL_REAL> *p_polylib = MPIPolylib<PL_REAL>::get_instance();

    // ユーザ定義ファクトリークラスを登録
    p_polylib->set_factory( new MyGroupFactory() );

    // 自 PE の並列実行関連初期化情報の設定
    stat = p_polylib->init_parallel_info( MPI_COMM_WORLD,
                                    myParaInfos[rank].bpos,
                                    myParaInfos[rank].bbsize,
                                    myParaInfos[rank].gcsize,
                                    myParaInfos[rank].dx
                                );
    if( stat != PLSTAT_OK ) return -1;

    // データロード
    stat = p_polylib->load_rank0( "./polylib_config.tpp" );
    if( stat != PLSTAT_OK ) return -1;

    // 時間発展計算ループ (100 ステップ実行)
    for( step=0; i<100; step++ ){
```

```
        // 現在ステップでの計算実行…
        {
            // 必要なポリゴン情報を検索して取得
            vector<Triangle<PL_REAL>*>* trias =
                 p_polylib->search_polygons(/*検索条件を設定*/);

            // 検索結果ベクタの後始末
            delete trias;
        }

        // 次計算ステップへ進むためにポリゴン情報を更新
        // move パラメタセットを設定
        PolylibMoveparams params;
        parmas.m_current_step = step;
        params.m_next_step     = step+1;
        params.m_delta_t       = 1.0;

        // move 実行
        stat = p_polylib->move(params);
        if( stat != PLSTAT_OK ) return -1;

        // migrate 実行
        stat = p_polylib->migrate();
        if( stat != PLSTAT_OK ) return -1;
    }

    // 各ランク毎にデータセーブ．STL はアスキー形式で出力．
    string fname;
    stat = p_polylib->save_parallel( &fname, "", "stl_a" );
    if( stat != PLSTAT_OK ) return -1;
    cout << "saved filename:" << fname << endl;

    // MPI 終了処理
    MPI_Finalize();
    return 0;
}
```

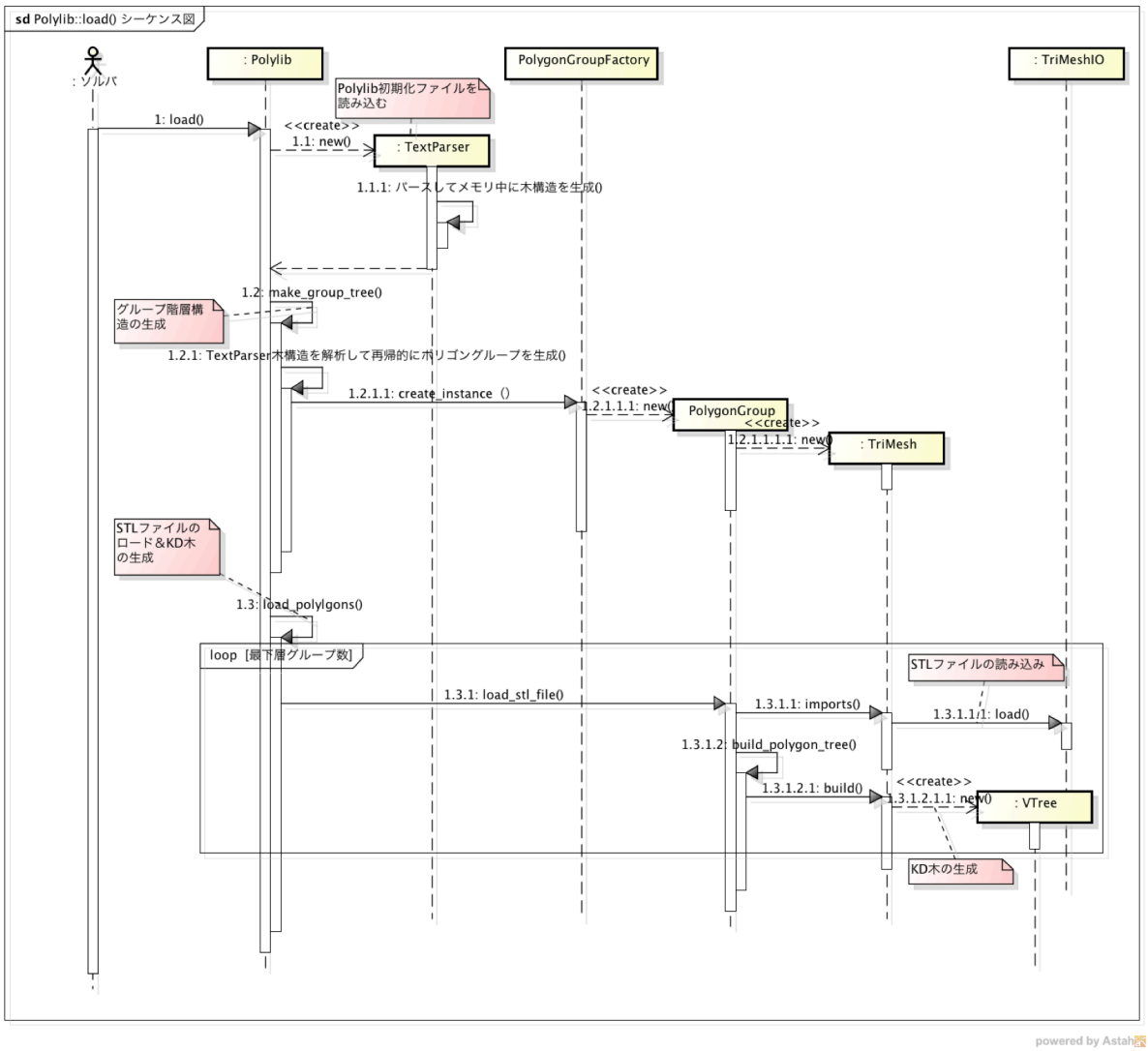
第 6 章

内部設計情報

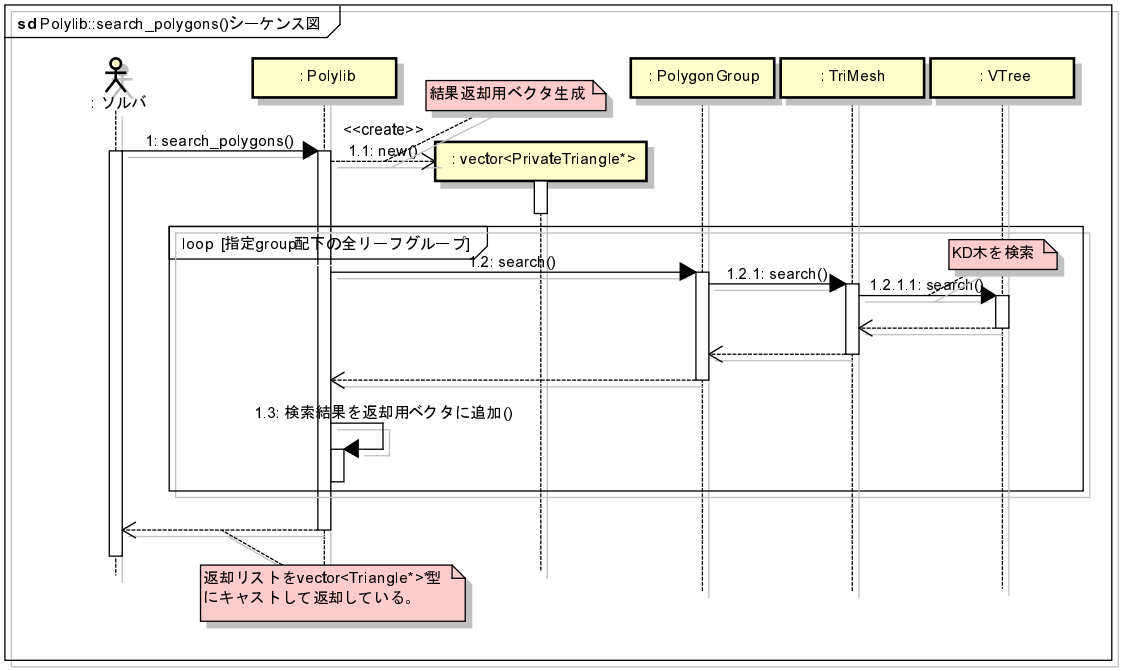
本節では，Polylib および MPIPolylib の内部設計について，UML シーケンス図を用いて説明します．

6.1　Polylib クラス

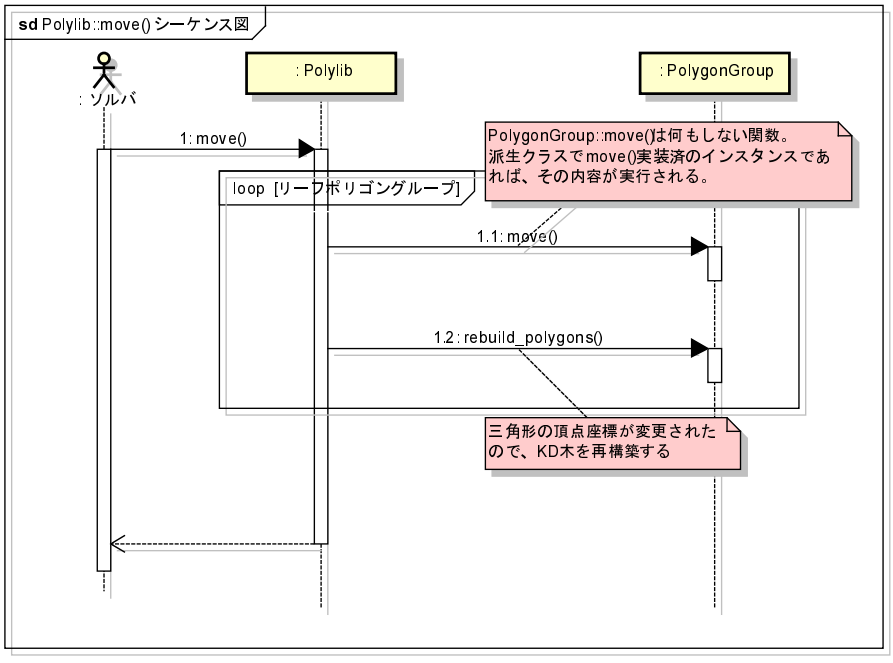
6.1.1　Polylib::load() の内部処理シーケンス



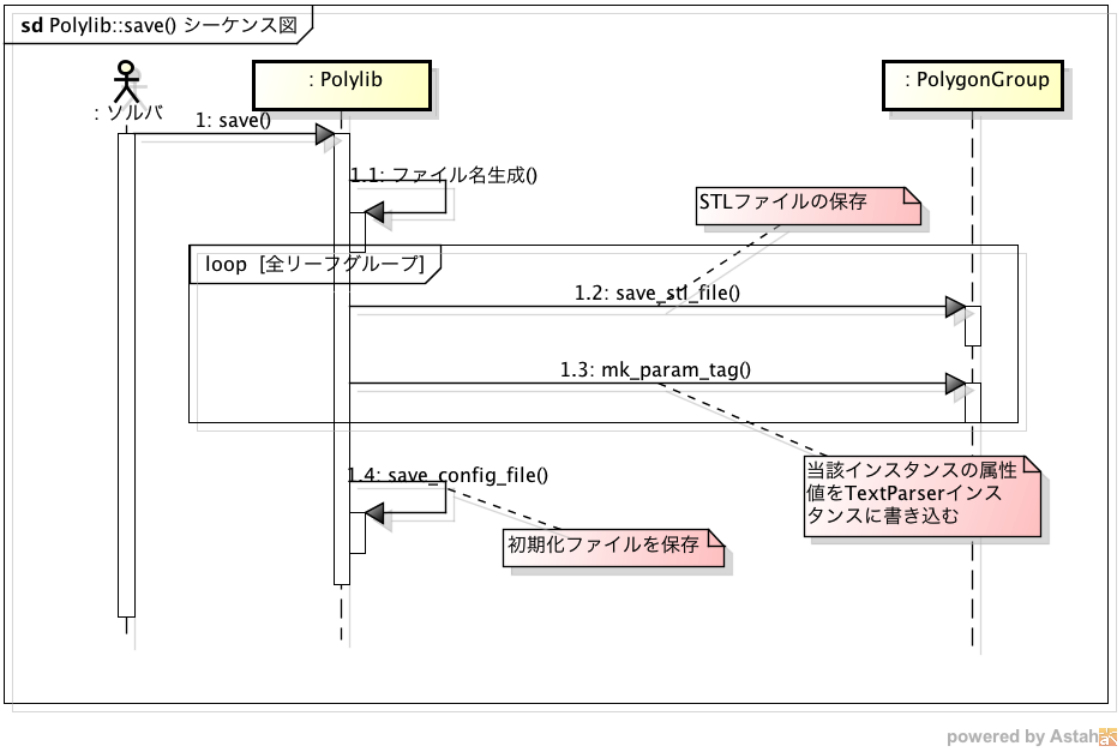
6.1.2　Polylib::search_polygons() の内部処理シーケンス



6.1.3　Polylib::move() の内部処理シーケンス

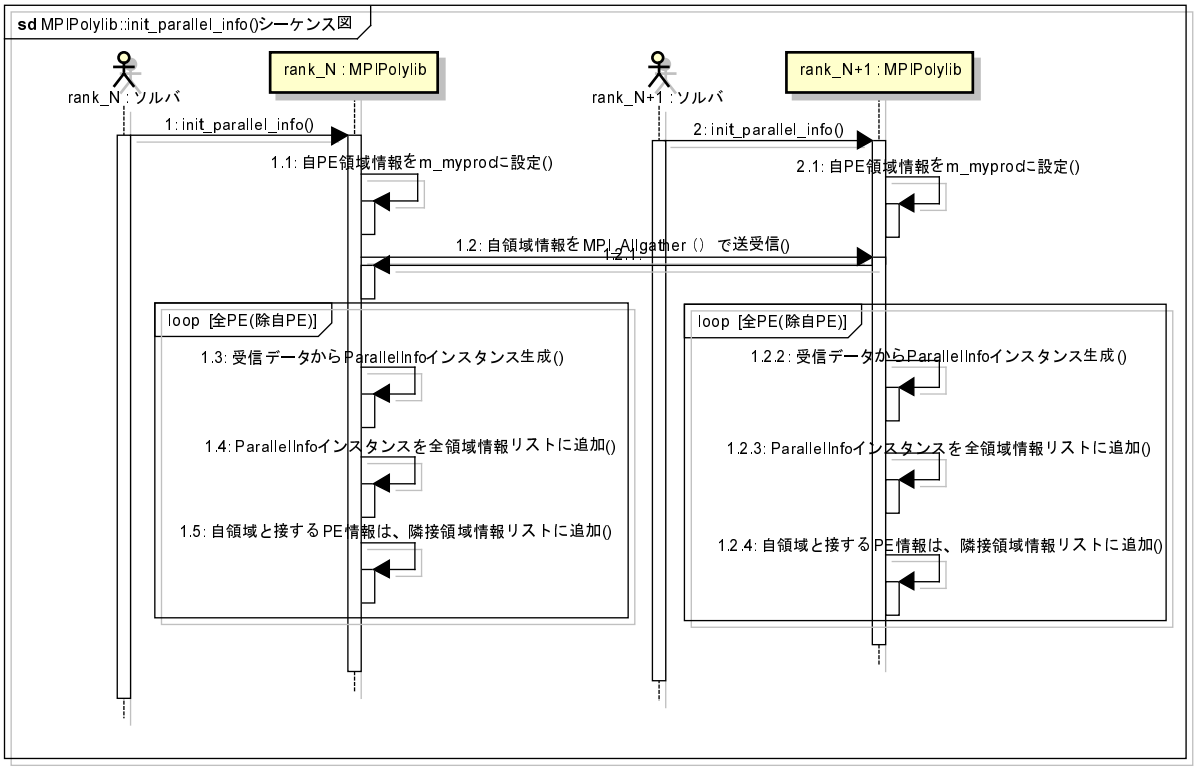


6.1.4　Polylib::save() の内部処理シーケンス

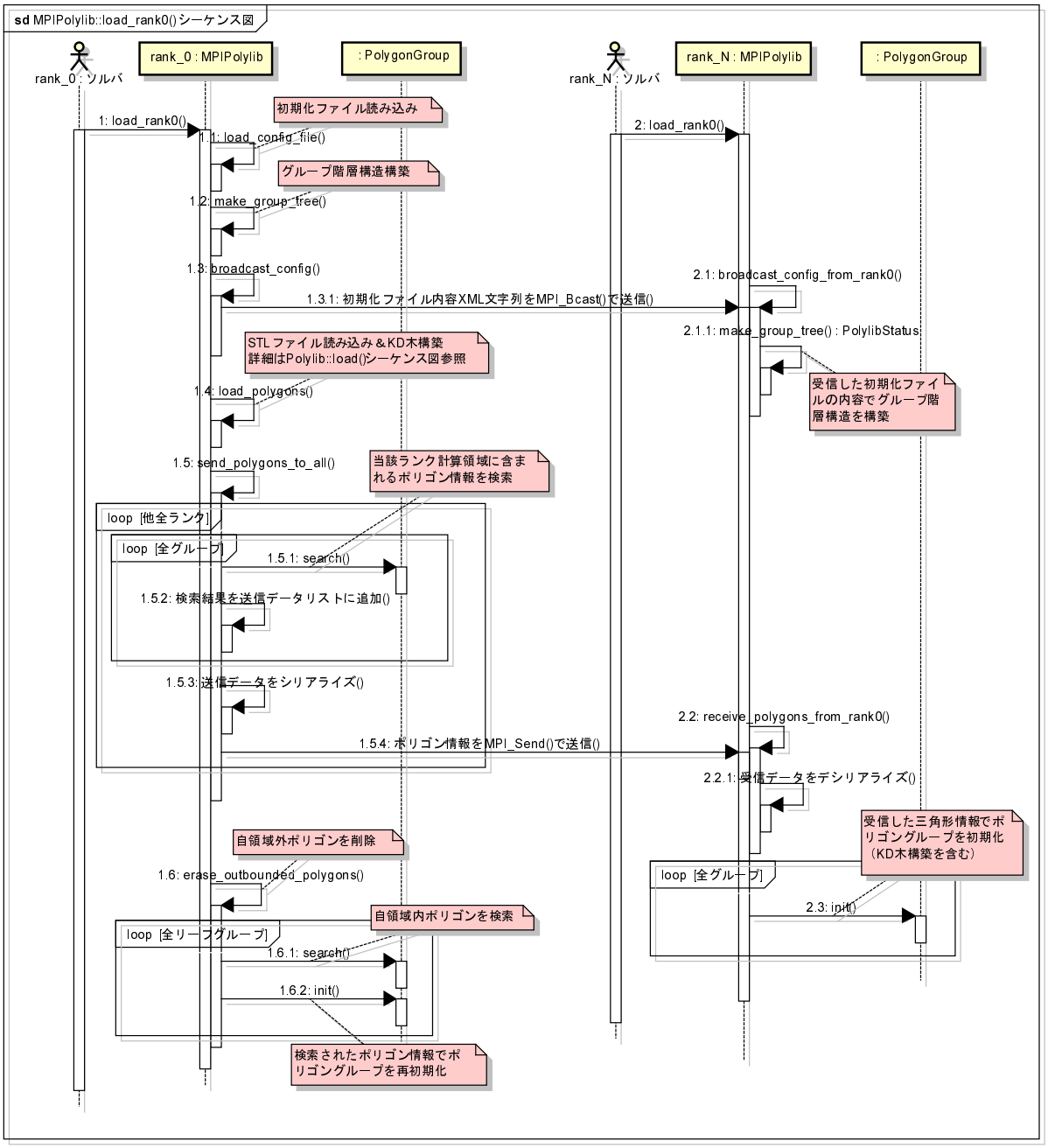


6.2　MPIPolylib クラス

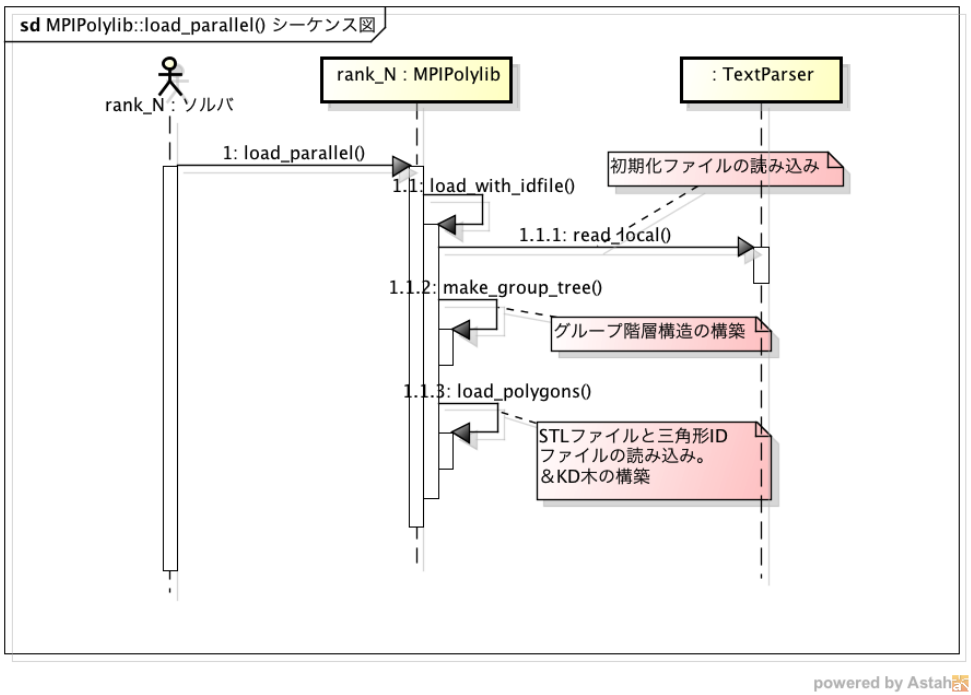
6.2.1　MPIPolylib::init_parallel_info() の内部処理シーケンス



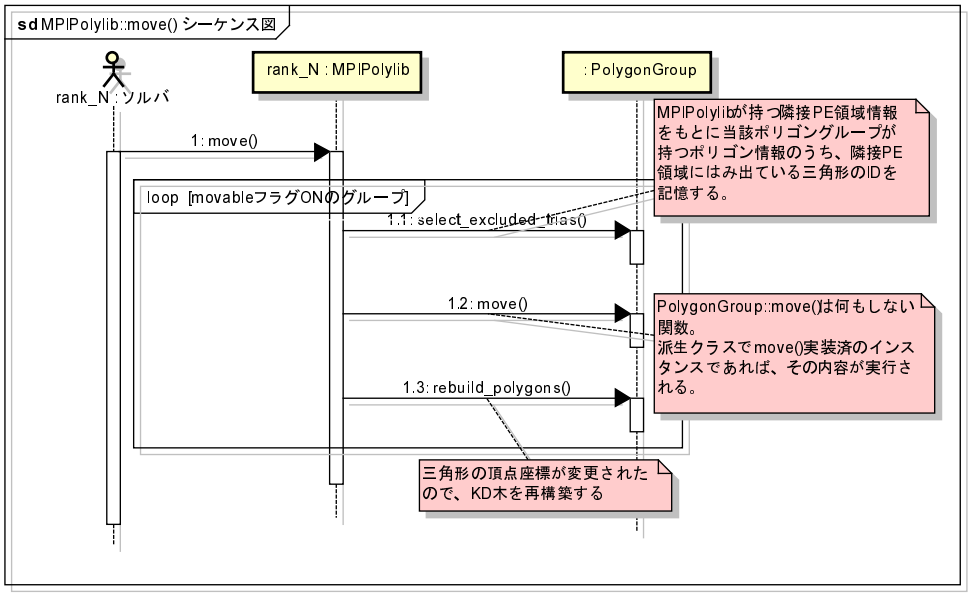
6.2.2　MPIPolylib::load_rank0() の内部処理シーケンス



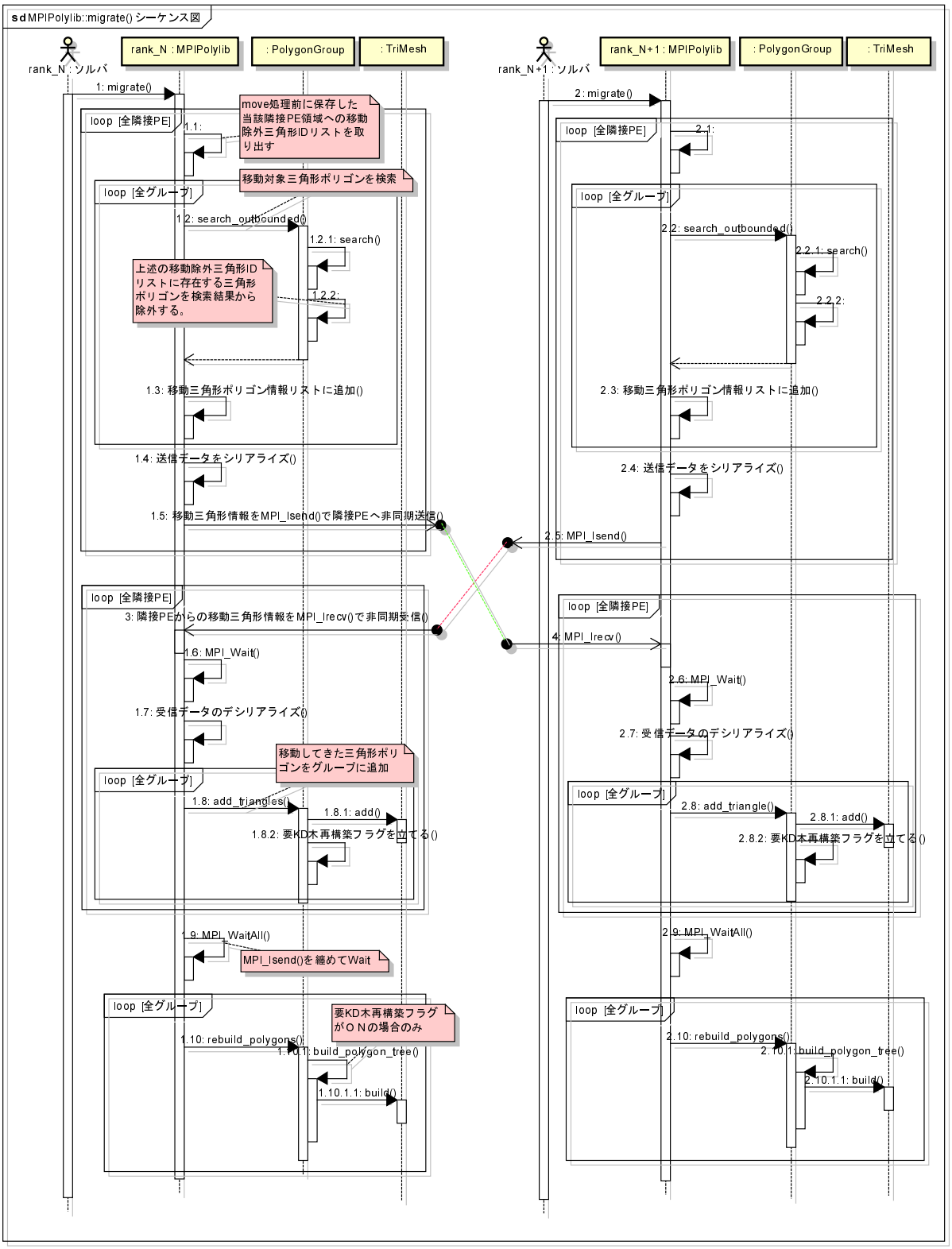
6.2.3　MPIPolylib::load_parallel() の内部処理シーケンス



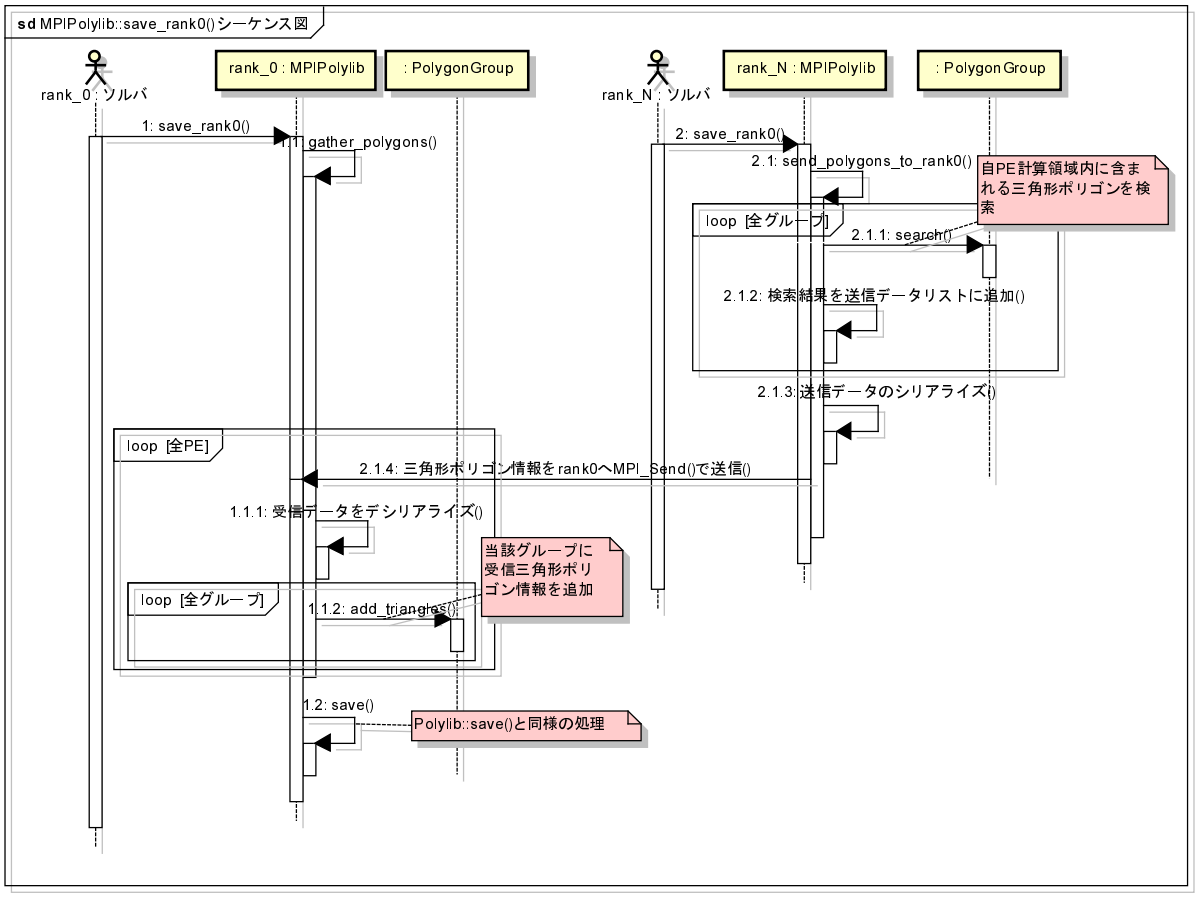
6.2.4　MPIPolylib::move() の内部処理シーケンス



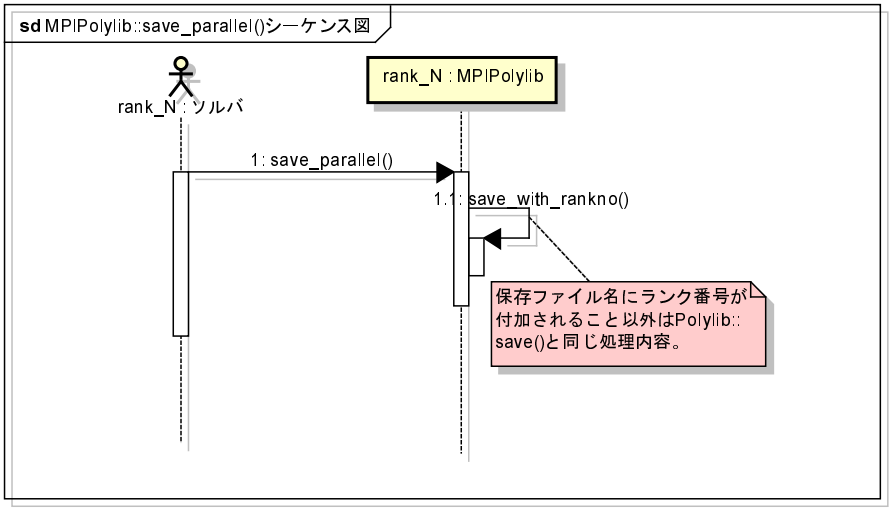
6.2.5　MPIPolylib::migrate() の内部処理シーケンス



6.2.6　MPIPolylib::save_rank0() の内部処理シーケンス



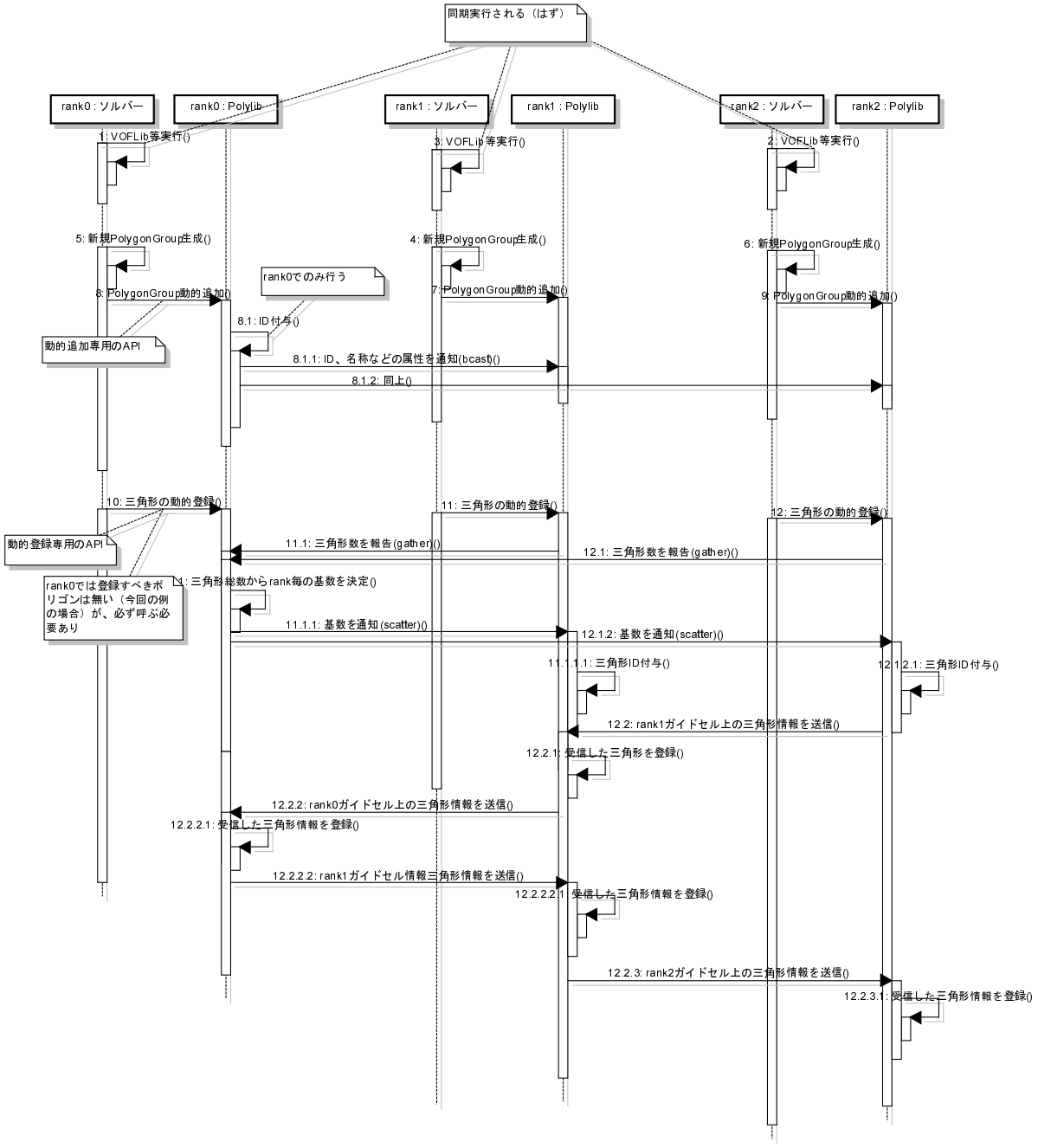
6.2.7　MPIPolylib::save_parallel() の内部処理シーケンス



6.2.8　（参考）MPIPolylib におけるポリゴンの動的登録に関する検討結果

Polylib は動的なポリゴングループの追加や削除に対応していません．これは開発時点でその必要性について検討段階であったためです．

以下 UML シーケンス図はソルバー実行中に VOFLib により新たにポリゴン情報が生成され，それを MPIPolylib に登録した際の内部処理案です．



6.3　Version 3.0.0 の修正

Polylib 3.0.0 のアップデートにあたり，改修内容は以下の通りである．

- ユーザ選択による実数データ切り替え方式の導入
- 重複する頂点座標データを持たない効率的なデータ構造の導入
- 入出力ファイルフォーマットの追加
- サンプルプログラムの製造
- 不要なコードの整理
- ビルド方式の変更

6.3.1　実数データ切り替え方式の導入

ライブラリ全体にわたり，実数データを変数に持つクラス及び関数をすべてテンプレート化しました．これにより，ユーザーが作成するライブラリのオブジェクトは，ユーザーがオブジェクト作成時に実行型を指定することができます．

C 言語用インターフェイスは，configure 時に作成するライブラリ内での実数データの型を指定してコンパイルし，ライブラリ作成をします．作成したライブラリは指定実数型のみ利用できます．

MPI 用のインターフェイスでは，関数にプロセス間で通信する変数の型を指定しなければなりません．その変数型は，configure 時に指定された変数型を元に通信時の変数の型を決定します．

実装

6.3.2　効率的なデータ構造の導入

Version 3.0.0 以前の Polylib では，三角形ポリゴンのオブジェクトに頂点座標データを保持していました．Version 3.0.0 では，頂点座標を格納するリストを別途用意し，三角形ポリゴンのオブジェクトは頂点座標データへのポインタを持つように実装しました．これにより，データ保持メモリを縮小することができます．**図 2.1** はこの機能を実現するための Polylib の主要クラスを示します．

今回，新たに追加及び大幅な変更があるクラスは，次の通りです．

- Vertex (新規)
  頂点座標のクラス．頂点追加時にインスタンスが生成され，VertexList に格納されます．
- VertexList (新規)
  頂点座標のインスタンスを格納するクラス．頂点追加時に，重複がないかどうか確認し，重複が無い場合にのみ頂点を追加します．
- Triangle(変更)
  頂点座標を格納するのを止め，該当する頂点の VertexList に保持されている Vertex クラスへのポインタを 3 頂点分保持します．　　get_vertex() は，Vertex<T>*の配列の先頭アドレスを返すように変更しました．
- PrivateTriangle() (変更)
  継承元が Triangle クラス. コンストラクタ等を Vertex クラスが受けとるように変更しましました．
- PrivateTriangle<T>::get_vertex() を扱っていた箇所 (変更)
  PrivateTriangle<T>::get_vertex() を扱っていた箇所で，Vec3<T> 型から，Vertex<T>**型への変更に対応しました．

尚，ファイル読み込み時にポリゴンデータを格納します．

実装

@todo 頂点座標の同一性の判断アルゴリズム

@todo 双方向リンク作成

6.3.3　入出力ファイルフォーマットの追加

Version 3.0.0 以前では STL フォーマットのアスキー形式とバイナリ形式はファイルフォーマットが異なっていました．Version3.0.0 では拡張子を stl とし，アスキー/バイナリの判断は一度読み込んで判断することにしました．
stl ファイルの他に，OBJ ファイルフォーマットに対して，読み込み書き出しが出きるようにしました．
OBJ フォーマットの読み込みは行頭の文字”v” で表される頂点座標と，行頭の文字”f” で表される多角形の面の行の頂点番号のみを読み込み，データとして格納します．その他の要素及び多角形の面の情報，テキスチャ番号/法線番号は，破棄します．面の情報の行に，三点分以上の番号の情報があっても，最初の 3 つのみを読み込み，残りは破棄する仕様です．
OBJ フォーマットの書き込みは，頂点リストの情報と三角形ポリゴンの情報から，頂点座標の行 (行頭文字”v”), 三角形ポリゴンの行 (行頭文字”f”) のみを書き出します．バイナリモードは，先頭にヘッダをつけた後，頂点座標，ポリゴンの頂点番号の順で書き出します．ヘッダは，ファイルフォーマットを判別する文字列です．

6.3.4　サンプルプログラムの製造

Polylib 各部の API を網羅的に動作させるサンプルプログラムを作成します．

6.3.5　不要なコードの整理

xml を利用していた時期のコードが残っているので，削除しました．

6.3.6　ビルド方式の変更

autotools と libtool に対応し，静的/動的ライブラリの生成も，configure 時に選択できるようにしました．

@todo インストールについて記述．

第 7 章

Appendix

その他，補足資料です.

7.1　OBJ ファイルフォーマット

7.1.1　アスキー形式

@todo ファイルフォーマットについて記述

7.1.2　バイナリ形式

@todo ファイルフォーマットについて記述

第 8 章

アップデート情報

本ライブラリのアップデート情報について記します.

8.1 アップデート履歴

- Version 3.0.0　2013-09-16
  - ポリゴンデータの省メモリ化
    重複する頂点座標データを持たない効率的なデータ構造の導入.
  - ポリゴンデータの実数型の選択機能
  - autoconf & automake 対応
  - 入出力ファイルフォーマットの追加
    obj ファイルの読み込み/書き出しを追加. stl ファイルのアスキーバイナリーの自動判別. これに伴い, ファイル拡張子は*.stl で統一.
  - サンプルプログラムの作成
  - 不要なコードの整理

- Version 2.6.8　2013-07-20
  - PolygonGroup::set_all_exid_of_trias() の修正
    m_id, m_id_defined の両方に値をセット.

- Version 2.6.7　2013-07-20
  - Version 情報取得メソッドの追加
    Version.h.in と getVersionInfo() を追加.

- Version 2.6.6　2013-07-17
  - type ラベルの追加
    PolygonGroup::m_type.
  - ポリゴンの exid に値をセットするメソッドを追加
    PolygonGroup::set_all_exid_of_trias() を追加.

- Version 2.6.5　2013-07-15
- Version 2.6.4　2013-06-27
- Version 2.6.3　2013-06-27
- Version 2.6.2　2013-06-26
- Version 2.6.1　2013-06-25

- Version 2.6　2013-06-24
  - autotools 導入
    並列版のみ autotools でビルド可能.

- Version 2.5　2013-06-17
- Version 2.4　2013-05-08
- Version 2.3　2013-03-25

- Version 2.2　2012-11-27
  - 直近ポリゴン検索機能の追加
    指定された点にもっとも近い三角形ポリゴンを検索する機能を追加.

- Version 2.1　2012-04-31
  - TextParser 書式の初期化ファイルに対応
    初期化ファイル書式を，XML 形式から TextParser 書式に変更.
  - id 付きバイナリ STL ファイル読み込みに対応
    FXgen が出力する id 付きバイナリ STL ファイルの読み込みに対応.
  - STL ファイル読み込み時の縮尺変換
    Polylib::load メソッドに引数 float scale=1.0 を追加し，ジオメトリデータを縮尺変換して読み込むオプションを追加.

- Version 2.0.3　2012-04-22
- Version 2.0.2　2010-11-17
- Version 2.0.1　2010-11-05

- Version 2.0.0　2010-06-30
  - ポリゴン移動機能の追加
    ユーザ定義によるポリゴン座標移動関数を用いた，時間発展計算実行中のポリゴン群の移動機能を追加．並列計算環境化においては，隣接 PE 計算領域間を移動したポリゴン情報を自動的に PE 間で融通.
  - 計算中断・再開への対応
    時間発展計算途中の計算中断時にポリゴン情報をファイル保存する機能を追加．ファイルの保存は，各ランク毎保存，もしくはマスターランクでの集約保存が選択可能．また，時間発展計算の再開時に利用することを想定し途中保存したファイルを指定してポリゴン情報の読み込みを行う機能も追加.
  - データ登録系 API の整理
    ポリゴングループの登録や STL ファイルの読み込みなどのポリゴンデータ登録処理を XML 形式の設定ファイルを利用．データ登録系 API を大幅に刷新.

- Version 1.0.0　2010-02-26
  - 初版リリース