KVS　Tutorial

目次

KVS　Tutorial 1

１．はじめに 2

２．インストールの仕方 3

2.1 インストール環境の準備 3

2.2 ライブラリのインストール 4

2.３．環境変数の準備 4

2.4 KVS のダウンロード 5

2.5 KVS コンパイルとインストール 6

３．KVS コマンドアプリケーション 8

3.1 KV プログラムのコンパイル 8

3.2 KVS 利用環境のチェック 9

3.3 データ変換 10

3.4 可視化ツール 11

４．KVS プログラムの作成 12

4.1 行列/ベクトルクラス 12

例 4.1 ３次元ベクトルの演算 12

例 4.2 行列の演算 15

問題 4.1　標準系を求めよう 19

4.2 オブジェクト 19

例 4.3 構造型ボリュームデータの生成 20

例 4.4 面データの可視化 26

問題 4.2　構造型ボリュームデータの表示 32

4.3 可視化パイプライン 32

例 4.5 ボリュームデータをレイキャスティングを用いてレンダリングしてみよう． 33

問題 4.3　GPU を利用した高速可視化ライブラリを使ってみよう． 37

例 4.6　粒子ベース・ボリュームレンダリングの実装 38

問題 4.4　可視化パイプラインを使ってプログラムを作ろう． 45

１．はじめに

本チュートリアルは可視化ライブラリ群 Kyoto Visualization System (KVS) の基本的な利用方法について説明します．KVS には可視化ライブラリだけでなくコマンドアプリケーションも用意されており，初めにコマンドアプリケーションを通して KVS の簡単な使用方法について学んだ後，KVS ライブラリを用いたプログラムの作成方法について習得していきます．本書で作成するプログラムは，基本的な KVS ライブラリの使用法として行列/ベクトルの演算，KVS で扱うデータ構造の操作とその可視化方法を示します．また，KVS ライブラリによる本格的な可視化プログラム作成のために可視化パイプラインについて説明し，それを利用したプログラムを作成していきます．

本チュートリアルは C++言語で作成されています．C++についての深い知識は必要としませんが C++言語の基本的な記述方法については C 言語の知識を前提として説明しています．

２．インストールの仕方

本章では，KVS　のコンパイルおよびインストールについて説明します．KVS をインストールするためには以下の手順が必要となります．

１．インストール環境（gcc や make）の準備
２．必要なライブラリのインストール
３．環境変数の準備
４．KVS のダウンロード
５．KVS のコンパイルとインストール

KVS は基本的に Mac OS X, Linux, Windows のマルチプラットフォームで利用可能ですが，コンパイルするためには C/C++コンパイラが必要となります．KVS をコンパイルする環境として本書では，Mac 及び Linux の場合には gcc/make を， Windows の場合には Microsoft Visual Studio を利用します．また，KVS を利用して可視化結果を表示するためには GLUT が必要です．KVS ではさらに, GPU を使った高速可視化機能を利用するために GLEW を用いているほか, CUDA や CAVE, SAGE 等のライブラリにも対応しています. 必要なライブラリは本体をコンパイルする前にインストールをしておいて下さい．

2.1 インストール環境の準備

Linux の場合

Linux では，ディストリビューションに応じたパッケージ管理ツールを利用して gcc/make および GLUT が簡単にインストール可能です．各パッケージの開発版（dev または devel キーワードが付いたパッケージ）をインストールして下さい．ただし、GLUT は freeglut という名前でパッケージ管理されていることが多いため，freeglut というパッケージがある場合はそちらをインストールしてください．

Mac OS X の場合

Mac OS X の場合は，Xcode をインストールすると gcc/make コマンドがインストールされます．付属 DVD から（ない場合は Apple ホームページにある iOS Dev Center から Xcode をダウンロードし）インストールして下さい．GLUT は既にインストールされていますので，あらためてインストーしなおす必要はありません．

Windows の場合

Windows の場合は，Microsoft Visual C/C++を利用します．無料版の Microsoft Visual C/C++ Express を利用することが可能ですので，
http://www.microsoft.com/japan/msdn/vstudio/express/
からインストーラをダウンロードし，指示に従いインストールして下さい． GLUT は以下からバイナリファイルをダウンロードしてインストールして下さい．

＊GLUT: Nate Robins - OpenGL - GLUT for Win32
http://www.xmission.com/~nate/glut.html

2.2 ライブラリのインストール

KVS では前述の通り様々な外部ライブラリをサポートしています．ここでは，GPU を使った高速可視化機能を利用するための GLEW のインストールについて説明します．Linux では GLUT 同様パッケージ管理ツールを用いて，Mac では MacPorts (http://www.macports.org/)を用いて，Windows では GLEW の Web ページ（ http://glew.sourceforge.net/ ）からバイナリファイルをダウンロードしてインストールすることができます．または，Linx/Mac では GLEW の Web ページからソースをダウンロードし解凍し，解凍したフォルダ内で

$ make
$ sudo make install

とすればコンパイルとインストールが完成します．この場合，ライブラリのインストール先は『/usr/lib』となります．

2.3．環境変数の準備

KVS を利用するためには，環境変数『KVS_DIR』を設定する必要があります．また，KVS でインストールされるアプリケーションを使うため，$KVS_DIR/bin を path に追加します．

Linx/Mac の場合

ここでは，KVS のインストール先をホームディレクトリ内の local ディレクトリとします．また，使っているシェルにあわせて以下の(i)または(ii)いずれかを設定して下さい．

(i) bash の場合
~/.bashrc に以下を追記する．

export KVS_DIR=$HOME/local/kvs
export PATH=$KVS_DIR/bin:$PATH

(ii) tcsh の場合
~/.tcshrc に以下を追記する.

setenv KVS_DIR $HOME/local/kvs
setenv PATH $KVS_DIR/bin:${PATH}

Windows の場合

Windows の場合は，システムのプロパティから設定します.「システムのプロパティ」→「詳細設定」→「環境変数」に以下を記述します．ここでは KVS のインストール先を C: 直下の kvs ディレクトリとします.

変数：「KVS_DIR」　値：「C: ¥kvs」
変数：「PATH」　値：「%PATH%;%KVS_DIR%¥bin」

2.4 KVS のダウンロード

KVS は日々進化しているツールです．最新のバージョンを容易にアップデートするために，ここでは，バージョン管理システム subversion を使ってソースをダウンロードします．ダウンロードの仕方は以下の通りです.

$ svn　checkout　http://kvs.googlecode.com/svn/trunk/　kvs-src

これにより KVS のソースファイルが kvs-src フォルダ内にダウンロードされます.
（補足）svn がタイムアウトになる場合
ネットワークのプロキシが設定されている場合，subversion のプロキシを設定して下さい.
ホームディレクトリにある .subversion ディレクトリ内の servers というファイルを開きます.
ファイル内の [global] 以下にある
http-proxy-host = defaultproxy.whatever.com
http-proxy-port = 7000
のコメントを外し，正しいプロキシのアドレスとポート番号を入力して下さい.

例　京都大学の場合）

http-proxy-host = proxy.kuins.net

http-proxy-port = 8080

例　立命館大学の場合）

http-proxy-host = proxy.ritsumei.ac.jp

http-proxy-port = 3128

作業フォルダに戻って再度　svn を実行すればダウンロードができるようになっています.

subversion は Linux や Mac，Windows ならば Cygwin 環境内にインストールされています．ない場合は http://code.google.com/p/kvs/downloads/list より直接ダウンロードして下さい.

2.5 KVS コンパイルとインストール

KVS のコンパイル及びインストール方法について説明します．前節でダウンロードして作成されたディレクトリ内（ここでは kvs-src）に移動して下さい．また，GLEW ライブラリを利用するために，『kvs.conf』内の変数

KVS_SUPPORT_GLEW　を　1　にします.

```
KVS_SUPPORT_CAVE = 0
KVS_SUPPORT_CUDA = 0
KVS_SUPPORT_GLEW = 1              <-- ここを　1 に変える
KVS_SUPPORT_GLUT = 1
KVS_SUPPORT_OPENCV = 0
KVS_SUPPORT_QT = 0
KVS_SUPPORT_SAGE = 0
KVS_SUPPORT_OPENCABIN = 0
```

Linux/Mac の場合

コンパイル方法はソースを kvs-src ディレクトリ内で make コマンドを実行し，エラーなくコンパイルできればインストール (make install)を実行します ．

```
$ make
$ make install
```

以上で先に設定した KVS_DIR ディレクトリにインストールされます.

Windows の場合

Windows では make の代わりに nmake コマンドを実行します．Microsoft Visual Studio のコマンドプロンプト上で以下を実行すればコンパイル及びインストールが完了します.

```
> cd C: ¥kvs
> nmake
> nmake install
```

３．KVS コマンドアプリケーション

KVS をインストールすると以下のコマンドラインアプリケーションがインストールされます．本章ではこのアプリケーションについて説明します.

- kvsmake: KVS を利用したプログラム向けのコンパイルコマンド
- kvscheck: KVS のバージョンやコンパイラの情報などをチェックするためのコマンド
- kvsconv: KVS で可視化を行うためのデータ変換コマンド
- kvsview: KVS で読み込み可能なデータを簡単に可視化するためのコマンド

3.1 KV プログラムのコンパイル

KVS では，KVS ライブラリ群を使用してプログラムをコンパイルするための Makefile を自動生成できます．KVS を利用するために必要な情報（インクルードパスおよびライブラリパスの設定やライブラリファイルのリンクなど）を組み込んだ Makefile を自動生成し，実行ファイルを作成します.

使用方法

kvsmake [options] <project_name/make_options>

Option について

-h : ヘルプメッセーを表示する

-g : プロジェクト名を指定し Makefile を生成する

-G : カレントディレクトリ名をプロジェクト名として Makefile を生成する

-v : VC 向けのプロジェクトファイルを生成する

例 3.1 プロジェクト名をカレントディレクトリ名としてコンパイル）

この例では KVS サンプルプログラムから水素原子の電子密度分布を表すボリュームデータを作成します．ダウンロードした KVS のソースファイル「kvs-src」中の「Example/Application/Hydrogen」へ移動してコンパイル・実行してみましょう.

```
$ cd Example/Application/Hydrogen
$ kvsmake -G
    （カレントディレクトリ名「Hydrogen」を実行ファイル名とるす Makefile の作成）
$ kvsmake
    （コンパイルができると実行ファイル「Hydrogen」が生成される）
$ ./ Hydrogen
    （水素原子の電子密度分布を表すボリュームデータ hydrogen.kvsml が作成される）
```

3.2 KVS 利用環境のチェック

KVS のバージョンやコンパイラの情報などをチェックするためのコマンド．KVS のバージョンや利用しているコンパイラの情報などを標準出力に表示します．また，KVS がサポートしている他のライブラリ（GLUT, Qt, SAGE など）のバージョン情報や，KVS で読み込み可能なデータファイルのチェックも行うことができます.

使用方法

kvscheck [options] <input value>

Option について

-h : ヘルプメッセージを表示する
-version : KVS のバージョンを表示する
-platform : プラットフォーム情報を表示する
-compiler : コンパイラ情報を表示する
-support : サポートしている他ライブラリの情報を表示する
-opengl : OpenGL に関する情報を表示する
-sizeof : データ型のバイト長を表示する
-file : データファイルの情報を表示する

例：１　KVS のバージョンをチェック）

```
$ kvscheck -version
KVS version: 1.0.0
```

例：２　現在作業しているプラットフォームの情報を表示）

```
$ kvscheck -platform
Platform:     Mac OS X
CPU:            AMD64 (64 bits)
Cores:         4 core(s)
Byte-order: Little endian
```

例：３　利用するコンパイラに関する情報を表示）

```
$ kvscheck -compiler
Compiler: GNU C/C++ (4.2.1)
```

3.3 データ変換

KVS で読み込み可能なデータを対象としてデータ変換を行う．KVS フォーマット（KVSML については付録を参照）へはこのコマンドを利用して変換することができる．

使用方法

Usage: kvsconv [options] <input value>

Option について

-h　　：ヘルプメッセージを表示する

-help　　：変換コマンドごとのヘルプメッセージを表示する

-fld2kvsml　: AVS Field データから KVSML データへ変換する

-ucd2kvsml　: AVS UCD データから KVSML データへ変換する

-img2img　：画像データの変換を行う

例：１　-fld2kvsml のオプションを表示する）

$ kvsconv -help fld2kvsml

Usage: kvsconv -fld2kvsml [options] <input data file>

Options:

-help　　: Output help message.

-output　　: Output filename. (default: <input_basename>.<output_extension>)

-fld2kvsml : AVS Field data to KVSML Structured volume object.

-e　　: External data file. (optional)

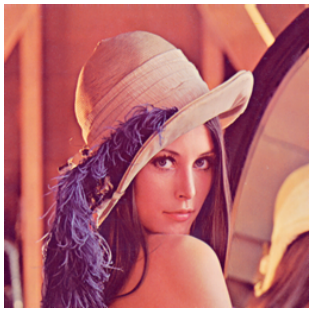
-b　　: Data file as binary. (optional)

例：2　AVS Field データ(lobster.fld)を KVSML データ（lobster.kvsml）に変換）

$ kvsconv -fld2kvsml lobster.fld -output lobster.kvsml

例：３　画像(lenna.bmp)をグレースケール化（lenna_gray.bmp））

$ kvsconv -img2img lenna.bmp -g 0 -output lenna_gray.bmp



lenna.bmp　　lenna_gray.bmp

3.4 可視化ツール

KVS で読み込み可能なデータに対して，指定する手法で可視化した結果を表示します．手法を指定しない場合がファイル形式に応じて可視化パイプラインを構築し表示します．

使用方法

Usage: kvsview [options] <input data file>

Option について

-h	: ヘルプメッセージを表示する
-help	: 可視化ツールごとのヘルプメッセージを表示する
-PointRenderer	: ポイントデータに対して PointRenderer を利用して表示する
-LineRenderer	: ラインデータに対して LineRenderer を利用して表示する
-PolygonRenderer	: ポリゴンデータに対して PolygonRenderer を利用して表示する
-Isosurface	: ボリュームデータに対して等値面抽出結果を表示する。
-SlicePlane	: ボリュームデータに対して任意断面を表示する。
-OrthoSlice	: ボリュームデータに対して直交断面を表示する。
-TransferFunction	: 伝達関数（カラーマップ）を表示する。
-ExtractEdges	: ボリュームデータに対して格子エッジを表示する。
-ExtractVertices	: ボリュームデータに対して格子点を表示する。
-ExternalFaces	: ボリュームデータに対してデータ境界面（最外表面）を表示する
-RayCastingRenderer	: ボリュームデータに対してレイキャスティングでの可視化結果を表示する
-ParticleVolumeRenderer	: ボリュームデータに対して PBVR での可視化結果を表示する
-Histogram	: ヒストグラムを表示する

４．KVS プログラムの作成

前章では，KVS コマンドアプリケーションを使ったボリュームデータの可視化方法について説明しました．本章では，本格的な KVS プログラムの作成を目的として，KVS の基本的な利用方法について説明します．KVS を使ってプログラムを作成するには，KVS で定義されているクラスごとにヘッダを呼ぶ必要があります．また，KVS が提供するクラスは全て kvs という名前空間に属しています．本章では，行列ベクトルクラスの使い方と，KVS プログラムで扱うオブジェクトについて，さらに，可視化対象データの読み込み，適切な可視化処理を施して Renderer を用いてディスプレイ上に３次元表示するまでまでの流れについて説明します．

4.1 行列/ベクトルクラス

物体を表示するためにはベクトルや行列の演算は必須となります．KVS では，2〜4 次元の行列/ベクトルクラスが独自に定義されており，各クラスで用意されている四則演算メソッドを使って行列/ベクトルの演算を容易に行うことができます．行列やベクトルクラスは使用する次数 $(N = 2, 3, 4)$ と型 $(T = \mathrm{i}, \mathrm{ui}, \mathrm{f}, \mathrm{d})$ に応じて

ベクトルのクラスのためのヘッダ　　#include <kvs/Vector*N*>

行列クラスのためのヘッダ　　#include <kvs/Matrix*NN*>

を呼ぶ必要があります．また，変数を宣言する場合には，

ベクトル型変数 x　　kvs::Vector*NT*　x;

行列型変数 A　　kvs::Matrix*NNT*　A;

で宣言します．

例 4.1 ３次元ベクトルの演算

２つの３次元ベクトル：

$\boldsymbol{x} = [1, 0, 0]^{\mathrm{T}}, \boldsymbol{y} = [2, 1, 1]^{\mathrm{T}}$

を用いて和，差，内積，外積を計算するプログラムを作成します．

1) 作業ディレクトリに「Vector」というディレクトリを作成して，移動して下さい．

```
$ mkdir  Vector
$ cd Vector
```

2) 「main.cpp」というファイルを作成して下さい．ここでは，3 次元ベクトルを使用するために <kvs/Vector3>をインクルードします．

```
#include <kvs/Vector3>
int main( void )
{
    return ( 0 );
}
```

3) ***x*** を 3 次元 double 型のベクトルとして宣言します．クラス名は kvs::Vector3d となり，ここで，3 は次元数，d は double 型を表しています．

```
    kvs::Vector3d x;
```

4) 定義した変数に値を代入します．kvs::Vector クラスでは C 言語の配列と同様に添字演算子[]を用いて値を代入することができます．

```
    x[0] = 1.0; x[1] = 0.0; x[2] = 0.0;
```

5) kvs::Vector クラスではコンストラクタを用いて変数宣言と同時に値を代入することができます．

```
    kvs::Vector3d y( 2.0, 1.0, 1.0 );
```

6) 和，差，内積，外積を計算します．kvs::Vector クラスにメソッドとして定義してあるため，それらを利用して簡単に計算することができます．

```
    // 和
    kvs::Vector3d sum = x + y;
    // 差
    kvs::Vector3d dif = x - y;
    // 内積
    double    prd = x.dot( y );
    // 外積
```

```
kvs::Vector3d crs = x.cross( y );
```

7) ベクトルの出力をするための演算子<< も定義されています.

```
std::cout << x << std::endl;
```

8) 以上をまとめると以下のようなプログラムになります.

```
#include <kvs/Vector3>

int main( void )
{
    kvs::Vector3d x;
    x[0] = 1.0; x[1] = 0.0; x[2] = 0.0;
    kvs::Vector3d y( 2.0, 1.0, 1.0 );

    // 和
    kvs::Vector3d sum = x + y;
    std::cout << "x + y = ( " << sum << " )"<< std::endl;
    // 差
    kvs::Vector3d dif = x - y;
    std::cout << "x - y = ( " << dif << " )"<< std::endl;
    // 内積
    double    prd = x.dot( y );
    std::cout << "x dot y =    " << prd << std::endl;
    // 外積
    kvs::Vector3d crs = x.cross( y );
    std::cout << "x cross y = ( " << crs << " )"<< std::endl;

    return ( 0 );
}
```

9) 8)で作成したプログラムをコンパイルします．前章で説明した通り，KVS を利用したコンパイルを行うためにまず"kvsmake -G"を実行し Makefile.kvs というメイクファイルを作成しま

す．その後，"kvsmake"を実行するとディレクトリ名「Vector」と同名の実行ファイルが作成されます．

```
$ kvsmake -G
$ kvsmake
$ ./Vector                    ※Windows 環境では Vector.exe となります
```

以上のように，KVS ではベクトルを扱うことができます．KVS では double 型 3 次元ベクトル kvs::Vector3d の他に，型としては i は int 型，ui は unsigned int 型，f は float 型を宣言することができます．また，メソッドとしては和，差などの演算の他にベクトルの長さを表す length() や長さ 1 のベクトルに正規化するための normalize() 等のメソッドが用意されています．

例 4.2 行列の演算

2 次形式：

$2x^2 + 2y^2 + 2z^2 + 2xy + 2yz + 2zx = 3$

の標準系を求めよう．

1) 作業ディレクトリに「QuadricSurface」というディレクトリを作成して，移動して下さい．

```
$ mkdir    QuadricSurface
$ cd QuadricSurface
```

2) 「main.cpp」というファイルを作成して下さい．ここでは，3×3 行列と 3 次元ベクトルを使用するために<kvs/Matrix33>と<kvs/Vector3>をインクルードします．

```
#include <kvs/Vector3>
#include <kvs/Matrix33>

int main( void )
{
   return ( 0 );
}
```

3) 2 次形式は行列とベクトルを用いて：

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = 3$$

で表すことができます．ここで，実対称行列

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

を double 型の 3 次正方行列として宣言します．クラス名は kvs::Matrix33d となり，ここで，3 は行数及び列数，d は double 型を表しています．kvs::Matrix クラスは，C 言語と同様に A[0][0] = 1; のように添字演算子[]を使うこともでき，また，kvs::Vector クラスと同様にコンストラクタを用いて変数宣言と同時に値を代入することもできます．

```
kvs::Matrix33d A( 2, 1, 1,
                  1, 2, 1,
                  1, 1, 2 );
```

4) 行列 A の固有値 $\lambda_1 = \lambda_2 = 1$，$\lambda_3 = 4$ より，正規直交な固有ベクトル

$$\boldsymbol{p}_1 = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \\ 0 \end{bmatrix}, \quad \boldsymbol{p}_2 = \begin{bmatrix} 1/\sqrt{6} \\ 1/\sqrt{6} \\ 2/\sqrt{6} \end{bmatrix}, \quad \boldsymbol{p}_3 = \begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}$$

を 3 次元 double 型のベクトルとして宣言します．これらの固有ベクトルを列ベクトルとして並べた直交行列 P

$$P = \begin{bmatrix} \boldsymbol{p}_1 & \boldsymbol{p}_2 & \boldsymbol{p}_3 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{6} & 1/\sqrt{3} \\ -1/\sqrt{2} & 1/\sqrt{6} & 1/\sqrt{3} \\ 0 & -2/\sqrt{6} & 1/\sqrt{3} \end{bmatrix}$$

を double 型の 3 次正方行列 kvs::Matrix33d として宣言します．各要素は kvs::Vector クラスに定義されている x, y, z 成分を戻り値とするメソッド x(), y(), z()を利用します．

```
// 固有ベクトル
kvs::Vector3d p1 ( 1/sqrt(2), -1/sqrt(2),  0.0 );
kvs::Vector3d p2 ( 1/sqrt(6), 1/sqrt(6), -2/sqrt(6) );
kvs::Vector3d p3 ( 1/sqrt(3), 1/sqrt(3), 1/sqrt(3) );
// 直交行列
```

```
kvs::Matrix33d P ( p1.x(), p2.x(), p3.x(),
                   p1.y(), p2.y(), p3.y(),
                   p1.z(), p2.z(), p3.z() );
```

5) $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$ とおけば，3)で記述した 2 次形式は

$$\begin{bmatrix} X & Y & Z \end{bmatrix} P^{\mathrm{T}} A P \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = 3$$

であらわすことができます．ここで，kvs::Matrix クラスには転置列を計算するメソッド transpose()が定義されていますが，行列自身を上書きしますのでコピーを保存しておきます．

```
kvs::Matrix33d P_tmp = P;
```

6) $P^{\mathrm{T}}AP$ を計算します．kvs::Matrix クラスには行列の出力をするための演算子<< も定義されています．

```
kvs::Matrix33d L = P. transpose() * A * P_tmp;
std::cout << L << std::endl;
```

7) $P^{\mathrm{T}}AP = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{bmatrix}$ より，この 2 次形式の標準系は：

$X^2 + Y^2 + 4Z^2 = 3$

となり，楕円形を表していることが分かります．以上をまとめると以下のようなプログラムになります．

```
#include <kvs/Vector3>
#include <kvs/Matrix33>

int main( int argc, char** argv )
{
```

```
    kvs::Matrix33d A( 2, 1, 1,
                      1, 2, 1,
                      1, 1, 2 );
    // 固有ベクトル
    kvs::Vector3d p1 ( 1/sqrt(2), -1/sqrt(2),    0.0 );
    kvs::Vector3d p2 ( 1/sqrt(6), 1/sqrt(6), -2/sqrt(6) );
    kvs::Vector3d p3 ( 1/sqrt(3), 1/sqrt(3), 1/sqrt(3) );
    // 直交行列
    kvs::Matrix33d P ( p1.x(), p2.x(), p3.x(),
                       p1.y(), p2.y(), p3.y(),
                       p1.z(), p2.z(), p3.z() );

    kvs::Matrix33d P_tmp = P;
    kvs::Matrix33d L = P.transpose() * A * P_tmp;
    std::cout << L << std::endl;
    std::cout << L[0][0] << " X^2 + " << L[1][1] << " Y^2 + " << L[2][2] << " Z^2 = 3 " << std::endl;
    return ( 0 );
}
```

8) 7)で作成したプログラムをコンパイルします．メイクファイルを作成（kvsmake -G）し，"kvsmake"を実行するとディレクトリ名「QuadricSurface」と同名の実行ファイルが作成されます．

```
$ kvsmake -G
$ kvsmake
$ ./ QuadricSurface
```

実行すると，対角行列と標準形が出力されます．

以上のようにして行列の演算を行うことができます．行列演算クラスとしては他に，LU 分解（<kvs/ LUDecomposer>）や QR 分解（<kvs/ QRDecomposer>），固有値分解（<kvs/ EigenDecomposer>）等のクラスも用意されています．

問題 4.1　標準系を求めよう

2 次形式：

$$2x^2 + 2y^2 + 5z^2 - 2xy + 4yz - 4zx = 2$$

の標準系を求めよう.

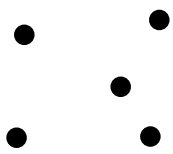
ヒント：2 次形式の行列とベクトルを用いて

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} 2 & -1 & -2 \\ -1 & 2 & 2 \\ -2 & 2 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = 2$$

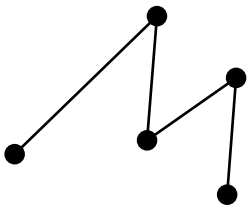
で表すことができます．また，固有値分解クラスを利用して直交行列を求めることもできます.

4.2 オブジェクト

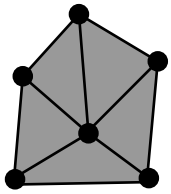
KVS 内で効率的にデータを扱うために様々なデータ構造（object）が用意されています．KVS で定義されている Object には図 4.1 に示すような点データ（PointObject），線データ（LineObject），面データ（PolygonObject）やボリュームデータを扱うための構造型ボリュームデータ（StructuredVolumeObject），非構造型ボリュームデータ（UnstructuredVolumeObject）等が用意されています.



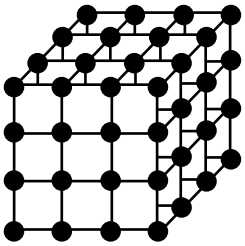
PointObject
点データ



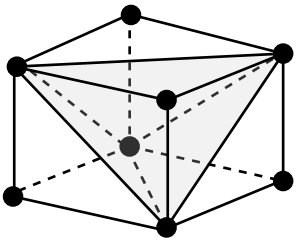
LineObject
線データ



PolygonObject
面データ



StructuredVolumeObject
構造型ボリュームデータ



UnstructuredVolumeObject
非構造型ボリュームデータ

図 4.1 KVS で扱うデータ構造

例 4.3 構造型ボリュームデータの生成

関数値：

$f(x, y, z) = 3 - x^2 - y^2 - 4z^2$

をフィールド値にもつ構造型ボリュームデータを作成し，AVF Field ファイルとして出力します．AVS Field ファイルは次のようなファイル形式です．

```
ndim    = 3                    // 空間の次元数
dim1    = 2                    // x 軸方向のグリッド数
dim2    = 2                    // y 軸方向のグリッド数
dim3    = 2                    // z 軸方向のグリッド数
nspace = 3                     // 各軸の次元サイズ
veclen = 1                     // 各要素のデータ数（スカラーデータならば 1）
data    = double               // データの型
field   = uniform              // ボリュームデータのフィールドタイプ（座標情報）
^L^L…以下フィールド値….
```

KVS で定義されている構造型ボリュームデータ（StructuredVolumeObject）クラスはメンバ変数としてグリッド数（kvs::Vector3ui），グリッドタイプ（enum 型で Uniform, Rectilinear, Curvilinear, Irregular があります），各要素のデータ数（size_t），フィールドデータ（kvs::AnyValueArray），グリッド座標（kvs::ValueArray<float>）をもちます．ここでは，特に kvs::AnyValueArray クラスのフィールドデータの登録の仕方について説明します．

1) 作業ディレクトリに「CreateField」というディレクトリを作成して，移動して下さい．

```
$ mkdir  CreateField
$ cd CreateField
```

2) 「main.cpp」というファイルを作成して下さい．関数値評価のためにインライン関数を作成します．

```
inline float func( float x, float y, float z) {
  return ( 3.0 - x*x - y*y - 4.0*z*z );
```

```
}

int main( void ) {
return ( 0 );
}
```

3) 構造型ボリュームデータのグリッド数を unsigned int 型の 3 次元ベクトルとして宣言します．ここでは，データのグリッド数は dim1 = dim2 = dim3 = 64 とします．

```
kvs::UInt32 gridNum = 64;
const kvs::Vector3ui resol(gridNum, gridNum, gridNum);
```

4) フィールドデータは kvs::AnyValueArray クラスを使います．kvs::AnyValueArray クラスにはデータ型やデータサイズ，データを格納する動的配列をメンバにもっています．メモリの割り当てには allocate()というメソッドが定義されており，ここでは 64×64×64 の float 型データとしてメモリを確保します．

```
kvs::AnyValueArray data;
if ( !data.allocate<float>( gridNum * gridNum * gridNum) ) {
    std::cout << "Cannot allocate memory for the data." << std::endl;
    exit(1);
}
```

5) 4) で宣言した kvs::AnyValueArray クラスではフィールドデータを直接配列に格納することはできません．代わりに，データへのポインタを使って操作します．

```
float* pdata = data.pointer<float>();
```

6) x, y, z の定義域を[-2, 2]として，グリッドサイズが各々64 となる格子点上で関数値を評価します．評価した関数値は pdata を使って値を格納していきます．

```
float min = -2.0, max = 2.0;
float dt = (float)(max - min)/(float)(gridNum-1);
```

```
kvs::UInt64 index = 0;
for(int i=0; i< 64; i++) {
   for(int j = 0; j<64; j++) {
      for(int k=0; k<64; k++) {
         float x = (float)min + (float)k*dt;
         float y = (float)min + (float)j*dt;
         float z = (float)min + (float)i*dt;
         // 関数値を評価して data へ格納
         pdata[ index++ ] =    func(x, y, z) ;
      }
   }
}
```

7) 構造型ボリュームデータ（StructuredVolumeObject）を作成します．ここでは，kvs::StructuredVolumeObject のコンストラクタを用いて変数宣言と同時に値を代入します．コンストラクタの要求する引数はグリッドサイズ（const kvs::Vector3ui），各要素のデータ数（const size_t），フィールドデータ値（const kvs::AnyValueArray）です．

```
size_t veclen = 1.0;
kvs::StructuredVolumeObject *volume =
   new kvs::StructuredVolumeObject ( resol,          // グリッドサイズ
                                     veclen,         // 各要素のデータ数
                                     data      );    // ボリュームデータ
```

8) 構造型ボリュームデータを AVS Field ファイルへエクスポートします．AVSField ファイルへエクスポートするために kvs::StructuredVolumeExporter クラスを使用し，kvs::AVSField クラスのインスタンスを作成します．kvs::StructuredVolumeExporter はファイルフォーマットタイプ（ここでは，kvs::AVSField）を指定し，引数に kvs::StructuredVolumeObject を渡します．

```
kvs::AVSField* field2 =
   new kvs::StructuredVolumeExporter<kvs::AVSField>( volume );
if( !field2 ) {      // エクスポートできたかチェックする
   std::cout << "Cannot export Resized volume data." <<std::endl;
   delete volume;
```

```
    return (false);
  }
```

出力できるファイルフォーマットタイプとしては他に，KVS 独自のファイル形式 KVSML(kvs::KVSMLObjectStructuredVolume)を指定することができます.

9) ファイルフォーマットを定義したクラスにはファイル出力のためのメソッド write()が定義されています．ファイル名は引数として渡します.

```
  field->write( "test.fld" )
```

10) 使用した KVS クラスをインクルードします．ここでは，kvs::Vector3ui, kvs::StructuredVolumeObject, kvs::StructuredVolumeExporter, kvs::AVSField を使いました.

```
#include <kvs/Vector3>
#include <kvs/StructuredVolumeObject>
#include <kvs/StructuredVolumeExporter>
#include <kvs/AVSField>
```

11) まとめると以下のようなプログラムになります.

```
#include <kvs/Vector3>
#include <kvs/StructuredVolumeObject>
#include <kvs/StructuredVolumeExporter>
#include <kvs/AVSField>

inline float func( float x, float y, float z) {
  return ( 3.0 - x*x - y*y - 4.0*z*z );
}

int main( int argc, char** argv )
{
  // グリッドサイズ
  kvs::UInt32 gridNum = 64;
```

```
const kvs::Vector3ui resol(gridNum, gridNum, gridNum);
// ボリュームデータの値を格納するためのメモリの割り当て
kvs::AnyValueArray data;
if ( !data.allocate<float>( gridNum * gridNum * gridNum) ) {
   std::cout << "Cannot allocate memory for the data." << std::endl;
   exit(1);
}
// data にを操作するためのポインタ変数
float* pdata = data.pointer<float>();

float min = -2.0, max = 2.0;
float dt = (float)(max - min)/(float)(gridNum-1);

kvs::UInt64 index = 0;
for(int i=0; i< 64; i++) {
   for(int j = 0; j<64; j++) {
      for(int k=0; k<64; k++) {
         float x = (float)min + (float)k*dt;
         float y = (float)min + (float)j*dt;
         float z = (float)min + (float)i*dt;
         // 関数値を評価して data へ格納
         pdata[ index++ ] =    func(x, y, z) ;
      }
   }
}

size_t veclen = 1.0;
// 構造型ボリュームオブジェクトの作成
kvs::StructuredVolumeObject *volume =
   new kvs::StructuredVolumeObject ( resol,
                                     veclen,
                                     data       );

// VolumeObject を AVSField に Export
```

```
  kvs::AVSField* field =
     new kvs::StructuredVolumeExporter<kvs::AVSField>( volume );
  if( !field ) {
     std::cout << "Cannot export Resized volume data." <<std::endl;
     delete volume;
     return (false);
  }
  // ファイルへ出力
  if ( !field->write( "test.fld" ) ) {       // ファイル書き込みができたかチェックする
     std::cout << "Cannot write to the file as AVS field format." << std::endl;
     delete volume;
     return( false );
  }

  return ( 0 );

 }
```

12) 11)で作成したプログラムをコンパイルします．メイクファイルを作成（kvsmake -G）し，"kvsmake"を実行するとディレクトリ名「CreateField」と同名の実行ファイルが作成されます．

```
$ kvsmake -G
$ kvsmake
$ ./ CreateField
```

実行すると，「test.fld」というファイル名の AVS Field ファイルが作成されます．どのようなデータが作成されたのか確認するため, KVS コマンドアプリケーション kvsview を使って零等値面を描画して確かめてみましょう．

```
$ kvsview -Isosurface -l 0 test.fld
```

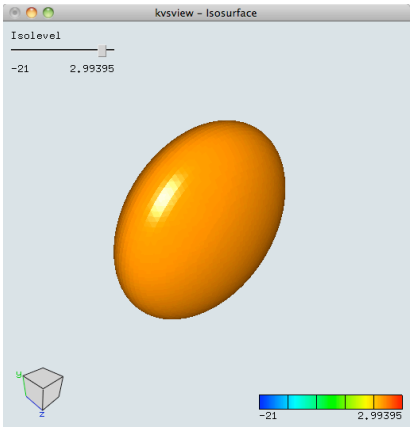


図 4.2　作成したボリュームデータの零等値面描画

例 4.4 面データの可視化

四面体を描画するプログラムを作成します．KVS オブジェクトの作成方法は例 4.3 と同様の手順で面データ（PolygonObject）の作成を行います．PolugonObject はメンバ変数として，ポリゴンの種類（enum 型で Triangle, Quadrangle があります），色の種類（enum 型で VertexColor, PolygonColor があります），法線の種類（enum 型で VertexNormal, PolygonNormal があります）接続情報（kvs::ValueArray<kvs::UInt32>），透明度（kvs::ValueArray<kvs::UInt8>），頂点座標（kvs::ValueArray<kvs::Real32>），色（kvs::ValueArray<kvs::UInt8> ），法線（kvs::ValueArray<kvs::Real32>）があります．この例では作成したオブジェクトを可視化の為に定義されている Screen クラスへ登録し，描画を行います．

1) 作業ディレクトリに「ViewPolygon」というディレクトリを作成して，移動して下さい．
   $ mkdir　ViewPolygon
   $ cd ViewPolygon

2) 「main.cpp」というファイルを作成して下さい．作成する四面体の頂点座標，頂点色，頂点法線，接続情報を配列に代入します．また，OpenGL による描画のために main 関数には引数をもたせて下さい．

```
int main( int argc, char** argv )
{
  // 頂点座標
  kvs::Real32 vertex[] = {
      -1.0, -1.0, -1.0,          // 第 0 頂点座標
```

```
        1.0,  0.0,  0.0,          // 第 1 頂点座標
        0.0,  1.0,  0.0,          // 第 2 頂点座標
        0.0,  0.0,  1.0,          // 第 3 頂点座標
    };
    // 頂点色
    kvs::UInt8 vertCol[] = {
          0,    0,    0,          // 第 0 頂点の色
        255,    0,    0,          // 第 1 頂点の色
          0,  255,    0,          // 第 2 頂点の色
          0,    0,  255,          // 第 3 頂点の色
    };
    // 頂点法線
    kvs::Real32 norm[] = {
     -0.577,   -0.577,  -0.577,       // 第 0 頂点の法線
      0.577,   -0.577,  -0.577,       // 第 1 頂点の法線
     -0.577,    0.577,  -0.577,       // 第 2 頂点の法線
     -0.577,   -0.577,   0.577,       // 第 3 頂点の法線
    };
    // 接続情報
    kvs::UInt32 tri[] = {
       0, 2, 1,
       0, 1, 3,
       0, 3, 2,
       3, 2, 1,
    };

return ( 0 );
}
```

3) 頂点や法線などを kvs::ValueArray へキャストして PolygonObjecto へ登録するために, 配列の値を std::vector として作成しなおします.

```
    std::vector<kvs::Real32>coords( vertex, vertex + 12 );
    std::vector<kvs::UInt32>connects( tri, tri + 12 );
```

```
std::vector<kvs::UInt8>colors( vertCol, vertCol + 12 );
std::vector<kvs::Real32>normals( norm, norm + 12 );
```

4) 面データ（PolygonObject）を作成します．ここでは，kvs::PolygonObject のコンストラクタを用いて変数宣言と同時に値を代入します．コンストラクタの要求する引数は頂点座標（kvs::ValueArray<kvs::Real32>），接続情報（kvs::ValueArray<kvs::UInt32>），色（kvs::ValueArray<kvs::UInt8> ），透明度（kvs::ValueArray<kvs::UInt8>），法線（kvs::ValueArray<kvs::Real32>），ポリゴンの種類，色の種類，法線の種類です． ここでは，ポリゴンの種類を Triangle，色と法線の情報はそれぞれ頂点に付加されているとし，また，透明度は常に一定であるとして，kvs::UInt8 型の変数を 1 つ渡しています．

```
kvs::PolygonObject* object =
    new kvs::PolygonObject(
            kvs::ValueArray<kvs::Real32>( coords ),      // 頂点座標
            kvs::ValueArray<kvs::UInt32>( connects ),    // 接続情報
            kvs::ValueArray<kvs::UInt8>( colors ),       // 色
            255,                                         // 透明度
            kvs::ValueArray<kvs::Real32>( normals ),     // 法線
            kvs::PolygonObject::Triangle,                // ポリゴンの種類
            kvs::PolygonObject::VertexColor,             // 色の種類
            kvs::PolygonObject::VertexNormal );          // 法線の種類
```

5) 描画する範囲を決定するために，座標値の最大値と最小値をあらかじめ計算しておきます．

```
object->updateMinMaxCoords();
```

6) OpenGL による可視化のために，GLUT の初期化を行います．KVS では，GLUT の関数を簡単に使用できるように kvs::glut::Application クラスによって定義することができます．

```
kvs::glut::Application app( argc, argv );
```

7) OpenGL では，ウィンドウの生成やサイズ・背景色の設定を行う関数を呼び，それぞれ設定を行います．KVS ではそれらの関数は kvs::glut::Screen のメソッドして定義されています．また，キーイベントやマウスイベントについても KVS 内で定義されており，ユーザが設定

を行わずともキーやマウスの操作が GLUT ウィンドウ上で可能となります．ここではウィンドウの左上の位置とウィンドウサイズを設定するメソッド setGeometry( )とウィンドウタイトルを設定するメソッド setTitle()使用しています．また，可視化したいオブジェクトは registerObject( )を用いてスクリーンクラスに登録します．その後，show()を呼ぶと適当な Renderer を用いてオブジェクトが描画されます．Renderer については次節で説明します．最後に，イベント待ち状態にするために，kvs::glut::Application クラスのメソッド run()を呼びます．

```
// Screen クラスの生成と設定
kvs::glut::Screen screen( &app );
screen.registerObject( object );
screen.setGeometry( 0, 0, 512, 512 );
screen.setTitle( "Polygon");
screen.show();
// イベント待ち
return( app.run() );
```

8) 使用した KVS クラスをインクルードします．ここでは，kvs::PolygonObject，kvs::glut::Screen，kvs::glut::Application を使いました．

```
#include <kvs/PolygonObject>
#include <kvs/glut/Application>
#include <kvs/glut/Screen>
```

9) まとめると以下のようなプログラムになります．

```
#include <kvs/PolygonObject>
#include <kvs/glut/Application>
#include <kvs/glut/Screen>

int main( int argc, char** argv )
{
  // 頂点座標
  kvs::Real32 vertex[] = {
```

```
    -1.0, -1.0, -1.0,
     1.0,  0.0,  0.0,
     0.0,  1.0,  0.0,
     0.0,  0.0,  1.0,
};
// 頂点色
kvs::UInt8 vertCol[] = {
       0,    0,    0,
     255,    0,    0,
       0,  255,    0,
       0,    0,  255,
};
// 頂点法線
kvs::Real32 norm[] = {
 -0.577,   -0.577,   -0.577,
  0.577,   -0.577,   -0.577,
 -0.577,    0.577,   -0.577,
 -0.577,   -0.577,    0.577,
};
// 接続情報
kvs::UInt32 tri[] = {
   0, 2, 1,
   0, 1, 3,
   0, 3, 2,
   3, 2, 1,
};

// std::vector コンテナに格納
std::vector<kvs::Real32>coords( vertex, vertex + 12 );
std::vector<kvs::UInt32>connects( tri, tri + 12 );
std::vector<kvs::UInt8>colors( vertCol, vertCol + 12 );
std::vector<kvs::Real32>normals( norm, norm + 12 );

// ポリゴンオブジェクトの作成
```

```
    kvs::PolygonObject* object =
        new kvs::PolygonObject(
                    kvs::ValueArray<kvs::Real32>( coords ),
                    kvs::ValueArray<kvs::UInt32>( connects ),
                    kvs::ValueArray<kvs::UInt8>( colors ),
                    255,
                    kvs::ValueArray<kvs::Real32>( normals ),
                    kvs::PolygonObject::Triangle,          //  PolygonType
                    kvs::PolygonObject::VertexColor,       // ColorType
                    kvs::PolygonObject::VertexNormal );    // NormalType
    // 頂点の最大値と最小値を計算
    object->updateMinMaxCoords();

    // glut の初期化
    kvs::glut::Application app( argc, argv );
    // Screen クラスの生成と設定
    kvs::glut::Screen screen( &app );
    screen.registerObject( object );
    screen.setGeometry( 0, 0, 512, 512 );
    screen.setTitle( "Polygon");
    screen.show();
    // イベント待ち
    return( app.run() );
}
```

10) 9)で作成したプログラムをコンパイルします．メイクファイルを作成（kvsmake -G）し，"kvsmake"を実行するとディレクトリ名「ViewPolygon」と同名の実行ファイルが作成されます．

```
$ kvsmake -G
$ kvsmake
$ ./ ViewPolygon
```

実行すると，図 4.3 のように四面体が描画されます．

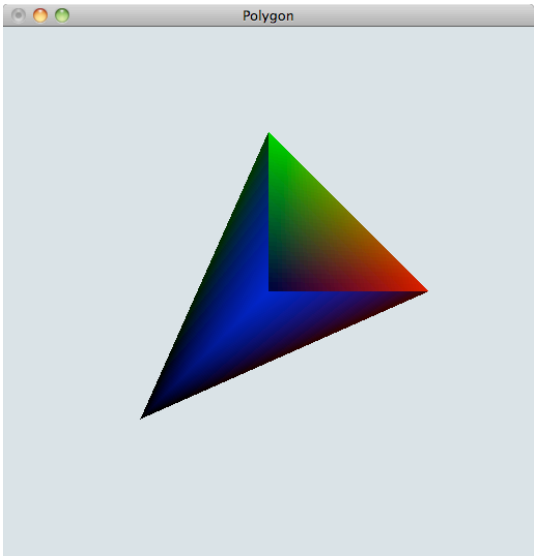


図 4.3 PolygonObject の可視化

問題 4.2　構造型ボリュームデータの表示

例 4.3 で作成した楕円形をフィールド値にもつ構造型ボリュームデータを表示させるプログラムを作成せよ．

4.3 可視化パイプライン

前節でも説明した通り，KVS 内部では KVS で定義されている適切なデータ構造（Object）を通してデータのインポート/エクスポートや可視化を行ってきました． これらの Object はデータ削減や補間，サンプリングなど適切な処理を施して Object を変換し，ディスプレイに表示していくことがあります．KVS を用いてこれらの処理を用いるためには，基本的な可視化パイプラインに沿ってプログラムを作成する必要がります．KVS における可視化パイラインは図 4.4 の様になります．可視化パイプラインの構成は以下の通りです．

1)　Importer

データを KVS 可視化パイプラインで扱えるデータ構造（Object）に変換します． たとえば，点群データを PointImporter で読み込み，PointObject へ変換，構造型ボリュームデータを StructuredVolumeImporter で読み込み StructuredVolumeObject へ変換するなど．Importer には他に，画像を読み込む ImageImporter，線データを読み込む LineImporter，面データを読み込む PolygonImporter，非構造型ボリュームデータを読み込む UnstructuredVolumeImporter がある．

2) Filter

次の Mapper のための前処理を行うために，Import された Object に対してデータ削減や補間処理を行います．例えば三重線形補間があります.

3) Mapper

Object に対して，Renderer のために点やポリゴンなどの幾何形状データに変換する処理を行います．例えば，Object に対してマーチングキューブ法で等値面を生成する場合は，MarchingCube を通してして PolygonObject に変換します．他にも，スライス面の作成（SlicePlane）やメトロポリスサンプリング（MetropolisSampling），等値面の作成（Isosurface）などがあります.

4) Renderer

Mapper で処理された Object に対して，適切な描画方法で画像を生成します．PolygonObject ならば PolygonRenderer, PointObject に対しては PointRenderer などがあります．他にも，VolumeObject に対して RayCastingRenderer が用意されています.

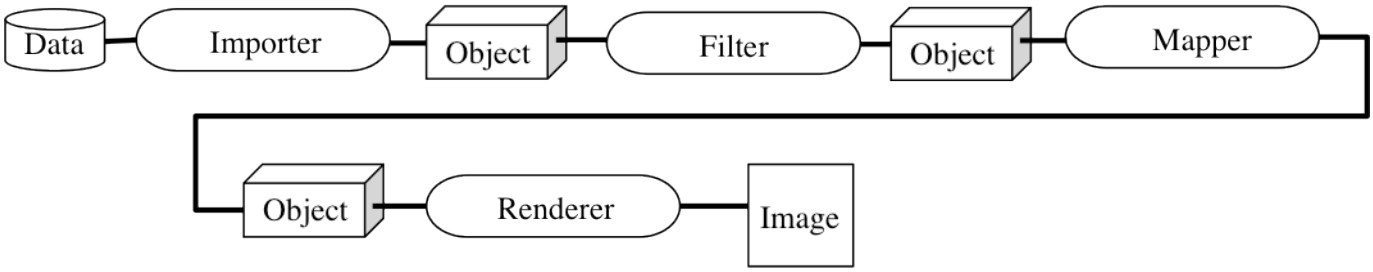


図 4.4　KVS における可視化パイプライン

例 4.5 ボリュームデータをレイキャスティングを用いてレンダリングしてみよう.

構造型ボリュームデータをインポートしてレイキャスティング法でレンダリングしてみましょう．ここで用いる可視化パイプラインは図 4.5 にあるように Importer に StructuredVolumeImporter，Renderer に RayCastingRenderer を用います．データをそのまま使うため，Filter と Mapper は個々では使用しません.

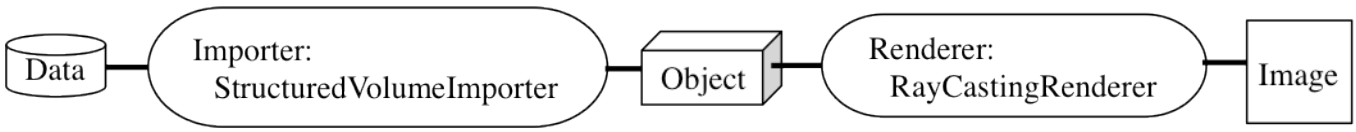


図 4.5　例 4.5 で用いる可視化パイプライン

1) 作業ディレクトリに「RayCasting」というディレクトリを作成して，移動して下さい.

```
$ mkdir    RayCasting
$ cd RayCasting
```

2) 「main.cpp」というファイルを作成して下さい．プログラム実行の際にコマンドライン引数としてファイルを読み込むようにするため，引数のチェックをします．

```
int main( int argc, char** argv )
{
    if( !(argc == 2 ) ) {
    std::cerr << "USAGE (1): ./RayCasting volume_data" << std::endl;
    exit(1) ;
  }
return ( 0 );
}
```

3) コマンドライン引数として与えられたファイル名をインポート（kvs::StructuredVolumeImporter）して構造型ボリュームデータ（kvs::StructuredVolumeObject）へ変換します．また，このとき正しいファイルが読み込めたのかチェックします．

```
    kvs::StructuredVolumeObject* object
      = new kvs::StructuredVolumeImporter( std::string( argv[1] ) );
    if ( !object )        // ファイルが正しく読み込めたのかチェック
    {
        kvsMessageError( "Cannot create a structured volume object." );
        return( false );
    }
```

4) Renderer を作成します．ここでは，Renderer に kvs::RayCastingRenderer を選択しています．また，正しく Renderer が作成できたかをチェックします．

```
    kvs::RayCastingRenderer* renderer = new kvs::RayCastingRenderer();
    if ( !renderer )
    {
        kvsMessageError( "Cannot create a ray casting renderer." );
```

```
        delete object;
        return( false );
    }
```

5) GLUT による可視化処理を行います．例 4.4 と同様に kvs::glut::Application と kvs::glut::Screen により，ウィンドウの初期化や記イベントの登録を行います．ここでは，Screen クラスのメソッド registerObject へ Object だけではなく，Renderer も登録していることに注意して下さい．これにより，ユーザが作成・設定した Renderer を用いて描画処理を行うことが可能になります．

```
    kvs::glut::Application app( argc, argv );
    kvs::glut::Screen screen( &app );
    // Screen クラスにオブジェクトと Renderer を登録
    screen.registerObject( object, renderer );
    screen.setGeometry( 0, 0, 512, 512 );
    screen.setTitle( "RayCastingRenderer" );
    screen.show();

    return( app.run() );
```

6) 使用した KVS クラスをインクルードします．ここでは，kvs::StructuredVolumeImporter, kvs::StructuredVolumeObject, kvs::RayCastingRenderer, kvs::glut::Screen, kvs::glut::Application を使いました．

```
#include <kvs/StructuredVolumeObject>
#include <kvs/StructuredVolumeImporter>
#include <kvs/RayCastingRenderer>
#include <kvs/glut/Application>
#include <kvs/glut/Screen>
```

7) まとめると，以下のようなプログラムになります．

```
#include <kvs/StructuredVolumeObject>
#include <kvs/StructuredVolumeImporter>
```

```
#include <kvs/RayCastingRenderer>
#include <kvs/glut/Application>
#include <kvs/glut/Screen>

int main( int argc, char** argv )
{
    if( !(argc == 2 ) ) {
    std::cerr << "USAGE (1): ./RayCasting volume_data" << std::endl;
    exit(1) ;
  }
    kvs::StructuredVolumeObject* object
      = new kvs::StructuredVolumeImporter( std::string( argv[1] ) );
    if ( !object )
    {
        kvsMessageError( "Cannot create a structured volume object." );
        return( false );
    }

    kvs::RayCastingRenderer* renderer = new kvs::RayCastingRenderer();
    if ( !renderer )
    {
        kvsMessageError( "Cannot create a ray casting renderer." );
        delete object;
        return( false );
    }

    // glut の初期化
    kvs::glut::Application app( argc, argv );
    // Screen クラスの生成と設定
    kvs::glut::Screen screen( &app );
    // Screen クラスにオブジェクトと Renderer を登録
    screen.registerObject( object, renderer );
    screen.setGeometry( 0, 0, 512, 512 );
    screen.setTitle( "RayCastingRenderer" );
```

```
    screen.show();

    return( app.run() );
}
```

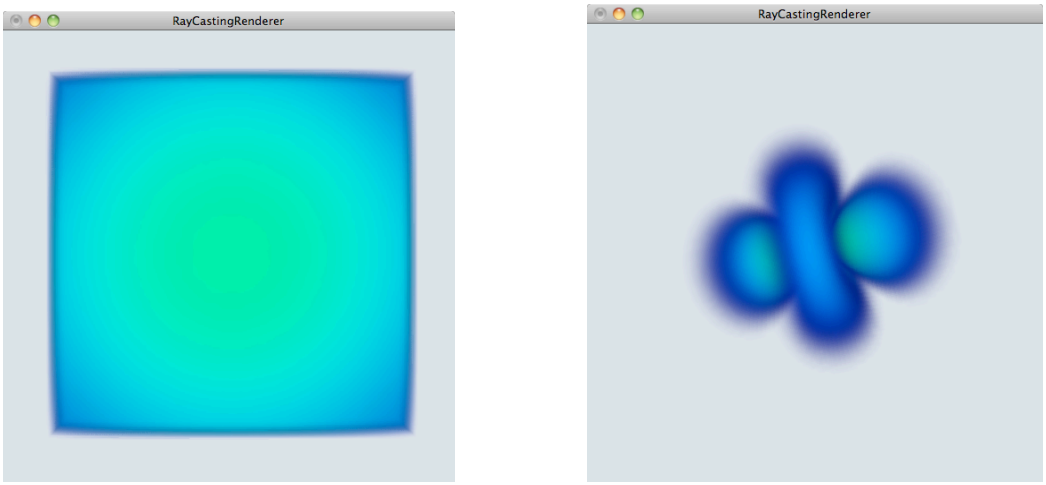
8) 7)で作成したプログラムをコンパイルします．メイクファイルを作成（kvsmake -G）し，"kvsmake"を実行するとディレクトリ名「RayCasting」と同名の実行ファイルが作成されます．

```
$ kvsmake -G
$ kvsmake
```

例 4.3 で作成したファイル「test.fld」をカレントディレクトリへコピーし，実行してみましょう．

```
$ ./ RayCasting test.fld
```

実行すると，図 4.6 (a)のような可視化結果が描画されます． 他にも，例 3.1 で作成した hydrogen.kvsml を可視化すると同図(b) のような結果が得られます．



(a) test.fld (b) hydrogen.kvsml

図 4.6 RayCassting 結果

問題 4.3 GPU を利用した高速可視化ライブラリを使ってみよう．

インストールの際に追加ライブラリとして GLEW をインストールしている場合には GLEW を利用した GPU による高速な Renderer を使うことができます．例 4.5 で作成したプログラムを改良し，kvs::glew::RayCastingRenderer を使ってレンダラを置き換えてみよう．

例 4.6 粒子ベース・ボリュームレンダリングの実装

粒子ベース・ボリュームレンダリングは「不透明発光粒子モデル」に基づくボリューム・レンダリング手法です．ソート処理の必要がないため，細雨の大きなボリュームデータの可視化に有利です．粒子ベース・ボリュームレンダリングは伝達関数で定まる不透明度分布と一致するサンプリング点分布を確率的に生成し，ピクセルごとの隠点処理と輝度値アンサンブル平均によって半透明画像の生成を実現します．粒子ベース・ボリュームレンダリングの実現手順は図 4.7 に示すとおり，粒子を生成・投影し，画像のアンサンブル平均（平均回数を以降リピートレベルと呼びます）を行って半透明画像を生成します．

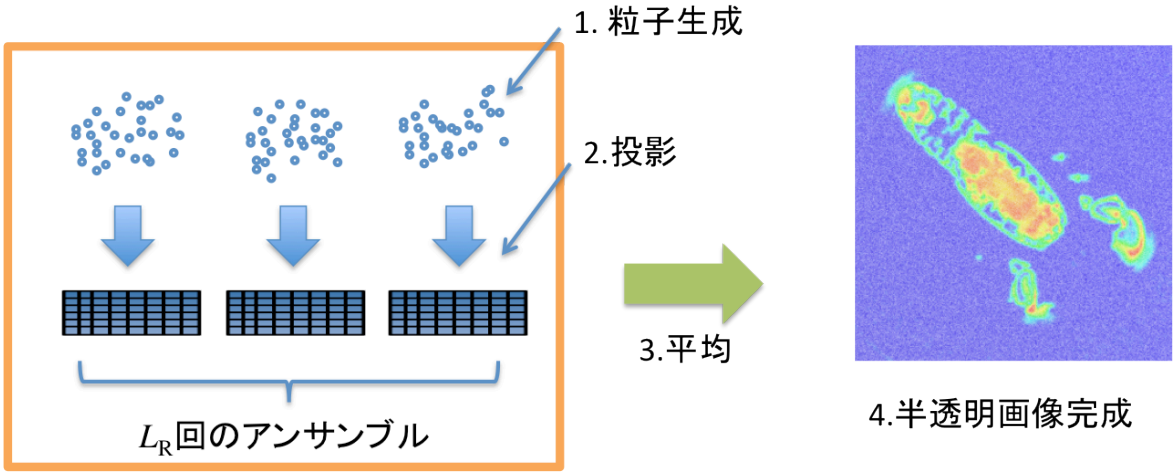


図 4.7 粒子ベース・ボリュームレンダリング手順

粒子ベース・ボリュームレンダリングを実現するための可視化パイプラインは図 4.8 のようになります．ここでは Importer に StructuredVolumeImporter を使用し，Mapper として CellByCellMetropolisSampling を用いて点群データを生成します．最後に Renderer に GLEW を用いた ParticleVolumeRenderer を使用します．

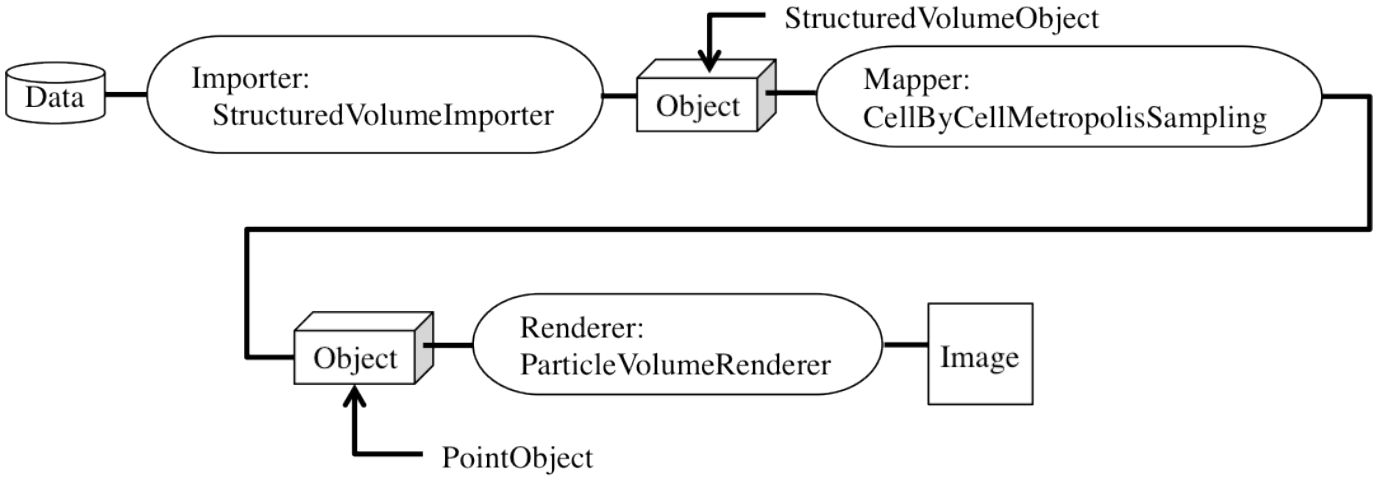


図 4.8 粒子ベース・ボリュームレンダリングのための可視化パイプライン

1) 作業ディレクトリに「PBVR」というディレクトリを作成して，移動して下さい．

```
$ mkdir    PBVR
$ cd PBVR
```

2) 「main.cpp」というファイルを作成して下さい．例 4.4 と同様にプログラム実行の際にコマンドライン引数としてファイルを読み込むようにするため，引数のチェックをします．また，このプログラムでは伝達関数を引数として受け取る場合も想定して作成しています．

```
int main( int argc, char** argv )
{
   if( !(argc == 2 || argc == 3) ) {
      std::cerr << "USAGE (1): PBVR volume_data" ;
         std::cerr << std::endl;
      std::cerr << "USAGE (2): PBVR volume_data transfer_fnction.kvsml" ;
         std::cerr << std::endl;
      exit(1) ;
   }
return ( 0 );
}
```

3) コマンドライン引数として与えられたファイル名をインポート（kvs::StructuredVolumeImporter）して構造型ボリュームデータ（kvs::StructuredVolumeObject）へ変換します．

```
kvs::StructuredVolumeObject* volume =
    new kvs::StructuredVolumeImporter( std::string( argv[1] ));
if ( !volume )          {
        kvsMessageError( "Cannot create a structured volume object." );
        return( false );
    }
```

4) 伝達関数（kvs::TransferFunction）を作成します．伝達関数は，ボリュームデータから粒子を生成するために必要な関数です．コマンドライン引数としてデータを受け取っていない場合には標準の伝達関数（図 4.9 参照）を用います．

```
kvs::TransferFunction* transfer_function = 0 ;

if ( argc == 2 ) {
    // rainbow color (default)
    transfer_function =    new kvs::TransferFunction ();
} else
if ( argc == 3 ) {
    transfer_function = new kvs::TransferFunction ( argv[2] );
}
```

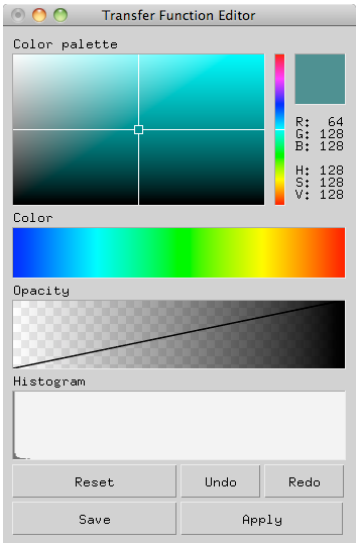


図 4.9 標準の伝達関数

5) Mapper として kvs::CellByCellMetropolisSampling を用いてサンプリングを行い，構造型ボリュームデータ（kvs::StructuredVolumeObject）から点群データ（kvs::PointObject）へ変換します．点群生成のための kvs::CellByCellMetropolisSampling クラスのコンストラクタの引数は，構造型ボリュームデータ（kvs::StructuredVolumeObject），サブピクセルレベル（size_t 型，ここではリピートレベルの平方根），サンプリングステップ（float 型），伝達関数（kvs::TransferFunction），セルの幅（float)です．

```
//リピートレベルとサブピクセルレベル
int repeat_level = 9 ;
int subpixel_level;
subpixel_level = (int) sqrt ( (double)repeat_level );
//   サンプリングステップとセルの幅
const float sampling_step = 0.5f;
const float object_depth = 1.0f;
// サンプリング
kvs::PointObject* object = new kvs::CellByCellMetropolisSampling(
                volume,                 // ボリュームデータ
                subpixel_level,         // サブピクセルレベル
                sampling_step,          // サンプリングステップ
                *transfer_function,     // 伝達関数
                object_depth );         // セルの幅
```

6) サンプリングによって生成された粒子数は kvs::PointObject クラスのもつメソッド nvertices() で確認することができます．設定したリピートレベルと伝達関数に応じて生成される粒子数が決定されます.

```
int num_particles = object->nvertices() ;
std::cout << "** Number of Particles: " << num_particles << std::endl;
```

7) Renderer を作成します．ここでは，Renderer に kvs::glew::ParticleVolumeRenderer を選択しています. また, アンサンブル平均をおこなうため, リピートレベルを Renderer に登録します. また，6) のサンプリング時とは違い，描画の際にはサブピクセル法を用いませんので, Renderer へ登録するサブピクセルレベルは 1 とします.

```
kvs::glew::ParticleVolumeRenderer* renderer =
   new kvs::glew::ParticleVolumeRenderer();        // Renderer の作成
renderer->setSubpixelLevel( 1 );                   // サブピクセルレベルを 1 にする
renderer->setRepetitionLevel ( repeat_level );     // リピートレベルを登録
```

8) GLUT による可視化処理を行います． kvs::glut::Application と kvs::glut::Screen により，ウィンドウの初期化や記イベントの登録を行い, Screen クラスのメソッド registerObject へ Object と Renderer を登録します.

```
kvs::glut::Application app( argc, argv );
// Screen クラスの生成と設定
kvs::glut::Screen screen( &app );
screen.setGeometry( 0, 0, 512, 512 );
// Object と Renderer の登録
screen.registerObject( object, renderer );
screen.setTitle( " Particle Volume Renderer " ) ;
screen.show();

return( app.run() );
```

9) 使用した KVS クラスをインクルードします．ここでは，kvs::StructuredVolumeImporter, kvs::StructuredVolumeObject, kvs::TransferFunction, kvs::CellByCellMetropolisSampling, kvs::PointObject, kvs::glew::ParticleVolumeRenderer と描画のためのクラス kvs::glut::Application、kvs::glut::Screen を使いました.

```
#include <kvs/StructuredVolumeImporter>
#include <kvs/StructuredVolumeObject>
#include <kvs/TransferFunction>
#include <kvs/CellByCellMetropolisSampling>
#include <kvs/PointObject>
#include <kvs/glut/Application>
#include <kvs/glew/ParticleVolumeRenderer>
#include <kvs/glut/Screen>
```

10) まとめると以下のようなプログラムになります.

```
#include <kvs/StructuredVolumeImporter>
#include <kvs/StructuredVolumeObject>
#include <kvs/TransferFunction>
```

```
#include <kvs/CellByCellMetropolisSampling>
#include <kvs/PointObject>
#include <kvs/glut/Application>
#include <kvs/glew/ParticleVolumeRenderer>
#include <kvs/glut/Screen>

int main( int argc, char** argv )
{
   if( !(argc == 2 || argc == 3) ) {
      std::cerr << "USAGE (1): PBVR volume_data" ;
         std::cerr << std::endl;
      std::cerr << "USAGE (2): PBVR volume_data transfer_fnction.kvsml" ;
         std::cerr << std::endl;
      exit(1) ;
   }
   // ボリュームデータのインポート
   kvs::StructuredVolumeObject* volume =
      new kvs::StructuredVolumeImporter( std::string( argv[1] ));
 if ( !volume )          {
            kvsMessageError( "Cannot create a structured volume object." );
            return( false );
      }

   //   伝達関数の初期化
   kvs::TransferFunction* transfer_function = 0 ;
   if ( argc == 2 ) {
      // rainbow color (default)
      transfer_function =    new kvs::TransferFunction ();
   } else
   if ( argc == 3 ) {
      transfer_function = new kvs::TransferFunction ( argv[2] );
   }

    //リピートレベルとサブピクセルレベル
```

```
  int repeat_level = 9 ;
  int subpixel_level;
  subpixel_level = (int) sqrt ( (double)repeat_level );
  //   サンプリングステップとセルの幅
  const float sampling_step = 0.5f;
  const float object_depth = 1.0f;
// サンプリング
  kvs::PointObject* object = new kvs::CellByCellMetropolisSampling(
                    volume,
                    subpixel_level,
                    sampling_step,
                    *transfer_function,
                    object_depth );

  // 生成した粒子数の表示
  int num_particles = object->nvertices() ;
  std::cout << "** Number of Particles: " << num_particles << std::endl;

  // Renderer の作成
  kvs::glew::ParticleVolumeRenderer* renderer =
     new kvs::glew::ParticleVolumeRenderer();
  renderer->setSubpixelLevel( 1 );
  renderer->setRepetitionLevel ( repeat_level );

  // glut の初期化
  kvs::glut::Application app( argc, argv );
  // Screen クラスの生成と設定
  kvs::glut::Screen screen( &app );
  screen.setGeometry( 0, 0, 512, 512 );
  // Object と Renderer の登録
  screen.registerObject( object, renderer );
  screen.setTitle( " Particle Volume Renderer " ) ;
  screen.show();
```

```
    return( app.run() );

}
```

11) 10)で作成したプログラムをコンパイルします．メイクファイルを作成（kvsmake -G）し，"kvsmake"を実行するとディレクトリ名「PBVR」と同名の実行ファイルが作成されます．

```
$ kvsmake -G
$ kvsmake
```

例 3.1 で作成したファイル「hydrogen.kvsml」をカレントディレクトリへコピーし，実行してみましょう．

```
$ ./PBVR hydrogen.kvsml
```

実行すると，図 4.10 のような可視化結果が描画されます．

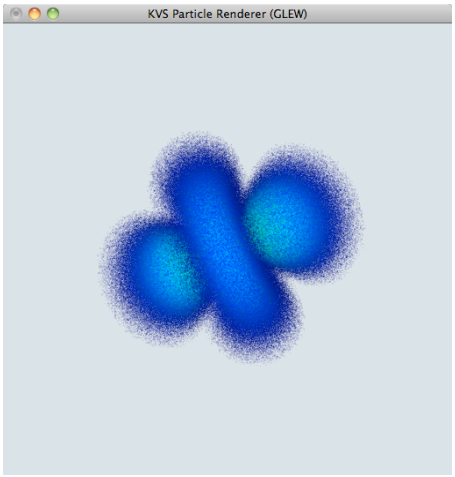


図 4.10 粒子ベース・ボリュームレンダリングによる可視化結果

問題 4.4 可視化パイプラインを使ってプログラムを作ろう．

以下のプログラムは登録された構造型ボリュームデータの一部を切り出す Filter を実装したプログラム「CropFLD (CropFLD.h, CropFLD.cpp)」である．このプログラムを用いてボリュームデータの一部を切り出すプログラムを作成せよ．

1) CropFLD.h

```
#ifndef _CropFLD_H_
#define _CropFLD_H_

#include <kvs/VolumeObjectBase>
#include <kvs/StructuredVolumeObject>
#include <kvs/AnyValueArray>
#include <kvs/Vector3>

class CropFLD : public kvs::StructuredVolumeObject
{
   kvs::Vector3ui m_grid_min;
   kvs::Vector3ui m_grid_max;

    typedef kvs::StructuredVolumeObject SuperClass;
    kvs::AnyValueArray data;

 public:
    CropFLD( const kvs::VolumeObjectBase* volume,
                const kvs::Vector3ui grid_min,
                const kvs::Vector3ui grid_max );
2)     virtual ~CropFLD( void );

 private:
    void cropVolume( const kvs::VolumeObjectBase* volume );

    template <typename T>
    void    extract( const kvs::StructuredVolumeObject* volume );

 private:
    CropFLD( void );
};

#endif
```

2) CropFLD.cpp

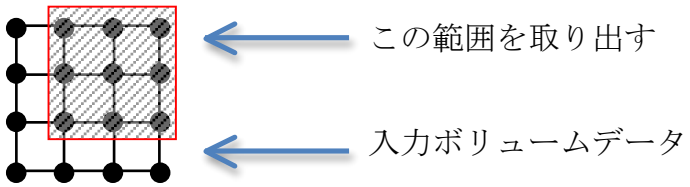
```
#include <cstdlib>
#include <kvs/MersenneTwister>
#include <kvs/TrilinearInterpolator>
#include <kvs/IgnoreUnusedVariable>
#include <kvs/AnyValueArray>
#include "CropFLD.h"

CropFLD::CropFLD(
                        const kvs::VolumeObjectBase* ORGvolume,
                        const kvs::Vector3ui grid_min,
                        const kvs::Vector3ui grid_max ):
   m_grid_min( grid_min ),
   m_grid_max( grid_max )
{

   SuperClass::setGridType( ORGvolume->gridType() );
   SuperClass::setVeclen( ORGvolume->veclen() );
   SuperClass::updateMinMaxCoords();

   this->cropVolume( ORGvolume );
}

void CropFLD::cropVolume(const kvs::VolumeObjectBase* ORGvolume)
{

   if ( ORGvolume->volumeType() == kvs::VolumeObjectBase::Structured )
      {
         // Down Cast Pointer from SuperClass
         const kvs::StructuredVolumeObject* structured_volume =
            kvs::StructuredVolumeObject::DownCast( ORGvolume );
          // データの型ごとに処理します
```

```
        const std::type_info& type = structured_volume->values().typeInfo()->type();
        if (          type == typeid( kvs::Int8     ) )
          this->extract<kvs::Int8>( structured_volume );
        else if ( type == typeid( kvs::Int16    ) )
          this->extract<kvs::Int16>( structured_volume );
        else if ( type == typeid( kvs::Int32    ) )
          this->extract<kvs::Int32>( structured_volume );
        else if ( type == typeid( kvs::Int64    ) )
          this->extract<kvs::Int64>( structured_volume );
        else if ( type == typeid( kvs::UInt8    ) )
          this->extract<kvs::UInt8>( structured_volume );
        else if ( type == typeid( kvs::UInt16 ) )
          this->extract<kvs::UInt16>( structured_volume );
        else if ( type == typeid( kvs::UInt32 ) )
          this->extract<kvs::UInt32>( structured_volume );
        else if ( type == typeid( kvs::UInt64 ) )
          this->extract<kvs::UInt64>( structured_volume );
        else if ( type == typeid( kvs::Real32 ) )
          this->extract<kvs::Real32>( structured_volume );
        else if ( type == typeid( kvs::Real64 ) )
          this->extract<kvs::Real64>( structured_volume );
        else
          {
            kvsMessageError("Unsupported data type '%s'.",
                            structured_volume->values().typeInfo()->typeName() );
          }
      }
    else // volume->volumeType() == kvs::VolumeObjectBase::Unstructured
      {       // 非構造型ボリュームデータへは対応しません
        kvsMessageError("Input volume is unStructured");
        return;
      }

}
```

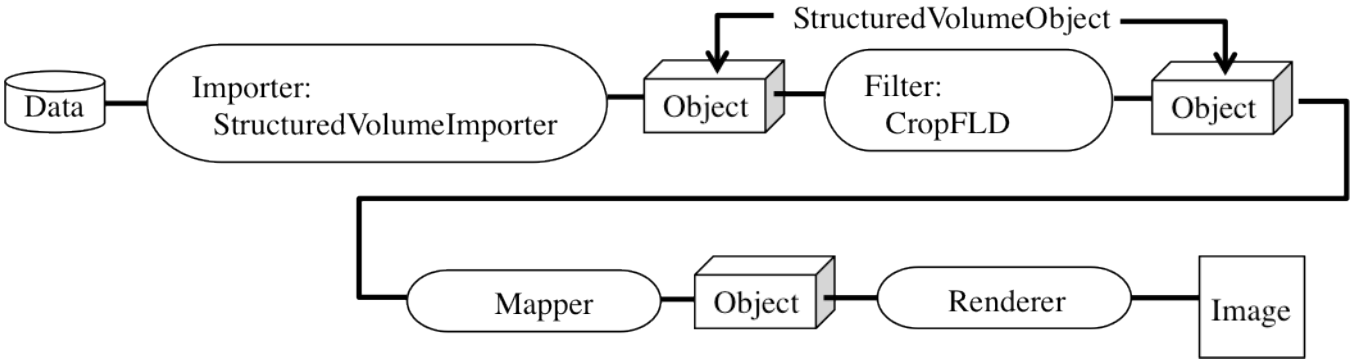
```
template <typename T>
void CropFLD::extract(const kvs::StructuredVolumeObject* ORGvolume)
{
  // 入力したボリュームデータのグリッド数
  const kvs::Vector3ui ORGncells(ORGvolume->resolution());
  const kvs::UInt64 ORGdim1 = static_cast<kvs::UInt64>( ORGncells.x() );
  const kvs::UInt64 ORGdim2 = static_cast<kvs::UInt64>( ORGncells.y() );
  const kvs::UInt64 ORGdim3 = static_cast<kvs::UInt64>( ORGncells.z() );

  kvs::Vector3ui resol;
  kvs::UInt64 nx, ny, nz;

  // 切り出した後のボリュームデータのグリッド数
  nx = m_grid_max.x() - m_grid_min.x() ;
  ny = m_grid_max.y() - m_grid_min.y() ;
  nz = m_grid_max.z() - m_grid_min.z() ;
  resol.set( nx, ny, nz);
 // グリッド数の登録
 SuperClass::setResolution( resol );
 // 入力したボリュームデータのフィールドデータを操作するためのポインタ
  const T* const ORGdata = reinterpret_cast<const T*>( ORGvolume->values().pointer() );

  kvs::UInt64 line_size   = static_cast<kvs::UInt64>(ORGvolume->nnodesPerLine());
  kvs::UInt64 slice_size = static_cast<kvs::UInt64>(ORGvolume->nnodesPerSlice());
  kvs::UInt32 ORGindex;

  //----- 切り出下後のフィールドデータを確保するためのメモリ割り当て
  if ( !data.allocate<T>( ORGdim1 * ORGdim2 * ORGdim3 ) ) {
     kvsMessageError("Cannot allocate memory for the data.");
     exit(1);
  }
  //----- deta を操作するためのポインタ
  T* pdata = data.pointer<T>();
```

```
    //---- 入力された範囲のフィールドデータを data へ登録
    kvs::UInt64 index = 0;
    for(kvs::UInt64 k = m_grid_min.z(); k < m_grid_max.z(); k++) {
        for(kvs::UInt64 j = m_grid_min.y(); j < m_grid_max.y(); j++) {
            for(kvs::UInt64 i = m_grid_min.x(); i < m_grid_max.x(); i++ ) {
                ORGindex = (i + j * line_size + k * slice_size); // Index Number of Original Volume Data
                pdata[ index++ ] = ORGdata[ORGindex];                    // Add to Data for New
FLD-Data
            }
        }
    }
    // 切り出したフィールドデータを登録
    SuperClass::setValues( data );
}

CropFLD::~CropFLD( void )
{
}

CropFLD::CropFLD( void )
{
}
```

ヒント：CropFLD クラスのコンストラクタが要求する引数は構造型ボリュームデータ（kvs::StructuredVolumeObjectBase），切り出すボリュームデータの範囲の始点（kvs::Vector3ui）と終点（kvs::Vector3ui）です.



可視化パイプラインは以下のようになります.



Filter 処理（CropFLD）をおこなった後の Object は kvs::StructuredVolumeObject のままですので，その後の Mapper や Penderer は例を参考にして作成して下さい.