Volume Extractor
開発用マニュアル

株式会社アイプランツ・システムズ

1.0 版

本資料について

目的

本資料では、Volume Extractor 3.0 をベースとして新規に機能を追加しようとする方を対象に、開発環境、機能の追加方法、デバッグ方法等を示し、最終的に機能を追加、実行できるようになることを目的としています。

前提

本資料を閲覧するにあたり、予め以下の知識について習得されていることをお勧めします。

- C++によるプログラミング
- Visual Studio 2005 による開発
- C#、.NET Framework 2.0 によるプログラミング
- OpenGL（描画方法に変更を加える場合）

構成

本資料は下記のように構成されております。

VolumeExtractor ソリューションの構成の説明

（2. クラス構成）

機能の追加方法

（3.1.プロジェクトの追加 ～ 3.5.パラメータクラスの作成）

フォーム（ダイアログ）によるボリュームデータ編集を目的とした機能の追加方法を説明します

追加した機能のメイン処理部分との関連付方法

（3.6.MainFrame への登録， 3.10.イベントの作成と登録）

追加した機能の表示処理部分との関連付方法

（3.7.WorkFrame への登録， 3.10.イベントの作成と登録）

ボリュームデータの操作方法

（3.8.ボリュームデータの操作方法 ～ 3.9.ImageData のメソッド等）

デバッグ開始までの準備

（4.ビルドとデバッグ）

リリースの方法

（5.リリース）

ファイルフォーマット等

（6.備考）

目次

1. 開発環境

Volume Extractor 3.0 の開発環境、フォルダ構成等について記します。

1.1. 環境

1.1.1. 使用開発環境

開発環境は下記の通りです。

Microsoft Visual Studio 2005 Professional Edition
　Version 8.0.50727.762　(SP.050727-7600)
Microsoft .NET Framework
　Version 2.0.50727

※ VS2005SP1 が適用された状態です。
※ 上記のバージョン以降での動作については未確認です。(2007/12/13 時点)
※ Microsoft Corp.から配布されております、"Visual C++ 2005 Express Edition"、"Visual C++ 2005 Express Edition"は対象外となります。

1.1.2. フォルダ構成

Volume Extractor 3.0 のフォルダ構成は以下の通りです。

```
VE30
  ├¥Bin_Release
  ├¥CPPSample
  ├¥CSharpSample
  ├¥dllset
  ├¥VEVoxelPaint
  ├¥VolumeExtractor
```

<u>VE30 フォルダ</u>

プロジェクト全体のルートフォルダになります。
このフォルダには、ソリューションファイルとして「VolumeExtractor.sln」が配置されています。
Visual Studio 2005 での開発の際には、このファイルを開きます。

<u>Bin_Release フォルダ</u>

実行環境のファイルを配置するフォルダです。
VolumeExtractor の開発環境では生成されない実行ファイル、設定ファイル等が配置されています。
Volume Extractor 実行時には、このフォルダ内のファイルと VolulmeExtractor の実行ファイ

ルを同じフォルダに置いて使用します。

dllset フォルダ

各種機能の DLL が配置されたフォルダです。

VolumeExtractor の実行ファイルを生成する際に自動的に読込まれますので、そのままにしておいてください。

CPPSampleフォルダ

C++ でのサンプルコードになります。

CSharpSample フォルダ

C# でのサンプルコードになります。

XY 平面、YZ 平面、XZ 平面の 3 方向からの2D 画像の生成と表示を行います。

ImageData クラス型のインスタンスの受け渡し、データ参照の際にご使用いただけます。

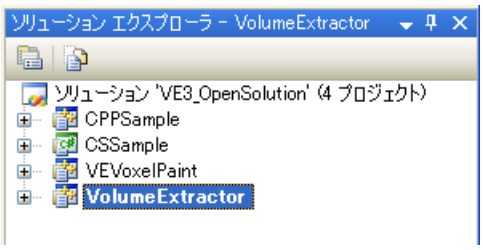
VEVoxelPaint フォルダ

VE の機能の一つ、3D 矩形塗りつぶし機能のコードです。(コードは C++で記述されております)

機能実装の際のサンプルとして参照下さい。

1.1.3. プロジェクト構成

ソリューション‘VE3_OpenSolution’の構成は下記の通りです。



CSSample

C#でのサンプルコードのプロジェクトです。

CPPSample

C++でのサンプルコードのプロジェクトです。

VEVoxelPaint

3D 矩形塗りつぶし機能のプロジェクトです。

VolumeExtractor

実行ファイル本体のプロジェクトです。

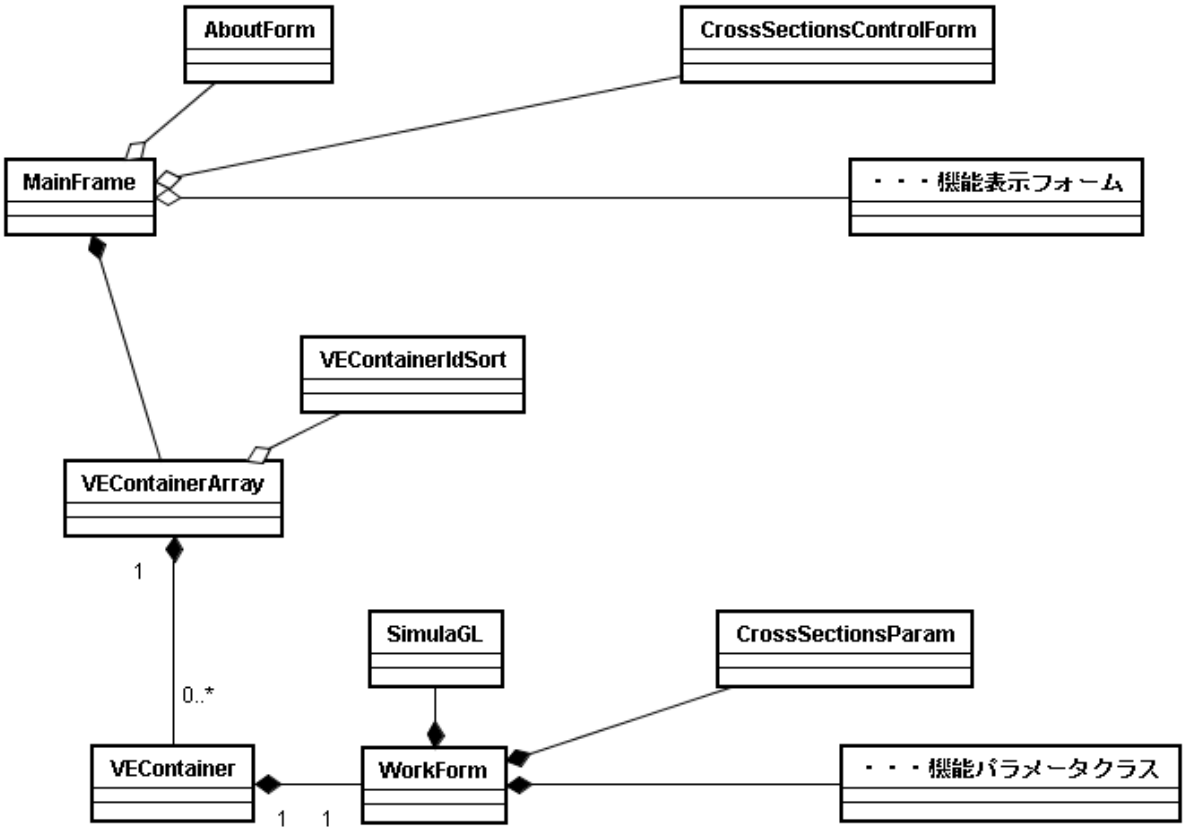
新しい機能のメニューへの追加、呼び出しなどは、このプロジェクトに対して行ないます。

2. クラス構成

2.1. VolumeExtractor のクラス構成

VolumeExtractor プロジェクトに含まれるクラスの関係を下図に示します。

実際の動作には、他のライブラリで定義されたクラスのインスタンスが MainFrame や WorkForm と関係しますが、ここでは省略いたします。



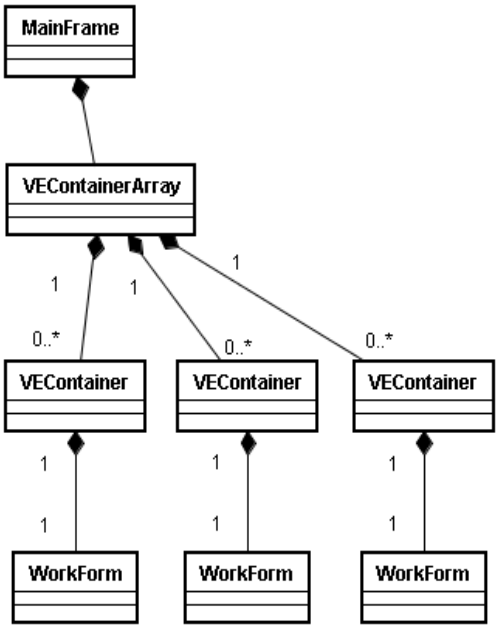
MainFrame	… メインフレームです
WorkForm	… 2D、3D 画像表示、作業用フォームクラス
AboutForm	… バージョン情報フォームクラス
CrossSectionsControlForm	… 3 断面表示フォームクラス
CrossSectionsParam	… 3 断面表示パラメータクラス
FVStateItem	… 各種機能フォームの位置情報用のアイテムクラス
FVState	… 各種機能フォームの位置情報入出力クラス
SimulaGL	… OpenGL インスタンスを表すクラス
VEContainer	… 各ボリュームデータ等の情報を内包したコンテナクラス
VEContainerArray	… 上記コンテナの配列クラス
VEContainerIdSort	… コンテナのソート用クラス
VEContainerLastActiveIdSort	… コンテナのソート用クラス

2.2. MainFrame

アプリケーションのメインフレームの処理を行なうクラスです。
各ウィンドウの管理、各種機能フォームの呼出し（ボタン押下処理）、各種機能のフォームイベントとWorkForm の処理をつなぐ役割をしています。
MainFrame では、コンテナというかたちでボリュームデータとそれに関する情報を管理しています。このコンテナを配列で管理することで、複数のボリュームの読込みと表現を実現してます。

MainFrame では、コンテナ配列「VEContainerArray」のインスタンスを保持し、新しいボリュームデータを読込むと、コンテナ「VEContainer」を作成してコンテナ配列の要素として管理します。
コンテナには、読込んだボリュームデータ（または DICOM データ等々）や、専用の WorkForm のインスタンスが登録されます。

下記は例として、3 つのボリュームデータが読込まれた時の状態を示すイメージです。
例えば、3 つのボリュームデータが読込まれていると、下記のようにそれぞれのボリュームデータに対して VEContainer と WorkForm が構成されます。



2.3. WorkForm

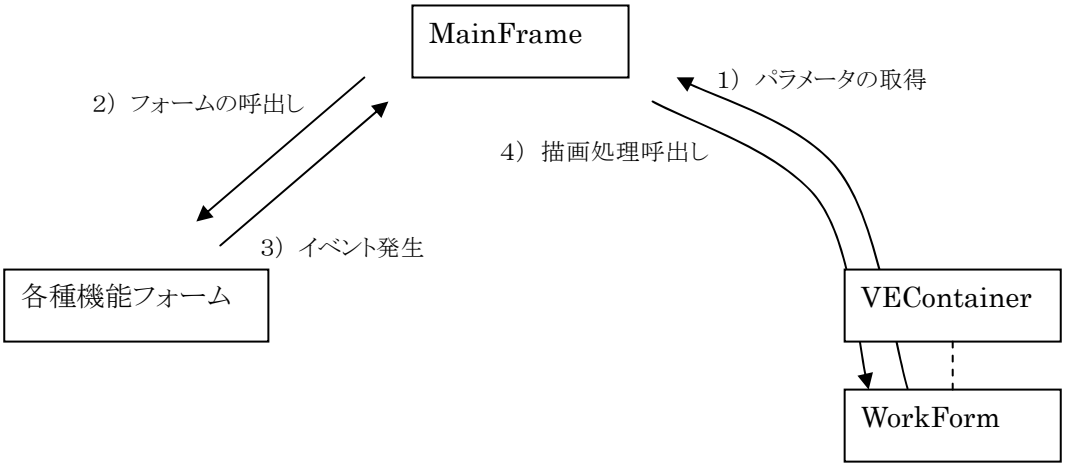
主に描画処理を行なうクラスです。
読込んだボリュームファイルの 2D、3D の描画処理が記述されます。
一つのボリュームデータに対して、一つの WorkForm インスタンスが生成されます。

WorkForm は、GUI 上に描画に関する状態を変化させるためのボタンが用意され（右図）、各種機能のパラメータのインスタンスが保持されていることからも、このクラスには基本的に描画に直接関係する処理やインスタンスを定義します。

またアプリケーション上の動作では、WorkForm と各種機能のフォームが直接連携して描画内容が変更されているように見えますが、実際には、MainFrame を介して描画処理が呼び出されています。

下記は、各種機能フォーム呼出し、描画の際の処理順序を示したイメージです。

1） MainFrame がボリュームデータや WorkForm の保持しているパラメータを取得
2） MainFrame が各種機能フォームを呼び出す
※この時必要に応じて、ボリュームデータやパラメータを渡す
3） 各種機能フォーム上での描画変更関連の処理実行時、MainFrame で予め用意したメソッドをイベントというかたちで呼び出す
4） MainFrame 上の関数から、WorkForm 上の描画処理メソッドを呼び出す



2.4. ボリュームデータと固有 ID

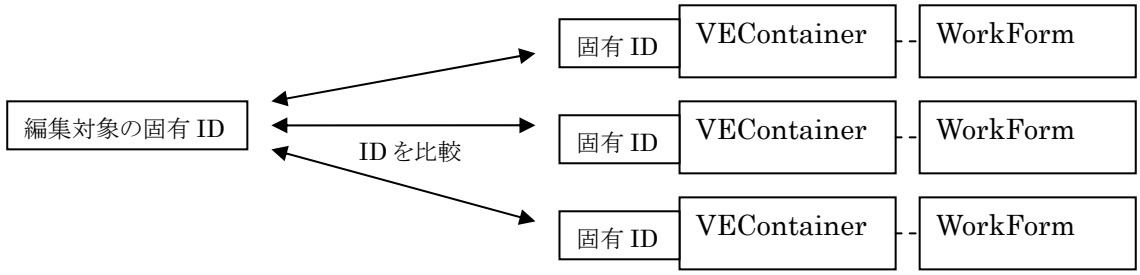
Volume Extractor では、同時に複数のボリュームデータを読込むことが可能であることから、各ボリュームデータに対して固有の ID（文字列）を設定し管理します。

この固有 ID で識別することにより、現在どのデータを表示しているのか、どのデータに対して編集を行なったのか等々を管理することができます。

この固有 ID については、新たに機能を実装する上で必要になってきますのでご注意ください。

どのデータに対して編集を行なうのかを「固有 ID」の情報を元に決定します

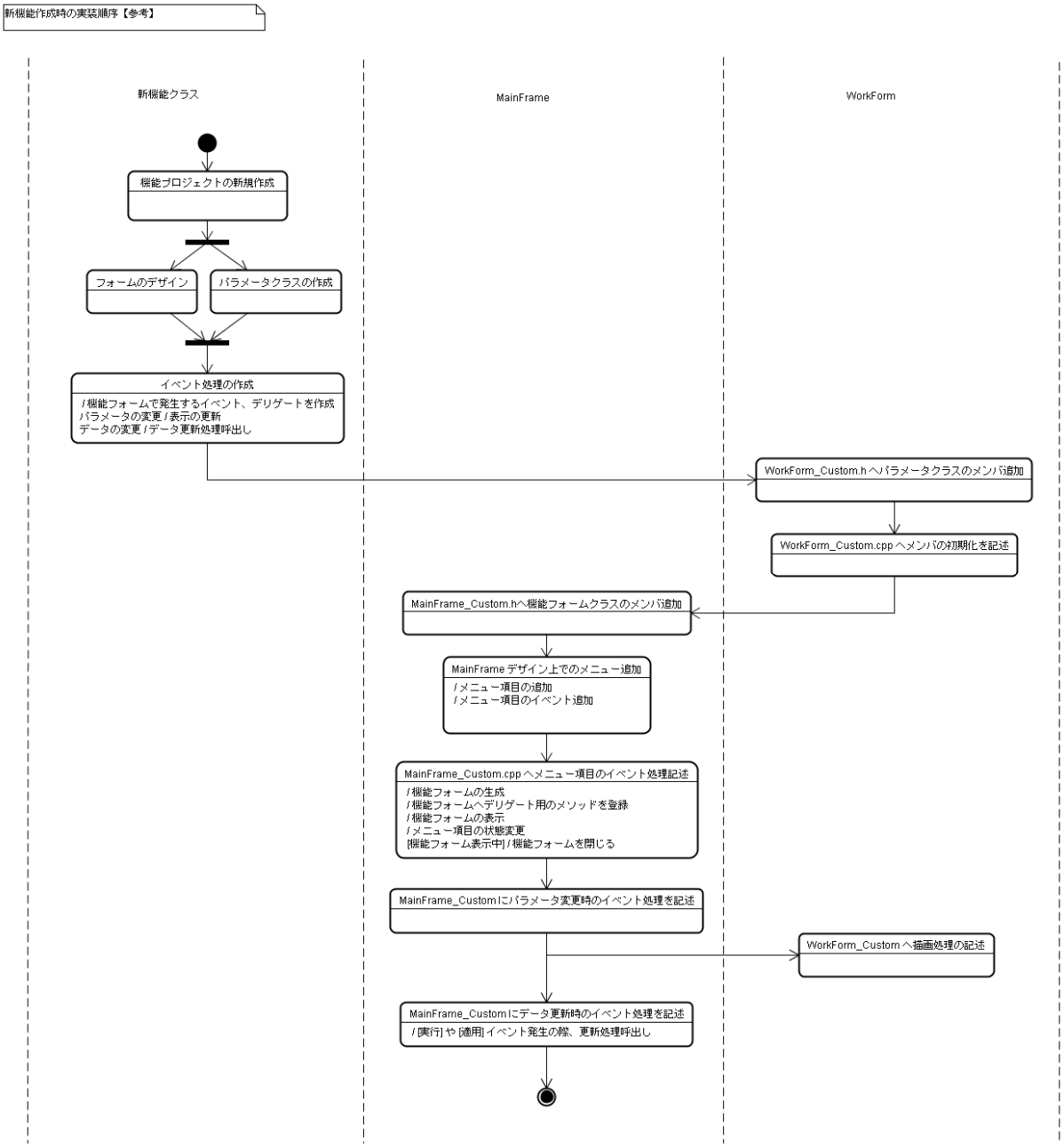
（イメージ図）



3. モジュールの作成

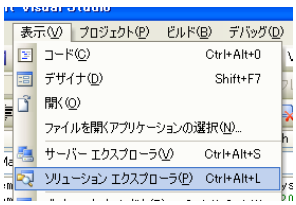
ここでは、ダイアログベースのインタフェースを作成して、Volume Extractor で機能を呼び出す方法までについて記します。新しく追加する機能は DLL ファイルとして作成し、Volume Extractor 実行ファイルから作成したライブラリファイルを呼び出す形式をとります。

主に下図の順で作成していきます。



※ 作業を効率的に行なうため、「ソリューションエクスプローラ」を開きます。

※ 「ソリューションエクスプローラ」は、[表示]－[ソリューション エクスプローラ]で表示されます。

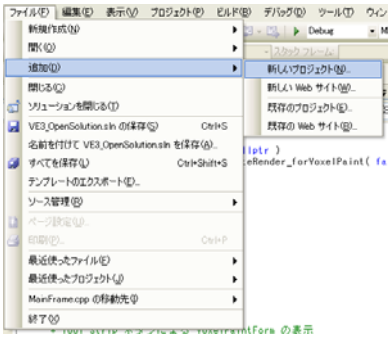


3.1. プロジェクトの追加

DLL ファイルを作成するためのプロジェクトを Volume Extractor のソリューションに追加します。

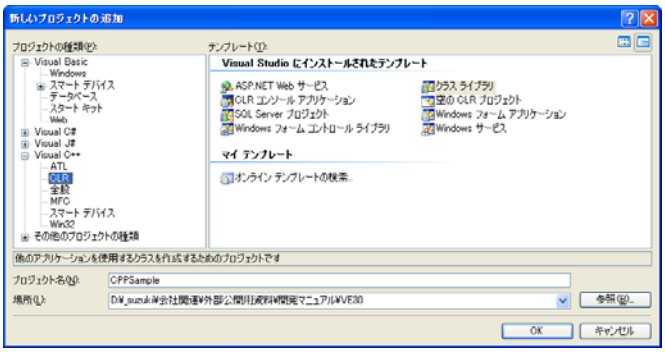
○ 手順

1） ［ファイル］－［追加］－［新しいプロジェクト］を選択して、「新しいプロジェクトの追加」ダイアログを表示します。



2） ［プロジェクトの種類］で使用言語［Visual C#］、［Visual C++］等を選択します。

3） C#の場合は、［プロジェクトの種類］で［Windows］を選択し、右側の［テンプレート］から［クラス ライブラリ］を選択します。C++の場合は、［プロジェクトの種類］で［CLR］を選択し、右側の［テンプレート］から［クラス ライブラリ］を選択します。



※新しいプロジェクト追加の際に自動で作成された“[プロジェクト名].h, ［プロジェクト名］.cpp”は、ここでは私用する予定が無いので、削除してしまっても構いません。（そのまま残しておいても構いません）

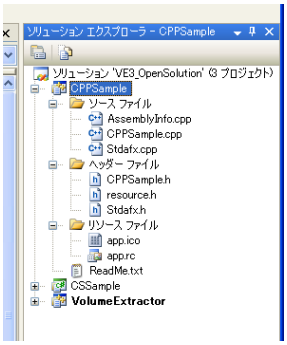
※Visual Basic、Visual J++での開発は行なっておりませんが、基本的には同様に追加可能です。

3.2. フォームの追加

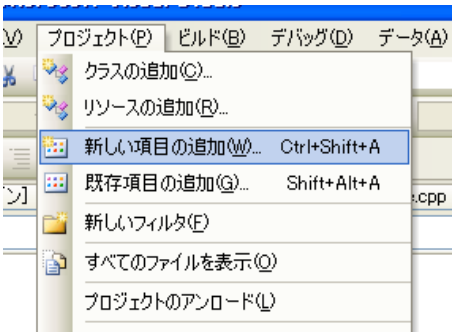
ユーザーインタフェース（UI）を追加します。

○ 手順

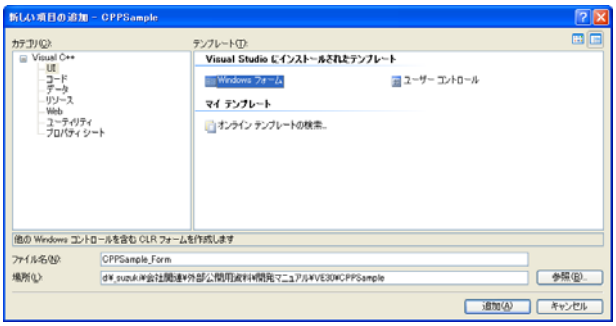
1） 「ソリューション　エクスプローラ」上で新しく追加したプロジェクトを選択します。



2) ［プロジェクト］－［新しい項目の追加］を選択します。



3) 「新しい項目の追加」ダイアログ上から［Windows フォーム］を選択します。（ファイル名は、機能や設計内容に合わせて任意の名称を指定してください。）

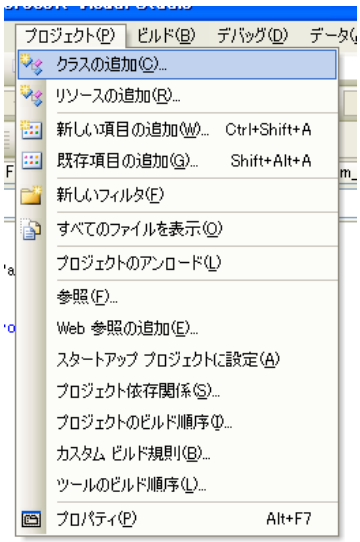


3.3. パラメータクラスの追加（新規クラスの追加）

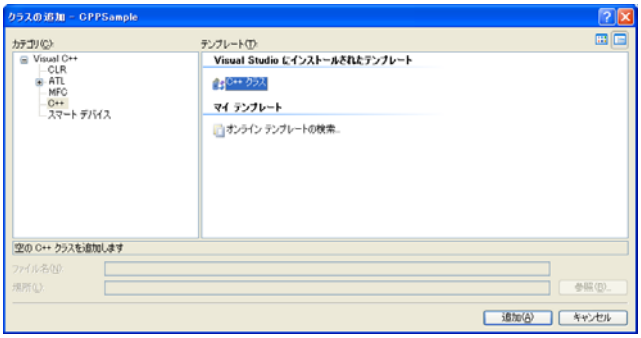
新規機能とメイン部分をつなぐパラメータクラスを作成します。（新規にクラスを追加する場合も同様です）

○ 手順

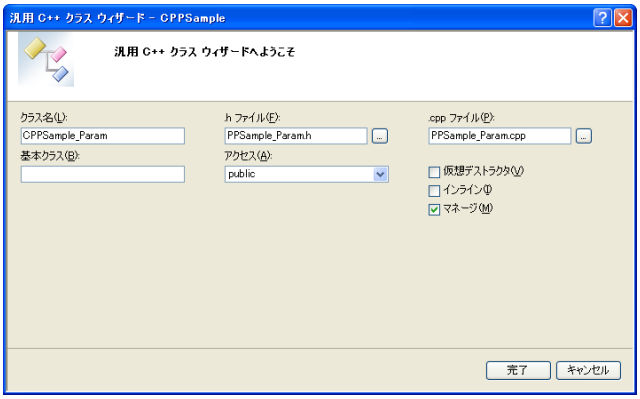
1） ソリューションエクスプローラ上でプロジェクトを選択し、[プロジェクト]-[クラスの追加]を選択します。



2） [クラスの追加]ダイアログ上で、[C++]-[C++クラスを選択します。]



3） クラス ウィザードが起動しますので、クラス名を指定します。（ここでは、“CPPSample_Param”とします）



3.4. 名前空間の決定

作成したプロジェクトのクラスに対して、名前空間(namespace)を設定します。
(新規に追加したばかりのクラスは、名前空間が設定されていない場合があります)
本資料では、“フォーム等の GUI”、“パラメータ”それぞれのクラスについて、下記の規則に従って名前空間を決定していきます。

- フォーム(ダイアログ)等の GUI のクラスの場合
  VolumeExtractor.VEForms

```
namespace VolumeExtractor {
    namespace VEForms {
        public class CPPSample_Form : public System::Windows::Forms::Form
        {
        };
    }
}
```

パラメータについては、下記のように“CustomParam”の名前空間へ設定して下さい。

- 機能のパラメータを示すクラスの場合
  VolumeExtractor.VETools.CustomParam.[機能名称]

```
namespace VolumeExtractor {
    namespace VETools {
        namespace CustomParam {
            public ref class CPPSample_Param
            {
            };
        }
    }
}
```

3.5. パラメータクラスの作成

ここでは、作成するモジュールの機能設定フォーム、MainFrame、WorkForm 等々、それぞれのクラスとのデータ受け渡しのための**パラメータクラス**の作成についての指針を記します。
このパラメータクラスは基本的にはデータをまとめたコンテナとしての動作を目的とします。

3.5.1. データを定義する

パラメータクラス作成にあたり、以下の点を押さえておいてください。

1) イメージデータ識別用の固有 ID(文字列)の格納先を確保
2) 機能フォーム上で何の情報を操作するのか？
3) またその操作時に必要なパラメータ情報は？
4) 機能フォームで必要な情報、パラメータのうち、描画に必要なものは？

以上の点を踏まえ、パラメータクラスに定義するデータを決定します。

```
public ref class CPPSample_Param
{
private:
        // 固有ID格納用
        System::String^  __id;

public:
        // プロパティの定義 (3.5.3参照)
        property Sytem::String^ ID
        {
                void set(System::String^ str)
                {
                        __id = str;
                }
                System::String^ get()
                {
                        return __id;
                }
        }

        //............
public:
        CPPSample_Param();
};
```

3.5.2. 参照クラスとして定義する

C++の場合、ref を付けておきます。
参照クラス(データ渡しではない)として扱われます。

```
public ref class CPPSample_Param
```

また“public”として宣言し、全体に公開しておきます。

※ C#の場合は

- クラスとして定義するとオブジェクトの参照が受け渡しの対象となります
- 構造体(struct)として定義すると値渡しとして認識されます

3.5.3. 変数へのアクセスは“property”として宣言

C++の場合、パラメータクラスに定義したデータへのアクセスは必要に応じて”property”として宣言し、set、get メソッドを必要に応じて定義し、外部からのアクセスに対応させます。
(C#の場合は、property キーワードが無くてもプロパティとして認識されます)
これは、CLR の機能として用意されており、通常の変数操作と同じように“＝”演算子で値の入力、参照などの操作ができます。
(C++の場合の、Get()、Set()等の関数と同じような役割とイメージしてください)

```
property System::String^ ID
```

3.6. 新規実装機能の記述対象ファイル

VolumeExtractor プロジェクト内では、MainFrame と WorkForm クラスについて処理を大別し、ソースコードを幾つかのファイルへ分散しています。
ただ、新しく追加する機能の呼び出しや実行のコードについては、下記のファイルでまとめて記述できるようにしておりますので、適宜ご使用下さい。（本資料内でも下記のファイルへ記述することを想定して説明していきます）

3.6.1. MainFrame クラス

MainFrame_Custom.h

新しく機能を追加する際の、メンバ変数、メソッドの宣言を記します。コンパイル時に MainFrame.h 内でインクルードされます。

MainFrame_Custom.cpp

新しく追加する機能に関するメソッドの定義を記します。

3.6.2. WorkForm クラス

WorkForm_Custom.h

新しく機能を追加する際の、WorkForm に関するメンバ変数、メソッドの宣言を記します。コンパイル時に WorkForm.h 内でインクルードされます。

WorkForm_Custom.cpp

新しく機能を追加する際の、WorkForm に関するメソッドの定義を記します。

3.6.3. 既存メソッドの呼出しタイミング

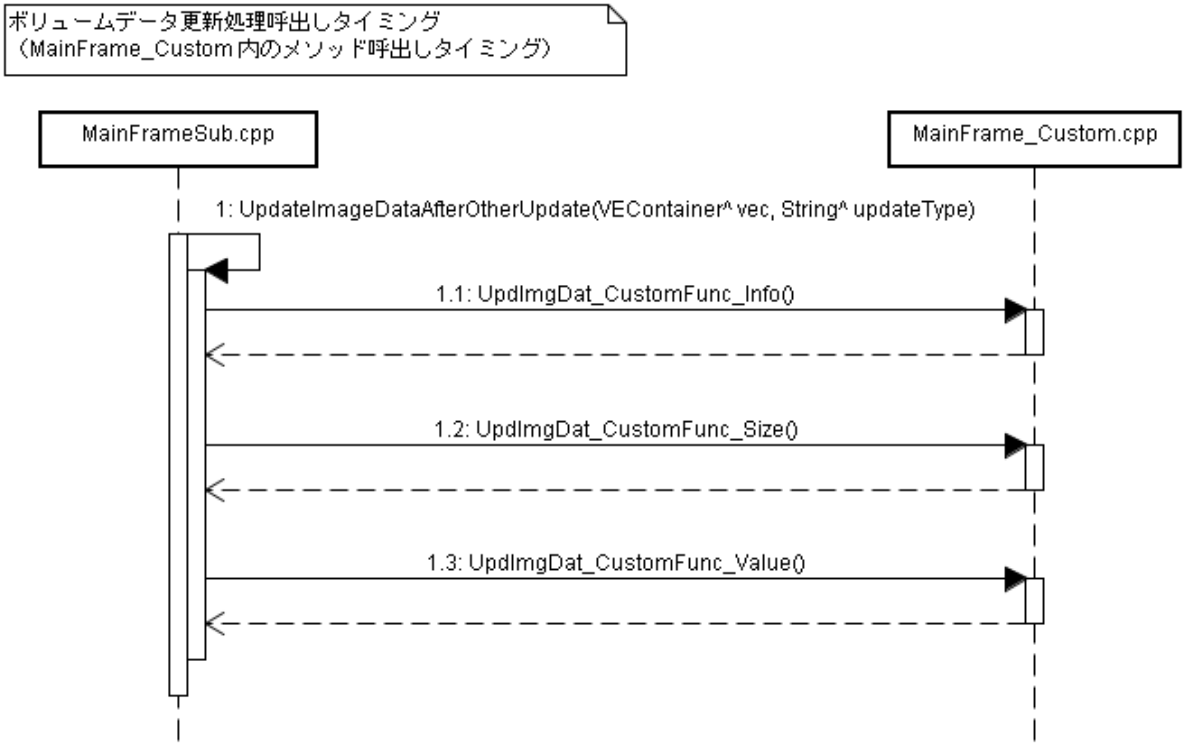
MainFrame_Custom、WorkForm_Custom の中に新しく機能を実装した際に記述すべき部分をピックアップしたメソッドを用意しています。

MainFrame_Custom には、ボリュームデータ更新時の、各機能パラメータ更新処理を記述するためのメソッドを用意しています。（UpdateImgDat_CustomFunc_Info(), UpdImgDat_CustomFunc_Size(), UpdImgDat_CustomFunc_Value() ）
WorkForm_Custom には、パラメータの初期化（init_CustomParam()）、削除（delete_Custom()）、WorkForm 名設定時（setWindowName_Custom()）、画像データ設定時（setImageData_Custom()）、描画処理（render_Custom_first(), render_Custom_preVR(), render_Custom_afterVR(), render_Custom_afterPolygon()）について、メソッドを用意していますので、適宜、使用して下さい。

各メソッドが呼び出されるタイミングは次のようになります。

3.6.3.1. ボリュームデータ更新用メソッドの呼出しタイミング

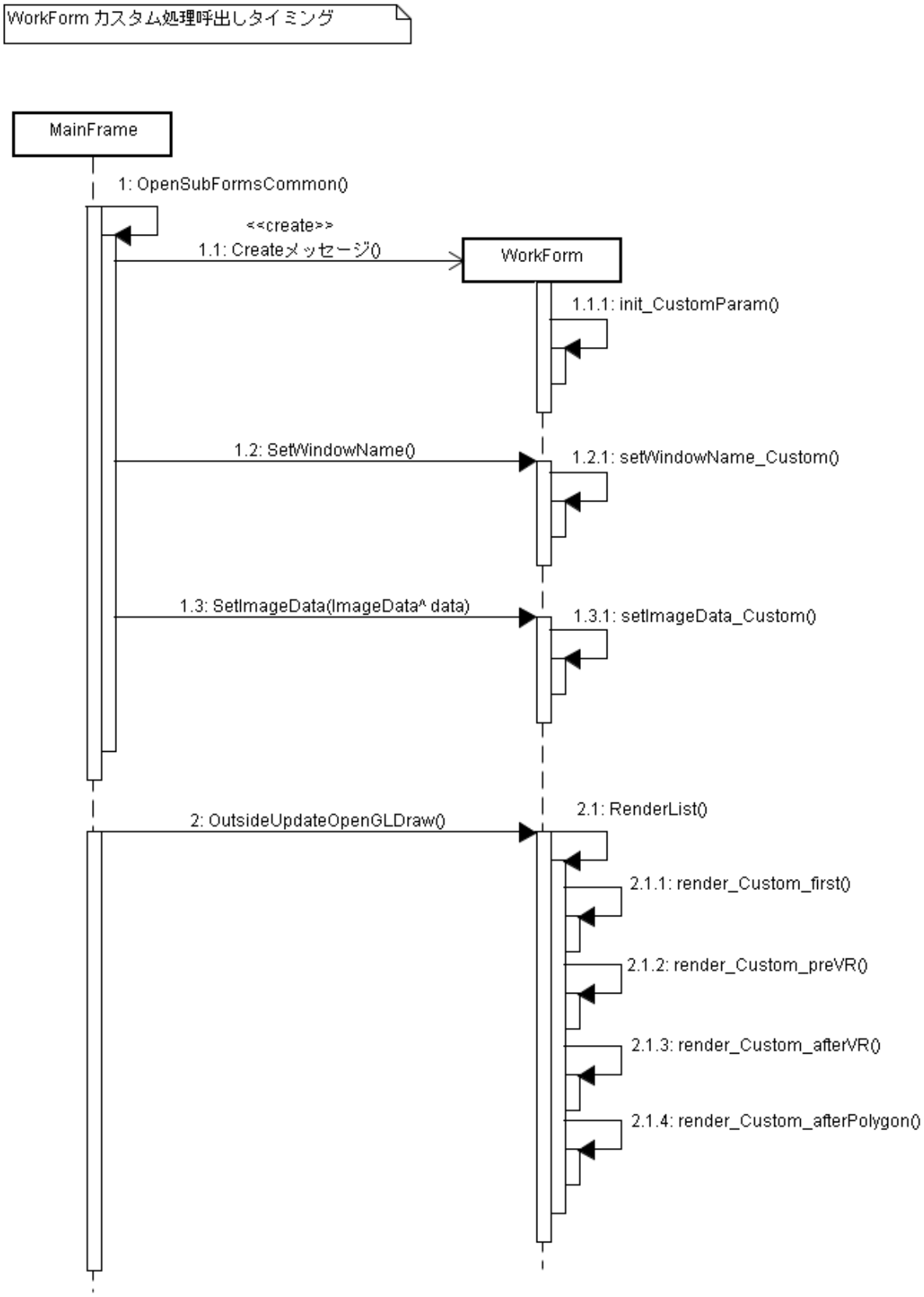
ボリュームデータ更新のための処理“UpdateImageDataAfterOhterdate”が呼び出された際、新規に実装した機能のパラメータを更新するメソッドが呼び出されるタイミングです。



MainFrame クラス内の UpdateImageDataAfterOtherUpdate 内部で、それぞれの場合（情報、サイズ、値）によって各メソッドが呼び出されます。

3.6.4. WorkForm カスタムメソッド呼出しタイミング

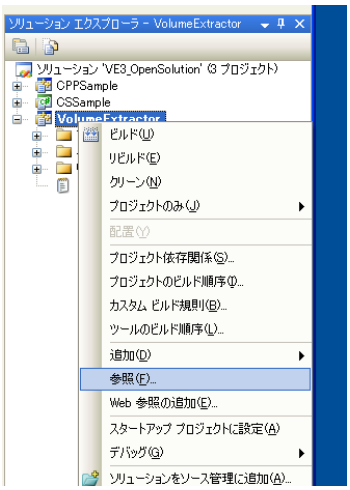
WorkForm のカスタム処理が呼び出されるタイミングを記します。



3.7. MainFrame への登録

3.7.1. 作成したモジュールを参照に追加

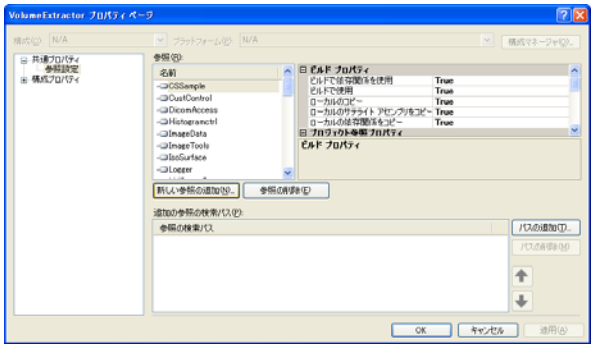
作成したモジュールを MainFrame から参照し、使用できるようにします。



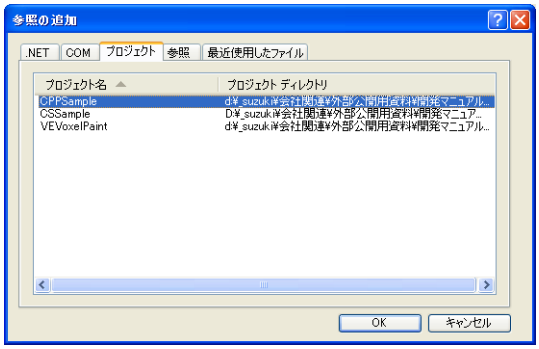
○ 手順

1） ソリューションエクスプローラ上で[VolumeExtractor]プロジェクトを選択、右クリックしてメニューの[参照]を選択します。

2） VolumeExtractor プロジェクトのプロパティを開き、参照設定―新しい参照の追加ボタンを押します。



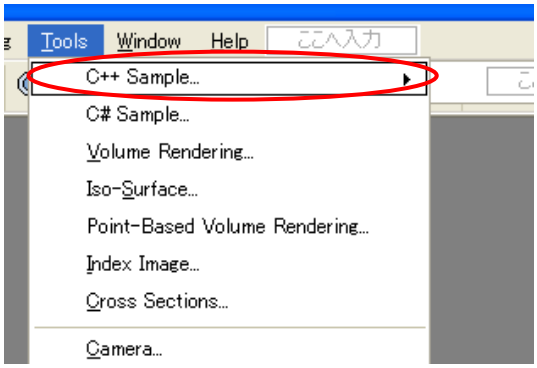
3） プロジェクトタブを選択して、先程作成したプロジェクトを選択、追加してください。



3.7.2. メニューへの追加

設定フォームを呼び出すためのメニューを追加します。MainFrame のデザイナを表示させ、メインメニュー部分に、機能を呼び出すための項目を追加します。

メニュー項目追加の際の指針としては、イメージデータ自体に変更を加える機能の場合は、“Edit”メニューへ、表示内容の変更や解析のような機能の場合は“Tools”メニューへそれぞれ追加します。

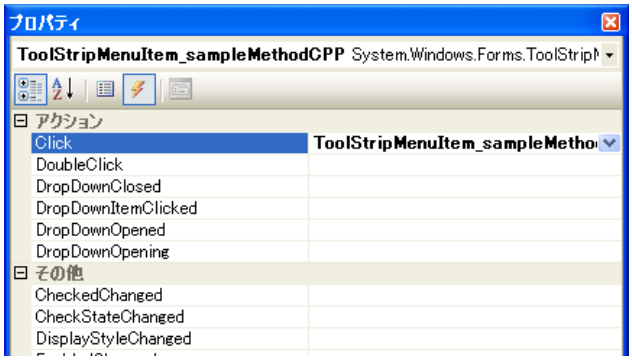


3.7.3. メニューイベントの追加

設定フォームを呼び出すためのイベントを追加します。

上記で追加したメニュー項目について、クリックした時のイベント(メソッド)として“Click”イベントを追加します。

MainFrame のデザイナ上で追加したメニュー項目を選択状態にしつつ、プロパティウィンドウ上のイベントにメソッドを追加します。



※ 追加したイベントに対応するメソッドは自動的に “MainFrame.h”に追加されますが、ヘッダファイルへはメソッドの宣言のみ記述し、定義(実際の処理コード)はソースファイルへ記述するという方針で進めます。(C#の場合はこの限りではなく、“*.cs”ファイルへ全て記述します)

下記のようなコードが“MainFrame.h”へ追加されますので・・・

```
private: System::Void ToolStripMenuItem_CPPSample_Click(System::Object^  sender,
  System::EventArgs^  e) { }
```

次のように変更します。

```
private: System::Void ToolStripMenuItem_CPPSample_Click(System::Object^  sender,
  System::EventArgs^  e);
```

処理を行なうコードを“MainFrame_Custom.cpp”へ記述します。

```
System::Void MainFrame::ToolStripMenuItem_CPPSample_Click(System::Object^  sender,
  System::EventArgs^  e)
{
    // イベント処理を行なうコードを追加
}
```

3.7.4. MainFrame_Custom.h へメンバ登録

フォームをメンバ変数として登録します。

```
VolumeExtractor::VEForms::CPPSample_Form^        __cppSampleForm;
```

3.7.5. メニューイベントへの記述内容

追加されたメニューの Click イベントでは主に

- 「機能フォームの生成と表示」
- 「メニュー項目のチェック状態変更」
- 「機能フォームのクローズ」

を目的とした処理を記述します。

「機能フォームの生成と表示」では、作成した機能の設定フォームの初期化、表示を行ないます。
設定フォームの初期化を行い、表示処理を実装してください。
既に設定フォームが表示されているときは、フォームを閉じる処理を記述します。

設定フォームの表示状態に合わせて、メニューのチェック状態も変更します。
(モードレスダイアログの状態で表示させる場合は、この処理は必要ありません)

下記は、フォームを表示するまでのサンプルです。

```
/// <summary>
/// メニュークリック時のメソッド
/// </summary>
System::Void MainFrame::ToolStripMenuItem_CPPSample_Click(
                                System::Object^  sender, System::EventArgs^  e)
{
  ↓ メニューのチェック状態を確認
  if( !ToolStripMenuItem_CPPSample->Checked )
  {
    if( __cppSampleForm!=nullptr )
    {
      delete __cppSampleForm;
      __cppSampleForm = nullptr;
    }
    ↓ フォームの作成
    __cppSampleForm = gcnew VolumeExtractor::VEForms::CPPSample_Form();
    __cppSampleForm->MdiParent = this;
    ↓ クローズイベント処理を登録します（イベント用のメソッドは別途用意します）
    __cppSampleForm->Closing += gcnew System::ComponentModel::CancelEventHandler
                                        (this, &MainFrame::CPPSample_Form_Closing);
    ↓ フォームをモードレスで表示します
    __cppSampleForm->Show();
    ↓ メニューをチェックした状態にします
    ToolStripMenuItem_CPPSample->Checked = true;
  }
  else
  {
    ↓ 既にフォームが表示されている場合に閉じる処理を実行します
    if( __cppSampleForm != nullptr )
    {
      __cppSampleForm->Close();
      delete __cppSampleForm;
```

```
        __cppSampleForm = nullptr;
      }
    }
}
```

↓ CPPSampleForm を閉じる際のイベントを記述します（ここでメニューのチェックを外しています）

```
/// <summary>
/// CPPSampleFormが閉じる時のメソッド
/// </summary>
System::Void MainFrame::CPPSample_Form_Closing(System::Object^ sender,
System::ComponentModel::CancelEventArgs^ e)
{
  ToolStripMenuItem_CPPSample->Checked = false;
}
```

※ フォームが閉じる時のイベントメソッドについては、上記のメソッドをヘッダファイル上で定義する必要があるので、“MainFrame_Custom.h”へ次の宣言を追加しておきます。

```
System::Void CPPSample_Form_Closing(System::Object^ sender,
                                    System::ComponentModel::CancelEventArgs^ e);
```

以上が、MainFrame への組込み処理の基本的な部分となり、メニュー上から機能フォームが呼び出せるようになります。
（初期化の情報にパラメータなどが必要な場合は、WorkForm への登録処理後に呼出し可能になるかと思います）

3.8. WorkForm への登録

WorkForm へは、基本的にパラメータオブジェクトを登録します。

登録する場合のパラメータオブジェクトは、次のような性質であることが望ましいです。

- ボリュームデータとパラメータオブジェクトが 1 対 1 で対応している場合
- 且つ描画処理に直接関連する場合

3.8.1. コンストラクタで初期化

パラメータオブジェクトの初期化を行ないます。

3.8.2. WorkForm::SetImageData へパラメータ初期情報設定の記述

ボリュームデータが登録された時にはじめて呼び出される処理です。
（大元は、MainFrame::OpenSubFormsCommon です）
ここではじめて、MainFrame よりボリュームデータが渡されます。
ボリュームデータ、また関連情報（縦横高のサイズ等）による初期化をここで行ないます。

3.8.3. パラメータオブジェクトのプロパティ作成

パラメータオブジェクトを MainFrame からも参照できるように、プロパティメソッドを作成します。
基本的にインスタンスは WorkForm で作成され、それを参照する形になりますので、ここでは、“get”を用意しておきます。

```
property [機能プロパティ名]
{
    [機能プロパティ名] get() { return [機能プロパティオブジェクト名]; }
}
```

3.9. ボリュームデータの操作方法

ボリュームデータは、ImageData というデータクラスで表現され、WorkForm と 1 対 1 で対応するデータとして VEContainer 単位で管理されています（厳密には WorkForm の内部データとして管理されます）。

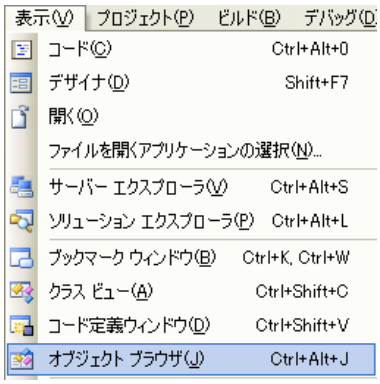
新しく実装する機能でボリュームデータを操作する場合には、この関連付けられた ImageData を取り出して（参照して）、操作することになります。

3.9.1. ImageData クラスの名前空間

データが格納された ImageData クラスの名前空間は下記の通りです。

```
VolumeExtractor::VEImageData
```

※ ImageData クラスも含め他のクラス等の名前空間についても、[表示]-[オブジェクト ブラウザ]より参照可能です。



3.9.2. データの参照方法

新しく実装する機能からボリュームデータを参照するためには、大きく下記の2つの方法があります。

【方法1】（C++）

VEArray から現在操作中の VEContainer を取り出し、

```
VEContainer->MdiImageData->IData
```

として参照し、新機能フォームの初期化時に参照を渡します

【方法2】

WorkForm 生成時に新機能のパラメータクラスに関連付けて、パラメータの受け渡しにより参照します

基本的にどちらの方法でも構いません。

※C#の場合は、“->”ではなく、“.”（ピリオド）に読みかえて下さい。

3.9.3. データの型と格納

輝度値のデータは、ImageData クラスの内部で Unsigned Char 型（1バイト）、Unsigned Short（2バイト）型、Short 型（2バイト）、Int 型（4 バイト）の 4 種類のうち、いずれかの型で、1 次元の配列として格納されています。

どの種類のデータであるかについては、ImageData クラスの“DataType”プロパティを参照することで知ることができます。

DataType の値の範囲と型種別は下記の通りです。

```
1 : Unsigned Char      1byte
2 : Unsigned Short     2byte
```

```
3 : Short          2byte
4 : Int            4byte
```

3.9.4. データの変更

配列に格納された輝度値のデータを直接、置き換えることができます。

但し、データの変更処理後は他の機能や表示内容を変更する必要があるので、データ変更処理後のイベントを発生させる必要があります。

イベントの発生方法、データ変更処理方法については、「3.11.イベントの作成と登録」を参照して下さい。

3.10. ImageData のメソッド等

3.10.1. メソッド

```
// リスケール適用後の値の取得
int       ApplyRescale(int)
// リスケール適用前の値の取得
int       UnapplyRescale(int)

bool      GetRealSize( ref float, ref float, ref float)
void      GetScale(ref float, ref float, ref float)
void      SetGridPoints( int, int, int )
// 格子間間隔と描画時移動値を求める
void      SetIntervalBetweenLatticesAndStartPoint()
```

3.10.2. プロパティ

```
// 輝度値データ配列（符号無バイト）
BYTE[]                Cfd                        {set; get;}
// 格納されているデータ種別
int                   DataType                   {set; get;}
// 元データファイル名
String                FileName                   {set; get;}
// 元データファイル名（複数）
String[]              FileNameSet                {set; get;}
// 元データファイル名（絶対パスでのファイル名）
String                FullFileName               {set; get;}
// 元データファイル名（絶対パスでのファイル名、複数）
String[]              FullFileNameSet            {set; get;}
// ボリュームデータサイズ（XYZの３方向のサイズを取得）
int[]                 GridPoints                 { get;}
// ボリュームデータサイズX方向
int                   GridX                      { get;}
// ボリュームデータサイズY方向
int                   GridY                      { get;}
// ボリュームデータサイズZ方向
int                   GridZ                      { get;}
// 輝度値データ配列（符号付４バイト）
int[]                 Ifd                        {set; get;}
// 3D上での画素間隔（１画素あたりのサイズ）
double[]              IntervalBetweenLattices    { get;}
double                IntervalBetweenLatticesX   {set; get;}
double                IntervalBetweenLatticesY   {set; get;}
double                IntervalBetweenLatticesZ   {set; get;}
// ボリュームとポリゴンの判別用
bool                  IsVolume                   {set; get;}
// 輝度最大値
int                   Max                        {set; get;}
// 輝度最大値
int                   Min                        {set; get;}
// ボクセル数
int                   Num                        { get;}
// 実際の画素間隔（mm）
double[]              Pitches                    { get;}
double                PitchX                     {set; get;}
double                PitchY                     {set; get;}
```

```
double                  PitchZ                      {set; get;}
// 1枚スライス画像の有無を取得
ImageDataPlaneType      PlaneType                   { get}
// ポリゴンのスケール
double                  PolygonScaleX               {set; get;}
double                  PolygonScaleY               {set; get;}
double                  PolygonScaleZ               {set; get;}
// 輝度値のリスケールの要素の取得
int                     RescaleIntercept            {set; get;}     // リスケール切片
int                     RescaleSlope                {set; get;}     //リスケール傾斜
// リスケールのタイプ設定・取得
ImageDataRescaleType    RescaleType                 {set; get;}
// 輝度値データ配列（符号無２バイト）
ushort[]                Sfd                         {set; get;}
// 輝度値データ配列（符号付２バイト）
short[]                 Ssfd                        {set; get;}
// 3D空間上でのボリューム描画開始位置
double[]                StartPoints                 { get;}
double                  StartPointX                 {set; get;}
double                  StartPointY                 {set; get;}
double                  StartPointZ                 {set; get;}
// WW/Lパラメータ
int                     WindowCenter                {set; get;}
int                     WindowWidth                 {set; get;}
```

3.11. イベントの作成と登録

新機能の設定フォーム上で実行ボタンを押した時や、パラメータを設定した時などのイベントを MainFrame や WorkForm で受け取り、処理を実行したい場合があります。
このような時に、“delegate”と“event”を使用して、設定フォーム上で発生したイベントを MainFrame、WorkForm で処理できるようにします。

参考 ： http://msdn2.microsoft.com/ja-jp/library/17sde2xt(VS.80).aspx

3.11.1. 新機能側への実装

3.11.1.1. delegate の作成

新規に作成した設定フォームに下記を定義します。

“delegate”で参照する関数の型を定義します。(C++形式で)

```
public delegate void SampleEventHandler( object^ sneder, eventArgs e );
```

“event” でイベントを定義します。

```
public event SampleEventHandler^ SampleEvent;
```

上記の“SampleEvent”を発生させたいタイミングの箇所に記述します。
例として、設定フォームの“Paint”へ記述する場合(あくまでもサンプルです。パフォーマンスを考えると、できれば Paint には記述しないほうが良いと思われます。)

```
void [設定フォームクラス名]::OnPaint( System::Object^ sender, eventArgs^ e )
{
    // なにかしら一通りの処理
    SampleEvent( this, e )
}
```

3.11.1.2. 必要最低限のイベント

以下のような場合には、新機能の設定フォーム上でイベントを発生させて、MainFrame 側、また WorkForm 側で対応する処理を記述して下さい。(他のオブジェクト(フォーム、データ)に影響のある場合に、イベントとして発生させる必要があります)

- ボリュームデータに変更を加えた時
- 設定フォーム上で、変更したパラメータに連動して表示状態が変わる時

3.11.2. MainFrame、WorkForm 上でのイベント処理の追加

新機能の設定フォーム上で作成したイベントについて、MainFrame 側にイベントに対応する処理を記述します。

3.11.2.1. メソッドの作成と登録方法

MainFrame 側に、機能設定フォームで定義した“delegate”のメソッドと、同じ型、同じ引数のメソッドを定義します(メソッド名は自由です)。このメソッドにイベントに対応する処理を記述します。

```
void CS_SampleForm_Event( object^ sender,  eventArgs^ e );
```

この MainFrame 側で定義したメソッドを、新機能の設定フォームのインスタンスに、イベントを処理するメソッドとして追加します。(関数ポインタのイメージです)

```
CS_SampleForm^  csForm = gcnew  CS_SampleForm();
csForm->SampleEvent +=
      gcnew  CS_SampleForm::SampleEventHandler( this, &MainFrame::CS_SampleForm_Event );
```

3.11.2.2. ボリュームデータの変更に伴う更新処理の呼出し

ボリュームデータに変更を加えた場合は、他のボリュームデータを参照しているオブジェクトも更新が必要になります。このような場合には以下のメソッドを呼出します。また更新処理についてメソッド内部への記述も必要です。

下記のようなボリュームデータの変更の場合、更新処理を呼び出します。

- ボリュームデータの値に変更を加えた場合
- ボリュームデータのサイズ情報(ピッチサイズ等も含みます)を変更した場合
- ボリュームデータのサイズ(データ自身の XYZ サイズ)を変更した場合

下記は、VoxelPaint(矩形塗りつぶし機能)の場合。

```
UpdateImageDataAfterOtherUpdate( vpForm->VoxelPaintParameter->Id, L"VoxelPaint");
```

新しく実装する機能の場合は、下記のように記述します。

```
UpdateImageDataAfterOtherUpdate( [データ固有 ID (文字列) ], [更新識別文字列] );
```

[データ固有 ID(文字列)] 部分には、編集対象のデータの固有 ID を設定します。
[更新識別文字列] には、下記の文字列のいずれかを指定します。

L"CSTM_UpdInfo"　・・・ ボリュームデータのサイズ情報等を変更した時に指定します
L"CSTM_UpdSize"　・・・ ボリュームデータのサイズ (XYZ サイズ) を変更した時に指定します。
L"CSTM_UpdValue"　・・・ ボリュームデータの値を変更した時に指定します

また、他の機能からも画像データ更新処理の呼び出しが実行される場合もあります。

上記の各種条件の場合に、機能のパラメータ等を変更する必要がある場合は、処理をソースコードへ記述します。
記述箇所は、 MainFrame_Cusotm.cpp 内の下記メソッド内です。

情報変更に伴う処理　　→ UpdImgDat_CustomFunc_Info()
サイズ変更に伴う処理　　→ UpdImgDat_CustomFunc_Size()
値変更に伴う処理　　→ UpdImgDat_CustomFunc_Value()

この更新処理全体の処理順序としては、イベント発生を受け取った際に下記処理を実装します。

1）カーソールを WAIT へ変更
2）メインフレームの操作を不可
3）各オブジェクトの情報更新処理（UpdateImageDataAfterOtherUpdate）
（ここで、UpdImgDat_CustomFunc_xxx() が呼び出されます）
4）必要であれば描画の変更（WorkForm 側に処理を実装）
5）メインフレームの操作を可
6）カーソールをデフォルトに戻す

【参考】VEVoxelPaint では同様の処理を MainFrameDelegate.cpp 内に記されている
VoxelPaintForm_ExecuteButtonClicked() で行っています。

3.11.2.3. 機能設定フォーム上のパラメータに応じた表示内容の変更処理呼出し

ボリュームデータを変更せず、3D 上の描画のみを変更したい場合はイベントの処理として、WorkForm の描画処理を呼び出します。

処理順序とてしては

- 操作対象の VE コンテナ（WorkForm）を取得
- 描画変更処理の呼出し（WorkForm 側にメソッドを定義）

【参考】VEVoxelPaint では同様の処理を MainFrameDelegate.cpp に記されている
VoxelPaintForm_ParamChanged() で行っています。このメソッドでは、WorkForm に実装した VoxelPaint 用の更新処理として UpdateRender_forVoxelPaint()を呼び出しています。

3.11.2.4. 描画処理の記述と呼出し

描画処理自体は、自身で記述する必要があります。描画処理を呼び出すための下記のメソッドを“WorkForm_Custom.cpp”へ用意しています。

render_Custom_first()
描画開始時に呼び出されます。
（フレームバッファがクリアされた直後のタイミングです）

render_Custom_preVR()
ボリュームレンダリング前に呼び出されます。
ほとんどの場合、ここで描画処理を呼び出します。

render_Custom_afterVR()
ボリュームレンダリング直後に呼び出されます。
（ポリゴンデータ描画前です）

render_Custom_afterPolygon()
ポリゴンデータ描画後に呼び出されます。

4.　ビルドとデバッグ

4.1. 依存関係の設定

新規に作成したモジュールと、VolumeExtractor プロジェクトの関係を設定します。

依存関係を設定することにより、複数のプロジェクトを一度にビルドする際の順序を決定することができます。

※　モジュールの参照を追加する際に“プロジェクト”として追加していた場合は、この依存関係の設定は自動的に行なわれます。“参照”として DLL 等を直接追加していた場合は、下記の手順が必要となります。

○　手順

1）［プロジェクト］－［プロジェクト依存関係］を開き、［プロジェクト］の箇所を「VolumeExtractor」にします。

2）今回作成したモジュールのプロジェクト名が［依存先］の一覧に表示されますので、チェックを入れます。

※　例として、A プロジェクトが B プロジェクトを参照、B プロジェクトが C プロジェクトを参照するような場合、C→B→A の順にビルドが行なわれる必要があります。この場合は、「A の依存先に B」、「B の依存先に C」をそれぞれ設定します。

4.2. ビルド構成

Volume Extractor 3.0 のソリューションでは、下記のビルド構成が含まれています。

- Deug
- Release
- FunctionLimit_Release
- VE_SMESHOMITTED_Release
- VE_USE_SERIALCHECK_Release

このうち、デバッグで使用する場合には“Debug”、リリースで使用する場合には“Release”を、［ビルド］－［構成マネージャ］で選択します。

4.3. デバッグ

ここでは、開発環境上でのデバッグ方法について記します。

4.3.1. 作業フォルダの設定

デバッグ時の作業フォルダを指定します。

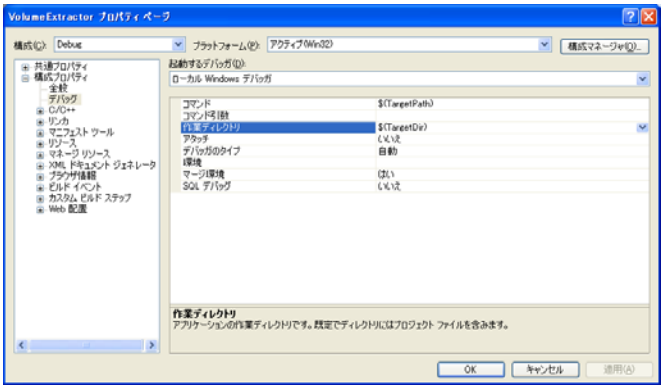
デバッグ時に実行する実行ファイルは、ここで指定されたフォルダを基準に関連モジュール等を参照します。

○ 手順

1） VolumeExtractor プロジェクトを選択し、プロジェクトのプロパティを開きます。

2） ［構成プロパティ］－［デバッグ］を選択します。

3） 右側に表示されるリストの「作業ディレクトリ」に“$(TargetDir) ”を記します。

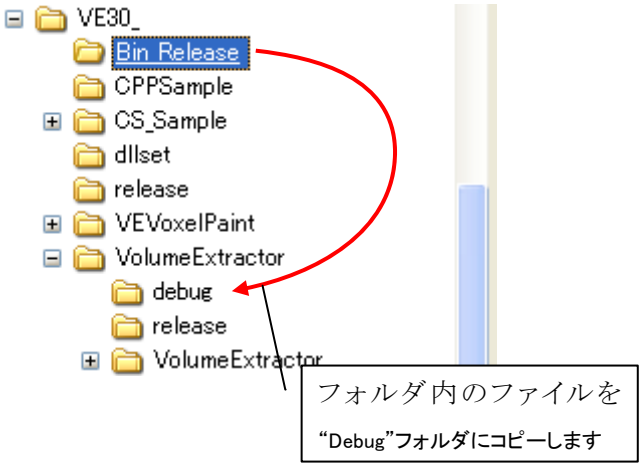


4.3.2. 必須リソースのコピー

実行ファイルが動作する際に必要なファイルを作業フォルダにコピーしておきます。

設定した作業ディレクトリに、“Bin_Release”フォルダ内のファイル（アプリケーション動作のための設定ファイル）をコピーします。

※ 実行の際には、各々のファイルに読み取り属性が無いことを確認して下さい。読み取り属性がある場合は、正常に起動しないことがあります。



4.3.3. ビルドとデバッグ実行

構成マネージャで“Debug”を選択します。

［ビルド］－［ソリューションのビルド］を実行して、実行ファイルを作成します。

［デバッグ］－［デバッグ開始］を選択して、デバッグを開始します。

5. リリース

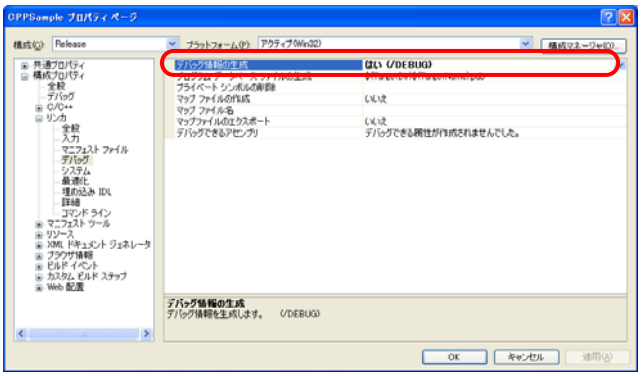
5.1. リリースのビルド方法

ここでは、デバッグ情報を付加しないリリースビルドの方法を示します。

Visual Studio 2005 では、リリースのビルドの場合でも、デフォルトでデバッグ情報が含まれたモジュールが作成され、実行速度にも影響します。
リリースビルド時に、このデバッグ情報が含まれないように以下のように設定します。(リリースのビルドでデバッグする場合を除きます)

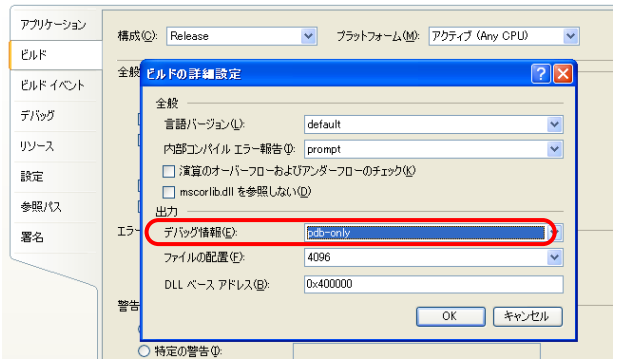
・ C++の場合

プロジェクトのプロパティフォームを開きます。
フォーム上部の「構成」が「Release」になっていることを確認します。
「構成プロパティ」-「リンカ」-「デバッグ」を表示します。
右の設定一覧の中から「デバッグ情報の設定」を選択し、設定値を「いいえ」にします。



・ C#の場合

プロジェクトのプロパティフォームを開きます。
左側タブ一覧から「ビルド」のタブを選択します。
上部の「構成」が「Release」になっていることを確認します。
「出力」の項目内の「詳細設定」ボタンを押下、「ビルドの詳細設定」ダイアログを表示します。
「出力」-「デバッグ情報」の値を「none」に設定します。

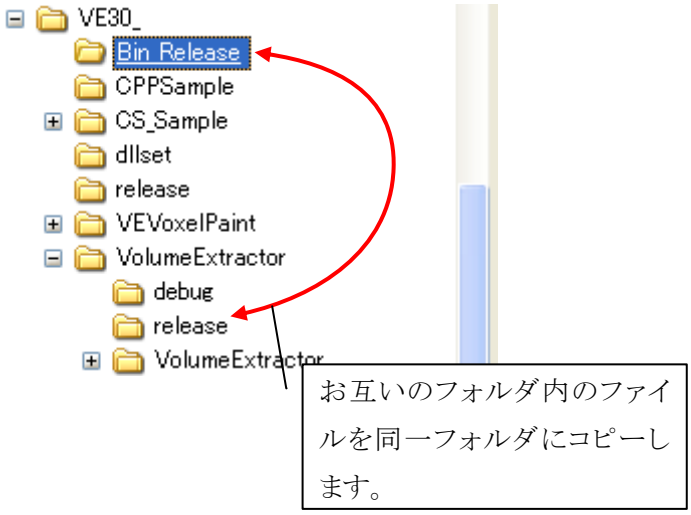


5.2. リリース方法

（実行ファイルと関連モジュールの配置方法を記述）

Volume Extractor 3.0 では、作成したモジュールは実行ファイル、設定ファイルを同じフォルダに配置することで動作します。

“Bin_Release”フォルダに、アプリケーションに必要な設定情報などが格納されており、“VolumeExtractor¥Release”フォルダ内に作成された実行ファイルやモジュールと同じフォルダに配置することでアプリケーションが動作します。



例えば、“Bin_Release”フォルダ内のファイルと“VolumeExtractor¥Release”フォルダ内のファイルを新しく用意したフォルダへコピーし、実行ファイル（VolumeExtractor.exe）を起動します。

※ 実行の際には、各々のファイルに読み取り属性が無いことを確認して下さい。読み取り属性がある場合は、正常に起動しないことがあります。

6. 備考

6.1. ファイルフォーマット

6.1.1. VDF フォーマット

Volume Extractor 用のファイルフォーマットです。

ファイル情報 ［256byte］
画像データ （Raw 形式） （縦x横x高x単位データサイズ）

6.1.1.1. ファイル情報

先頭の 256byte にはファイル情報として下記の情報が ASCII で格納されます。

各項目は、スペース(0x20)で区切られ格納されます。

	項目	内容
ファイル情報 ［256byte］	ファイルヘッダ	“VDF_1.0_VE12.8”
	StartPoint(描画開始位置)タグ	“sp”
	開始位置 X 座標	数値(実数)
	開始位置 Y 座標	数値(実数)
	開始位置 Z 座標	数値(実数)
	グリッドサイズタグ	“n”
	X 軸サイズ	数値(整数)
	Y 軸サイズ	数値(整数)
	Z 軸サイズ	数値(整数)
	ピッチサイズタグ	“pitch”
	X 方向 Pitch	数値(実数)
	Y 方向 Pitch	数値(実数)
	Z 方向 Pitch	数値(実数)
	データタイプタグ	“dt”
	データタイプ(1～4)	数値(1～4)
	情報終了識別子	¥n(0x0A)
	(予備)	0x00

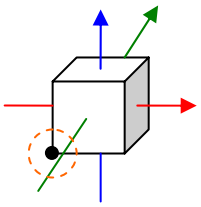
・ ファイルヘッダ

文字列“VDF_1.0_VE12.8”固定

・ StartPoint

3D 空間上での配置位置の指定

右図のようにモデルの 3D 空間上での開始点を指定します。



・　グリッドサイズ

X 軸、Y 軸、Z 軸、各方向のピクセル（ボクセル）数のサイズ

・　データタイプ

1ピクセル（ボクセル）のデータで使用されるデータの型

1 ： Byte 型（1byte）

2 ： Unsigned Short 型（2byte）

3 ： Short 型（2byte）

4 ： int 型（4byte）

・　情報終了識別子

ファイル情報の最後に“¥n”（0x0A）を格納

・　（予備）

この部分は、今後の拡張等のための箇所として用意されています

0x00 で埋めます。

6.1.1.2. 画像データ

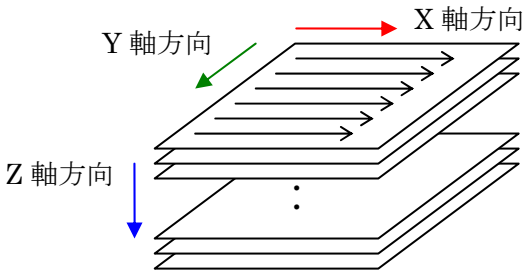
画像データが RAW 形式で格納されます。

各データ要素は「データタイプ」で指定されたサイズになり、データ全体としては下記のサイズになります。

（グリッド X 軸サイズ）×（グリッド Y 軸サイズ）×（グリッド Z 軸サイズ）×（データタイプの指定 byte 数）

格納順序は、X 軸方向の要素から格納開始され、Y 軸方向、Z 軸方向へと格納されています。

（右図イメージ参照）



※VE での表示は、Z 軸方向を上向きにしていますのでご注意下さい。

6.1.2. VOL フォーマット

拡張子、“vol” と “vif” の 2 つのファイルにより構成されたボリュームデータファイルです。
vif がファイル情報部分、vol が画像データ部分となります。

6.1.2.1. vif

vif ファイルにはファイル情報として下記の内容が ASCII として 5 行構成で格納されます。
(改行コードは ¥r¥n(0x0D0A))

行数	項目	記述内容
1	ファイルヘッダ文字列	“VIF 1.0 VE12.8”
2	StartPoint	“start_pt [X 座標] [Y 座標] [Z 座標]”
3	画像(グリッド)サイズ	“size [X 方向] [Y 方向] [Z 方向]”
4	各方向のピッチ	“pitch [X 方向] [Y 方向] [Z 方向]”
5	データタイプ	“data_type [1～4]”

2 行目以降の各項目は、内容を示す文字列(”start_pt”等)が記述され、続けて値(数値)が格納されます。

文字列と値の間は半角スペース(0x20)2 文字分で区切り、各値(数値)の間は半角スペース 1 文字分で区切ります。

【例】

```
VIF 1.0 VE12.8
start_pt  -0.5 -0.5 -0.5
size  512 512 469
pitch  0.1693333 0.1693333 0.64
data_type  2
```

6.1.2.2. vdf

画像データ要素が RAW 形式で格納されます。
格納されるデータのサイズ、順序等は VDF のデータ部分と同じです。(VDF フォーマット参照)

6.1.3. 出力サンプルコード

VDF、VOL、VIF の出力サンプルコードを下記に記します。

※　エラー処理等は考慮されていませんので、ご注意下さい

```
///
/// Raw データの書き出し
/// 下記のメソッド内で使用します。
///
private: bool SaveRAW(BinaryWriter^ bw, array<unsigned char>^ data){
    // Raw データ書き出し
    int count = 0;
    while(count < data->Length){
        bw->Write(data[count]);
        count++;
    }
    return true;
}

///
/// VOL ファイルの書き出し
///
private: bool SaveVOL(
  String^ path,                       // 出力先のファイルパス
  array<unsigned char>^ data          // 出力元データ（1 次元配列のボリュームデータ）
){
    // VOL 書き出し
    FileStream^ fs = gcnew FileStream(
                     path, System::IO::FileMode::Create,
                     System::IO::FileAccess::Write, System::IO::FileShare::None);
    BinaryWriter^ bw = gcnew BinaryWriter(fs);

    // VOL ファイルは、実際のところ Raw データそのもの
    SaveRAW(bw, data);

    bw->Close();
    fs->Close();

    return true;
}
```

```
///
/// VIF ファイルの書き出し
///
private: bool SaveVIF(
  String^ path,                        // 出力先のファイルパス
  int x, int y, int z,                 // データグリッドサイズ
  double spx, double spy, double spz, // データ開始位置
  double ptx, double pty, double ptz  // データピッチ
){
    // VIF 書き出し
    FileStream^ fs = gcnew FileStream(
                        path, System::IO::FileMode::Create,
                        System::IO::FileAccess::Write, System::IO::FileShare::None);
    StreamWriter^ sw = gcnew StreamWriter(fs);
    String^ tmpstr = L"";
    sw->NewLine = L"¥r¥n";   // 改行コード
    //全部で 5 行
    // 1 行目
    sw->WriteLine(L"VIF 1.0 VE12.8");
    // 2 行目
    tmpstr = L"start_pt  "
            + spx.ToString() + L" " + spy.ToString() + L" " + spz.ToString();
    sw->WriteLine(tmpstr);
    // 3 行目
    tmpstr = L"size  "
            + x.ToString() + L" " + y.ToString() + L" " + z.ToString();
    sw->WriteLine(tmpstr);
    // 4 行目
    tmpstr = L"pitch  "
            + ptx.ToString() + L" " + pty.ToString() + L" " + ptz.ToString();
    sw->WriteLine(tmpstr);
    // 5 行目
    tmpstr = L"data_type  " + L"1";   // サンプルでは unsigned char 型に固定
    sw->WriteLine(tmpstr);

    sw->Close();
    fs->Close();

    return true;
}
```

```
///
/// VDF ファイルの書き出し
///
private: bool SaveVDF(
  String^ path,                          // 出力先のファイルパス
  int x, int y, int z,                   // データグリッドサイズ
  double spx, double spy, double spz,    // データ開始位置
  double ptx, double pty, double ptz,    // データピッチ
  array<unsigned char>^ data             // 出力元データ（1 次元配列のボリュームデータ）
){
    // VDF 書き出し
    FileStream^ fs = gcnew FileStream(
                path, System::IO::FileMode::Create,
                System::IO::FileAccess::Write, System::IO::FileShare::None);
    BinaryWriter^ bw = gcnew BinaryWriter(fs);

    // ヘッダ文字列    unsigned char 型の例（データタイプ=1。ヘッダ長さは 256 固定）
    String^ str_header = L"VDF_1.0_VE12.8"
            + " sp "    + spx.ToString() + " " + spy.ToString() + " " + spz.ToString()
            + " n "     + x.ToString()   + " " + y.ToString()   + " " + z.ToString()
            + " pitch " + ptx.ToString() + " " + pty.ToString() + " " + ptz.ToString()
            + " dt "    + "1"
            + "¥n";

    // String を Char クラス配列に変換
    array<Char>^ transfer_header = str_header->ToCharArray();
    // 実際に書き出す Char クラス配列
    array<Char>^ put_header_array = gcnew array<Char>(256);
    // 書き出しに使用する配列に、データを転送
    Array::Copy(transfer_header, put_header_array, transfer_header->Length);

    // 未使用列は、0 で埋める
    int count = transfer_header->Length;
    while(count < 256){
        put_header_array[count] = 0;
        count++;
    }

    // ヘッダを書き出す
    count = 0;
    while(count < 256){
        bw->Write(put_header_array[count]);
        count++;
    }

    // データ部分を書き出す。データ部は、Raw データと同じ
    SaveRAW(bw, data);

    bw->Close();
    fs->Close();
    return true;
}
```

6.2. 外部プロセスの起動

CLR の場合に、外部プロセスを呼出す方法を記述します。

6.2.1. ファイル名を直接実行

“System.Diagnostics.Process” クラスを使用します。

【例】実行ファイル名と引数を直接指定します。

```
System::Diagnostics::Process::Start("test.exe", "-a -b");
```

6.2.2. プロセス情報を設定して実行

“ProcessStartInfo” を使用します。

【例】実行ファイル名と引数を“ProcessStartInfo”に格納して指定します。

```
System::Diagnostics::ProcessStartInfo^ pi =
    gcnew System::Diagnostics::ProcessStartInfo();
pi->FileName  = "TEST.exe";
pi->Arguments = "-a -b";
System::Diagnostics::Process::Start( pi );
```

6.2.3. 実行完了を待つ

外部プロセスを呼び出し、実行が完了するまで処理を待ちます。

```
System::Diagnostics::Process^ p = System::Diagnostics::Process::Start([ファイル名等]);
p->WaitForExit();
```

詳しい使用方法については、MSDN ライブラリ等を参照下さい。

6.3. バージョン決定指針

VolumeExtractor 3.0 でのモジュールのバージョン決定方法についての指針を記します。

1.0.0.0

左から、「メジャー」、「マイナー」、「リビジョン」、「ビルド」とします。

・メジャー　：　根本的な仕様変更などがあった場合に +1
（基本的には "1"）

・マイナー　：　大幅な仕様変更や機能追加した場合に +1
（外部にファイルやモジュールが必要になった等、外部から見て明らかに変更が判別できる場合）

・ビルド　：　仕様変更や機能追加した場合に +1
（主に内部処理への変更や、既存の機能の改善等）

・リビジョン　：　不具合修正等を行った場合に +1