Medv4D

project documentation

Medv4D: project documentation

Table of Contents

List of Figures

Preface

aaaa

Chapter 1. Library Common

aaaaaaa

Chapter 2. Library Imaging

Main goal of this library is effective implementation of pipeline computation on input datasets. Whole computation should be as parallel as possible in order to utilize resources available in modern processors, etc.

Design of interfaces and class hierarchies is aimed to extensibility and code reusability. Centra

2.1. Compilation

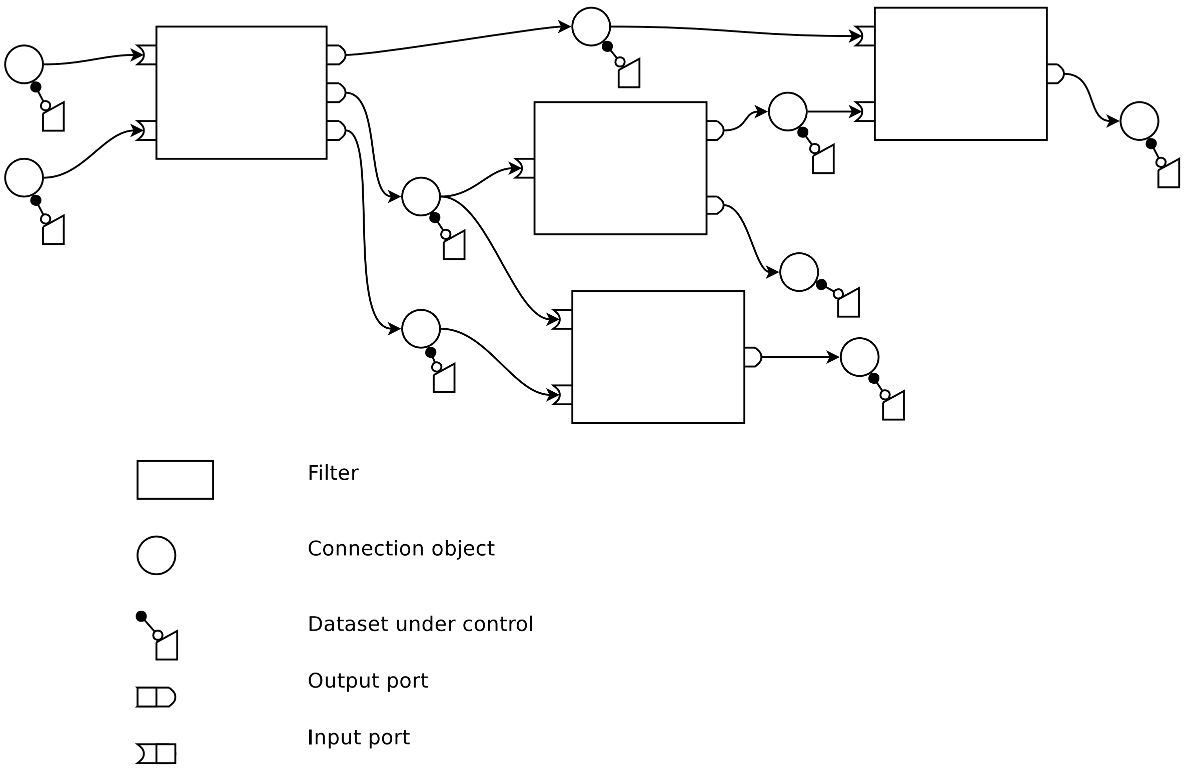
2.1.1. Dependencies

This library should be linked together with libCommon. So all its dependencies come with it.

2.2. Architecture

All declarations are in namespace M4D::Imaging. Whole design of class hierarchies is Figure 2-1

Figure 2-1. Pipeline scheme



Pipeline should be built from objects of certain types, which we will now discuss.

2.2.1. Datasets

Actual data are stored in proper descendants of class `AbstractDataSet`. Its hierarchy is shown in . Purpose of these classes is to provide access methods for reading and writing informations of certain type and optional synchronization.

By optional synchronization is meant set of synchronization methods, which are not called by access methods in class. And user should use these methods only in situations requiring synchronization. This less comfortable though, but more effective.

`AbstractPipeFilter`. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

`AbstractImageFilter`. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

`AbstractImageSliceFilter`. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

2.2.2. Filters

aaaaaaa

2.2.3. Connections

aaaaaaa

2.2.4. Ports

aaaaaaa

2.3. Usage

2.3.1. How to build a pipeline

First thing we should know, is how to create filtering pipeline with supposed behaviour using prepared filters. This section will be brief tutorial to pipeline construction. We show possible usage on example class that will have all desired abilities normaly distributed across application.

Before we start construction, we must decide if we want handle storing and deallocation of all objects. There is prepared container for all pipeline objects, which can handle deallocation of stored objects and semiautomatically connects filters, and other objects with ports used as communication channel. For our example we use this `PipelineContainer`. For manual control and construction see its source code.

```
#include "Imaging/Image.h"
#include "Imaging/PipelineContainer.h"

const unsigned Dim = 3
typedef uint16 ElementType;

typedef M4D::Imaging::Image< ElementType, Dim > ImageType;

class ExamplePipelineHandler
{
public:
    //Default constructor - pipeline construction
    ExamplePipelineHandler();

    //Method which pass image to pipeline and start computation.
    void
```

```
    FilterImage( ImageType::Ptr image );

    //This method will be called, when filters finish.
    //Parameter tells, if computation passed withou problems.
    //Implementation of this method depends on its purpose (notify user, ...).
    void
    FinishNotification( bool succesfully )
        { /*Put your code here.*/ }
protected:
    M4D::Imaging::PipelineContainer   _pipeline;
    M4D::Imaging::AbstractPipeFilter  *_firstFilter;
    M4D::Imaging::AbstractImageConnectionInterface  *_inConnection;
    M4D::Imaging::AbstractImageConnectionInterface  *_outConnection;
};
```

Actual pipeline construction should proceed in three basic steps:

- Allocation of filters
- Establishing connections
- Setting message hooks

Now we discuss these steps in detail. With example implementation.

Allocation of filters. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

Establishing connections. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

Passing input datasets and execution. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

```
#include "ExamplePipelineHandler.h"

ExamplePipelineHandler
::ExamplePipelineHandler()
{
    _firstFilter = new
    M4D::Imaging::AbstractPipeFilter *secondFilter = new

    _pipeline.AddFilter( _firstFilter );
    _pipeline.AddFilter( secondFilter );

    _pipeline.MakeConnection( _firstFilter, 0, secondFilter, 0, true );

    _inConnection =
        dynamic_cast< M4D::Imaging::AbstractImageConnectionInterface * >(
             &(_pipeline.MakeInputConnection( _firstFilter, 0, false )) );

    _outConnection =
        dynamic_cast< M4D::Imaging::AbstractImageConnectionInterface * >(
```

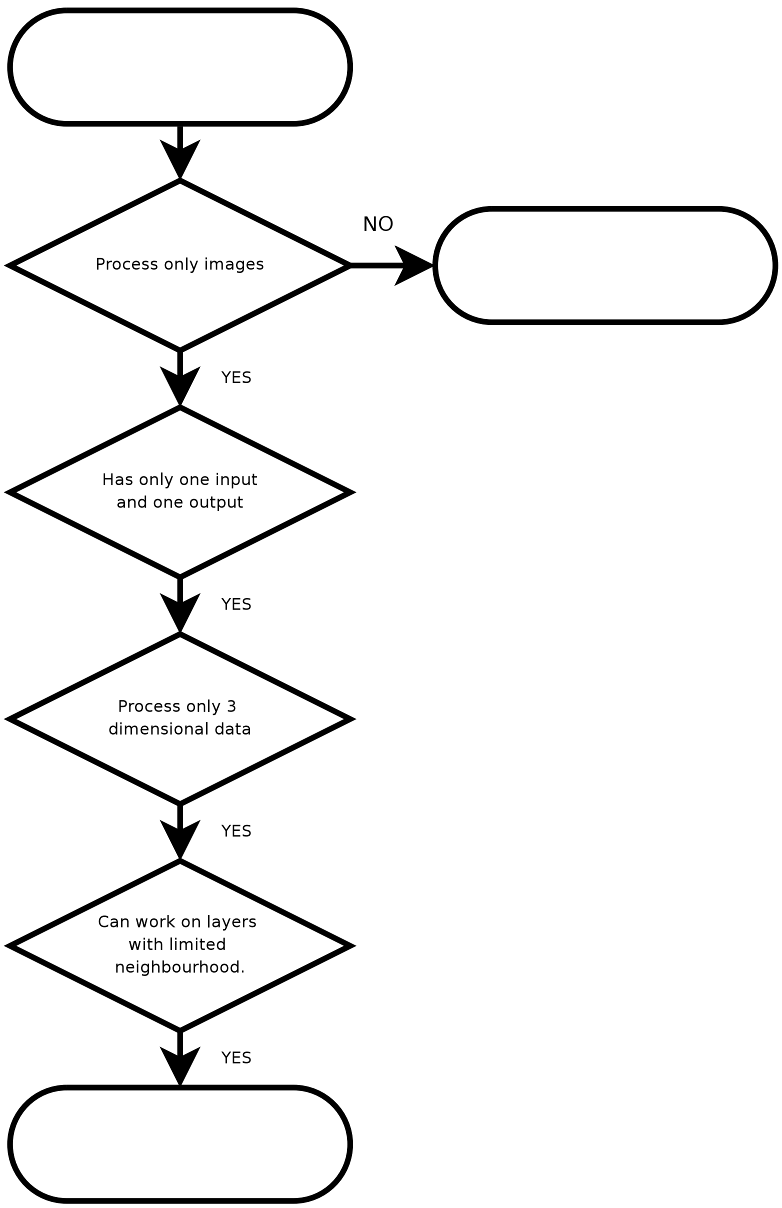
```
            &(_pipeline.MakeOutputConnection( secondFilter, 0, true )) );
}


void
ExamplePipelineHandler
::FilterImage( ImageType::Ptr image )
{
    _inConnection->PutImage( image );

    _firstFilter->Execute();
}
```

2.3.2. Creating new filter

One of the main advantage of this library is its extensibility. The biggest potential is easy way to add new filters with all features promised by pipeline design (synchronization, parallel execution, etc.). This is achieved through set of filter abstract classes, each designed for special purpose. Author of new filter have to consider few aspects (dimensionality, type of input/output data, way the filter computes, etc.) and choose right ancestor class for his filter.

To make decision easier you can use prepared flowchart Figure 2-2 and find most suitable ancestor class.

Figure 2-2. Filter decisive flowchart



Now if you have chosen right ancestor class, you can start with actual implementation of your filter. You should keep few rules and concepts, which can not only help you with writing, but even some other parts of library will just work without handling extra issues. This is achieved by generic design of whole library, templates are used almost everywhere.

We now introduce these rules and concepts, and show practical examples from library sources. First of all try design your filters as generic as possible. So try maximally use dataset traits (now only available `ImageTraits`), template specialializations, etc. All ancestor classes are templated (with exception of `AbstractPipeFilter`), so it makes that easier.

Try to keep these rules, when designing filter interface:

- Filter class has public typedef to predecessor class with name `PredecessorType`.
- All properties of filter are in one public nested `struct Properties` deriving from `PredecessorType::Properties`. In case, that set of properties is empty, make at least public typedef to `PredecessorType::Properties` with name Properties.
- Prepare default constructor, and constructor with pointer to `Properties`. Default constructor creates default set of properties. In both constructors pointer to instance of `Properties` is passed to predecessor constructor in list of initializers. Using constructor with parameter will completely initialize filter with passed properties, no other method is needed to call.

AAAAAAAAAAAAAAAAAAAAAAAAA

- Pointer to `Properties` structure is stored as protected member *`_properties`*, but its type is `AbstractFilter::Properties`. So if you want access members of your properties structure you must either cast to right type every time, or put preprocessor macro `GET_PROPERTIES_DEFINITION_MACRO` to private section of your class declaration. Now method `GetProperties()` returning reference to `Properties` is available for usage.
- To make declaration of Get/Set methods easier three macros are prepared: `GET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )`, `SET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )` and `GET_SET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )`. These macros will be unwinded into inline declarations of `TYPE Get’NAME’()const`, `void Set’NAME’( TYPE value )` and both. Parameter *`NAME`* is used in name of Get/Set method and parameter *`MEMBER_NAME`* is name of `Properties` member accessed by these two methods.

```
template< typename ElementType >
class ThresholdingFunctor
{
public:
    void
    operator()( const ElementType&    input, ElementType& output )
    {
        if( input < bottom || input > top ) {
            output = outValue;
        } else {
            output = input;
```

```
        }
    }

    ElementType    bottom;
    ElementType    top;

    ElementType    outValue;
};

template< typename ImageType >
class ThresholdingFilter
: public AbstractImageElementFilter<
    ImageType,
    ImageType,
    ThresholdingFunctor< typename ImageTraits< ImageType >::ElementType >
    >
{
public:
    typedef ThresholdingFunctor
        < typename ImageTraits< ImageType >::ElementType >     Functor;

    typedef Imaging::AbstractImageElementFilter
        < ImageType, ImageType, Functor >                      PredecessorType;

    typedef typename ImageTraits< ImageType >::ElementType     InputElementType;

    struct Properties : public PredecessorType::Properties
    {
        Properties(): bottom( 0 ), top( 0 ), outValue( 0 ) {}

        InputElementType    bottom;
        InputElementType    top;

        InputElementType    outValue;

        void
        CheckProperties() {
            _functor->bottom = bottom;
            _functor->top = top;
            _functor->outValue = outValue;
        }

        Functor    *_functor;
    };

    ThresholdingFilter( Properties  * prop );
    ThresholdingFilter();

    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, Bottom, bottom );
    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, Top, top );
    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, OutValue, outValue );
protected:
```

```
private:
    GET_PROPERTIES_DEFINITION_MACRO;

};

template< typename ImageType >
ThresholdingFilter< ImageType >
::ThresholdingFilter()
    : PredecessorType( new Properties() )
{
    GetProperties()._functor = &(this->_elementFilter);
}

template< typename ImageType >
ThresholdingFilter< ImageType >
::ThresholdingFilter( typename ThresholdingFilter< ImageType >::Properties *prop )
    : PredecessorType( prop )
{
    GetProperties()._functor = &(this->_elementFilter);
}
```

2.3.3. Defining new dataset type

- Designing dataset interfaces and synchronization system
- Implementing connection objects and input/output ports
- Creating filter base classes working on this dataset type