Ettention Developer Guide (1.0.4)

Checkout

Download the ettention software package from www.ettention.org.

Building

General information: ettention uses cmake as a makefile generator. Visual studio solution files are generated from the cmake files and the file hierarchy is built automatically. Precompiled packages for the dependencies are automatically downloaded and installed in the folder “ettention_dependencies”, by default located in the same directory as the folder with the ettention source code. This means, if ettention is located in c:\development\ettention, the dependencies will by default be located in c:\development\ettention_dependencies. Dependencies are Boost 1_55_0, clAmdFft, FreeImage, OpenCL and Google’s gtest.

Before starting with the building process you need to get

- cmake [http://www.cmake.org/download/] and
- Microsoft Visual Studio 2013 (Release version 12.0)

Step by step building process:

I. After having downloaded the ettention software package extract the files to your favorite location.
II. Open a console and change to the directory containing ettention. Execute create_solution_msvc2013_x64.bat. This downloads the dependencies and calls cmake to create the solution files.
III. Use cmake to generate ettention by specifying source and destination (from now on the build folder). Cmake will take care for the whole configuration and generate everything needed by Visual Studio.
IV. Go to the build folder and open the Visual Studio Solution “ettention_cmake.sln”.
V. Build the solution (Ctr+Shift+B).

You can also test the behavior of ettention with the included test projects

I. libReconstructionUnitTest
II. libReconstructionIntegrationTest

by setting on of them as StartUp project and executing it (Ctrl+F5). Passing all these tests guarantees the intended behavior of ettention as it is delivered.

Plugins should bring their own test suite, and several plugin specific test suites are already included:

I. STEMPluginTest
II. ContourPluginTest
III. WBPPluginTest
IV. WeightedSIRTPluginTest

How To’s

Write a plugin

Ettention is extended via plugins. Every plugin contains one main class, derived from the Plugin interface. The plugin class is used to instantiate classes that are used to extend ettention. Currently, ettention supports extending the following concepts via plugins:

- reconstruction algorithms
- forward projections
- back projections
- data sources (i.e. input file formats)
- volume writer (i.e. output file format)
- projection iterators
- parameter sources (typically used to support new config file formats)

The plugin class is implemented in a header and looks roughly like this:

YourNewPlugin.h

```
#ifdef YourNewPlugin_EXPORTS
#define YourNewPLUGIN_API __declspec(dllexport)
#else
#define YourNewPLUGIN_API __declspec(dllimport)
#endif
```

The dll declarations are required to export the declared symbols. A detailed discussion of the topic can be found here: https://msdn.microsoft.com/de-de/library/a90k134d.aspx

```
#include "framework/plugins/Plugin.h"

namespace ettention
{
    namespace yournew
    {

    class PLUGIN_API YourNewPlugin : public ettention::Plugin {
```

In order to create a new plugin, subclass ettention::Plugin. Every class inside your plugin must be declared using the PLUGIN_API statement. The ettention::Plugin interface contains methods to instantiate class factories and is used as the entry point for your extensions. For example, in order to create a new reconstruction algorithm, you have to override the method instantiateReconstructionAlgorithm.

```
public:
    YourNewPlugin(void);

    virtual ~YourNewPlugin(void);
    virtual std::string getName() override;

    virtual ReconstructionAlgorithm* instantiateReconstructionAlgorithm(
    std::string identifier, CLAbstractionLayer* al, ParameterSource* ) override;
```

```
	} // class
	} // namespace yournew
} // namespace ettention
```

A plugin does not have to provide every possible feature. In order to allow plugins that provide for example a new reconstruction algorithm but no new forward projection, the ettention::Plugin class provides empty implementations for most functions. The only method you have to override is the getName member.

YourNewPlugin.cpp

```
#include "stdafx.h"

#include "YourNewPlugin.h"
#include "YourReconstructionAlgorithm.h"

namespace ettention {
    namespace yournew {

        YourNewPlugin::YourNewPlugin()
        {
        }

        YourNewPlugin::~YourNewPlugin()
        {
        }

        std::string YourNewPlugin::getName()
        {
                return "YourNew Plugin";
        }

        ReconstructionAlgorithm* YourNewPlugin::instantiateReconstructionAlgorithm(
        std::string identifier, CLAbstractionLayer* al, ParameterSource* parameterSource )
        {
                if (identifier == "yournew") {
                        return new YourReconstructionAlgorithm ( al, framework );
                return NULL;
        }

        } // namespace yournew
} // namespace ettention
```

In the example above, the plugin provides a new reconstruction algorithm. Apart from the implementation of the plugin class, the main file of the plugin also requires the entry function for your plugin. This function must be named initializePlugin and return a pointer to the plugin class.

```
extern "C"
YOURNEWPLUGIN_API ettention::Plugin* initializePlugin()
{
        return new YourNewPlugin();
}
```

As mentioned above, a plugin can provide a number of different, extensible objects. Besides the already given functions in the example you can add one or more of the following ones:

- `virtual ReconstructionAlgorithm* instantiateReconstructionAlgorithm( std::string identifier, CLAbstractionLayer* al, ParameterSource* );`

- ```
  virtual ParticleDetectionAlgorithm* instantiateParticleDetectionAlgorithm( std::string identifier, CLAbstractionLayer* al, ParameterSource* );
  ```

- ```
  virtual ProjectionSet* instantiateProjectionIterator(std::string identifier, ImageStackDatasource* source);
  ```
- ```
  virtual std::vector<ParameterSource*> instantiateParameterSource();
  ```

- ```
  virtual void registerImageStackDataSourcePrototype( ImageStackDatasourceFactory* factory );
  ```
- ```
  virtual void registerForwardProjectionCreator( ForwardProjectionFactory* factory );
  ```
- ```
  virtual void registerBackProjectionCreator(BackProjectionFactory* factory);
  ```

As a final step you have to make sure to set all necessary information for cmake. That means:

I. You have to “add subdirectories and dependencies” in the equally named section of the CMakeLists file in the ettention folder.

```
add_subdirectory(plugins/YourNewPlugin)
add_dependencies(YourNewPlugin libReconstruction)
add_dependencies(YourNewPlugin AnyFurtherDependence)
```

II. You have to create a CMakeLists for YourNewPlugin in “ettention/plugins/YourNewPlugin”.

That’s it. The plugin is operational now. Re-run cmake and the plugin is built together with the ettention solution upon the next build. However, we strongly encourage writing unit-tests for every step taken, so you might want to check out the next tutorial as well.

Write a test case

Having written YourNewPlugin you have to test it. Ettention uses the Google C++ Testing Framework for that cause. Further information on GoogleTest can be found at https://code.google.com/p/googletest/, where you can also find some tutorials.

For ettention to be able to execute YourNewPluginTest some things have to be done. You have to create a folder test in “ettention/plugin/YourNewPlugin”. Be reminded that YourPlugin already exists as demo in ettention.

YourNewPluginTest also needs a CMakeLists file. An example and some explanations follow:

- You need to specify the name of your Test class
  ```
  cmake_minimum_required(VERSION 2.8)
  project(YourNewPluginTest)
  ```

- You must set your source files destination and names
  ```
  set(SOURCE_GENERAL
      “src/source1.cpp”
      “src/source2.cpp”
  )
  ```

- You have to set the include directories of YourNewPlugin
  ```
  set(YOURNEWPLUGIN_INCLUDE_DIR "${CMAKE_CURRENT_SOURCE_DIR}/../src")
  set(YOURNEWPLUGIN_INCLUDE_DIR ${YOURNEWPLUGIN_INCLUDE_DIR} PARENT_SCOPE)
  ```

- Add YourNewPlugin include directory
  ```
  include_directories(“..”,..,”${YOURNEWPLUGIN_INCLUDE_DIR}”)
  ```

- Add YourNewPlugin to the target link libraries
  ```
  target_link_libraries(${PROJECT_NAME}
      libReconstruction
      YourNewPlugin
  gtest
  “${OPENCL_LIBRARIES}”)
  ```

Now, YourNewPluginTest should be integrated by cmake and executed in ettention.

Example: Weighted Back Projection

In the following, we demonstrate how the building blocks described above are used to assemble new algorithms. A first example is the well-known Weighted Back Projection (WBP) algorithm. WBP is a direct inversion method, which corresponds to a “single iteration algorithm” in Ettention.

The Ettention framework is built around the idea of assembling building blocks that can be combined as “pipelines”. As a consequence, most algorithms consist of two steps: a setup step, where the required building blocks are instantiated, and an execution step, where the execution of the individual blocks is triggered. WBP requires two main components: a filter step, which is implemented as a convolution with the Ram-Lak filter, and a real-space back projection. More advanced filter operations such as combinations with Hamming windows can be easily implemented by exchanging the filter kernel. The setup of the pipeline is implemented in the constructor of the WBP class.

```
WBP::WBP(Framework* framework, WBP_ParameterSet* parameterSet)
{
    // open the input stack specified by a configuration file and extract projection geometry
    inputStack = framework->openInputStack(parameterSet);
    geometricSetup = new GeometricSetup(inputStack);

    // allocate an output volume, resolution is specified by configuration
    volumeOnCpu = new Volume(VoxelType::FLOAT_32,
                             parameterSet->Get<VolumeParameterSet>());

    // allocate objects on the GPU
    volume = new GPUMappedVolume(framework->getCLStack(), volumeOnCpu);
    sourceImage = new GPUMapped<Image>(framework->getCLStack(), inputStack->getResolution());

    // allocate compute kernels and generate filter mask
    filterSize = Vec2ui(31, 31);
    ramLakGenerator = new GenerateRamLakKernel(framework->getCLStack(), filterSize);
    ramLakGenerator->launchKernel();
    convolution = new ConvolutionOperator(framework->getCLStack(),
                                          ramLakGenerator->getOutput(), sourceImage);
    backprojection = new BackProjectionOperator(framework, parameterSet, geometricSetup,
                                                volume, convolution->getOutput());
}
```

The execution is implemented in the run method. It consist of an iteration over all input projections. Each input projection is first filtered by means of a convolution and subsequently, the back projection is executed.

```
void WBP::Run()
{
    // iterate over input stack, copy to GPU, filter each image, and perform back projection
    for (int i = 0; i < inputStack->GetNumberOfProjections(); i++)
    {
        sourceImage->setObjectOnCPU(inputStack->loadProjection(i));
        convolution->launch();
        backprojection->launch();
    }
}
```

Example: Weighted SIRT

Based on the building blocks introduced with the WBP algorithm, we can now consider more interesting examples. The weighted SIRT algorithm combines the ideas of using a convolution with a ramp filter in an iterative approach for tomographic reconstruction. The algorithm can be implemented in Ettention by extending the SIRTOperator class. In addition to the SIRT functionality, a convolution with a Ram-Lak filter is required. More complex filters can be created the same way.

```
WeightedSIRT::WeightedSIRT(Framework* framework,
                           AlgebraicParameterSet* parameterSet)
  : SIRTOperator(framework, parameterSet)
{
    ramLakGenerator = new GenerateRamLakKernel( framework->getClStack(), filterSize );
    ramLakGenerator->launchKernel();
    convolution = new ConvolutionOperator(framework->getClStack(),
                                          ramLakGenerator->getOutput(), NULL, false);
}
```

The convolution operator can now be used to replace the computeResidual function of the SIRT algorithm. In this implementation, the residual is first generated by computing the per pixel difference between real projection and virtual projection, as done for the SIRT algorithm. The difference is then filtered by computing the convolution with a Ram-Lak filter kernel.

```
void WeightedSIRT::ComputeResidual(void)
{
    // compute per pixel difference of real- and virtual projection
    compareKernel->SetInput(state->GetCurrentRealProjection(), state->GetVirtualProjection());
    compareKernel->launchKernel();

    // filter the residual
    convolution->setInput(compareKernel->getOutput());
    convolution->setOutput(state->GetResidual());

    convolution->launch();
}
```

After the modification of the residual value, the SIRT operator can proceed as normal.

The examples given above are achieved by recombining existing building blocks on C++ level. Some more advanced algorithms require implementing new building blocks, such as customized forward and back projections using OpenCL.

Example: Make use of Geometric Prior Knowledge

In the following example, we demonstrate how an algorithm using geometric prior knowledge can be implemented using ettention. The algorithm loads a “mask volume” containing the outer contour of the irregular reconstruction volume. The mask volume has a voxel value of one inside the reconstruction area and zero outside. As the prior knowledge algorithm cannot be implemented using existing operators, it requires writing new kernels. In the following section, we therefore also demonstrate how to write new compute kernels in the framework.

The forward projection uses a ray tracing implementation based on the digital differential analyzer (DDA) algorithm. A configurable DDA implementation is contained in ettention. In order to use it for new forward projections, four define statements are needed to specify how a ray is initialized, how parameters are passed to a ray, what filter operation should happen per voxel and what result is returned from the ray. For a normal forward projection, the ray simply accumulates the gray values of the voxels along that ray. In our example, the ray accumulates two values: the first contains the gray values along the mask, the second value accumulates the mask values, i.e. it contains the length of the intersection between the ray and the irregular reconstruction area.

```
#define INIT_RAY              float accValue = 0.0f; \\
                              float accRayLength = 0.0f;
#define ADDITIONAL_DDA_PARAMS MEMORY3D(volume),MEMORY3D(mask)
#define FILTER_OPERATOR       accValue += deltaT * READ_MEMORY3D(volume, volumeCoordinate); \\
                              accRayLength += deltaT * READ_MEMORY3D(mask, volumeCoordinate);
#define RETURN_RESULT         return (float2) ( accValue, accRayLength );

#include "DDARaytracer.hl"
```

With the configured DDA ray tracer, implementing the modified forward projection is now straight forward.

```
__kernel void forward( ... )
{
    // compute memory address of result
    const uint2 gid = (uint2) ( get_global_id(0), get_global_id(1) );
    unsigned int imageIndex = gid.y * projectionResolution.x + gid.x;


    // compute ray start and direction for this pixel
    const float4 source = getRaySourcePosition( ... );
    const float4 ray = getRayDirection( projectionGeometry->ray );

    // traverse the ray using DDA
    float2 traversalResult = ddaTraverseRay(source, ray, volumeGeometry,
                                            MEMORY3D_PARAMS(volume), MEMORY3D_PARAMS(mask) );

    // store gray value and mask value
    projection[imageIndex] = traversalResult.x;
    volume_traversal_length[imageIndex] = traversalResult.y;
}
```

The reconstruction algorithm also requires a back projection operation. The back projection is implemented voxel by voxel as follows. The center of each voxel

```
__kernel void back( ... )
```

```
{
    // compute memory address of result
    const uint3 gid = (uint3)(get_global_id(0), get_global_id(1), get_global_id(2));
    const uint voxelIndex = Flatten3D(gid, volumeGeometry->subVolumeResolution);

    // compute voxel center position and project to image plane
    float3 samplingPos = GetVoxelCenter(gid, volumeGeometry);
    float2 projected = TransformCoordG(transform, samplingPos).xy;

    // read residual value and ray length (for normalization)
    float rayLengthValue = READ_MEMORY2D(rayLength, projected);
    float residualValue = READ_MEMORY2D(projection, projected);

    // compute correction term
    const float correction = residualValue * lambda / rayLengthValue;;

    // correct the voxel value.
    float volumeValue = READ_FLOAT(volume,  voxelIndex);
    float maskValue   = READ_FLOAT(mask, voxelIndex);

    // if maskValue is zero, the product is zero also and the voxel remains unchanged
    volumeValue += correction * maskValue;
    STORE_FLOAT(volume, voxelIndex, volumeValue);
}
```