Zebra
Aurora™ Vision

Aurora Vision Library 5.3

Introduction

Created: 6/8/2023

Product version: 5.3.4.94078

Table of content:

- Overview
- Programming Conventions
- Aurora Template Library

Overview

Introduction

Aurora Vision Library is a machine vision library for C++ and .NET programmers. It provides a comprehensive set of functions for creating industrial image analysis applications – from standard-based image acquisition interfaces, through low-level image processing routines, to ready-made tools such as template matching, measurements or barcode readers. The main strengths of the product include the highest performance, modern design and simple structure making it easy to integrate with the rest of your code.

The scope of the library encompasses:

- **Image Processing**
  High performance, any-shape ROI operations for unary and binary image arithmetics, refinement, morphology, smoothing, spatial transforms, gradients, thresholding and color analysis.
- **Region Analysis**
  Robust processing of pixel sets that correspond to foreground objects: extraction, set arithmetics, refinement, morphology, skeletonization, spatial transformations, feature extraction and measurements.
- **Path Analysis**
  Subpixel-precise alternative to region analysis, particularly suitable for shape analysis. Provides methods for contour extraction, refinement, segmentation, smoothing, classification, global transformations, feature extraction and more.
- **Profiles**
  Auxiliary toolset for analysis of one-dimensional sequences of values, e.g. image sections or path-related distances.
- **Histograms**
  Auxiliary toolset for value distribution analysis.
- **Geometry 2D**
  Exhaustive toolset of geometric operations compatible with other parts of the library. Provides operations for measuring distances and angles, determining intersections, tangents and feature.
- **1D Edge Detection**
  Detection of edges, ridges and stripes (paired edges) by the means of 1D edge scanning, i.e. by extracting and analysing a profile along a specified path.
- **2D Edge Detection**
  Detection of edges by the means of 2D edge tracing, i.e. by extracting and refining locally maximal image gradients.
- **Fourier Analysis**
  Suitable both for educational experimentation and industrial application, this toolset provides methods for Fourier transform and image processing in the frequency domain.
- **Template Matching**
  Efficient, robust and easy to use methods for localizing objects using a gray-based or an edge-based model.
- **Barcodes**
  Detection and recognition of many types of 1D codes.
- **Datacodes**
  Detection and recognition of QR codes and DataMatrix codes.
- **Hough Transform**
  Detection of analytical shapes using the Hough transform.
- **Image Segmentation**
  Automated extraction of object regions using gray or edge information.
- **Multilayer Perceptron**
  Artificial neural networks.
- **Optical Character Recognition**
  Text recognition or validation, including dot print.
- **Shape Fitting**
  Subpixel-precise detection of analytical shapes, whose rough locations are known.

Relation between Aurora Vision Library and Aurora Vision Studio

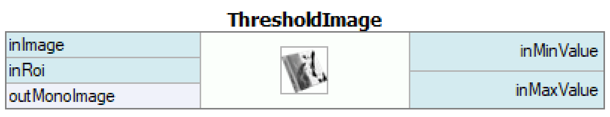
Each function of the Aurora Vision Library is the basis for the corresponding filter available in **Aurora Vision Studio**. Therefore, it is possible (and advisable) to use the Aurora Vision Studio as a convenient, drag & drop prototyping tool, even if one intends to develop the final solution in C++ using Aurora Vision Library. Moreover, for extended information about how to use advanced image analysis techniques, one can refer to Machine Vision Guide from the documentation of **Aurora Vision Studio**.

In the table below we compare the ThresholdImage function with the ThresholdImage filter:

Aurora Vision Library:

```
void ThresholdImage
(
 const Image&   inImage,
 Optional<const Region&> inRoi,
 Optional<real>   inMinValue,
 Optional<real>   inMaxValue,
 real     inFuzziness,
 Image&     outMonoImage
);
```

Aurora Vision Studio:



Key Features

Performance

In Aurora Vision Library careful design of algorithms goes hand in hand with extensive hardware optimizations, resulting in performance that puts the library among the fastest in the world. Our implementations make use of **SSE instructions** and **parallel computations** on multicore processors.

Modern Design

All types of data feature **automatic memory management**, errors are handled explicitly with **exceptions** and **optional types** are used for type-safe special values. All functions are **thread-safe** and use **data parallelism** internally, when possible.

Consistency

The library is a simple collection of types and functions, provided as a single DLL file with appropriate headers. For maximum readability function follow consistent naming convention (e.g. the VERB + NOUN form as in: ErodeImage, RotateVector). All results are returned via reference output parameters, so that many outputs are always possible.

Example Program

A simple program based on the Aurora Vision Library may look as follows:

```cpp
#include <AVL.h>

using namespace atl;
using namespace avl;

int main()
{
 try
 {
  InitLibrary();
  Image input, output;
  LoadImage("input.bmp", false, input);
  ThresholdImage(input, NIL, 128, NIL, 0, output);
  SaveImage(output, NIL, "output.bmp", false);
  return 0;
 }
 catch (const atl::Error&)
 {
  return -1;
 }
}
```

Please note that Aurora Vision Library is distributed with a set of example programs, which are available after installation.

Programming Conventions

Organization of the Library

Aurora Vision Library is a collection of C++ functions that process machine vision related types of data. Each function corresponds to a single data processing operation, e.g. DetectEdges_AsPaths performs a Canny-like 2D edge detection. As a data processing library, it is not particularly object-oriented. It does use, however, modern approach to C++ programming with automatic memory management, exception handling, thread safety and the use of templates where appropriate.

Namespaces

There are two namespaces used:

- **atl** – the namespace of types and functions related to Aurora Template Library.
- **avl** – the namespace of types and functions related to Aurora Vision Library as the whole.
- **avs** – Aurora Vision Studio Code Generator equivalents of Aurora Vision Library functions. Not recommended to use.

Enumeration Types

All enumeration types in Aurora Vision Library use C++0x-like namespaces, for example:

```
namespace EdgeFilter
{
    enum Type
    {
        Canny,
        Deriche,
        Lanser
    };
}
```

This has two advantages: (1) some identifiers can be shared between different enumeration types; (2) after typing "EdgeFilter::" IntelliSense will display all possible elements of the given enumeration type.

Example:

```
atl::Array<avl::Path> edges;
avl::Image image, gradientImage;

avl::DetectEdges_AsPaths(image, atl::NIL, avl::EdgeFilter::Canny,
                         2.0f, atl::NIL, 60.0f, 30.0f, atl::NIL, 30.0f, 0.0f, atl::NIL, 0.0f, edges, gradientImage);
```

Function Parameters

Contrary to standard C++ libraries, machine vision algorithms tend to have many parameters and often compute not single, but many output values. Moreover, diagnostic information is highly important for effective work of a machine vision software engineer. For these reasons, function parameters in Aurora Vision Library are organized as follows:

1. First come **input parameters**, which have "in" prefix.
2. Second come **output parameters**, which have "out" prefix and denote the results.
3. The last come **diagnostic output parameters**, which have "diag" prefix and contain information that is useful for optimizing parameters (not computed when the diagnostic mode is turned off).

For example, the following function invocation has a number of input parameters, a single output parameter (*edges*) and a single diagnostic output parameter (*gradientImage*).

```
atl::Array<avl::Path> edges;
avl::Image image, gradientImage;

avl::DetectEdges_AsPaths(image, atl::NIL, avl::EdgeFilter::Canny,
                         2.0f, atl::NIL, 60.0f, 30.0f, atl::NIL, 30.0f, 0.0f, atl::NIL, 0.0f, edges, gradientImage);
```

Diagnostic Output Parameters

Due to efficiency reasons the diagnostic outputs are only computed when the diagnostic mode is turned on. This can be done by calling:

```
avl::EnableAvlDiagnosticOutputs(true);
```

In your code you can check if the diagnostic mode is turned on by calling:

```
if (avl::GetAvlDiagnosticOutputsEnabled())
{
 //...
}
```

Optional Outputs

Due to efficiency reasons computation of some outputs can be skipped. In function TestImage user can inform function that computation of **outMonoImage** is not necessary and function can omit computation of this data.

the TestImage Header with last two optional parameters:

```
void avl::TestImage
(
 avl::TestImageId::Type inImageId,
 atl::Optional<avl::Image&> outRgbImage = atl::NIL,
 atl::Optional<avl::Image&> outMonoImage = atl::NIL
)
```

Example of using optional outputs:

```
avl::Image rgb, mono;

// Both outputs are computed
avl::TestImage(avl::TestImageId::Baboon, rgb, mono);

// Only RGB image is computed
avl::TestImage(avl::TestImageId::Baboon, rgb);

// Only mono image is computed
avl::TestImage(avl::TestImageId::Baboon, atl::NIL, mono);
```

In-Place Data Processing

Some functions can process data *in-place*, i.e. modifying the input objects instead of computing new ones. There are two approaches used for such functions:

1. Some filters, e.g. the image drawing routines, use "io" parameters, which work simultaneously as inputs and outputs. For example, the following function invocation draws red circles on the *image1* object:
   ```
   avl::DrawCircle(image1, circle, atl::NIL, avl::Pixel(255, 0, 0), style);
   ```
2. Some filters, e.g. image point transforms, can be given the same object on the input and on the output. For example, the following function invocation negates pixel values without allocating any additional memory:
   ```
   avl::NegateImage(image1, atl::NIL, image1);
   ```

Please note, that simple functions like NegateImage can be executed even 3 times faster in-place than when computing a new output object.

Work Cancellation

Most of long-working functions can be cancelled using CancelCurrentWork function. Cancellation technique is thread-safe, so function CancelCurrentWork can be called from different thread.

To check cancellation status use the IsCurrentWorkCancelled function.

```
void ProcessingThread()
{
 while (!avl::IsCurrentWorkCancelled())
 {
  std::cout << "Iteration start" << std::endl;
  avl::Delay(10000); // Function with cancellation support
  std::cout << "Iteration complete" << std::endl;
 }
 std::cout << "Processing thread stop" << std::endl;
}

int main()
{
 avl::InitLibrary();
 std::thread thread {ProcessingThread};

 std::cout << "Press Enter to stop execution." << std::endl;
 std::cin.get();

 // Cancel work in ProcessingThread and in avl::Delay
 avl::CancelCurrentWork();

 thread.join();

 return 0;
}
```

Library Initialization

For reasons related to efficiency and thread-safety, before any other function of the AVL library is called, the InitLibrary function should be called first:

```
int main()
{
 avl::InitLibrary();
 //...
}
```

Debug Preview

For diagnostic purposes it is useful to be able to preview data of types Image and Region. You can achieve this by using functions from the Debug Preview category. They can be helpful in debugging programs and displaying both intermediate and final data of such types.

```
avl::Image image;
avl::LoadImage("hello.png", false, image);

// Show loaded image in new window.
avl::DebugPreviewShowNewImage(image);

// Wait until window is closed.
avl::DebugPreviewWaitForWindowsClose();
```

Aurora Template Library

Aurora Vision Library is based on the Aurora Template Library – a simplified counterpart of the C++ Standard Template Library, which avoids advanced templating techniques mainly by using raw pointers instead of abstract iterators. This makes Aurora Vision Library portable to embedded platforms, including the ones that do not support C++ templates fully.

Please note, that the following types should only be parametrized with fundamental types (int, float, etc.) or types from avl or atl namespace. Const and/or reference types are also allowed, as long as template type accepts such type (e.g. Array<T> cannot be parametrized with reference type).

Array<T>

The Array<T> type strictly corresponds to std::vector<T>. It is a random-access, sequential container with automatic memory reallocation when growing.

Here is a simplified version of the public interface is depicted: Array.h

Optional<T>

The Optional<T> type provides a consistent way of representing an optional value, something for which NULL pointers or special values (such as -1) are often used. Many APIs provide optional values using default values of parameters. This type is inspired by boost::optional<T> class from the Boost Library, but is designed mostly for input parameters, not only for function results.

In Aurora Vision Library it is used to represent optional regions of interest in image processing operations and many other input parameters that can be determined automatically when not provided by an user.

Documentation for this type is presented in Optional.h.

Sample use:

```
atl::Optional<avl::Point2D> p;
p = avl::Point2D(10, 25);        // normal value
p = atl::NIL;               // NIL value
if (p != atl::NIL)
{
 avl::Point2D q = p.Get();    // access to a non-nil value
 p.Get().x = 15;         // direct access to a field
}
```

Conditional<T>

This type of data is especially used to determine invalid results. Many functions in C return special value as -1 or NULL when their result is invalid. Type Conditional<T> is very similar to Optional<T>, but it is mostly used in outputs.

Documentation for this type is presented in Conditional.h.

Sample use:

```
atl::Conditional<int> result;
avl::ParseInteger("Test1", avl::NumberSystemBase::Base_10, result); // Parsing textual data

if (result != atl::NIL)   // If textual data is not valid integer result has value atl::NIL
 printf("Valid integer.");
else
 printf("Invalid integer. Value: %d", result.Get());
```

Dummy<T>

Dummy<T> class is used to create a temporary object that will be released after its use. It is mostly used to create a temporary object to pass its reference to a function. Such temporary objects are helpful when not all values returned by a function are important and we don't plan to use them.

Sample use:

```
avl::Region region;
avl::Circle2D circle = avl::Circle2D(50.0f, 50.0f, 50.0f);

avl::CreateCircleRegion(circle, atl::NIL, 100, 100, region);

// Second parameter is not used.
avl::Segment2D minorAxis;
avl::RegionEllipticAxes(region, atl::Dummy<avl::Segment2D>(), minorAxis);

std::cout << "Minor axis length: " << minorAxis.Length();
```

Zebra
Aurora™ Vision