mViz, comprehensive machine vision libraries

For processing, analysis, guidance and identification

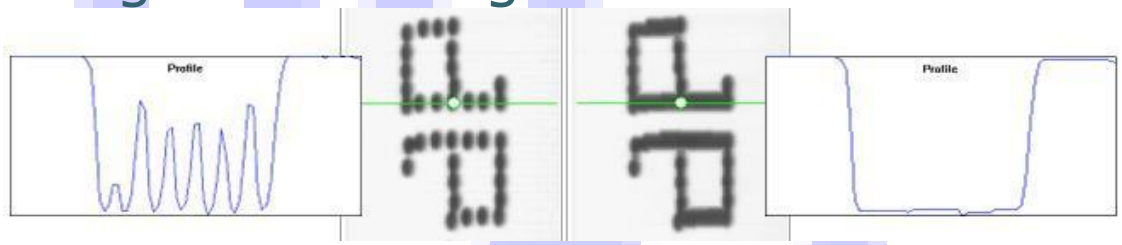
Compatible with the Microsoft C++, C# and Visual Basic environments

Linux gcc, 32 and 64 bits

mViz is not just "another image processing package". We want full customer satisfaction and we know that the learning curve for a first project can be costly. This is why free consultancy hours are included to help you establish a proof of concept very quickly and at a low risk.

mViz is licensed by separate modules or in bundles, run-times per machine, with an attractive pricing scheme. Feature enhancements, porting and special adaptations are possible on request.

Image Processing



The processing functions are used to prepare the image by reducing unwanted nuisances such as noise or blur, or enhance some properties such as connexity of characters, contrast… They usually turn images into other images.

This function set comprises point-to-point operations such as pixel arithmetic, lookup table transforms or shading correction. It also handles geometric transforms like rotation and scaling that allow f.i. deskewing or size normalization.

Point-to-point arithmetic & logic, grayscale & color transforms, shading correction, convolutions, grayscale morphology, geometric transforms, polar unwarping

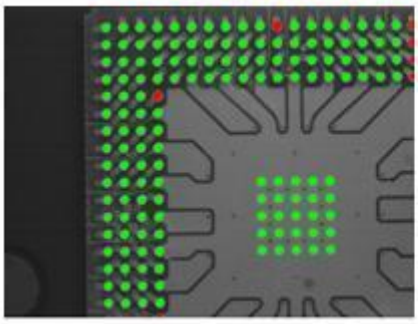
```
// Sample code

// Rotate 30°, scale +20% around the center, with interpolation
Geometry::Rotate(XY(320, 240), XY(320, 240), 30, 1.2, Source, Rotated, true);

// Two points shading correction with Black and White reference images
Operator::Correct(Black, White, Source, Corrected, 0, 255);
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Addition of two images	0.055
Pixel-wise maximum of two images	0.036
Pixel-wise greater-or-equal comparison	0.041

Automatic binarization	0.88
Two-point shading correction	1.0
Sobel magnitude	0.30
Prewitt magnitude	0.29
Binomial low-pass (3x3)	0.15
Uniform low-pass (25x25)	1.9
Sharpening filter (3x3)	0.16
Uniform high-pass (25x25)	2.0
User-defined convolution (3x3)	9.9
User-defined convolution (5x5)	17
Morphological dilation (3x3 square)	0.079
Morphological opening (7x7 diamond)	0.46
Morphological top-hat (5x5 octagon)	0.34
Downsampling (2:1)	0.98
Upsampling (1:2)	4.6
Rotation (nearest neighbor)	0.71
Rotation (bilinear)	1.8
Polar unwarping (bilinear)	5.5

Image Analysis



Area	Center X	Center Y	Ellipse Width	Ellipse Height	Ellipse Angle	Eccentricity
218	382,537	70,032	17,707	15,883	39,495	0,594
218	544,628	357,716	16,780	16,619	-53,532	0,194
218	576,229	325,560	16,959	16,438	1,446	0,343
218	318,560	70,592	17,821	15,861	62,623	0,610
217	700,977	67,502	17,389	16,202	85,909	0,496
217	480,636	358,581	16,706	16,619	38,097	0,144
217	318,438	38,401	17,165	16,246	71,530	0,444
217	577,221	421,493	16,906	16,413	-3,182	0,334
217	608,498	357,106	16,795	16,534	78,295	0,246
216	732,801	67,106	16,782	16,646	-44,115	0,179

The analysis functions extract condensed information from images or regions in order to characterize and classify 1D or 2D features.

They encompass histogram analysis, straight or curved profile extraction, and blob analysis. When numerical values are obtained, they can be used to compare objects to known references and discriminate between them.

Profile processing, histogram statistics, blob analysis, region masking, contouring

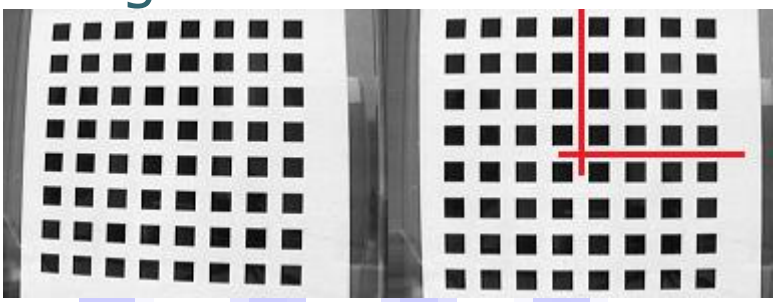
```
// Sample code

// Get the trimmed gray-level mean with 10% trimming
double Mean= Histogram::TrimmedMean(0.1, Source);

// Accumulate the histogram of the first (largest) blob
Blobs.PixelHistogram(0, Source, Histo);
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Cumulate 16 rows	0.011
Trace a blob contour	0.033
Compute a contour area	0.0040
Fill a contour	0.14
Mean of an image	0.18
Entropy of an image	0.35

Compute an image histogram	0.33
Variance of a histogram	<0.001
Skewness of a histogram	<0.001
Best threshold from an histogram	0.0040
Segment an image (200 blobs)	0.50
Evaluate the gray-level averages (200 blobs)	0.00100
Evaluate the gray-level maxima (200 blobs)	<0.001
Evaluate the ellipse of inertia (200 blobs)	0.012
Evaluate the Feret box (200 blobs)	0.0030
Sort blobs by area (200 blobs)	0.0060
Compute a blob histogram	0.24

Image Calibration



An image is said to be calibrated when a mapping correspondence has been established between the pixel coordinates and some real-world coordinates system established on the observe surface.

The image calibration set provides support to accurately compute the direct and inverse matching from a number of "anchor points", and to deal with perspective distortion as well as optical deformations.

Scaling, isometry, Affinity, Perspective & Distortion Transforms

```
// Sample code

// Adjust a perspective transform between the source and calibrated points
Model.Append(XY(RawX, RawY), XY(GridX, GridY));
Model.Fit(Perspective);

// Undistort the source image, without bilinear interpolation
Model.Register(Source, Calibrated, false);
```

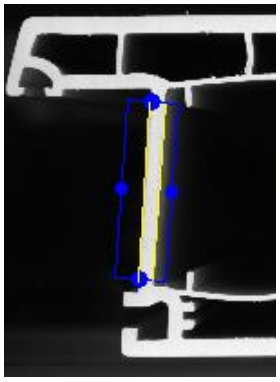
Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Affine calibration (nearest neighbor)	0.71
Affine calibration (bilinear)	1.9
Perspective calibration (bilinear)	8.7
Distortion calibration (bilinear)	8.4
Distortion and perspective calibration (bilinear)	9.4

Edge Gauging

The edges of objects usually correspond to sharp transitions in the intensities in the image.

Gauging enables very accurate measurements of edges in order to determine positions, distances, sizes, angles… with a subpixel precision.



This function set is specialized in profile analysis (peaks in a 1D signal) and fitting of straight or circular lines. It puts emphasis in robust selection of the relevant edge points.

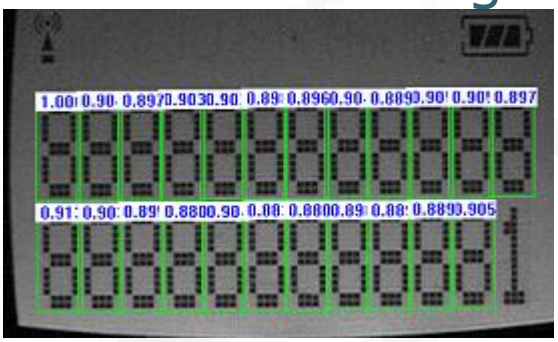
Automatic edge detection, outlier rejection, point model fitting, robust sub-pixel measurement

```
// Sample code

// Place a straight line measurement gauge
Gauge.Beg= XY(139, 88); Gauge.End= XY(343, 87);

// Fit the line and get the angle
Gauge.Detect(Source, BestStep);
double Angle= Gauge.FittedAngle;
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Detect edges along a profile (100 pixels)	<0.001
Fit a straight edge (21 profiles)	0.012
Fit a circular edge (63 profiles)	0.062

Pattern Matching



Pattern matching, also called template matching, is an excellent approach to locate known objects in an image. It works by providing a sample image of the item to be located and then searching for similar shapes in the target image.

The pattern matching tool of mViz supports translation, rotation and scaling and tolerates linear changes in light intensity. It is based on the well-known normalized correlation score.

Model training, sub-pixel location, rotation & scaling

```
// Sample code

// Train the pattern from a region of interest
Source.Window(41, 170, 219, 194);
Template.Train(Source);
```

```
// NGC search in the whole image with a rotation tolerance
Template.MinAngle= -30; Template.MaxAngle= 30;
Template.Find(Source);
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Search (160x160 template)	0.60
Search ±30° rotation (160x160)	6.4
Search ±30° rotation, ±10% scaling (160x160)	18

Geometric Matching

By contrast to standard pattern matching, the geometric finder uses edge information rather than area information. This provides better robustness to occlusion, clutter, blur, non-linear intensity changes.

The geometric and standard pattern matching tools share a common API and can easily be traded for one another.

Model training, sub-pixel location, full rotation & scaling

```
// Sample code

// Train the pattern from a region of interest
Source.Window(41, 170, 219, 194);
Template.Train(Source);

// Geometric search in the whole image with a rotation tolerance
Template.MinAngle= -30; Template.MaxAngle= 30;
Template.Find(Source, true);
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Search (160x160 template)	0.94
Search ±30° rotation (160x160)	5.5
Search ±30° rotation, ±10% scaling (160x160)	9.6

Character Reading

Many industrial applications require part identification. One of the ways to distinguish object is by printing a human-readable serial number on the surface or on a stickled label.

The character reading set serves two purposes: either to be able to read the content of the marking (OCR), or to verify that a given content has been properly printed (OCV). mViz uses user pre-trained fonts for the most accurate recognition. A priori information on the text layout can be used to further increase reliability.

Font training, deskewing, auto-segmentation, printed character recognition & verification

```
// Sample code

// Load the pre-recorded font from a file
OCR.Read("Fonts\\OCR-B.fnt");

// Perform the recognition from a region of interest and get the text
Source.Window(41, 170, 219, 194);
OCR.CharsRead(Source);
char* String= OCR.AsciiString;
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Read characters (among 54 in font)	1.4
Read characters (adaptive threshold)	3.7

Bar Code Reading

A well-known alternative to printed characters are the barcodes. They come in numerous flavors and allow storing a number of digits and/or alphabetic characters. Depending on the type, the payload ranges from a few digits to a few tens of characters.

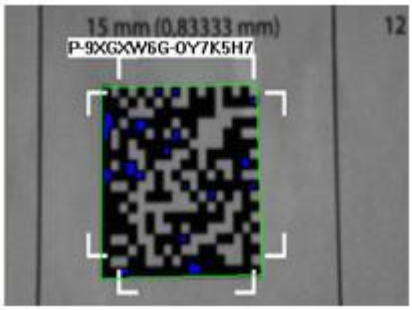
Automatic code location, recognition and decoding, scale and rotation invariant, EAN/UPC, GS1 Databar, Code 39, (color) Pharmacode… symbologies

```
// Sample code

// Find the Pharmacode from the whole image
Barcode.DetectPharmacode= true;
bool Success= Barcode.Decode(Source);

// Read the decided string
char* String= Barcode.AsciiString;
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Read a EAN128 code	0.79
Read a EAN128 code (6 directions)	2.7

Dot Code Reading



Another alternative to printed characters appeared more recently and is known as dot codes or 2D codes. They encode more information in the same space by allowing variations in two dimensions rather than one. To ensure data integrity, they also embed error detection and correction mean.

Automatic code location, recognition and decoding, scale and rotation invariant, Data Matrix, Aztec Code, QR… symbologies

```
// Sample code

// Find the Data Matrix code from the whole image
bool Success= DataMatrix.Decode(Source);

// Read the decided string
char* String= DataMatrix.AsciiString;
```

Typical running times (in milliseconds, 640x480 images, Core i7 @ 2.2 GHz)

Read a Data Matrix code	6.0