Image Blurring and Rotation with Intel® Integrated Performance Primitives

Legal Information

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Getting Technical Support

If you did not register your Intel® software product during installation, please do so now at the Intel® Software Development Products Registration Center. Registration entitles you to free technical support, product updates, and upgrades for the duration of the support term.

For general information about Intel technical support, product updates, user forums, FAQs, tips and tricks and other support questions, please visit http://www.intel.com/software/products/support/.

NOTE
If your distributor provides technical support for this product, please contact them rather than Intel.

For technical information about the Intel IPP library, including FAQ's, tips and tricks, and other support information, please visit the Intel IPP forum: http://software.intel.com/en-us/forums/intel-integrated-performance-primitives/ and browse the Intel IPP knowledge base.

Overview

Discover how to use Intel® Integrated Performance Primitives (Intel® IPP) image processing functions to implement image blurring and rotation in your application.

About This Tutorial	This tutorial demonstrates how to: • Implement box blurring of an image with the Intel IPP filtering functions • Rotate an image with the Intel IPP functions for affine warping • Set up environment to build the Intel IPP application • Compile and link your image processing application
Estimated Duration	10-15 minutes
Learning Objectives	After you complete this tutorial, you should be able to: • Understand the basic concepts of Intel IPP image processing • Use Intel IPP functions for image filtering • Use Intel IPP functions for image geometry transformation • Develop an image processing application that loads an image from a BMP file and applies rotation and blurring after user presses arrow keys • Set up environment to build the application • Compile and link your code with Intel IPP
More Resources	• *Intel IPP Developer Reference* contains detailed descriptions of functions syntax, parameters, and return values • *Intel IPP Developer Guide* provides detailed guidance on Intel IPP library configuration, development environment, domain dependencies, and linkage modes • The sample code for this tutorial can be downloaded from the Intel® Software Product Samples and Tutorials website The above documents are available at the Intel IPP documentation page or in the Intel IPP documentation directory. In addition, you can find more resources at https://software.intel.com/en-us/intel-ipp.

Next >

Introduction to the Intel® Integrated Performance Primitives Library

Intel® Integrated Performance Primitives (Intel® IPP) is an extensive library of software functions to help you develop multimedia, data processing, and communications applications. These ready-to-use functions are highly optimized using Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instruction sets.

Intel IPP supports application development for various Intel® architectures. By providing a single cross-architecture application programmer interface, Intel IPP permits software application repurposing and enables porting to unique features across Intel® processor-based desktop, server, mobile, and handheld platforms. Use of the Intel IPP primitive functions can help drastically reduce development costs and accelerate time-to-market by eliminating the need of writing processor-specific code for computation intensive routines.

One key area of the Intel IPP library is image processing, which includes various operations on two-dimensional signals like filtering, geometric transforms, color conversion, and morphological transforms.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Previous Next

Intel® IPP Image Processing Basics

This section explains some of the basic concepts used in the image processing part of Intel® IPP:

- Representing an image
- Processing regions of interest (ROIs)
- Initializing function data
- Setting image border type

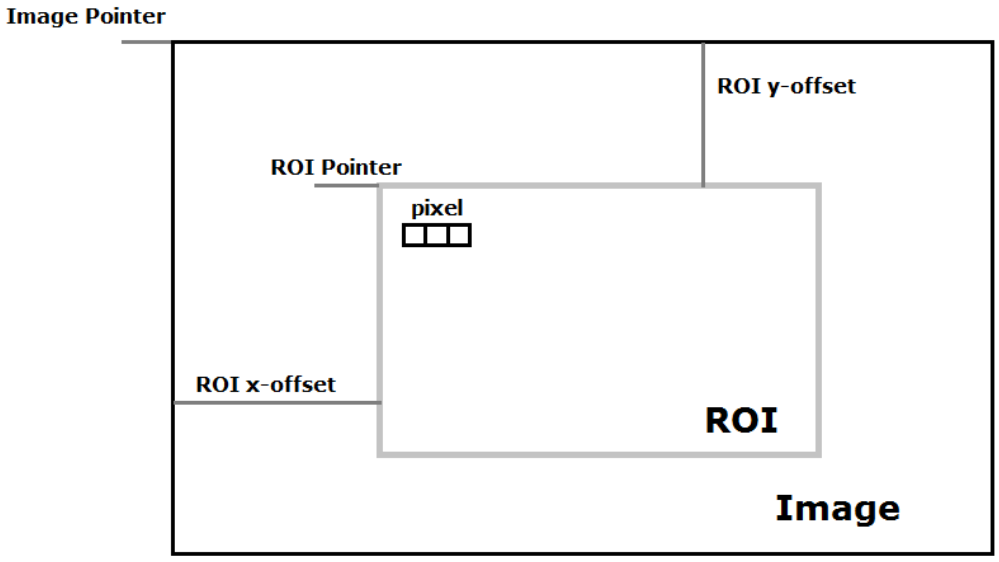
Representing an Image

An image can be represented as lines of pixels. Depending on the image format, one pixel can keep one or more integer or floating-point values. Each image row contains the same number of pixels. An image step is a value that is equal to distance, in bytes, between the starting points of consecutive lines in the image.

Processing Regions of Interest (ROIs)

Most Intel IPP image processing functions can operate not only on entire images but also on image areas. Image region of interest (ROI) is a rectangular area that can be either some part of the image or the whole image. Intel IPP functions that support ROI processing have the `R` descriptor in their names.

ROI of an image is defined by the size and offset from the image origin as shown in the figure below. The origin of the image is in the top left corner, with `x` values increasing from left to right and `y` values increasing downwards.



Initializing Function Data

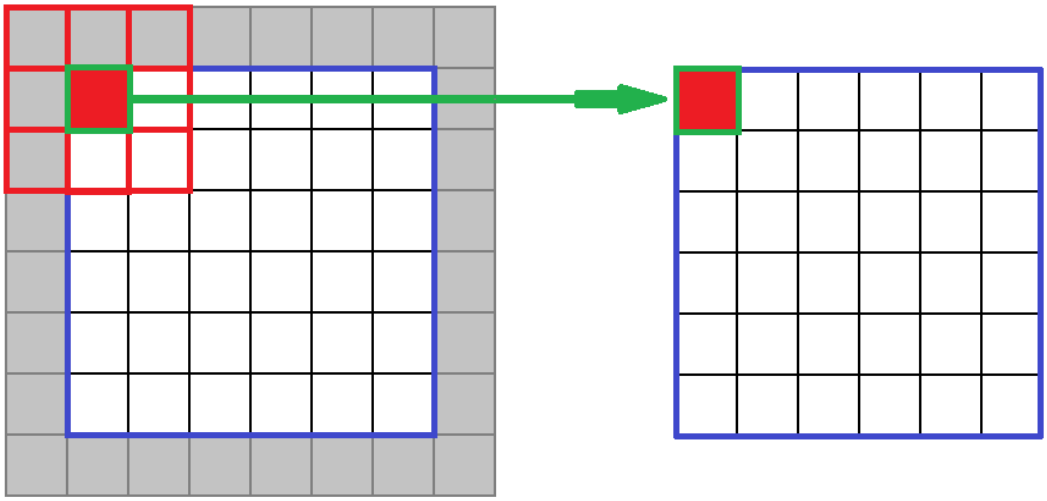
Most Intel IPP functions do not perform memory allocation and require external memory buffers to contain pre-computed values or to keep temporary data for algorithm execution. To get size of required buffers, use the following auxiliary functions:

Use This	To Do This
<*processing function*>`GetSize`	Get buffer size for precomputed data structure that is initialized by <*processing function*>`Init`.

Use This	To Do This
<*processing function*>`GetBufferSize`	Obtain size for temporary buffer that is passed to the processing function.

Setting Image Border Type

Many image processing algorithms sample a pixel by floating point (*x*, *y*) coordinates to compute an intermediate or output image. To improve the image quality, you can apply filters that use neighborhood pixels to calculate the sampled pixel value. Thus, as shown in the figure below, to obtain an image with size 6x6 by using 3x3 filter kernel (8 neighborhood pixels are used), the source image of size 8x8 is required.



Each filtering operation reduces image size. To keep the image size and process image border pixels, it is required to extend the image artificially. There are several types of border processing in Intel IPP:

Replicated borders	Border pixels are replicated from the source image edge pixels
Constant borders	Values of all border pixels are set to a constant.
Transparent borders	Destination pixels that have inverse transformed location out of the source image are not processed.
Borders in memory	Source image border pixels are obtained from the source image pixels in memory.
Mixed borders	Combination of transparent borders and borders in memory is applied.

To set border processing methods for Intel IPP functions, use the `borderType` (or `border`) and `borderValue` parameters. To get information about the list of supported border types for a particular function, refer to the *Intel IPP Developer Reference*.

‹ Previous Next ›

Blurring an Image Using ippiFilterBoxBorder

A box blur of an image is a filtering process that sets each pixel in a destination image to the average value of all pixels of a source image in a rectangular neighborhood of the specified mask size. This operation has the effect of blurring or smoothing of the source image.

Intel IPP `ippiFilterBoxBorder` function implements box blurring of an image. Before calling the processing function, you need to allocate memory for the work buffer. You can get the required memory size for the specified parameters using the `ippiFilterBoxBorderGetBufferSize` auxiliary function. The code extract below demonstrates how to use the box blur functionality of Intel IPP:

```
…
IppStatus status  = ippStsNoErr;
IppiSize maskSize = {3,3};
IppiSize srcSize   = {0};
Ipp8u* pBuffer    = NULL;
Ipp8u* pSrc       = NULL;
Ipp8u* pDst       = NULL;
int srcStep, dstStep;
…
/* assigning parameters */
…
/* Get work buffer size */
status = ippiFilterBoxBorderGetBufferSize(srcSize, maskSize, ipp8u, 3, &bufSize);

pBuffer = ippsMalloc_8u(bufSize);

/* Filter the image */
if (status >= ippStsNoErr) status = ippiFilterBoxBorder_8u_C3R(pSrc, srcStep, pDst, dstStep,
srcSize, maskSize, ippBorderRepl, NULL, pBuffer);

if (pBuffer) ippsFree(pBuffer);
…
```

For more information about the `ippiFilterBoxBorder` API and auxiliary functions, refer to the *Intel IPP Developer Reference*.

‹ Previous Next ›

Rotating an Image Using Intel IPP Warp Functions

Starting with the 9.0 version, Intel IPP enables you to use the `ippiWarpAffine<Interpolation>` functions to implement rotation of an image using pre-calculated affine coefficients based on rotation parameters (angle, `x`-, `y`-shifts). To obtain affine coefficients for the specified rotation parameters, you can apply the `ippiGetRotateTransform` function. This function computes the affine coefficients for the transform that rotates an image by the specified angle around (0, 0) and shifts the image by the specified `x`– and `y`– shift values.

You can apply the computed bounding box to change the `x`-, `y`-shifts of the rotated image to fit the transformed image to the destination ROI. To compute the bounding box for the affine transform, use the `ippiGetAffineBound` function.

Before calling the warp affine processing function, you need to:

- Compute the size of the specification structure for affine warping with the specified interpolation method using the `ippiWarpAffineGetSize` function
- Initialize the specification structure using the `ippiWarpAffine<Interpolation>Init` function
- Compute the size of the temporary work buffer required for warping using `ippiWarpAffineGetSize` and pass pointer to the buffer to the processing function

The code example below demonstrates how to perform image rotation using the Intel IPP affine warping functions:

```
IppStatus warpAffine(Ipp8u* pSrc, IppiSize srcSize, int srcStep, Ipp8u* pDst, IppiSize dstSize,
int dstStep, const double coeffs[2][3])
{
    int specSize = 0, initSize = 0, bufSize = 0;
    Ipp8u* pBuffer  = NULL;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {0, 0};
    IppiBorderType borderType = ippBorderConst;
    IppiWarpDirection direction = ippWarpForward;
    Ipp64f pBorderValue[numChannels];
    IppiWarpSpec* pSpec = NULL;
    IppStatus status = ippStsNoErr;

    for (int i = 0; i < numChannels; ++i) pBorderValue[i] = 255.0;

    /* Spec and init buffer sizes */
    status = ippiWarpAffineGetSize(srcSize, dstSize, ipp8u, coeffs, ippLinear, direction,
borderType,
        &specSize, &initSize);

    /* Allocate memory */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    /* Affine transform data initialization */
    if (status >= ippStsNoErr) status = ippiWarpAffineLinearInit(srcSize, dstSize, ipp8u,
coeffs, direction, numChannels, borderType, pBorderValue, 0, pSpec);

    /* Get work buffer size */
    if (status >= ippStsNoErr) status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);
```

```
    pBuffer = ippsMalloc_8u(bufSize);

    /* Affine transform processing */
    if (status >= ippStsNoErr) status = ippiWarpAffineLinear_8u_C3R(pSrc, srcStep, pDst,
dstStep, dstOffset, dstSize, pSpec, pBuffer);

    /* Free memory */
    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}
```

For more information about the Intel IPP Warp functions, refer to the *Intel IPP Developer Reference*.

‹ Previous Next ›

Creating an Application for Image Blurring and Rotation

The code example below represents a simple application that loads an image from the BMP file, rotates and blurs it after user presses left/right or up/down arrow keys.

The source code for this application can be downloaded from the Intel® Software Product Samples and Tutorials website.

> **NOTE** The `ipp_blur_rotate.cpp` sample code has been developed for Intel® IPP installed within Intel® Parallel Studio XE. You can use the provided code, except for the graphical rendering part, as a starting point for your application that can be built within other product suites like Intel® System Studio.

```
/* C++ source code is found in ipp_blur_rotate.cpp */

#include "ipps.h"
#include "ipp_blur_rotate.h"

#include "bmpreader.h"

#include <math.h>
#include <stdio.h>

char titleBuffer[256];

video *v;

#if (defined unix || defined UNIX)
#include <X11/keysym.h>
#define VK_LEFT     XK_Left
#define VK_UP       XK_Up
#define VK_RIGHT    XK_Right
#define VK_DOWN     XK_Down
#define VK_ESCAPE   XK_Escape
#elif defined __APPLE__
#define VK_LEFT     0xf702
#define VK_UP       0xf700
#define VK_RIGHT    0xf703
#define VK_DOWN     0xf701
#define VK_ESCAPE   0x1B
#endif

IppStatus warpAffine(Ipp8u* pSrc, IppiSize srcSize, int srcStep, Ipp8u* pDst, IppiSize dstSize,
int dstStep, const double coeffs[2][3])
{
    /* IPP functions status */
    IppStatus status = ippStsNoErr;

    /* number of image channels */
    const Ipp32u numChannels = 3;

    /* border value to extend the source image */
```

```
    Ipp64f pBorderValue[numChannels];

    /* sizes for WarpAffine data structure, initialization buffer, work buffer */
    int specSize = 0, initSize = 0, bufSize = 0;

    /* pointer to work buffer */
    Ipp8u* pBuffer  = NULL;

    /* pointer to WarpAffine data structure */
    IppiWarpSpec* pSpec = NULL;

    /* set offset of the processing destination ROI */
    IppiPoint dstOffset = {0, 0};

    /* border type for affine transform */
    IppiBorderType borderType = ippBorderConst;

    /* direction of warp affine transform */
    IppiWarpDirection direction = ippWarpForward;

    /* set border value to extend the source image */
    for (int i = 0; i < numChannels; ++i) pBorderValue[i] = 255.0;

    /* computed buffer sizes for warp affine data structure and initialization buffer */
    status = ippiWarpAffineGetSize(srcSize, dstSize, ipp8u, coeffs, ippLinear, direction,
borderType,
        &specSize, &initSize);

    /* allocate memory */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    /* initialize data for affine transform */
    if (status >= ippStsNoErr) status = ippiWarpAffineLinearInit(srcSize, dstSize, ipp8u,
coeffs, direction, numChannels, borderType, pBorderValue, 0, pSpec);

    /* get work buffer size */
    if (status >= ippStsNoErr) status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);

    /* allocate memory for work buffer */
    pBuffer = ippsMalloc_8u(bufSize);

    /* affine transform processing */
    if (status >= ippStsNoErr) status = ippiWarpAffineLinear_8u_C3R(pSrc, srcStep, pDst,
dstStep, dstOffset, dstSize, pSpec, pBuffer);

    /* free allocated memory */
    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}

void ipp_blur_rotate::process(const drawing_memory &dm)
{
    IppStatus status = ippStsNoErr;
    /* number of image channels */
    int numChannels = 3;
```

```
    /* perform filtering */
    if (bFilterUpdate)
    {
        /* temporary work buffer */
        Ipp8u* pBuffer = NULL;
        /* buffer size for filtering */
        int bufSize = 0;

        /* Get work buffer size */
        status = ippiFilterBoxBorderGetBufferSize(srcSize,maskSize,ipp8u,3,&bufSize);

        /* allocate buffer memory */
        pBuffer = ippsMalloc_8u(bufSize);

        /* Image filtering */
        if (status >= ippStsNoErr) status = ippiFilterBoxBorder_8u_C3R(pSrc, srcStep, pBlur,
blurStep, srcSize, maskSize, ippBorderRepl, NULL, pBuffer);

        /* filtration flag is dropped */
        bFilterUpdate = false;
        /* rotation operation should be applied after filtration */
        bRotateUpdate = true;

        /* free buffer memory */
        ippsFree(pBuffer);
    }

    /* perform rotation */
    if (bRotateUpdate)
    {
        IppiSize roiSize = {dstSize.width / 2, dstSize.height };
        Ipp8u* pDstRoi = pBlurRot;
        IppiRect srcRoi = {0};
        srcRoi.width  = srcSize.width;
        srcRoi.height = srcSize.height;

        /* affine transform coefficients */
        double coeffs[2][3]  = {0};

        /* affine transform bounds */
        double bound[2][2]   = {0};

        /* compute affine transform coefficients by angle and x- and y-shifts */
        if (status >= ippStsNoErr) status = ippiGetRotateTransform(angle, 0, 0, coeffs);

        /* get bounds of transformed image */
        if (status >= ippStsNoErr) status = ippiGetAffineBound(srcRoi, bound, coeffs);

        /* fit source image to dst */
        coeffs[0][2] = -bound[0][0] + (dstSize.width / 2.f  - (bound[1][0] - bound[0][0])) / 2.f;
        coeffs[1][2] = -bound[0][1] + (dstSize.height - (bound[1][1] - bound[0][1])) / 2.f;

        /* perform affine processing for the blurred image */
        if (status >= ippStsNoErr) status = warpAffine(pBlur, srcSize, blurStep, pDstRoi,
roiSize, blurRotStep, coeffs);

        /* set destination ROI for the not blurred image */
        pDstRoi = pBlurRot + roiSize.width * numChannels;
```

```
        /* perform affine processing for the original image */
        if (status >= ippStsNoErr) status = warpAffine(pSrc, srcSize, srcStep, pDstRoi, roiSize,
blurRotStep, coeffs);

        /* rotation flag is dropped */
        bRotateUpdate = false;

        /* needs to redraw the image */
        bRedraw       = true;
    }

    if (bRedraw)
    {
        drawing_area area( 0, 0, dstSize.width, dstSize.height, dm) ;

        /* pass information message to window's title */
    #if defined _WIN32
        sprintf_s(titleBuffer, sizeof(titleBuffer)/sizeof(titleBuffer[0]), "Intel(R) IPP: blur +
rotate tutorial : rotation angle %.0f : box filter mask size {%d, %d}", angle - 360.0 *
floor(angle / 360.0), maskSize.width, maskSize.height);
    #elif (defined unix || defined UNIX)
        sprintf(titleBuffer, "Intel(R) IPP: blur + rotate tutorial : rotation angle %.0f : box
filter mask size {%d, %d}", angle - 360.0 * floor(angle / 360.0), maskSize.width,
maskSize.height);
    #elif defined __APPLE__
        sprintf(titleBuffer, "Intel(R) IPP: blur + rotate tutorial : rotation angle %.0f : box
filter mask size {%d, %d}", angle - 360.0 * floor(angle / 360.0), maskSize.width,
maskSize.height);
    #endif
        v->title = titleBuffer;
        v->show_title();

        // fill the rendering area
        for (int y = 0; y < dstSize.height; ++y)
        {
            Ipp8u* dstt = pBlurRot + y * blurRotStep;
            area.set_pos( 0, y );
            for (int x = 0, j = 0; x < dstSize.width; ++x, j += 3)
            {
                area.put_pixel( v->get_color(dstt[j+2], dstt[j+1], dstt[j]) );
            }
        }

        bRedraw = false;
    }
}

/* Key processing */
void ipp_blur_rotate::onKey( int key )
{
    if (pSrc == NULL || pBlur ==NULL || pBlurRot == NULL) return;

    /* up or down arrow key is pressed */
    if (key == VK_UP || key == VK_DOWN)
    {
        /* max size of mask for image blurring */
        const IppiSize maxMaskSize = {31,31};
```

```
        /* increase or decrease mask size on 2 depending on the key */
        maskSize.width  = (key == VK_DOWN) ? maskSize.width  - 2 : maskSize.width  + 2;
        maskSize.height = (key == VK_DOWN) ? maskSize.height - 2 : maskSize.height + 2;

        /* check that both mask width and mask height are positive */
        if (maskSize.width  < 1) maskSize.width  = 1;
        if (maskSize.height < 1) maskSize.height = 1;

        /* check that both mask width and mask height are at more the maximum mask size */
        if (maskSize.width  > maxMaskSize.width)  maskSize.width  = maxMaskSize.width;
        if (maskSize.height > maxMaskSize.height) maskSize.height = maxMaskSize.height;

        /* filtration operation should be applied */
        bFilterUpdate = true;
    }

     /* left or right arrow key is pressed */
    if(key == VK_RIGHT || key == VK_LEFT)
    {
        /* increase or decrease angle on 2 depending on the key */
        angle = (key == VK_LEFT) ? angle + 2 : angle - 2;
        /* rotation operation should be applied after filtration */
        bRotateUpdate = true;
    }

    if (key == VK_ESCAPE) v->running = false;
}

bool ipp_blur_rotate::loadFileBMP( const char* bmpImageFile )
{
    /* check the pointer */
    if (bmpImageFile == NULL) return false;

    /* number of image channels */
    int numChannels = 0;

    /* Read data from a file (only bmp format is supported) */
    Status status = ReadFile(bmpImageFile, &pSrc, srcSize, srcStep, numChannels);

    if (numChannels != 3)
    {
        status = STS_ERR_UNSUPPORTED;
    }

    if (status == STS_OK)
    {
        /* set blurred image step */
        blurStep = srcStep;

        /* set rotated image size to keep whole rotated image */
        dstSize.width  = static_cast<int>(sqrt((float)srcSize.width  * srcSize.width +
srcSize.height  * srcSize.height) + 0.5f) * 2;
        dstSize.height = static_cast<int>(sqrt((float)srcSize.width  * srcSize.width +
srcSize.height  * srcSize.height) + 0.5f);

        /* set image step for blurred and rotated image */
        blurRotStep = dstSize.width * numChannels;
```

```
        /* Memory allocation for the intermediate images */
        pBlur = ippsMalloc_8u(srcStep * srcSize.height);

        /* Memory allocation for the intermediate images */
        pBlurRot = ippsMalloc_8u(dstSize.width * numChannels * dstSize.height);

        maskSize.width  = 3;
        maskSize.height = 3;

        bFilterUpdate = bRotateUpdate = bRedraw = true;

        return true;
    }

    return false;
}

ipp_blur_rotate::ipp_blur_rotate()
    : angle(0.0), pSrc(NULL), pBlur(NULL), pBlurRot(NULL),
      bFilterUpdate(false), bRotateUpdate(false), bRedraw(false)
{
    dstSize.width  = srcSize.width  = 0;
    dstSize.height = srcSize.height = 0;
}

ipp_blur_rotate::~ipp_blur_rotate()
{
    ippsFree(pSrc);
    ippsFree(pBlur);
    ippsFree(pBlurRot);
}
```

‹ Previous Next ›

Building the Application

NOTE The `ipp_blur_rotate.cpp` sample code has been developed for Intel® IPP installed within Intel® Parallel Studio XE. You can use the provided code, except for the graphical rendering part, as a starting point for your application that can be built within other product suites like Intel® System Studio.

Setting Up the Build Environment

Before you invoke the compiler component of the suite that you installed with Intel® IPP, you must set certain environment variables that define the location of compiler-related components. The Intel® C++ Compiler includes the `compilervars` scripts that you can run to set environment variables:

- On Windows*, you can find the `compilervars.bat` batch file at
- On Linux OS* and macOS*, you can find the `compilervars.sh` shell script at

For more information about setting environment variables for different product suites, refer to https://software.intel.com/en-us/articles/setting-up-the-build-environment-for-using-intel-c-or-fortran-compilers.

Compile and Link Your Code

To compile and link the code examples in this tutorial, do the following:

- On Windows*:
  1. Unzip the `ipp_blur_rotate_sample.zip` archive downloaded from the Intel® Software Product Samples and Tutorials website.
  2. Run the Microsoft* Visual Studio* command prompt and change the directory to the `Makefile` location. By default, `Makefile` is located in the root of the `ipp_blur_rotate_sample.zip` archive.
  3. Run the `nmake -f Makefile_win` command.
- On Linux* OS and macOS*:
  1. Unzip the `ipp_blur_rotate_sample.tgz` archive downloaded from the Intel® Software Product Samples and Tutorials website.
  2. Run the bash shell and change the directory to the `Makefile` location. By default, `Makefile` is located in the root of the `ipp_blur_rotate_sample.tgz` archive.
  3. Run the `make -f Makefile_lin` command on Linux* OS and `make -f Makefile_mac` on macOS*.

To run the application, run the following command from the directory where `Makefile` resides:

- On Windows*: `ipp_blur_rotate.exe`
- On Linux*: `./ipp_blur_rotate`
- On macOS*: `./ipp_blur_rotate.app/Contents/MacOS/ipp_blur_rotate`

‹ Previous Next ›

Summary

Here are some important things to remember when developing an image processing application with Intel IPP:

- Before calling the processing function, compute the size of required buffers using the <*processing function*>`GetSize` or <*processing function*>`GetBufferSize` auxiliary functions, and, if required, initialize the function data using <*processing function*>`Init`.
- For the functions that perform neighborhood operations, set the appropriate border processing method in the *`borderType`* (or *`border`*) and *`borderValue`* parameters.
- When you link to the `ippi` domain library, you must also link to the libraries on which it depends: `ippcore`, `ipps`, and `ippvm`.

Previous