1> Example: GetRotateTransform function.

```
IppStatus RotateExample_8u_C3R(Ipp8u* pSrc, IppiSize srcSize, Ipp32s srcStep, Ipp8u*
pDst, IppiSize dstSize,
    Ipp32s dstStep, double angle, double xShift, double yShift)
{
    IppiWarpSpec* pSpec = 0;
    double coeffs[2][3];
    int specSize = 0, initSize = 0, bufSize = 0;
    Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {0, 0};
    IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderConst;
    IppiWarpDirection direction = ippWarpForward;
    Ipp64f pBorderValue[numChannels];

    status = ippiGetRotateTransform (angle, xShift, yShift, coeffs);
    if (status < ippStsNoErr) return status;

    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);

    if (status < ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    status = ippiWarpAffineLinearInit(srcSize, dstSize, ipp8u, coeffs, direction,
numChannels, borderType, pBorderValue, 0, pSpec);

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Resize processing */
    status = ippiWarpAffineLinear_8u_C3R(pSrc, srcStep, pDst, dstStep, dstOffset,
dstSize, pSpec, pBuffer);
```

```
    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}
```

2> Example: ippiWarpPerspective<Interp> functions.

```
IppStatus warpPerspectiveExample_8u_C3R(Ipp8u* pSrc, IppiSize srcSize, Ipp32s srcStep,
Ipp8u* pDst, IppiSize dstSize,
    Ipp32s dstStep, const double coeffs[3][3], IppiInterpolationType interpolation)
{
    IppiWarpSpec* pSpec = 0;
    Ipp8u* pInitBuf = 0;
    int specSize = 0, initSize = 0, bufSize = 0; Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {0, 0}; IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderConst;
    IppiWarpDirection direction = ippWarpForward;
    Ipp64f pBorderValue[numChannels];
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter coefficients for cubic
interpolation */
    IppiRect srcRoi = ippRectInfinite; /* Region of interest is not specified */

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

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

    if (status != ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    switch (interpolation)
    {
    case ippNearest:
        status = ippiWarpPerspectiveNearestInit(srcSize, srcRoi, dstSize, ipp8u, coeffs,
direction, numChannels, borderType, pBorderValue, 0, pSpec);
        break;
    case ippLinear:
        status = ippiWarpPerspectiveLinearInit(srcSize, srcRoi, dstSize, ipp8u, coeffs,
direction, numChannels, borderType, pBorderValue, 0, pSpec);
        break;
    case ippCubic:
        pInitBuf = ippsMalloc_8u(initSize);
        if (pInitBuf == NULL)
        {
            ippsFree(pSpec);
            return ippStsNoMemErr;
        }
        status = ippiWarpPerspectiveCubicInit(srcSize, srcRoi, dstSize, ipp8u, coeffs,
direction, numChannels, valB, valC, borderType, pBorderValue, 0, pSpec, pInitBuf);
        ippsFree(pInitBuf);
        break;
    default:
        return ippStsInterpolationErr;
    }

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Warp processing */
    switch (interpolation)
    {
    case ippNearest:
        status = ippiWarpPerspectiveNearest_8u_C3R(pSrc, srcStep, pDst, dstStep,
dstOffset, dstSize, pSpec, pBuffer);
        break;
    case ippLinear:
        status = ippiWarpPerspectiveLinear_8u_C3R(pSrc, srcStep, pDst, dstStep,
dstOffset, dstSize, pSpec, pBuffer);
        break;
    case ippCubic:
        status = ippiWarpPerspectiveCubic_8u_C3R(pSrc, srcStep, pDst, dstStep, dstOffset,
dstSize, pSpec, pBuffer);
        break;
    }

    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}
```

3> Example: ippiWarpQuad<Interp> Init functions.

```
IppStatus warpQuadExample_8u_C3R(Ipp8u* pSrc, IppiSize srcSize, Ipp32s srcStep, const
double srcQuad[4][2], Ipp8u* pDst, IppiSize dstSize,
    Ipp32s dstStep, const double dstQuad[4][2], IppiInterpolationType interpolation,
IppiWarpTransformType warpTransformType)
{
    IppiWarpSpec* pSpec = 0;
    int specSize = 0, initSize = 0, bufSize = 0; Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {0, 0}; IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderTransp; /* Transparent border : destination
image pixels mapped to the outer source image pixels are not changed. */
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter coefficients for cubic
interpolation */
    Ipp8u* pInitBuf = 0;
    /* Spec and init buffer sizes */
    status = ippiWarpQuadGetSize(srcSize, srcQuad, dstSize, dstQuad, warpTransformType,
ipp8u, interpolation, borderType, &specSize, &initSize);

    if (status < ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    switch (interpolation)
    {
    case ippNearest:
        status = ippiWarpQuadNearestInit(srcSize, srcQuad, dstSize, dstQuad,
warpTransformType, ipp8u, numChannels, borderType, 0, 0, pSpec);
        break;
    case ippLinear:
        status = ippiWarpQuadLinearInit(srcSize, srcQuad, dstSize, dstQuad,
warpTransformType, ipp8u, numChannels, borderType, 0, 0, pSpec);
        break;
    case ippCubic:
        pInitBuf = ippsMalloc_8u(initSize);
        if (pInitBuf == NULL)
        {
            ippsFree(pSpec);
            return ippStsNoMemErr;
        }
        status = ippiWarpQuadCubicInit(srcSize, srcQuad, dstSize, dstQuad,
warpTransformType, ipp8u, numChannels, valB, valC, borderType, 0, 0, pSpec, pInitBuf);
        ippsFree(pInitBuf);
        break;
    default:
        return ippStsInterpolationErr;
    }

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Warp processing */
    switch (warpTransformType)
    {
```

```
    case ippWarpAffine:
        switch (interpolation)
        {
        case ippNearest:
            status = ippiWarpAffineNearest_8u_C3R(pSrc, srcStep, pDst, dstStep,
dstOffset, dstSize, pSpec, pBuffer);
            break;
        case ippLinear:
            status = ippiWarpAffineLinear_8u_C3R(pSrc, srcStep, pDst, dstStep, dstOffset,
dstSize, pSpec, pBuffer);
            break;
        case ippCubic:
            status = ippiWarpAffineCubic_8u_C3R(pSrc, srcStep, pDst, dstStep, dstOffset,
dstSize, pSpec, pBuffer);
            break;
        }
        break;
    case ippWarpPerspective:
        switch (interpolation)
        {
        case ippNearest:
            status = ippiWarpPerspectiveNearest_8u_C3R(pSrc, srcStep, pDst, dstStep,
dstOffset, dstSize, pSpec, pBuffer);
            break;
        case ippLinear:
            status = ippiWarpPerspectiveLinear_8u_C3R(pSrc, srcStep, pDst, dstStep,
dstOffset, dstSize, pSpec, pBuffer);
            break;
        case ippCubic:
            status = ippiWarpPerspectiveCubic_8u_C3R(pSrc, srcStep, pDst, dstStep,
dstOffset, dstSize, pSpec, pBuffer);
            break;
        }
        break;
    }

    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}
```

4> Examples: How to replace the old ippiWarpAffine and WarpAffineBack functions with the new Intel APIs

```
IppStatus ippiWarpAffineSample_32f_C3R(const Ipp32f* pSrc, IppiSize srcSize, int srcStep,
IppiRect srcROI, Ipp32f* pDst, int dstStep, IppiRect dstROI, const double coeffs[2][3],
int interpolation)
{
    IppiWarpSpec* pSpec = 0;
    Ipp8u* pInitBuf = 0;
    int specSize = 0, initSize = 0, bufSize = 0; Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {0, 0};
    IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderTransp;
    IppiWarpDirection direction = ippWarpForward;
    IppiSize dstRoiSize = {dstROI.width, dstROI.height};
    double cf[2][3];
    IppiSize srcRoiSize;
    Ipp32f* pSrcRoi = (Ipp32f*)((Ipp8u*)pSrc + srcROI.y * srcStep) + srcROI.x *
numChannels;
    Ipp32f* pDstRoi = (Ipp32f*)((Ipp8u*)pDst + dstROI.y * dstStep) + dstROI.x *
numChannels;
```

```
    IppiInterpolationType interp;
    int borderSize = 0;
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter */

    if (srcROI.x < 0 || srcROI.y < 0 || srcROI.x >= srcSize.width || srcROI.y >=
srcSize.height)
        return ippStsRectErr;

    if (dstROI.x < 0 || dstROI.y < 0)
        return ippStsRectErr;

    /* Clip the source roi */
    if (srcROI.x + srcROI.width  > srcSize.width ) srcROI.width  = srcSize.width  -
srcROI.x;
    if (srcROI.y + srcROI.height > srcSize.height) srcROI.height = srcSize.height -
srcROI.y;

    srcRoiSize.width  = srcROI.width;
    srcRoiSize.height = srcROI.height;

    switch (interpolation)
    {
    case IPPI_INTER_NN:
        interp = ippNearest;
        break;
    case IPPI_INTER_LINEAR:
        interp = ippLinear;
        break;
    case IPPI_INTER_CUBIC:
        interp = ippCubic;
        borderSize = 1;
        break;
    default:
        return ippStsInterpolationErr;
    }

    /* compute new coefficients with taking into account ROI offsets*/
    cf[0][0] = coeffs[0][0]; cf[0][1] = coeffs[0][1]; cf[0][2] = coeffs[0][2] +
coeffs[0][0] * srcROI.x + coeffs[0][1] * srcROI.y - dstROI.x;
    cf[1][0] = coeffs[1][0]; cf[1][1] = coeffs[1][1]; cf[1][2] = coeffs[1][2] +
coeffs[1][0] * srcROI.x + coeffs[1][1] * srcROI.y - dstROI.y;

    /* define border type depending on the source ROI */
    if (srcROI.x >= borderSize) borderType = (IppiBorderType) (borderType |
ippBorderInMemLeft);
    if (srcROI.y >= borderSize) borderType = (IppiBorderType) (borderType |
ippBorderInMemTop);
    if (srcROI.x + srcROI.width <=  srcSize.width  - borderSize) borderType =
(IppiBorderType) (borderType | ippBorderInMemRight);
    if (srcROI.y + srcROI.height <= srcSize.height - borderSize) borderType =
(IppiBorderType) (borderType | ippBorderInMemBottom);

    /* Spec and init buffer sizes */
    status = ippiWarpAffineGetSize(srcRoiSize, dstRoiSize, ipp32f, cf, interp, direction,
borderType, &specSize, &initSize);

    if (status < ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }
    /* Memory allocation */
    pInitBuf = ippsMalloc_8u(initSize);
    if (pInitBuf == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpAffineNearestInit(srcRoiSize, dstRoiSize, ipp32f, cf, direction,
numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpAffineLinearInit(srcRoiSize, dstRoiSize, ipp32f, cf, direction,
numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpAffineCubicInit(srcRoiSize, dstRoiSize, ipp32f, cf, direction,
numChannels, valB, valC, borderType, 0, 0, pSpec,pInitBuf);
        break;
    }

    ippsFree(pInitBuf);

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstRoiSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Warp processing */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpAffineNearest_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpAffineLinear_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpAffineCubic_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    }
    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}


IppStatus ippiWarpAffineBackSample_32f_C3R(const Ipp32f* pSrc, IppiSize srcSize, int
srcStep, IppiRect srcROI, Ipp32f* pDst, int dstStep, IppiRect dstROI, const double
coeffs[2][3], int interpolation)
{
    IppiWarpSpec* pSpec = 0;
    Ipp8u* pInitBuf = 0;
    int specSize = 0, initSize = 0, bufSize = 0; Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {0, 0};
    IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderTransp;
    IppiWarpDirection direction = ippWarpBackward;
    IppiSize dstRoiSize = {dstROI.width, dstROI.height};
    double cf[3][3];
    IppiSize srcRoiSize;
    Ipp32f* pSrcRoi = (Ipp32f*)((Ipp8u*)pSrc + srcROI.y * srcStep) + srcROI.x *
numChannels;
    Ipp32f* pDstRoi = (Ipp32f*)((Ipp8u*)pDst + dstROI.y * srcStep) + dstROI.x *
numChannels;
    IppiInterpolationType interp;
    int borderSize = 0;
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter */

    if (srcROI.x < 0 || srcROI.y < 0 || srcROI.x >= srcSize.width || srcROI.y >=
srcSize.height)
        return ippStsRectErr;

    /* Clip the source roi */
    if (srcROI.x + srcROI.width  > srcSize.width ) srcROI.width  = srcSize.width  -
srcROI.x;
    if (srcROI.y + srcROI.height > srcSize.height) srcROI.height = srcSize.height -
srcROI.y;

    srcRoiSize.width  = srcROI.width;
    srcRoiSize.height = srcROI.height;

    switch (interpolation)
    {
    case IPPI_INTER_NN:
        interp = ippNearest;
        break;
    case IPPI_INTER_LINEAR:
        interp = ippLinear;
        break;
    case IPPI_INTER_CUBIC:
        interp = ippCubic;
        borderSize = 1;
        break;
    default:
        return ippStsInterpolationErr;
    }

    /* compute new coefficients with taking into account ROI offsets*/
    cf[0][0] = coeffs[0][0]; cf[0][1] = coeffs[0][1]; cf[0][2] = coeffs[0][2] +
coeffs[0][0] * dstROI.x + coeffs[0][1] * dstROI.y - srcROI.x;
    cf[1][0] = coeffs[1][0]; cf[1][1] = coeffs[1][1]; cf[1][2] = coeffs[1][2] +
coeffs[1][0] * dstROI.x + coeffs[1][1] * dstROI.y - srcROI.y;
```

```
    /* define border type depending on the source ROI */
    if (srcROI.x >= borderSize) borderType = (IppiBorderType) (borderType |
ippBorderInMemLeft);
    if (srcROI.y >= borderSize) borderType = (IppiBorderType) (borderType |
ippBorderInMemTop);
    if (srcROI.x + srcROI.width <=  srcSize.width  - borderSize) borderType =
(IppiBorderType) (borderType | ippBorderInMemRight);
    if (srcROI.y + srcROI.height <= srcSize.height - borderSize) borderType =
(IppiBorderType) (borderType | ippBorderInMemBottom);

    /* Spec and init buffer sizes */
    status = ippiWarpAffineGetSize(srcRoiSize, dstRoiSize, ipp32f, cf, interp, direction,
borderType, &specSize, &initSize);

    if (status < ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }

    /* Memory allocation */
    pInitBuf = ippsMalloc_8u(initSize);
    if (pInitBuf == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpAffineNearestInit(srcRoiSize, dstRoiSize, ipp32f, cf, direction,
numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpAffineLinearInit(srcRoiSize, dstRoiSize, ipp32f, cf, direction,
numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpAffineCubicInit(srcRoiSize, dstRoiSize, ipp32f, cf, direction,
numChannels, valB, valC, borderType, 0, 0, pSpec,pInitBuf);
        break;
    }

    ippsFree(pInitBuf);

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstRoiSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }
    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Warp processing */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpAffineNearest_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep, 
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpAffineLinear_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep, 
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpAffineCubic_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep, 
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    }

    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}
```

5> Examples: How to replace the old ippiWarpPerspective and WarpPerspectiveBack functions with the new Intel IPP APIs:

```
IppStatus ippiWarpPerspectiveSample_32f_C3R(const Ipp32f* pSrc, IppiSize srcSize, int 
srcStep, IppiRect srcROI, Ipp32f* pDst, int dstStep, IppiRect dstROI, const double 
coeffs[3][3], int interpolation)
{
    IppiWarpSpec* pSpec = 0;
    Ipp8u* pInitBuf = 0;
    int specSize = 0, initSize = 0, bufSize = 0; Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {dstROI.x, dstROI.y};
    IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderTransp;
    IppiWarpDirection direction = ippWarpForward;
    IppiSize dstSize  = {dstROI.x + dstROI.width, dstROI.y + dstROI.height};
    IppiSize dstRoiSize = {dstROI.width, dstROI.height};
    IppiSize srcRoiSize;
    Ipp32f* pSrcRoi = (Ipp32f*)((Ipp8u*)pSrc + srcROI.y * srcStep) + srcROI.x * 
numChannels;
    Ipp32f* pDstRoi = (Ipp32f*)((Ipp8u*)pDst + dstROI.y * dstStep) + dstROI.x * 
numChannels;
    IppiInterpolationType interp;
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter */

    if (srcROI.x < 0 || srcROI.y < 0 || srcROI.x >= srcSize.width || srcROI.y >= 
srcSize.height)
        return ippStsRectErr;

    if (dstROI.x < 0 || dstROI.y < 0)
```

```
        return ippStsRectErr;

    /* Clip the source roi */
    if (srcROI.x + srcROI.width  > srcSize.width ) srcROI.width  = srcSize.width  -
srcROI.x;
    if (srcROI.y + srcROI.height > srcSize.height) srcROI.height = srcSize.height -
srcROI.y;

    srcRoiSize.width  = srcROI.width;
    srcRoiSize.height = srcROI.height;

    switch (interpolation)
    {
    case IPPI_INTER_NN:
        interp = ippNearest;
        break;
    case IPPI_INTER_LINEAR:
        interp = ippLinear;
        break;
    case IPPI_INTER_CUBIC:
        interp = ippCubic;
        break;
    default:
        return ippStsInterpolationErr;
    }

    /* Spec and init buffer sizes */
    status = ippiWarpPerspectiveGetSize(srcRoiSize, srcROI, dstSize, ipp32f, coeffs,
interp, direction, borderType, &specSize, &initSize);

    if (status < ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }

    /* Memory allocation */
    pInitBuf = ippsMalloc_8u(initSize);
    if (pInitBuf == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpPerspectiveNearestInit(srcRoiSize, srcROI, dstSize, ipp32f,
coeffs, direction, numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpPerspectiveLinearInit(srcRoiSize, srcROI, dstSize, ipp32f,
coeffs, direction, numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpPerspectiveCubicInit(srcRoiSize, srcROI, dstSize, ipp32f,
coeffs, direction, numChannels, valB, valC, borderType, 0, 0, pSpec,pInitBuf);
        break;
    }
    ippsFree(pInitBuf);

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Warp processing */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpPerspectiveNearest_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpPerspectiveLinear_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpPerspectiveCubic_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    }

    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}

IppStatus ippiWarpPerspectiveBackSample_32f_C3R(const Ipp32f* pSrc, IppiSize srcSize, int
srcStep, IppiRect srcROI, Ipp32f* pDst, int dstStep, IppiRect dstROI, const double
coeffs[3][3], int interpolation)
{
    IppiWarpSpec* pSpec = 0;
    Ipp8u* pInitBuf = 0;
    int specSize = 0, initSize = 0, bufSize = 0; Ipp8u* pBuffer  = 0;
    const Ipp32u numChannels = 3;
    IppiPoint dstOffset = {dstROI.x, dstROI.y};
    IppStatus status = ippStsNoErr;
    IppiBorderType borderType = ippBorderTransp;
    IppiWarpDirection direction = ippWarpBackward;
    IppiSize dstSize  = {dstROI.x + dstROI.width, dstROI.y + dstROI.height};
    IppiSize dstRoiSize = {dstROI.width, dstROI.height};
    IppiSize srcRoiSize;
    Ipp32f* pSrcRoi = (Ipp32f*)((Ipp8u*)pSrc + srcROI.y * srcStep) + srcROI.x *
numChannels;
    Ipp32f* pDstRoi = (Ipp32f*)((Ipp8u*)pDst + dstROI.y * srcStep) + dstROI.x *
numChannels;
```

```
    IppiInterpolationType interp;
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter */

    if (srcROI.x < 0 || srcROI.y < 0 || srcROI.x >= srcSize.width || srcROI.y >=
srcSize.height)
        return ippStsRectErr;

    if (dstROI.x < 0 || dstROI.y < 0)
        return ippStsRectErr;

    /* Clip the source roi */
    if (srcROI.x + srcROI.width  > srcSize.width ) srcROI.width  = srcSize.width  -
srcROI.x;
    if (srcROI.y + srcROI.height > srcSize.height) srcROI.height = srcSize.height -
srcROI.y;

    srcRoiSize.width  = srcROI.width;
    srcRoiSize.height = srcROI.height;

    switch (interpolation)
    {
    case IPPI_INTER_NN:
        interp = ippNearest;
        break;
    case IPPI_INTER_LINEAR:
        interp = ippLinear;
        break;
    case IPPI_INTER_CUBIC:
        interp = ippCubic;
        break;
    default:
        return ippStsInterpolationErr;
    }

    /* Spec and init buffer sizes */
    status = ippiWarpPerspectiveGetSize(srcRoiSize, srcROI, dstSize, ipp32f, coeffs,
interp, direction, borderType, &specSize, &initSize);

    if (status < ippStsNoErr) return status;

    /* Memory allocation */
    pSpec = (IppiWarpSpec*)ippsMalloc_8u(specSize);

    if (pSpec == NULL)
    {
        return ippStsNoMemErr;
    }

    /* Memory allocation */
    pInitBuf = ippsMalloc_8u(initSize);
    if (pInitBuf == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Filter initialization */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpPerspectiveNearestInit(srcRoiSize, srcROI, dstSize, ipp32f,
coeffs, direction, numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpPerspectiveLinearInit(srcRoiSize, srcROI, dstSize, ipp32f,
coeffs, direction, numChannels, borderType, 0, 0, pSpec);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpPerspectiveCubicInit(srcRoiSize, srcROI, dstSize, ipp32f,
coeffs, direction, numChannels, valB, valC, borderType, 0, 0, pSpec,pInitBuf);
        break;
    }

    ippsFree(pInitBuf);

    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    /* work buffer size */
    status = ippiWarpGetBufferSize(pSpec, dstSize, &bufSize);
    if (status < ippStsNoErr)
    {
        ippsFree(pSpec);
        return status;
    }

    pBuffer = ippsMalloc_8u(bufSize);
    if (pBuffer == NULL)
    {
        ippsFree(pSpec);
        return ippStsNoMemErr;
    }

    /* Warp processing */
    switch (interpolation)
    {
    case IPPI_INTER_NN:
        status = ippiWarpPerspectiveNearest_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_LINEAR:
        status = ippiWarpPerspectiveLinear_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    case IPPI_INTER_CUBIC:
        status = ippiWarpPerspectiveCubic_32f_C3R(pSrcRoi, srcStep, pDstRoi, dstStep,
dstOffset, dstRoiSize, pSpec, pBuffer);
        break;
    }

    ippsFree(pSpec);
    ippsFree(pBuffer);

    return status;
}
```

6> Examples: how to replace the old ippiWarpAffineQuad and WarpPerspectiveQuad functions with the new Intel IPP new APIs:

Note: The code sample uses the function warpQuadExample_8u_C3R from the code example for ippiWarpQuad<Interp> Init functions. For Affine transform the parameter warpTransformType should be ippWarpAffine. For Perspective transform the parameter warpTransformType should be ippWarpPerspective.

```
IppStatus ippiWarpQuadSample_8u_C3(const Ipp8u* pSrc, IppiSize srcSize, int srcStep,
IppiRect srcROI, const double srcQuad[4][2], Ipp8u* pDst, int dstStep, IppiRect dstROI,
    const double dstQuad[4][2], int interpolation, IppiWarpTransformType warpTransform)
{
    int i;
    int numChannels = 3;
    IppiBorderType borderType = ippBorderTransp;
    IppiSize srcRoiSize;
    IppiSize dstRoiSize = {dstROI.width, dstROI.height};
    Ipp8u* pSrcRoi = (Ipp8u*)((Ipp8u*)pSrc + srcROI.y * srcStep) + srcROI.x *
numChannels;
    Ipp8u* pDstRoi = (Ipp8u*)((Ipp8u*)pDst + dstROI.y * srcStep) + dstROI.x *
numChannels;
    double srcQuadRoi[4][2];
    double dstQuadRoi[4][2];
    IppiInterpolationType interp;
    int borderSize = 0;
    Ipp64f valB = 0.0, valC = 0.5; /* Catmull-Rom filter */

    switch (interpolation)
    {
    case IPPI_INTER_NN:
        interp = ippNearest;
        break;
    case IPPI_INTER_LINEAR:
        interp = ippLinear;
        break;
    case IPPI_INTER_CUBIC:
        interp = ippCubic;
        break;
    default:
        return ippStsInterpolationErr;
    }

    if (srcROI.x < 0 || srcROI.y < 0 || srcROI.x >= srcSize.width || srcROI.y >=
srcSize.height)
        return ippStsRectErr;

    if (dstROI.x < 0 || dstROI.y < 0)
        return ippStsRectErr;

    /* Clip the source roi */
    if (srcROI.x + srcROI.width  > srcSize.width ) srcROI.width  = srcSize.width  -
srcROI.x;
    if (srcROI.y + srcROI.height > srcSize.height) srcROI.height = srcSize.height -
srcROI.y;

    srcRoiSize.width  = srcROI.width;
    srcRoiSize.height = srcROI.height;

    switch (interpolation)
    {
    case IPPI_INTER_NN:
        interp = ippNearest;
        break;
    case IPPI_INTER_LINEAR:
        interp = ippLinear;
        break;
    case IPPI_INTER_CUBIC:
        interp = ippCubic;
        borderSize = 1;
        break;
    default:
        return ippStsInterpolationErr;
    }
    /* define border type depending on the source ROI */
    if (srcROI.x >= borderSize) borderType = (IppiBorderType) (borderType | 
ippBorderInMemLeft);
    if (srcROI.y >= borderSize) borderType = (IppiBorderType) (borderType | 
ippBorderInMemTop);
    if (srcROI.x + srcROI.width <=  srcSize.width  - borderSize) borderType = 
(IppiBorderType) (borderType | ippBorderInMemRight);
    if (srcROI.y + srcROI.height <= srcSize.height - borderSize) borderType = 
(IppiBorderType) (borderType | ippBorderInMemBottom);

    /* Shift quadrangles depending on the processing regions of interest */
    for (i = 0; i < 4; ++i)
    {
        srcQuadRoi[i][0] = srcQuad[i][0] - srcROI.x; srcQuadRoi[i][1] = srcQuad[i][1] - 
srcROI.y;
        dstQuadRoi[i][0] = dstQuad[i][0] - dstROI.x; dstQuadRoi[i][1] = dstQuad[i][1] - 
dstROI.y;
    }

    return warpQuadExample_8u_C3R(pSrcRoi, srcRoiSize, srcStep, srcQuadRoi, pDstRoi, 
dstRoiSize, dstStep, dstQuadRoi, interp, warpTransform);
}
```