```cpp

#include <jni.h>
#include <string>
#include "native-lib.h"
#define _USE_MATH_DEFINES // for C++
#include <math.h>
#include <android/log.h>


#define  LOG_TAG    "DEBUG"
#define  LOGD(...)  __android_log_print(ANDROID_LOG_DEBUG,LOG_TAG, __VA_ARGS__)
extern "C"
//jstring
/**
 * Created by tangmiao on 11/27/2016.
 * This transform was basically trans-form the color value of every pixel into another
 * color value for each channel of RGB image.  The transform was based on the specified
 * piece-wise function.  Since there were three channels, there were also three piecewise functions.
 * Every piecewise function was given by eight numbers, which represented 4 points(x value and value)
 * onthe linear piece wise function plot.  Since there were three channels, we would be given
 * an array including 24 numbers in total.  These number would determine how the original figure will be transformed.
 */
jbyteArray

Java_edu_asu_msrs_artcelerationlibrary_ArtTransformService_ColorFilterFromJNI(
        JNIEnv *env,
        jobject /* this */,
        jbyteArray array,
        jintArray intArgs) {

    jbyte* pixels = env->GetByteArrayElements(array, 0); //pass byte array to pointer
    int length = env-> GetArrayLength(array);
    int* piecewiseArray = env->GetIntArrayElements(intArgs,NULL);



        for (int i = 0; i < length/4; i++) {
            pixels[4*i+1] = ArrayOperater(pixels[4*i+1],0, piecewiseArray);
            pixels[4*i+2] = ArrayOperater(pixels[4*i+2],8, piecewiseArray);
            pixels[4*i+3] = ArrayOperater(pixels[4*i+3],16, piecewiseArray);
        }

        env->SetByteArrayRegion (array, 0, length, pixels); // c++ to java, return java
        env->ReleaseByteArrayElements(array, pixels, 0);
```

```
            return array;


}
//Function: transform the pixel values according input args
//Input: Original image pixels, different channel indexes, and piecewiseArray
//Output: all the  image pixels after processed
jbyte ArrayOperater(jbyte pixel1,int colorshift, int* piecewiseArray) {

        int pixel = pixel1 & 0xFF;

        if (pixel< 0) {
            pixel = 0;
        }else if (pixel >= 0 || pixel < piecewiseArray[0+colorshift]) {
            pixel = (pixel)*(piecewiseArray[1+colorshift])/(piecewiseArray[0+colorshift]);
        }else if (pixel >= piecewiseArray[0+colorshift] || pixel < piecewiseArray[2+colorshift]) {
            pixel= piecewiseArray[0+colorshift]+(pixel-piecewiseArray[0+colorshift])*((piecewiseArray[3+colorshift]-piecewiseArray[1+colorshift])/(piecewiseArray[2+colorshift]-piecewiseArray[0+colorshift]));
        }else if (pixel >= piecewiseArray[2+colorshift] || pixel < piecewiseArray[4+colorshift]) {
            pixel = (piecewiseArray[2+colorshift]+(pixel-piecewiseArray[2+colorshift])*((piecewiseArray[5+colorshift]-piecewiseArray[3+colorshift])/(piecewiseArray[4+colorshift]-piecewiseArray[2+colorshift])));
        }else if (pixel >= piecewiseArray[4+colorshift]|| pixel < piecewiseArray[6+colorshift]) {
            pixel = (piecewiseArray[4+colorshift]+(pixel-piecewiseArray[4+colorshift])*((piecewiseArray[7+colorshift]-piecewiseArray[5+colorshift])/(piecewiseArray[6+colorshift]-piecewiseArray[4+colorshift])));
        }else if (pixel >= piecewiseArray[6+colorshift]|| pixel < 255){
            pixel = (piecewiseArray[6+colorshift]+ (pixel - piecewiseArray[6+colorshift])* (255 - piecewiseArray[7+colorshift])/(255 - piecewiseArray[6+colorshift]));
        } else {
            pixel = 255;
        }


        return  (jbyte)pixel;
    }

/*The Gaussian Blur transforms the input pixel values using Gaussian weighted kernelvector.
The vector is first applied to the x direction and then apply to the y direction.The radius
determines how many terms are to multiply by the Gaussian weight vector.*/
extern "C"
jbyteArray Java_edu_asu_msrs_artcelerationlibrary_ArtTransformService_GaussianBlurFromJNI(
            JNIEnv *env,
        jobject /* this */,
        jbyteArray array,
        int w,
        int h,
        jintArray args1,
        jfloatArray args2) {

    jbyte* b = env->GetByteArrayElements(array, 0); //pass byte array to pointer
    int length = env-> GetArrayLength(array);
    int *intArray = env->GetIntArrayElements(args1, NULL);
    float *floatArray = env->GetFloatArrayElements(args2, NULL);


    float** red = create2DArray(w,h);
    float** green = create2DArray(w,h);
    float** blue = create2DArray(w,h);



    convertToInt(b,w,h,red,green,blue);


    processOne(red,w,h,floatArray[0],intArray[0]);
    processTwo(red,w,h,floatArray[0],intArray[0]);

    processOne(green,w,h,floatArray[0],intArray[0]);
    processTwo(green,w,h,floatArray[0],intArray[0]);

    processOne(blue,w,h,floatArray[0],intArray[0]);
    processTwo(blue,w,h,floatArray[0],intArray[0]);




    for (int pixel = 0, row = 0, col = 0; pixel < h*w*4; pixel += 4) {

        b[pixel + 0] = (jbyte)(red[row][col]);
        b[pixel + 1] = (jbyte)((green[row][col]));
        b[pixel + 2] = (jbyte)((blue[row][col]));
        b[pixel + 3] = (jbyte)255;
        col++;
        if (col == w) {
            col = 0;
            row++;
        }
    }

    cleanupArray(red, h);
    cleanupArray(green, h);
```

```cpp
        cleanupArray(blue, h);


         env->SetByteArrayRegion (array, 0, length, b); // c++ to java, return java
         env->ReleaseByteArrayElements(array, b, 0);
         return array;
    }


//Function: extracts color values from byte array and stores them into 2d array
//Input: image byte array, img width, img height, color 2d arrays
//Output: null

void convertToInt(jbyte* b, int w, int h, float** red, float** green, float** blue) {
        for (int pixel = 0, row = 0, col = 0; pixel < h*w*4; pixel += 4) {
            red[row][col] = b[pixel + 0] & 0xff;
            green[row][col] = b[pixel + 1] & 0xff;
            blue[row][col] = b[pixel + 2] & 0xff;
            col++;
            if (col == w) {
                col = 0;
                row++;
            }
        }
    }


//Function: creates 2d color array
//Input: array size
//Output: 2d array

float** create2DArray(int w, int h){

    float** color = new float*[h];
    for(int i = 0; i < w; ++i)
        color[i] = new float[w];
    return color;
}


//Function: release 2d color array
//Input: array size
//Output: null

void cleanupArray(float** array, int h){

for(int i = 0; i < h; ++i) {
    delete[]  array[i];
}
```

```
delete [] array;

}

//Function: Gaussian transform step one
//Input: 2d color array, img width, img height, sigma, radius
//Output: null

void processOne(float** color, int w, int h, float sigma, int r) {
        for (int row = 0; row < h; row++ ){
            for (int col = 0; col < w; col++){
                color[row][col] = color[row][col]*gKernel(0,sigma);

                for (int k = 1; k<=r; k++){
                    if((row < k) || ( row + k >= h)){
                        color[row][col] += 0;
                    } else{
                        color[row][col] += color[row+k][col]*(gKernel(k,sigma));
                        color[row][col] += color[row-k][col]*(gKernel(-k,sigma));

                    }

                }
            }
        }

    }



//Function: Gaussian transform step two
//Input: 2d color array, img width, img height, sigma, radius
//Output: null

     void processTwo(float** color, int w, int h, float sigma,int r) {
        for (int row = 0; row < h; row++ ){
            for (int col = 0; col < w; col++){
               color[row][col] = color[row][col]*(gKernel(0,sigma));

                for (int k = 1; k<=r; k++){
                    if((col < k) || ( col + k >= w)){
                        color[row][col] += 0;
                    } else{
                        color[row][col] += color[row][col+k]*(gKernel(k,sigma));
                        color[row][col] += color[row][col-k]*(gKernel(-k,sigma));

                    }

                }
```

```

      }
    }

  }

//Function: find Gaussian kernel with given radius and sigma value
//Input: radius, sigma
//Output: Gaussian kernel

 float gKernel(int k, float t){

    float g;
    g = (float)exp(-(k*k)/(2*(t*t)));
    g = g*(float)1/(float)sqrt(2*M_PI*t*t);

    return g;
  }

```