

COMPARISON OF ROBUST COLOR RECOGNITION ALGORITHMS

Samuel Kosolapov

ORT Braude Academic College of Engineering, Karmiel, Israel

Abstract

A number of color recognition algorithms are known - from using classic HSL space to modern AI algorithms. However, when analyzing real-life industrial, agricultural or medical images (for example images of fruits, flowers – to be sorted by color, human skin – to be analyzed for the presence of suspicious colors) - camera noise makes color recognition non-reliable, especially in dark regions. In this paper reliability of a number of well-known color recognition algorithms was compared. In order to lessen the amount of false recognitions, some modifications to algorithms in test were added. Provided comparisons of the algorithms in test were executed by using real-life images.

Keywords: image processing, color recognition, Hue, scalar product, normalized correlation

1. INTRODUCTION

The wide proliferation of digital cameras of different sizes and resolutions lead to a creation of a big number of applications using different image processing algorithms. Some applications use color recognition algorithms. It must be noted, that, generally, image processing algorithms can be roughly divided into algorithms goal of which is to present specific image in a way that is optimal for a human observer, whereas the goal of other algorithms is to come to some “decision” - based on properties of the image in test. Later algorithms are often referred to as “Algorithms for Machine Vision”. For example, in the photo of “red tea roses” (see Fig.1), a human observer can easily point to “red roses” and easily distinguish them from green leaves. Considering, that modern digital cameras in most cases create images of high color fidelity, one can conclude, that in this case no additional image processing is needed. Even if it is known, that specific camera, used to get this photo, creates some color distortion, those distortions will not prevent the human observer from easily distinguishing red roses from a green leaves and other colored objects that can be seen on this photo.



Fig. 1. Photo of “red tea roses”

This photo was taken by a forward camera of a smartphone “Samsung Galaxy Note 10+”. In order to provide an adequate presentation of the images, by taking into account publication requirements, high-resolution JPG photo was rescaled to the CGA size 240x320 true color BMP image.

However, if, for example, the goal is to point some robotic arm to collect red roses only, the software of this robotic arm must contain some image processing algorithms. In this specific case, those algorithms may recognize (and mark) “red roses” as “interesting objects” having specific “red” color, whereas leaves and other objects having other colors may be marked as “non-interesting regions”. In that case, the resulting image must be a binary image having two colors only, for example – “black pixels” marking pixels “recognized as red” and “white pixels” marking pixels “non recognized as red”

A number of classical color recognition algorithms are known. Some algorithms are based on using HSL/HSV color spaces (Russ & Neal 2017, Jain et al. 1995, Foley et al. 1997). Other algorithms use additional techniques, like scalar product in the RGB space, normalized correlation in RGB space, color jumps (Russ & Neal 2017, Kosolapov 2007, Kosolapov et al. 2013).

The problem with most of the classical approaches is the low reliability of color recognition. For example, in the dark regions of the image - when the noise of the digital cameras become significant, color recognition by using Hue map and by using scalar product become problematic and often leads to false recognitions and to missed recognitions.

To cope with this problem, classical algorithms were modified in an attempt to make them more robust. In this paper details of the implementation and properties of three modified algorithms (Hue-map, Scalar Product, Normalized Correlation) will be discussed.

2. MODIFIED ALGORITHMS USED

2.1. Input and output variables and arrays

In order to provide analysis and comparison of the selected algorithms, a special test application was created by using Microsoft Visual Studio 2019. Plain classic C language was selected because this language enables (at the end) to create fast and memory-effective image processing algorithms, that can be easily ported to a wide range of microprocessors and microcontrollers – from simplest to dedicated DSP systems.

The input of all algorithms is a true-color image declared as a three-dimensional array of bytes (“unsigned char”) having fixed sizes: `NUMBER_OF_ROWS = 240`, `NUMBER_OF_COLUMNS = 320` and `NUMBER_OF_COLORS = 3` (known as “R”, “G”, “B”).

Intermediate and final images were declared as a two-dimensional arrays of bytes having fixed sizes: `NUMBER_OF_ROWS = 240`, `NUMBER_OF_COLUMNS = 320`.

In order to lessen the number of parameters in the functions, all arrays were declared as global arrays. This practice is not recommended by purist C educators, however, it is instrumental when stack size is limited. Even Microsoft Visual Studio 2019 default stack size enables to put inside the stack only one or two images of VGA size. Dynamic allocation was not used, because it is not always available in small systems.

Additionally, in order to lessen the number of parameters in the functions, a number of structures were declared. In Fig.2 presented global arrays and structures that will be used later. In the arrays and structures “H” stands for “Hue”, “S” for “Saturation”, “L” for “Lightness”, “V” for “Value”. Considering, that the “Saturation” calculation is different for HSL and HSV models, structure `sHSLV` contains two fields for “Saturation”: “SL” – for saturation in the HSL model, and “SV” – for “Saturation” in the HSV model. Original pixels are stored as bytes, so that `sRGB` structure is adequate for operations with BMP files – load image from BMP file and store image to BMP file. However, this structure (containing “unsigned char” fields) is problematic for many mathematical calculations, hence structure `sdRGB`, fields of which are “double values”, was used in the calculations.

It must be noted, that in this paper values of “Saturation” will not be used, and values of “Lightness” and “Value” will be used only as intermediate variables, needed to calculate “Hue”.

Most of the modifications of classic algorithms deal with low values of “Value” and with low magnitudes of “RGB” vectors. The idea of those modifications is that color of “dark” pixels cannot be

calculated in a reliable way, hence, those pixels must be marked as “dark” and excluded from the color analysis. Of course, in this approach, some pixels are excluded from the color recognition, but this effectively prevents “false recognitions” in the dark regions. An additional idea is to exclude pixels, the color of which is close to gray. To do this additional modifications (to be explained later) were provided.

```
22 unsigned char H[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
23 unsigned char S[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
24 unsigned char L[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
25 unsigned char V[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
26
27 unsigned char selected[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
28 unsigned char scalarProduct[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
29 unsigned char deGrayedScalarProduct[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
30
31 // Declare True Color Image
32 unsigned char TrueColorImage[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS][NUMBER_OF_COLORS];
33
34 struct sHSLV
35 {
36     double H; // H is the same for HSL and HSV
37     double SL; // Saturation for HSL
38     double SV; // Saturation for HSV
39     double L;
40     double V;
41 };
42
43 struct sRGB
44 {
45     unsigned char r;
46     unsigned char g;
47     unsigned char b;
48 };
49
50 struct sdRGB
51 {
52     double r;
53     double g;
54     double b;
55 };
```

Fig. 2. Global Arrays and Structures

2.2. Important Functions

In Fig.3 presented a modified code of the calculation of the “Hue” of the specific pixel. It is well known, that in some situations, the value of “Hue” cannot be calculated. The correct approach would be to mark the value of “Hue” for specific pixels as “NaN” (“not a number”). However, support of NaN is not always available in small systems, hence, it was decided, that problematic pixels will have a “Hue” value equal to “0”. This marking must be taken into account in the treatment of “Hue” for the goals of color recognition. Specifically, modifications are in lines 115-118 (“hue” cannot be calculated for the gray pixels) and in lines 140-145 (pixel is “dark”). Some problem with marking “bad pixels” with value ‘0’ is in that “0” is a legitimate value of “Hue” (pure “RED” color). The code in lines 135-138 changes the value of “0” to “1”, which can be considered as “accurate enough for the goals of this analysis”.

It must be noted, that non-relevant for the goal of this paper “calculations of saturation” are hidden.

An important parameter of the function “ConvertRGBtoHSLV” is Vlimit. This parameter is an empirical one, and, as such, must be empirically set for the specific image in test.

This function is used to build an “H-map” – gray image, and gray levels of its pixels specify “color”. (See Fig. 4). The classical interpretation of “Hue” is “angle” in the HSL or HSV coordinates. In most algorithms, this angle is the range [0..360]. However, in order to present values of “Hue” in a human-friendly form, “Hue” values in line 173 are scaled to “unsigned char” ranges [0..255].

```
70 struct sHSLV ConvertRGBtoHSLV(struct sRGB pixel, double Vlimit)
71 {
72     // normalized values in the range 0..1
73     double r = (pixel.r / 255.0);
74     double g = (pixel.g / 255.0);
75     double b = (pixel.b / 255.0);
76
77     sHSLV hslv;
78     { ... }
85     double min = Min(Min(r, g), b);
86     double max = Max(Max(r, g), b);
87     double delta = max - min;
88
89     // Calculations of V (Value) and L (Lightness)
90     hslv.V = max;
91     hslv.L = (max + min) / 2;
92     { ... }
103    { ... }
114    // Calculate Hue
115    if (delta == 0) // Pixel is Gray
116    {
117        hslv.H = 0; // or NaN - in this case color cannot be defined
118        return hslv;
119    }
120    if (hslv.V == r) // RED is the highest color
121    {
122        hslv.H = 60 * (0 + (g - b) / delta);
123    }
124    { ... }
135    // Zero is reserved for "bad" situations
136    if (hslv.H == 0)
137    {
138        hslv.H = 1;
139    }
140    // In this case calculation of the H is not reliable
141    if (hslv.V < Vlimit)
142    {
143        hslv.H = 0;
144    }
145    return hslv;
146 }
```

Fig. 3. Calculation of Hue by RGB and Vlimit

Non-relevant for calculations of Hue lines are hidden in {...}

It must be taken into account that scaling down may create additional “0” values of “Hue”. This situation is resolved in the code written in lines 170-177.

Function CreateHMap (presented in Fig. 4) creates H-Map that can be easily analyzed by humans. However, for the “recognition” adequate for the “Machine Vision”, an additional step is needed. Function “SelectColorByH” (presented at Fig. 5) creates a “binary” image, in which pixels, the color of which are close to the color of the “selected” value are marked as “black”, whereas pixels, the color of which is not close to this specific color, are marked as “white”. For the goal of the analysis of the modifications provided, “bad” pixels are marked as having a “Light Gray” color (200) – see code in lines 194 -199. Code in line 197 must be excluded from the final “Machine Vision” code.

```
151 void CalculateHmap( double Vlimit)
152 {
153     sRGB pixel;
154     sHSLV hslv;
155
156     int row, col;
157     for (row = 0; row < NUMBER_OF_ROWS; row++)
158     {
159         for (col = 0; col < NUMBER_OF_COLUMNS; col++)
160         {
161             pixel.r = TrueColorImage[row][col][R];
162             pixel.g = TrueColorImage[row][col][G];
163             pixel.b = TrueColorImage[row][col][B];
164
165             hslv = ConvertRGBtoHSLV(pixel, Vlimit);
166             if (hslv.H == 0)
167             {
168                 H[row][col] = 0;
169             }
170             else
171             {
172                 H[row][col] = (unsigned char)(int)(255.0 * hslv.H / (360.0));
173                 if (H[row][col] == 0) // zero because of 250/360 scaling
174                 {
175                     H[row][col] = 1; // because 0 is reserved for bad values
176                 }
177             }
178         }
179     }
180 }
```

Fig. 4. Calculation of Hue Map

Value “0” is reserved for “bad” values of “Hue”

An important empirical parameter of the function SelectColorByH (see Fig. 5) is “tolerance” – how the “Hue” value of the specific pixel must be close to the selected “Hue” value. This parameter is an empirical one, and, as such must, be set empirically for the specific image in test.

Using of “tolerance” parameter creates a calculation problem. The origin of this problem is in the obvious fact, that color of a pixel having a “Hue” value equals to “1” is very close to the color of a pixel having a “Hue” value equals to “359”. Actually, the problem is that the difference between two angles can be calculated in two ways. Code in lines 201-215 eliminates this ambiguity by calculating the “minimal” difference between two angles. At the end, resulting pixel values are “0” and “255”.

Code in lines 217-224 creates a resulting binary image which can be easily treated by systems having “Machine Vision” software. In the above example (see Fig. 1), the robotic arm must be pointed only to the reliably recognized red pixels marked now as “black” pixels. Of course, the real-life robotic system

must additionally process this “binary” image – for example, calculate “size” and “center” of the “black spot”, and only then “grab” detected “red rose”. Implementation of this algorithm is obvious to anybody skilled in the art but is out of the scope of this paper.

An alternative to using “Hue” values is to treat “RGB” pixel as a 3D vector in the “RGB” space. Then, by evaluating scalar product between “selected pixel having specific “RGB” values and “pixels in test”, it is possible to state if the color of both pixels are similar (if the value of the scalar product is close to “1”), or not (in the opposite case).

```
182 void SelectColorByH( unsigned char selectedH, unsigned char tolerance)
183 {
184     int val1, val2, val3, val4;
185     int diff, diff1, diff2, diff3, diff4;
186
187     int row, col;
188     for (row = 0; row < NUMBER_OF_ROWS; row++)
189     {
190         for (col = 0; col < NUMBER_OF_COLUMNS; col++)
191         {
192             val1 = H[row][col];
193
194             // Bad values are colored as "light gray"
195             if (val1 == 0)
196             {
197                 selected[row][col] = 200;
198                 continue;
199             }
200
201             val2 = 255 - val1;
202
203             val3 = selectedH;
204             val4 = 255 - val3;
205
206             diff1 = abs(val1 - val3);
207             diff2 = abs(val1 - val4);
208
209             diff3 = abs(val2 - val3);
210             diff4 = abs(val2 - val4);
211
212             diff = diff1;
213             if (diff2 < diff) diff = diff2;
214             if (diff3 < diff) diff = diff3;
215             if (diff4 < diff) diff = diff4;
216
217             if (diff <= tolerance)
218             {
219                 selected[row][col] = 0;
220             }
221             else
222             {
223                 selected[row][col] = 255;
224             }
225         }
226     }
```

Fig. 5. Recognition of the color by Hue ranges

“Recognized pixels are marked as “black”. Non-recognized pixels are marked has “white” “Bad Values” are marked by “Light Gray” color 200

Fig. 6 presented the code of the function “CalculateScalarProductValue”. Arguments of this function are two RGB pixels (pixel1 and pixel2) and two empiric parameters: “scalarProductLimit” and “magnitudeLimit”. Code in lines 243-250 excludes pixels having low magnitude, whereas code in lines 256-263 discards pixels having low value of the scalar product.

```
230 unsigned char CalculateScalarProductValue(sdRGB pixel1, sdRGB pixel2,
231                                         double scalarProductLimit,
232                                         double magnitudeLimit )
233 {
234     double value;
235
236     double magnitude1 = sqrt( pixel1.r * pixel1.r
237                               + pixel1.g * pixel1.g
238                               + pixel1.b * pixel1.b);
239     double magnitude2 = sqrt( pixel2.r * pixel2.r
240                               + pixel2.g * pixel2.g
241                               + pixel2.b * pixel2.b);
242
243     if (magnitude1 < magnitudeLimit)
244     {
245         return 255;
246     }
247     if (magnitude2 < magnitudeLimit)
248     {
249         return 255;
250     }
251
252     value = pixel1.r * pixel2.r
253            + pixel1.g * pixel2.g
254            + pixel1.b * pixel2.b;
255     value = value / (magnitude1 * magnitude2);
256     if (value >= scalarProductLimit)
257     {
258         return 0;
259     }
260     else
261     {
262         return 255;
263     }
264 }
```

Fig. 6. Modified Calculation of Scalar Product

Fig.7 presented the code of the function “CalculateScalarProduct”. This function creates a binary map: pixels, the color of which is closed to the color of exemplary “otherPixel”, are marked as “black”, whereas other pixels are marked as white. Actually, this function calculate the scalar product for all pixels of the image in test by using the function “CalculateScalarProductValue” described above.

On Fig. 8 presented code that implement the idea of “Normalized Correlation”. Important difference here is that at the first step mean value of “R”, “G”, “B” values is calculated, and, then, this mean value is subtracted from the values of “R”, “G”, “B”. Those calculations executed in the function “DeGrayPixel”. After de-graing procedure, previously described function “CalculateScalarProductValue” is applied for all pixels of the image in test.

```
266 void CalculateScalarProduct(sdRGB otherPixel,  
267                             double scalarProductLimit,  
268                             double magnitudeLimit)  
269 {  
270     sdRGB thisPixel;  
271  
272     int row, col;  
273     for (row = 0; row < NUMBER_OF_ROWS; row++)  
274     {  
275         for (col = 0; col < NUMBER_OF_COLUMNS; col++)  
276         {  
277             thisPixel.r = TrueColorImage[row][col][R];  
278             thisPixel.g = TrueColorImage[row][col][G];  
279             thisPixel.b = TrueColorImage[row][col][B];  
280  
281             scalarProduct[row][col] = CalculateScalarProductValue(  
282                 thisPixel, otherPixel,  
283                 scalarProductLimit, magnitudeLimit);  
284         }  
285     }  
286 }
```

Fig. 7. Recognition by Scalar Product

3. RESULTS

The following exemplary results were obtained by processing the photo of red roses presented in Fig. 1. The goal of the processing was to recognize pixels having selected (red) color and mark them as “black”, whereas all other pixels were marked as “white”, so that the result is binary: “rose” or “not rose”.

The left image on Fig. 9 presents Hue Map as a human-friendly gray level image. Gray level on this image encodes “color”, so that objects having the same color, must have the same gray level. Visually comparing original color photo (presented in Fig. 1) with Hue Map, human observer easily understand, that white pixels correspond to “red” color, so that if the goal is to “grab rose”, regions having “white” gray level must be “considered as an object to be grabbed”.

Parameter Vlimit of the function “calculateHMap” (presented in Fig. 4) was set to 10.0/255.0.

The right image on Fig.9 represents results obtained by using the function “SelectColorByH” as presented in Fig. 5. The following parameters were found as adequate: “Hue value to be found” was set to 253 – nearly pure RED color, tolerance was set to 10, which means that pixels having HUE values in the range [242..255] and values in the range [1..8] (after excluding problematic pixels having “0” Hue value) were considered as “recognized by color of rose”.

It can be seen, that the right image in Fig. 9 contains a significant number of “false” recognitions. Most of those “false” pixels can be eliminated by using a median filter. Additional option is to additionally modify algorithm presented in Fig. 3, in order to provide better filtering out of gray pixels.

Left image on Fig. 10 represents results of recognition by using Scalar Product (code of which was presented in Fig. 6 and Fig. 7). Recognition parameters used were: scalarProductLimit=0.95 and magnitudeLimit=20. It can be seen, that no “false” recognitions can be seen, but some clearly “red” pixels were not properly recognized as “red”. One reason to be checked may be too strict limits used.

Right image on Fig.10 represents results of recognition by using function DeGrayedScalarProduct (described on Fig. 8). It can be seen that most of the “RED” pixels are properly recognized as “RED”,

however some “false” recognitions are present, albeit in a smaller amount comparing with algorithms using the “Hue Map”.

```
288 sdRGB DeGrayPixel(sdRGB pixel)
289 {
290     sdRGB deGrayedPixel;
291
292     double mean = (pixel.r + pixel.g + pixel.b) / 3;
293     deGrayedPixel.r = pixel.r - mean;
294     deGrayedPixel.g = pixel.g - mean;
295     deGrayedPixel.b = pixel.b - mean;
296
297     return deGrayedPixel;
298 }
299
300
301 void CalculateDeGrayedScalarProduct(sdRGB otherPixel,
302                                     double scalarProductLimit,
303                                     double magnitudeLimit)
304 {
305     sdRGB thisDPixel;
306     sdRGB otherDPixel = DeGrayPixel( otherPixel );
307
308     int row, col;
309     for (row = 0; row < NUMBER_OF_ROWS; row++)
310     {
311         for (col = 0; col < NUMBER_OF_COLUMNS; col++)
312         {
313             thisDPixel.r = TrueColorImage[row][col][R];
314             thisDPixel.g = TrueColorImage[row][col][G];
315             thisDPixel.b = TrueColorImage[row][col][B];
316
317             thisDPixel = DeGrayPixel(thisDPixel);
318
319             deGrayedScalarProduct[row][col] =
320                 CalculateScalarProductValue(
321                     thisDPixel, otherDPixel,
322                     scalarProductLimit, magnitudeLimit);
323         }
324     }
325 }
```

Fig. 8. Recognition by modified scalar product by using de-Grayed pixels

4. DISCUSSION AND CONCLUSIONS

Results described in this paper can be considered only as preliminary, however, it is clear, that modifications of the classic color recognition algorithms described in this paper make them more

reliable. It is clear, that to make additional conclusions about the quality of the modified algorithms, more images must be analyzed. Additionally, more tests with different values of the parameters of the algorithms used in this paper must be provided.

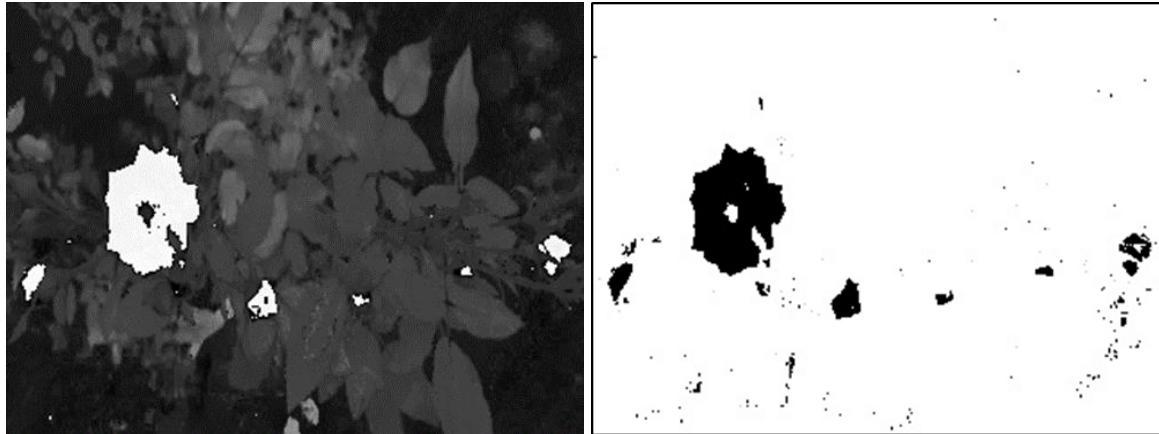


Fig. 9. “Hue” Map (left) and Recognition Map by using “Hue” value (right) : pixels having recognized “red” color are drawn as black

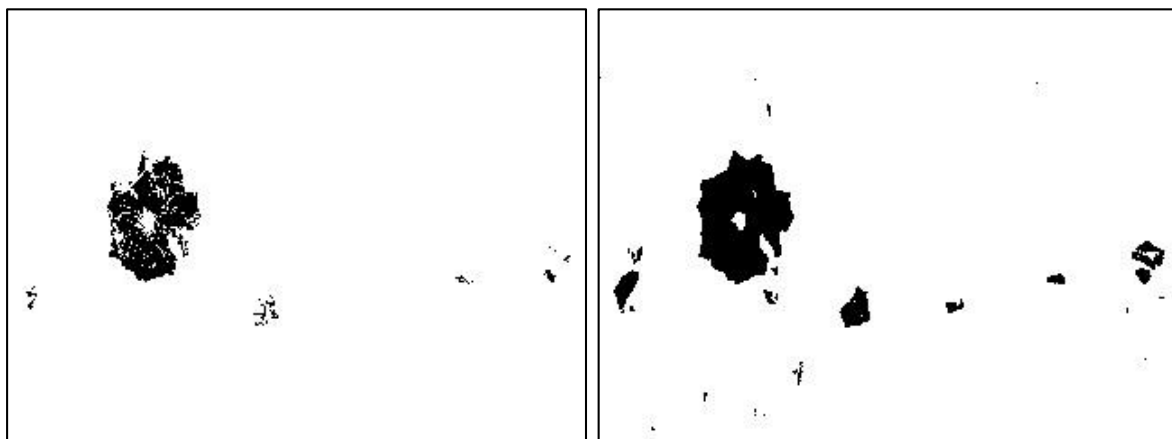


Fig. 10. Recognition by using Scalar Product (left image) and by using DeGrayed Scalar Product (right image). Pixels having recognized “red” color are drawn as black

REFERENCES

1. Russ, JC, Neal FB, 2017, *The Image Processing Handbook*, CRC Press.
2. Jain, R, Kasturi, R, Schunck, DG 1995, *Machine Vision*, MIT Press and McGraw Hill, Inc.
3. Foley JD, van Dam, A, Feiner SK, & Hughes JF 1997, *Computer Graphics Principles and Practice. Second Edition in C*, ADDISON-WESLEY.
4. Kosolapov, S, 2007, Color edge based system and method for determination of 3d surface topology. US Patent 20070085849A1.
5. Kosolapov, S, Gershikov, E, Sabag, N, 2013, *Feasibility of a Camera-Based Instant Feedback System*, CONTENT 2013, The Fifth International Conference on Creative Content Technologies, ThinkMind, Valencis, Spain, pp 23-29.