Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium, Volume 22, No. 1, ISSN 1726-9679
ISBN 978-3-901509-83-4, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria, EU, 2011
Make Harmony between Technology and Nature, and Your Mind will Fly Free as a Bird
Annals & Proceedings of DAAAM International 2011

OPTIMIZED RGB TO HSV COLOR CONVERSION USING SSE TECHNOLOGY

KOBALICEK, P[etr] & BLIZNAK, M[ichal]

Abstract: *Current approaches to conversion of pixels from RGB to HSV color space in C++ language rely in most cases on C++ code using built-in language constructs. Although the C++ compiler is able to translate the C++ code into SSE enhanced assembly, it doess not use instructions designed for parallel data processing. In this paper a completely new algorithm designed for conversion of pixels from RGB to HSV color space using optimized and branchless SSE code is introduced. The presented algorithm is more than 50% faster compared to the implementation written in pure C++.*
Key words: *RGB, HSV, optimization, parallel, processing, SSE*

1. INTRODUCTION

HSV (hue, saturation, and value) is hexcone coordinate representation of points in RGB (red, green, and blue) color model as described in Color Spaces for Computer Graphics (Joblove&Greenberg, 1978). HSV color space is often used in digital image processing, analysis, and color picking tools, where it's more intuitively satisfying or more convenient to an artist than the RGB colorcube (Smith, 1978).

Current approaches to conversion of pixels from RGB to HSV color space in C++ language rely in most cases on a code which uses conditional expressions preventing the C++ compiler to take advantage of instructions designed for parallel data processing. Although the specialized software can take advantage of multi-core CPUs, the performance of single core remains unoptimized. Successful attempts were given to improve the RGB to HSV conversion by using GPU (Engel, 2004; Ferrugia et al., 2006), but there was no attempt to improve the conversion by using CPU and enhanced SSE instruction set.

This paper introduces a completely new algorithm for conversion from RGB to HSV color space designed to use the SSE technology. The main goal was to create a fully branchless algorithm able to process four pixels in parallel which is compatible with pure C++ implementation.

2. TRADITIONAL RGB TO HSV ALGORITHM

Traditional way how to calculate HSV from RGB is based on evaluation of value (1), minimum (2), and chroma (3).

$$V = \max(R, G, B) \tag{1}$$

$$m = \min(R, G, B) \tag{2}$$

$$c = V - m \tag{3}$$

To prevent division by zero in case that the color is achromatic it is needed to run through a set of conditions to calculate the saturation (4), and finally, the hue (5).

$$S = \begin{Bmatrix} if\ (c == 0)\ 0 \\ else\ \frac{c}{V} \end{Bmatrix} \tag{4}$$

$$H = \begin{Bmatrix} if\ (c == 0)\ 0 \\ else\ if\ (V == R)\ 1 + \frac{G-B}{6c} \\ else\ if\ (V == G)\ \frac{2}{6} + \frac{B-R}{6c} \\ else\ \frac{4}{6} + \frac{R-G}{6c} \end{Bmatrix} mod\ 1 \tag{5}$$

3. OPTIMIZED RGB TO HSV ALGORITHM

Let P_{ARGB} be a pixel in RGB color space and let P_{AHSV} be a pixel in HSV color space. The domain of all pixel components in both color spaces is [0, 1]. The capacity of one SSE register is 4 single-precision floating points which matches the count of color components in P_{ARGB} and P_{AHSV} pixels. The algorithm processes four pixels per iteration, thus an index was added to distinguish between the processed pixels. The naming of variables matches the pixel components, thus R_1 is a red component of the first P_{ARGB} pixel and H_4 is a hue component of the last P_{AHSV} pixel. The alpha component is not changed by the algorithm so it can be assigned to both P_{ARGB} and P_{AHSV} pixels or it can be completely ignored.

Firstly, it is needed to load all four input pixels into four SSE registers and transpose them to prepare color components for further processing (6). The instructions used to do the transposition are UNPCKHPS, UNPCKLPS, MOVLHPS, and MOVHLPS.

$$\begin{bmatrix} A_4 & A_3 & A_2 & A_1 \\ R_4 & R_3 & R_2 & R_1 \\ G_4 & G_3 & G_2 & G_1 \\ B_4 & B_3 & B_2 & B_1 \end{bmatrix} \leftarrow \begin{bmatrix} B_1 & G_1 & R_1 & A_1 \\ B_2 & G_2 & R_2 & A_2 \\ B_3 & G_3 & R_3 & A_3 \\ B_4 & G_4 & R_4 & A_4 \end{bmatrix} \tag{6}$$

After the transposition a minimum and maximum values of RGB components are computed in parallel by using MINPS and MAXPS instructions (7). The register which contains the maximum values will be used later to construct the output HSV pixels. The minimum and maximum values are then used to calculate the chroma (8) and saturation (9) using SUBPS and DIVPS instructions as follows:

$$\begin{matrix} [V_{4\ldots1}] \\ [m_{4\ldots1}] \end{matrix} \leftarrow \begin{matrix} [\max(R_{4\ldots1}, G_{4\ldots1}, B_{4\ldots1})] \\ [\min(R_{4\ldots1}, G_{4\ldots1}, B_{4\ldots1})] \end{matrix} \tag{7}$$

$$[C_{4\ldots1}] \leftarrow [V_{4\ldots1} - m_{4\ldots1}] \tag{8}$$

$$[S_{4\ldots1}] \leftarrow \left[\frac{C_{4\ldots1}}{V_{4\ldots1}}\right] \tag{9}$$

If the input pixel is achromatic then the resulting saturation will be infinite, which is caused by division by zero. Our algorithm is designed to ignore such case and to correct the saturation together with the hue, which is calculated next.

The original equation for hue calculation (5) is based on conditions and branches. This is not suitable for parallel

processing, thus new equation (10) was defined. The equation completely removes branches but increases register usage. The conditions were replaced by parallel comparisons and branching was replaced by bitwise operators.

$$H = (h_{base} + \left(\frac{R_m+G_m+B_m}{6c}\right)) \; mod \; 1.0 \qquad (10)$$

The variable h_{base} is an offset; variables R_m, G_m, and B_m are RGB input components which may be cleared or negated. The SSE instruction set contains CMPEQPS, CMPNEQPS, and CMPLEPS instructions which can be used for parallel comparison. The result of parallel comparison is a mask, thus instructions ANDPS, XORPS designed for bit manipulation are needed to correctly use the mask in further calculations. The masks used to adjust the h_{base}, R_m, G_m, and B_m are shown in (11).

$$\begin{gathered}[X_{4\ldots1}] \leftarrow [V_{4\ldots1} \neq R_{4\ldots1}] \\ [Y_{4\ldots1}] \leftarrow [(V_{4\ldots1} == R_{4\ldots1}) | (V_{4\ldots1} \neq G_{4\ldots1})] \\ [Z_{4\ldots1}] \leftarrow [(V_{4\ldots1} == R_{4\ldots1}) | (V_{4\ldots1} == G_{4\ldots1})]\end{gathered} \qquad (11)$$

The X, Y, and Z masks are designed in a way that two masks are always set to ones and one is always set to zeros. This assumption is based on the original equation (5) where two of the three RGB components always contribute to the hue and one (the greatest one) is ignored. First of the two remaining components is negated and the second is kept as is. The h_{base}, R_m, G_m, and B_m variables are calculated by using (12).

$$\begin{gathered}[h_{base\,4\ldots1}] \leftarrow [((-\frac{4}{6}\&X_{4\ldots1}) \wedge sign(!\,Z_{4\ldots1})) + 1.0] \\ [R_{m\;4\ldots1}] \leftarrow [(R_{4\ldots1}\&X_{4\ldots1}) \wedge sign(!\,Y_{4\ldots1})] \\ [G_{m\,4\ldots1}] \leftarrow [(G_{4\ldots1}\&Y_{4\ldots1}) \wedge sign(!\,Z_{4\ldots1})] \\ [B_{m\,4\ldots1}] \leftarrow [(B_{4\ldots1}\&Z_{4\ldots1}) \wedge sign(Y_{4\ldots1})]\end{gathered} \qquad (12)$$

The h_{base} variable is initially set to -4/6, cleared to zero if X contains zeros and negated if Z contains zeros. Addition of 1 ensures that the final hue will be always positive. Hue greater than or equal to 1 will be normalized at the end of the algorithm to the [0, 1) domain, thus it is safe to increment it by 1, because this addition will not involve the result.

The last step is a correction. We have already mentioned that our algorithm processes the achromatic cases, thus the S and H can be infinite at the moment. The correction is based on checking whether the chroma is zero. In such case the S and H variables are cleared by performing bitwise AND operation (13).

$$\begin{gathered}[S_{4\ldots1}] \leftarrow [S_{4\ldots1}\&\,(C_{4\ldots1} \neq 0.0)] \\ [H_{4\ldots1}] \leftarrow [H_{4\ldots1}\&\,(C_{4\ldots1} \neq 0.0)]\end{gathered} \qquad (13)$$

The last part of the correction is to normalize H to be in the interval [0, 1); see modulo used in (10). We know that the current interval of H is [0, 2), but there is no fraction or modulo instruction available in SSE instruction set. If we leverage the assumption that H is always less than 2 then 1 can be subtracted from H in case that it is greater than or equal to 1 (14).

$$[H_{4\ldots1}] \leftarrow [H_{4\ldots1} - ((H_{4\ldots1} \geq 1.0)\;\&\;1.0)] \qquad (14)$$

The last step is to transpose the content of SSE registers to obtain values of four P_{AHSV} pixels (15).

$$\begin{bmatrix} V_1 & S_1 & H_1 & A_1 \\ V_2 & S_2 & H_2 & A_2 \\ V_3 & S_3 & H_3 & A_3 \\ V_4 & S_4 & H_4 & A_4 \end{bmatrix} \leftarrow \begin{bmatrix} A_4 & A_3 & A_2 & A_1 \\ H_4 & H_3 & H_2 & H_1 \\ S_4 & S_3 & S_2 & S_1 \\ V_4 & V_3 & V_2 & V_1 \end{bmatrix} \qquad (15)$$

4. RESULTS

The first part of testing was to compare the performance of the presented algorithm. The C++ code and SSE enhanced code were written to determine whether there is a performance gain. The performance was measured by conversion of 125 million pixels from RGB to HSV color space. The run time of the standard algorithm implemented in C++ was 2028 [ms] in contrast to the run time of the presented algorithm enhanced by SSE which was only 920 [ms] (Fig. 1). The test was performed on Intel Core2 Duo T7500@2.2GHz mobile CPU running Windows 7. Similar results were obtained running the test on different hardware and different operating systems.

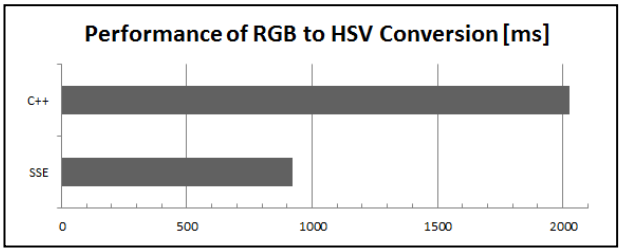

Fig. 1. C++ and SSE performance of RGB to HSV conversion

There are two reasons why our algorithm is faster. Firstly, the SSE enhanced code processes 4 pixels in parallel. Secondly, the SSE enhanced code is completely branchless, preventing branch misprediction which can be costly.

The second part of testing was to compare the numerical stability of our algorithm, because IEEE 754 floating-point operations are not associative (Monniaux, 2008). The maximum absolute error of our algorithm compared to the pure C++ implementation was ~1.2 * 10^{-7}. Based on the second test our algorithm can be regarded as numerically stable.

5. CONCLUSION

More efficient algorithm for conversion of pixels from RGB to HSV color space has been found. The algorithm was implemented in C++ language using SSE intrinsics for parallel data processing. For comparison the traditional algorithm was also implemented in pure C++ language. As shown, our algorithm is more than 50% faster compared to the pure C++ version and is numerically stable.

Although the algorithm was implemented using SSE intrinsics available for IA32 and AMD64 architectures only, it is possible to rewrite the code to use any other SIMD instruction set in the future.

6. REFERENCES

Engel, W. F. (2004). *ShaderX2: Shader Programming Tips & Tricks with DirectX 9*, Wordware Publishing, Inc., ISBN: 1-55622-988-7

Ferrugia J. P., Horain P., Guehenneux E., Alusse Y. (2006). *GPUCV: A Framework for Image Processing Acceleration with Graphics Processors*, Multimedia and Expo, 2008 IEEE International Conference, ISBN: 1-4244-0366-7

Joblove, G. H.; Greenberg, D. (1978). *Color Spaces for Computer Graphics*, ACM SIGGRAPH Computer Graphics, DOI>10.1145/965139.807362

Monniaux D. (2008). *The pitfalls of verifying floating -point computations (ver. 5)*, ACM Transactions on Programming Languages and Systems, DOI>10.1145/1353445.1353446

Smith A. R. (1978) *Color Gamut Transform Pairs*, ACM SIGGRAPH Computer Graphics, DOI>10.1145/800248.807361