Real-Time Specular Highlight Removal Using the GPU

Costin A. Boiangiu, Razvan M. Oita, and Mihai Zaharescu

***Abstract*—**This paper focuses on a computationally-intense image preprocessing technique, specular light removal, operation which is essential for the correctness of many of the modern computer vision algorithms. The specular component is different from other lightning components in the way that it is dependent on the position of the viewer. This causes apparent changes in object shape during camera movement. Removal of this component has been implemented in multiple ways, but usually the processing takes a long time. We port an existing algorithm on a SIMT GPU architecture and prove that such complex algorithms can work in real-time, thus being of high importance for robot vision.

***Keywords*—**Computer vision, DirectX11, GPU porting, specular Removal.

I. Introduction

SEPARATION of diffuse and specular lighting components is an important subject in the field of computer vision, since many algorithms in the field assume perfect diffuse surfaces. For example, a stereo matching algorithm would expect to find similar changes in the positions of the detected regions of interest; however, the specular reflection actually moves over the surface and doesn't remain stuck to the same part of the object. Segmentation algorithms are highly influenced by the strong and sharp specular lights; the specular lit object will thus manifest two very different regions. Object tracking can also be disturbed by the sudden appearance of a strong specular light on an object that seemed constantly lit for a number of frames. 3D object reconstruction from three images of the same scene being lit from different angles is very easy to achieve, by calculating the normals from the diffuse light intensities, but specular lights again disturb the simple light calculations. In conclusion, any image-based classification is influenced by the variations in lighting [7].

In the real world almost all dielectrics exhibit specular highlights (except those with extremely rough surfaces or a low Fresnel reflectance at normal incidence, like chalk), while metals exhibit only specular reflection and no diffuse contribution. Thus, it is important to develop algorithms focused on removing specular highlights from textured surfaces.

The algorithm presented in this article improves on an existing algorithm by Tan and Ikeuchi [1], by adapting their method to run on a modern GPU. This is done using the newest features of the DirectX11 API, allowing for random read and write access from and to buffers containing various types of data. This paper continues the processing described in [8].

II. Related work

In computer graphics certain models are used to approximate the intensity distribution of lighting components. Diffuse reflections approximately follow Lambert's Law [2], their intensity is determined by the angle between the surface normal at a point and the vector from the light source. Specular reflections can be modeled based on a micro-facet model, such as the Torrance-Sparrow model [3], which considers surfaces as consisting of tiny perfect reflectors, averaging their normals using a normal distribution function.

There are many existing algorithms based on reflectance, models that separate reflection components, but most of them require additional images besides the original one (ex: [6]). This is because the general case of retrieving two intrinsic images from a single one is ill posed, meaning that we have two unknowns and a single equation. The ability to reconstruct those two images is more or less the result of an estimation process, but the helping hand comes from the fact that a color image contains not a single channel, but three different color channels. The physical properties of the color components for the specular light are different from those of the diffuse light. Two examples are color and polarization: the reflected light is very polarized, meaning that it can be filtered with a polarizing filter. This technique is used to test the effectiveness of software based solutions. And the most important part for us, the specular light bounces off the surface as soon as it touches it, meaning that it doesn't diffuse through the object to be affected by the object's color. Thus, the specular reflection has the color of the light source, while the diffuse light, that penetrated the object deeper and got scattered by it, has a mixture of the light source color and the object's color.

In their paper [1], Tan and Ikeuchi introduced an algorithm based on Shafer's dichromatic reflection model (model explained here: [4]). They used a single input image, normalized the illumination color using its chromaticity, thus

C. A. Boiangiu is with the Computer Science Department from "Politehnica" University of Bucharest, Romania (e-mail: costin.boiangiu@cs.pub.ro)

R. M. Oita is with the Computer Science Department from "Politehnica" University of Bucharest, Romania (e-mail: razvan.oita@siggraph.org)

M. Zaharescu is with the Computer Science Department from "Politehnica" University of Bucharest, Romania (e-mail: mihai.zaharescu@cs.pub.ro)

obtaining an image with a pure white specular component. By shifting the intensity and maximum chromaticities of pixels non-linearly, while retaining their hue, a specular-free image was generated. This specular-free image has diffuse geometry identical to the normalized input image, but the surface colors are different. In order to verify if a pixel is diffuse, they used intensity logarithmic differentiation on both the normalized image and the specular-free image. This diffuse-pixel check is used as a termination condition inside an iterative algorithm that removes specular components step by step until no specular reflections exist in the image. These processes are done locally, by sampling a maximum of only two neighboring pixels.

Most of the presented methods can't be used in real-time. There are, however, implementations that process an image in less than one second [5]. The algorithm we chose to implement also has the drawback of being very slow.

III. GPU Processing Overview

In the past years, processing on the GPU has started to become widely adopted. This is because the GPU has a very different architecture from the CPU, one that is very suited for processing images. Image processing usually means applying the same identical operations over the entire surface of the image: on every pixel or on multiple local regions, overlapped or tiled.

Modern GPUs have multiple Processing Units (SIMD Cores), the equivalent of a CPU core. Each Processing unit contains multiple Processing elements (Shader Cores). Each shader core can work on vectors of multiple elements. Quantitative, the total number of theoretical operations that can be done in a cycle, is in the order of thousands, where on a modern PC CPU it is around 4.

Another advantage is memory speed. Even though accessing GPU RAM takes orders of magnitude more than accessing the GPU Cache (similarly on the CPU), the GPU takes advantage of its possibility to run millions of threads seemingly in parallel. When a group of threads is waiting for a response from the slower memory, with very little overhead, a new group of threads is started. This is like processing a 36x36 tile, and while we are waiting for the data to come, we start a new tile in the same processors and put the old threads on wait.

The problem is that mapping the desired problem to the card architecture is seldom possible. From the thousands of threads that can potentially be running in parallel, we may end up with hundreds or tens. But this also gives a two orders of magnitude speedup over the CPU, on simple image processing.

For complex problems, the speedup can become sub-unitary, as resources are wasted and memory transfer is slow. This paper tries to map a problem on a GPU architecture in order to obtain a real speedup by massively parallelizing the operations.

IV. The GPU Algorithm

The algorithm we decided to port to the GPU is a specularity removal problem, because this is a widely-used (and sometimes necessary) preprocessing phase for a lot of computer vision algorithms. We chose an algorithm that works on single images, in order to be integrated easily in existing applications, it is robust, so it doesn't produce unrecognizable results when the input data doesn't fit the requirements, and most importantly, it has the possibility to be split into multiple individual tasks, in order to be able to benefit from the highly parallel structure of the GPU.

We went with Tan and Ikeuchi's [1] proposed solution. The aforementioned authors started from the reflection model containing both the specular and diffuse spectral functions and energy diffusion and rewrote it in order to find distinguishable characteristics between the two illumination types.

The entire technique can be split into two processing approaches:

1) A single pixel processing step that plots the colors in a different coordinate system in order to estimate the amount of diffuse and specular components. This phase is perfect for parallelization.
2) A phase that restores the specular free images, verifying the amounts of specular reflection of neighboring pixels. This phase is not as easy to port on the GPU, as neighboring elements need to be accessed simultaneously, and because the processing is done on just one of the neighboring pixels, without previously knowing which is the one that needs to have its specular value lowered. Another limiting factor is that the algorithm is iterative, imposing a strict parallelization limit, but we hope that by speeding up each of the iterations enough, the entire algorithm can gain enough speed.

Details regarding the intricacies of the original algorithm can be found in [1]. Some important points will be emphasized in the following paragraphs. Also, advanced applications of GPGPU based parallelization are described in [5, 6].

All the steps of the algorithm are implemented using pixel shaders, Shader Model 5.0 with the newest compute features of DirectX 11.

Image data is held using a structured buffer. DirectX 11 allows the creation of buffers that can be bound as output to the pixel shader or compute shader parts of the pipeline. This means that a pixel shader can write to a buffer, addressing its contents exactly like one would use an array in C++. A structured buffer contains elements of custom defined structures. In our case it contains the color data and a flag used to indicate the type of pixel (specular, diffuse, noise, etc.). In order to obtain the conflict free random access the project uses the new DirectX 11 Unordered Access View buffer.

The notion of chromaticity is important and it basically represents a normalized RGB value. Maximum chromaticity is the normalized maximum color component (R, G, or B).

A new space is introduced, called the maximum chromaticity intensity space, where the maximum chromaticity is plotted on the horizontal axis and the maximum intensity of the color components on the vertical axis. It is observed that in

this space, the specular pixels' chromaticities are lower than the diffuse pixels' chromaticities. Thus, converting a specular pixel to a diffuse pixel is similar to shifting its chromaticity with respect to a certain larger diffuse chromaticity. This is called the specular-to-diffuse mechanism and it is a one-pixel operation, used to both generate the specular-free image, as well as to reduce the specularity of the input image.

The basic idea of the method is illustrated in figure 1.

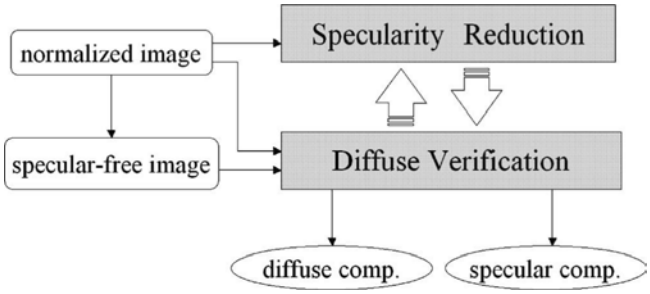

Fig. 1 Processing flow. Image taken from [1].

Given a normalized input image, firstly a specular-free image is generated as mentioned above. Based on these two images, a diffuse verification process is run. It basically verifies if the input image has diffuse-only pixels. If it does, the process terminates. Otherwise, specular reduction is applied, decreasing the intensity of the specular pixels until they become diffuse pixels. Specularity reduction and diffuse verification are both done iteratively until there is no more specularity in the input image.

A. *Specular-Free Image*

The specular-free image is generated on the GPU using a pixel shader. The operations used by the specular- to-diffuse mechanism map perfectly to HLSL functions used to manipulate pixel data. The image is rendered to a texture that is then bound to the following passes as a Shader Resource View.

B. *Pixel Flag Initialization*

This pass actually represents the diffuse verification process. It runs a pixel shader that samples the normalized input image by loading pixel data from the image buffer UAV and the specular-free image render target texture. Two neighbors are sampled from each image also, the vertical neighbor below and the horizontal neighbor to the right of the current pixel.

To determine the type of pixel and initialize the flag accordingly, intensity logarithmic differentiation is used. A pixel can either represent a specular highlight running in horizontal direction or vertical direction, or it can be a diffuse pixel. The flags are properly set for the current pixel by writing to the structured buffer UAV.

This shader also counts the number of specular pixels. This number is used as a termination condition for the loop that runs the specular reduction iterations. This is done by using a buffer with a single element bound as an UAV to the pixel shader output. Whenever a pixel is flagged as specular, an atomic increment is performed on the single element in this buffer. Then, the UAV buffer is copied to a second buffer built as a staging resource. This means it can be memory mapped and its contents can be read by the CPU code.

C. *Specular Reduction*

This pass runs two pixel shaders. Since a given pixel can apply the specular-to-diffuse mechanism either on itself, or on one of its neighbors, and pixel shader code is run in parallel on multiple threads, we need to determine which pixels need to execute code. The first shader does this, it checks whether a pixel is “specular” or not, then using data from its neighbors, the shader code determines which of the pixels needs to be shifted with respect to the other (based on their maximum chromaticities). Then the pixel is flagged, either as modifying itself, or one of its neighbors.

The second shader reads these flags and runs specular-to-diffuse shader code only on the pixels that are flagged for execution.

If run inside a single shader, the fact that a pixel thread is not aware if it was modified by a neighbor (due to the fact that it is run in parallel), it means that read-after-write hazards may appear. This generates undefined behavior, but usually the reduction process still converges to a mostly correct result, because the specular-to-diffuse mechanism generates spatially coherent results. However, if the algorithm is run in its entirety during a single frame, the visual result varies slightly from frame to frame. This is why two passes are necessary.

The rendering loop was written in such a way, that an iteration of specular reduction is executed during a frame. This allows one to view the reduction over multiple frames, clearly seeing how the specular highlights are removed.

V. Results

The GPU algorithm runs in about 250 milliseconds on a Radeon 7950 with 1792 stream processors running at 900 MHz, for the fish image in figure 2 (with a resolution of 640x480 pixels), and 50 seconds on one i5 CPU core running at 1.9 GHz, if all the iterations are executed in a single frame. The speedup is very large (200:1), however, we also want to point out that the CPU code was written just for validating GPU results and not having optimization in mind. An optimized CPU code may decrease the time by an order of magnitude, but not make real time. The results are presented in the following and include:

1) Dinosaur image (Fig. 2): a comparison with the original algorithm. Strong JPEG compression artifacts are visible in our result because we started from the compressed image offered in the original PDF paper. These are not visible when images with normal compression ratios are offered at the input.
2) Fish image (Fig. 3): showing the input, the specular free image constructed during single pixel processing, that generates a recolored specular free result, and the final output generated during the iterative phase that diminishes

the specular light at every iteration.

3) Various computer generated and real images: for testing the robustness and how the algorithm fails (Fig. 4, Fig. 5, and Fig. 6). The last test (Fig. 6) contains numerous grey regions, which Tan and Ikeuchi mentioned that are not suitable for this specularity removal algorithm [1].

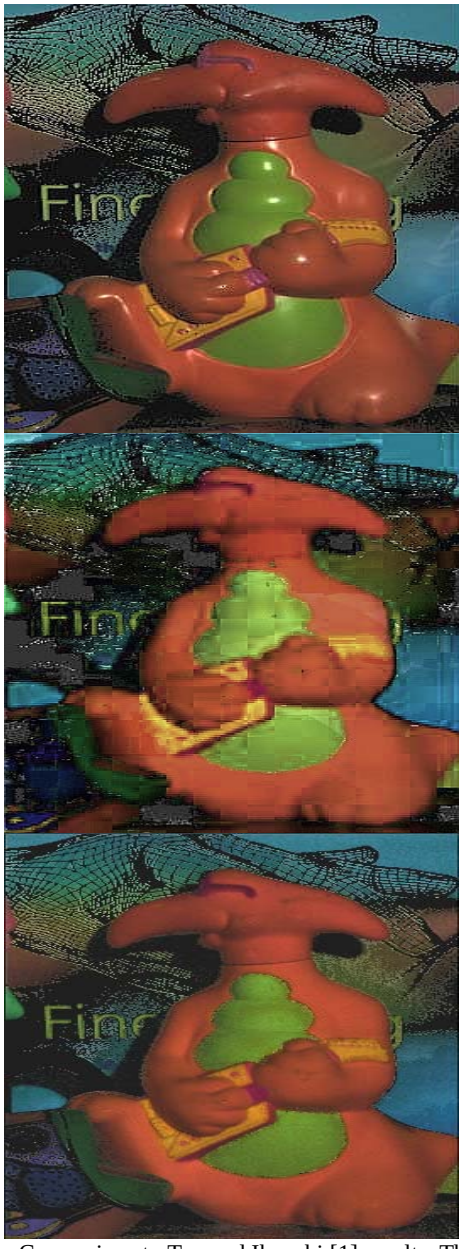

Fig. 2 Comparison to Tan and Ikeuchi [1] results. The tiles observed on our results are most likely the result of using the highly compressed, small resolution image copied from the source paper as input image. The following tests don't show those artifacts. Despite these, the specular removal gives similar results

Fig. 3 Fish image: Normalized input image; Specular-free image; Final output image with no specular highlights

Fig. 4 Results for a computer generated scene. The black regions are the cause of clipping in the original image.

Fig. 5 Results for a real scene. Problems can be seen in the bottom left corner where an object with small chrominance is visible.

Fig. 6 Results for a real scene containing non-colored objects.

A segmentation test was made with the original images and the images with specular light removal. The results are presented in Fig. 7.

Specific-purpose segmentation algorithms that will fully benefit from the specularity removal phase may be found in [9][10].

In our examples, the object count difference is not that high, but the important observation is that the specular object element will not remain in the same position during the movement of the camera, thus an object delimitation is much harder.

Even more, frame matching can be disturbed by the moving light elements.

Another processing that will improve segmentation results is

the homomorphic filter. This will minimize the differences in diffuse lighting, thus generating far less elements in the final result.

Fig. 7 Segmentation results with and without specular light.

VI. Conclusions

In this paper we offer the results of porting an existing specular removal algorithm to the GPU for reducing the computation time.

Our implementation of the algorithm works on most of the images, but it seems to be influenced by high compression noise in small images. One category of images on which it fails is on the ones containing non-colored objects. Tan and Ikeuchi mentioned this issue at the beginning of their paper [1], and the cause is obvious: the deduction of the specularity of each pixel is done on the basis of its color. If the object has no color, it is assumed to be highly specular. We mentioned at the beginning that we wanted to choose a robust algorithm that does not fail when offered improper input data. Even though the grey objects become darker, this will not become an issue of more importance for most computer vision algorithms than the specular light itself. The rest of the objects remain unchanged, so the algorithm can be safely used in the preprocessing phase.

Some quality loss is obvious in the output result because of minor simplifications in order to split the algorithm, but the overall gain in speed from the order of tens of seconds (on our implementation on the CPU) to less than a second is important.

Finally, this paper shows that there could be a large number of algorithms, that were overlooked in the past because of large running times but could benefit from today's technology for being incorporated in existing processing phases for robotic viewing.

VII. Future Work

The specular reduction process can be implemented as a single-pass by using a compute shader. Immediately after marking the pixels that change with specific flags, a barrier is placed for synchronization. After that, only the flagged pixels execute code.

It is also recommended for the image to be split into tiles, each tile representing a thread group. The tiles need to overlap each other with 1-pixel borders to ensure no tears are visible.

Even higher speeds can be achieved by utilizing multiple devices, each for a different image. [6]

As from the processing itself, a homomorphic filter applied on top of the specular free image could reduce the number of segments generated by most of the segmentation algorithms running in any computer vision processing pipeline.

References

[1] R. T. Tan, K. Ikeuchi, *"Separating reflection components of textured surfaces using a single image"*, Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 27, no. 2, pp. 178,193, Feb. 2005.

[2] J. H. Lambert, *"Photometria Sive de Mensura de Gratibus Luminis"*, Colorum et Umbrae. Augsberg, Germany: Eberhard Klett, 1760.

[3] K. E. Torrance, E. M. Sparrow, *"Theory for Off-Specular Reflection from Roughened Surfaces,"* J. Optics Soc. Am., vol. 57, pp. 1105-1114, 1966.

[4] S. Shafer, *"Using Color to Separate Reflection Components,"* Color Research and Applications, vol. 10, pp. 210-218, 1985.

[5] Q. Yang, S. Wang, N. Ahuja, *"Real-time specular highlight removal using bilateral filtering"*, Proceedings of the 11th European conference on Computer vision: Part IV (ECCV'10), Springer-Verlag, Berlin, Heidelberg, 87-100, 2010.

[6] A. Artusi, F. Banterle and D. Chetverikov, *"A Survey of Specularity Removal Methods"*, Computer Graphics forum vol. 30, number 8 pp. 2208–2230, 2011.

[7] S. N. Tica, C. A. Boiangiu, A. Tigora, *"Automatic Coin Classification"*, International Journal of Computers, NAUN, vol. 8, pp. 82-89, 2014.

[8] C. A. Boiangiu, R. M. Oiţă, M. Zaharescu, *"Single-Image Specular Highlight Removal on the GPU"*, Proceedings of the 6th International Conference on Applied Informatics and Computing Theory (AICT '15), Salerno, Italy, June 27-29, 2015, WSEAS Press, pp. 152-157.

[9] S. Basar, A. Adnan, N. H. Khan, S. Haider, *"Color Image Segmentation Using K-Means Classification on RGB Histogram"*, Proceedings of the 13th International Conference on Telecommunications and Informatics (TELE-INFO '14) Istanbul, Turkey, December 15-17, 2014, WSEAS Press, pp. 257-263.

[10] C. Fares, *"Hybrid Algorithm for Image Segmentation"*, Proceedings of the 13th International Conference on Applications of Computer Engineering (ACE '14), Lisbon, Portugal, October 30 - November 1, 2014, WSEAS Press, pp. 146-149.