GPU Accelerated Stylistic Augmented Reality

Rifat Aras and Yuzhong Shen

Abstract – With the introduction of programmable graphics pipeline, the highly parallel processing power of graphical processing units (GPU) is being used not only for special graphics effects but also for general purpose computation in areas such as molecular dynamics simulation, stock options pricing, and image processing. In this work, we utilize this power to increase the immersion level in an augmented reality (AR) application. To accomplish this task, the visual gap between real world and virtual objects are closed by applying non-photorealistic filtering/rendering techniques to both the real world video image and the rendered virtual object. The inherent requirement of real-time operation of AR is satisfied by implementing the mentioned techniques on GPU by using NVIDIA's CUDA.

Index Terms – non-photorealistic rendering, GPGPU, CUDA, augmented reality

I. INTRODUCTION

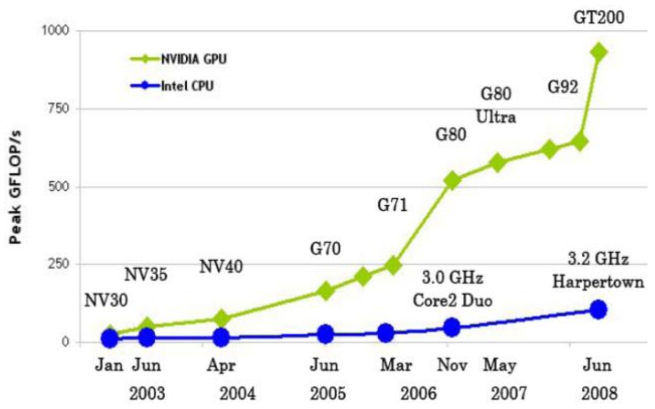
Augmented Reality (AR) can be defined as a variation of virtual environments, in which the user is allowed to see the real world that is enriched with superimposed virtual objects [1]. AR has been an active field in virtual reality research arena that plays an important role in a wide variety of usage scenarios from engineering maintenance assistance [2] to entertainment applications [1]. With the recent emergence of mobile platforms with high computing power, innovative outdoor AR applications were also added to the AR literature [3].

Although augmented virtual objects enrich the real world they are superimposed onto, the visual gap between the real world and rendered virtual objects limits the level of immersion. Two approaches can be adopted in order to close this gap: photorealistic rendering of virtual objects, or converting the real world to a non-photorealistic environment and augmenting non-photorealistic rendered virtual objects onto this altered environment [4]. Although we have the case of photorealistically rendered virtual objects or characters on top of real world scene in computer graphics assisted feature movies, it should be noted that this process is a labor intensive and offline process, which is not suitable for the real-time nature of AR. Because of this, the latter approach is adopted in this work.

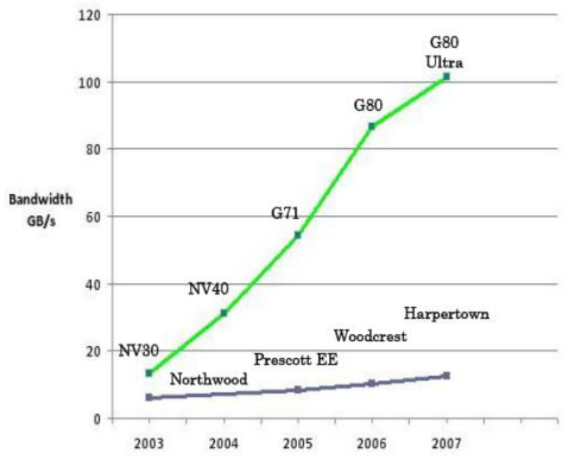
As stated above, an apparent requirement of AR is real-time operation. Although, there is an arsenal of real-time AR applications in the literature [5], our prescribed requirement of closing the visual gap between real-world and virtual objects need some sort of acceleration, which can be supplied by the programmable graphics hardware. In the last decade, graphics processing units (GPUs) have evolved into powerful parallel streaming computer architectures, offering enormous computing capabilities and memory bandwidth. GPUs are growing at a much faster rate than CPUs (central processing units, which are the main microprocessor on the motherboard) in terms of computational capability and memory bandwidth, as shown in Fig. 1. The transformation of graphics pipeline from fixed-function to user controlled programmable pipeline enabled the highly parallel GPU computing power to be used in areas other than custom shading such as image processing, computational biology, or physically based simulations. However, the early general purpose computations on GPUs (GPGPU) had some limitations and difficulties because of the need of performing these via graphic APIs (vertex and fragment shaders). With the introduction of Compute Unified Device Architecture (CUDA) by NVIDIA, this tedious task of mapping algorithms to vertex and fragment operations transformed into a collection of C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware (particularly the management of threads) from developers [6]. In this work, CUDA programming paradigm is utilized to apply an artistic cartooning filter to real-time video footage to increase the sense of immersion in an AR application.

Specifically, our main contributions are: (i) implementing the proposed AR pipeline in real-time by utilizing NVIDIA's CUDA paradigm; (ii) performing the virtual object rendering operation independent of non-photorealistic image filtering operation, thus ensuring the persistence of 3-D features of the virtual object.

The remainder of the paper is organized as follows. A brief literature review of the previous work is provided in Section II. Section III discusses GPU hardware capabilities and CUDA basics. In Section IV, the algorithms and used techniques are presented, followed by Section V presenting some visual and performance results. Finally, Section VI concludes the paper and discusses future work.



(a)



(b)

Fig. 1. The comparison of (a) floating point operation performance, (b) memory access bandwidth of CPU and GPU architectures over years [7].

II. PREVIOUS WORK

Non-photorealistic or artistic filtering of images is a well studied field in image processing world. The main adopted strategy is to use an edge preserving smoothing filter. Father of such filters is considered to be the non-linear Kuwahara filter [8], in which the window around the target pixel is divided into rectangular regions and the output value of the target pixel is set to be the mean value of the neighboring region with the lowest variance. Bilateral filtering [9], which is an extension to the Gaussian smoothing that not only considers spatial distance but also the chromatic distance, and median filter are other well studied edge preserving smoothing filters.

There are also various examples of real-time non-photorealistic filter applications on video footage. In the work of Winnemöller et al. [10], images were first converted to the LAB color space, then processed by multiple passes of bilateral filtering, and finally were blended with difference of Gaussian edges. To increase the level of abstraction, they also applied luminance quantization to the image. For real-time operation, their algorithm was implemented in GPU fragment programs. Another cartoon-like stylization technique that uses the GPU power is the work of Fischer and Bartz [4]. Their technique consists of applying a simplified bilateral filter - that only uses chromatic distance – iteratively onto a scaled down AR frame that also contains the virtual object rendered on the real world. Chen et al. [11] adopted a different approach to the problem of NPR rendering of AR scenes. Their work makes use of Voronoi diagrams to imitate watercolor paintings. After the Voronoi cells are extracted from AR frames containing virtual objects, they are colored with the average color of the original image within the cell.

III. A BRIEF CUDA PRIMER

High-quality graphics and realistic visual effects are important characteristics of many of the latest computing devices, such as personal computers, game consoles, and mobile phones. It is commonplace that computer graphics applications, such as video games and scientific visualizations, contain enormous amount of graphical objects, e.g., millions of triangle vertices and pixels and their associated properties, and therefore entail massive computational power to process them. In order to accelerate graphics computations, e.g., geometric transformations, lighting, and rasterization, GPUs are specialized to contain massively parallel computing capabilities to process numerous graphical objects such as triangles and pixels at the same time. The last decade has witnessed tremendous advances in commodity GPU architecture and computing capabilities, which started with simple 2D graphics accelerators and have grown into a computational powerhouse for demanding 3D applications. The reasons behind the difference in the computational capabilities between GPUs and CPUs are twofold. First, the transistors integrated on GPUs have outnumbered these on CPUs. Second and more importantly, GPUs have more transistors that are devoted to data processing rather than data caching and flow control. Previously only available on expensive high-end workstations, high performance GPUs are now available on consumer desktop computer for $100 or less.

GPUs' massively parallel computing capabilities can be harnessed for general purpose computations in addition to the traditional computer graphics applications. Indeed, GPUs are especially well-suited to address problems that can be expressed as data-parallel computations. Modern GPUs contain many multiprocessors that can process tens of thousands of concurrent threads for massively parallel computing. General-purpose computing on graphics processing units, or GPGPU for short, is a very active research area that studies the methods and algorithms for solving various problems and has been successfully applied to address a wide range of diverse problems, such as signal and image processing, computational fluid dynamics, physics simulation, data based, agent based modeling, computational finance, and computational biology [12-14].

The first generation GPGPU applications used specialized computer graphics programming languages such as GPU assembly language, which was difficult and tedious to use and error-prone. High-level shading languages, such as C for graphics (Cg) by NVIDIA, High Level Shading Language (HLSL) by Microsoft, and OpenGL Shading Language (GLSL), were released later. However, they were designed for computer graphics computations and it was not convenient to represent and solve general problems in such languages. The situation changed since NVIDIA, currently the world leader of GPU market, released Compute Unified Device Architecture (CUDA), which is an extension to the C programming language [7, 12, 15-18] for massively parallel computing using GPUs. The users do not need to have in-depth knowledge of programmable shaders in order to use CUDA for general computations. The availability of CUDA tremendously reduced the difficulties using GPU for general purpose computations. CUDA is available for a wide range of NVIDIA graphics products, including Tesla series for high-performance computing, Quadro series for professional graphics, and the GeForce series for consumer market.

CUDA provides two programming interfaces: C for CUDA and the CUDA driver API [7, 16-17]. C for CUDA contains a minimal set of extensions to the popular C programming language for GPGPU parallel computing, while the CUDA driver API is a lower-level C API that provides better level of control, but is more difficult to program and debug. As its name indicates, CUDA provides a unified hardware abstraction for a wide array of GPUs whose architecture can vary significantly. Compared with other parallel computing architectures, such as the Message Passing Interface (MPI), the CUDA architecture significantly simplifies parallel computing, since it automates many parallel computing tasks that were performed by the application programmers, such as thread generation and management.

CUDA programming model utilizes three major abstractions, namely, threads, memory, and barrier synchronization, to achieve high performance parallelism. Threads are the smallest parallel processing units in the CUDA architecture and data elements are mapped to parallel threads. CUDA devices with a compute capability of 1.3 support 1024 concurrent active threads per multiprocessor. Kernels in CUDA are C functions that are executed in parallel by different threads, that is, the same kernel can be executed simultaneously by tens of thousands of threads. Threads can be organized into hierarchical structures using thread blocks, which can be further organized into a block grid. Threads in the same block are executed concurrently and can cooperate through barrier synchronization and shared access to a memory space private to the block. To increase scalability, CUDA requires the thread blocks to be executed independently, in any order, or in parallel, enabling the CUDA program performance to increase and scale with the number of multiprocessors in a GPU.

CUDA devices use several different types of memories, both physically and logically, including registers, shared memory, global memory, constant memory, texture memory, and local memory. Different types of memory have significantly different bandwidth and access latency and thus must be utilized with great caution to achieve the best performance. Measures of memory optimization include, but not limited to, minimizing data transfer between host (CPU) and device (GPU), using coalesced (patterned) access to global memory, and maximum utilization of shared memory.

IV. ALGORITHMS

The focus of this work is to implement real-time non-photorealistic rendering (NPR) effects for real-world camera image and virtual objects in the AR domain. AR related tracking issues are handled by the ArToolkitPlus tracking library developed by Graz University of Technology [19]. In addition to the original AR processing pipeline, our technique introduces two GPU accelerated steps: (1) application of the NPR filter to the camera image, and (2) non-photorealistic rendering of virtual objects on top of the cartoonized camera image.

For each frame of the process, an image is captured from a camera, which is first fed to the tracker library to obtain camera pose information (ModelView matrix in OpenGL parlance). The same camera image is then fed to the implemented CUDA program to apply threshold blur filter followed by a Sobel edge detector filter. The two buffers obtained from threshold blur filter and Sobel edge detector filter are blended together to form the cartoonized real-world basis (Fig. 5). Finally, a virtual object that is shaded by an OpenGL Shading Language (GLSL) based Gooch [20-21] shader is rendered on top of the cartoonized real-world image using the camera pose information obtained from the tracker library (Fig. 2).

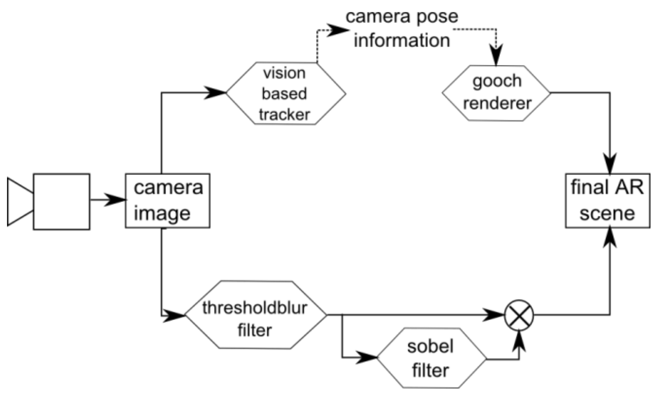


Fig. 2 Proposed AR pipeline.

A. Threshold Blur Filter

Threshold blur filter is a variant of the well known Gaussian blur filter, in which a pixel is included in the convolution only if it differs from the center pixel by less than a given threshold value. The introduced threshold value ensures preserved edges and eliminated noises, unnecessary details, and textures (Fig. 3).

Let $f\colon \mathbb{R}^2 \rightarrow \mathbb{R}^3$ be the color function of an image that maps the coordinate of pixels (x, y) to a (R, G, B) color vector. Then the threshold blur filter for the central pixel located at (x', y') with the given threshold value φ would be represented by (1).

$$\frac{1}{2\pi\sigma^2}\sum_{x,y}\begin{cases} e^{\frac{-x^2+y^2}{2\sigma^2}} & , \quad |f(x',y') - f(x,y)| \le \varphi \\ 0 & , \quad |f(x',y') - f(x,y)| > \varphi \end{cases}$$
(1)

A filter kernel is separable if it can be factorized into two vectors. With separable filters, convolutions that would originally need n^2 operations can be performed in $2n$ operations. Gaussian blur filter is a separable filter, whose factorized row and column vectors are normal distributions having the same mean and standard deviation as the original 2-D Gaussian blur filter kernel.

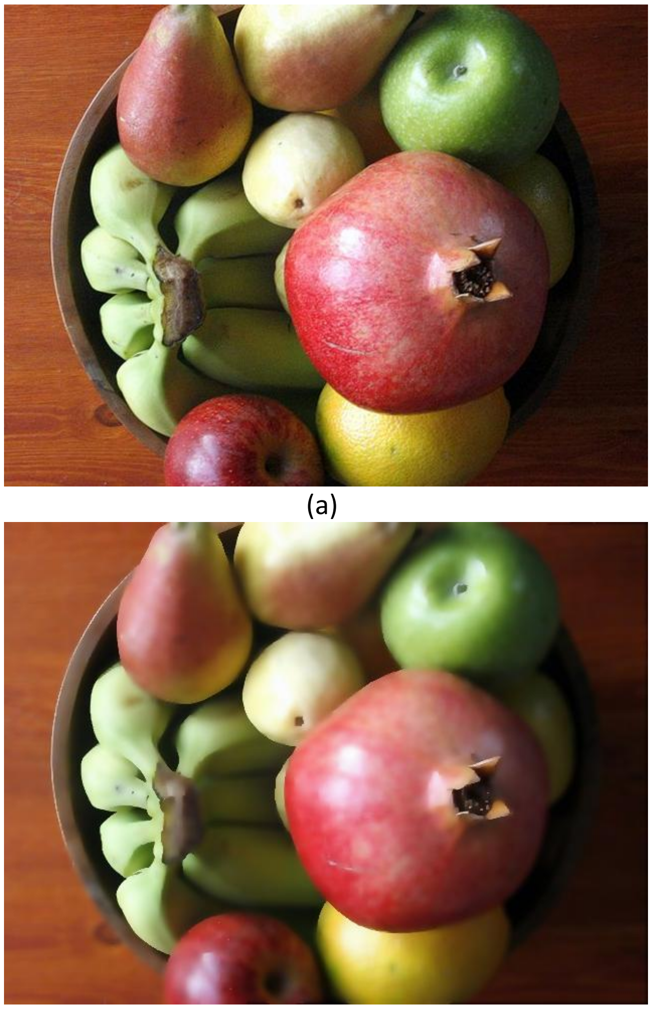


Fig. 3. Threshold blur filter is applied on an image. The edges are preserved whereas noise and textures are reduced. (a) Original image. (b) Result of applying Gaussian blur filtering to the original image.

As stated before, threshold blur is a variant of the Gaussian blur, which means that it shares the separability property. We have taken advantage of this property and implemented threshold blur as two separate CUDA kernels for row pass and column pass. The captured camera image is copied to the GPU device memory as an OpenGL pixel buffer object (PBO), which is first filtered with the row component and then with the column component. In order to harness the parallel processing power of the GPU to a maximum extent, CUDA related optimizations such as bank conflict free shared memory usage and coalesced global memory accesses are performed [7]. As stated before, shared memory is a low-latency memory portion that is shared among the threads of a thread block. Shared memory is divided into 16 equally sized banks that can be accessed simultaneously in order to achieve high bandwidth requirement. However, if two memory access requests are issued for the same shared memory bank, then there is a bank conflict, and the access is serialized.

Tesla C1060 GPU has a video memory bandwidth of 102 GB/s. This high bandwidth can only be utilized when memory accesses are coalesced into single memory transactions. Coalesced global memory access requirements change from architecture to architecture and described in the CUDA Programming Guide [7] in a detailed way.

B. Sobel Filter

Sobel filter is a gradient approximating filter that convolves a source image with row and column masks (S_x *and* S_y) (2).

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (2)$$

These masks look for edges in both the horizontal and vertical directions, which are then combined into a single metric. Defining I as the source image, $*$ as the 2-D convolution operator, and G_x, G_y as horizontal and vertical edge approximations, the single edge metric G can be found as shown in (3).

$$\begin{gathered} G_x = I * S_x, G_y = I * S_y \\ G = \sqrt{{G_x}^2 + {G_y}^2} \end{gathered} \quad (3)$$

In our implementation, Sobel filter is applied after the threshold blur filter pass. The filtered image that already resides on GPU global memory is copied onto shared memory by individual CUDA threads. After all of the corresponding pixels of the image are copied onto the shared memory, each pixel is convolved first by the row mask and then by the column mask. The results of these convolutions are stored as two different values, which are used to find the final magnitude of the edge (Fig. 4).

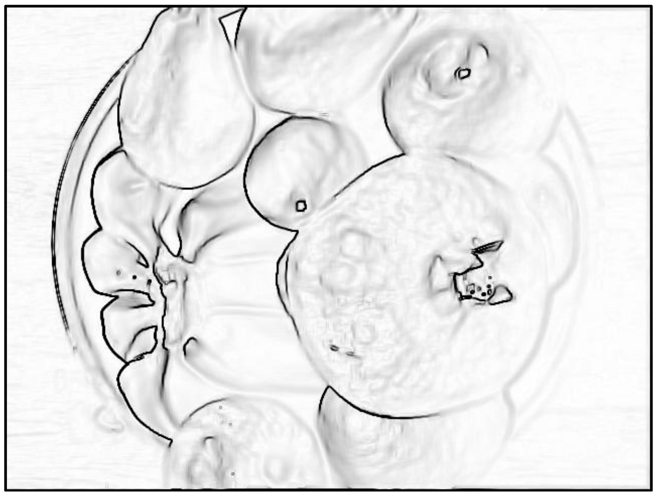


Fig. 4. The result of the sobel filter applied to the sample image.

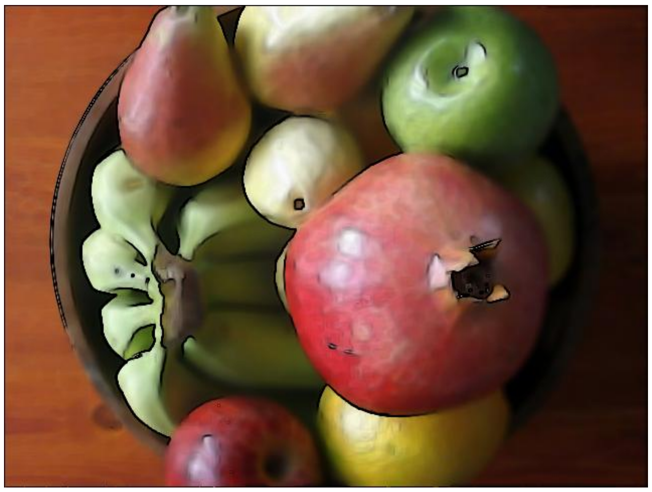


Fig. 5. The output buffers from threshold blur and sobel filters are blended together to obtain the cartoonized image.

C. Gooch Shading

Since the introduction of programmable graphics pipeline, lots of custom shaders have been implemented to create sophisticated onscreen effects rather than using the flat or Gouraud shading that the graphics libraries provide by default. One class of custom shaders is NPR shaders. Gooch shader is a member of NPR shaders [21], which we use to render virtual objects in an AR scene to increase the sense of immersion. Gooch shader is composed of two basic steps. Explicit rendering of black curves that represent silhouette, boundary, and crease edges and Phong lighting model based rendering of the object with a cool color to indicate surfaces facing away from the light source and a warm color to indicate surfaces facing toward the light source [20] (Fig. 6).

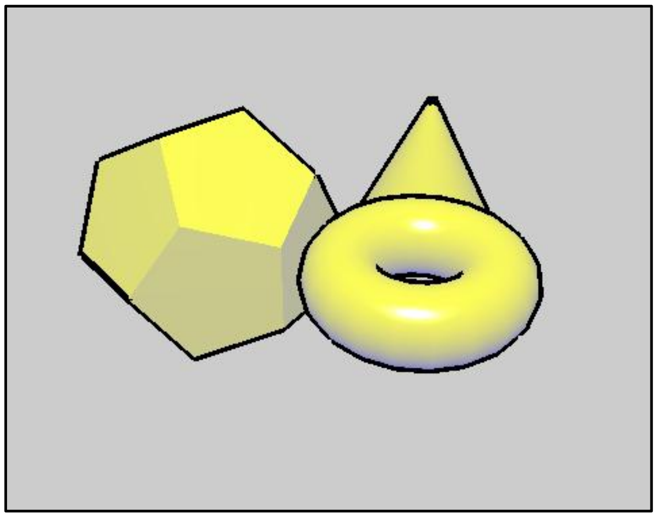


Fig. 6. Gooch shader is used to render different geometric 3-D objects.

V. RESULTS

The proposed technique was tested in several AR scenes on a laptop computer system with an Intel Core2Duo processor running at 2.2 GHz and an NVIDIA 8600M GT graphics card. The webcam that is used to capture video images to be processed works at a resolution of 640 by 480 pixels. Virtual object models used in AR scenes are imported in OBJ file format.

In Table 1, the GPU processing times of individual steps in our pipeline are listed. Per frame of operation, a total of four CUDA kernels run on the GPU. ThresholdBlurFilter_Row and Column pass kernels apply the threshold blur filter to a given image. SobelFilter kernel detects the edges of the blurred image, which are then blended by the BlendKernel.

TABLE 1
GPU PROCESSING TIMES PER FRAME

Kernel	**Average Processing Time (microseconds)**
ThresholdBlurFilter_RowPass	3982
ThresholdBlurFilter_ColPass	3628
SobelFilter	2077
BlendKernel	490

For comparison of individual steps in our pipeline with their CPU counterparts, OpenCV [22] library is used that provides various optimized implementations of filters. Our ThresholdBlur implementation was compared to OpenCV's Bilateral filter, which have similar computation complexities, and our Sobel implementation was compared to the library's Sobel filter implementation. GPU over CPU performance results show about 100 times performance increase for the edge preserving blur filter and about 25 times performance increase for the Sobel filter (Fig. 7).

The overall system frame rate of our AR pipeline including the video image capture, buffer copying operations, vision based marker tracking, non-photorealistic filtering of real world video image, and non-photorealistic rendering of virtual objects is on average varies from 35 to 55 FPS according to the rendered virtual object complexity on the hardware mentioned above.

Fig. 8 shows still images of test AR scenes. The top row contains the original AR frame, whereas the bottom row contains the final image after the original frame passed through our pipeline.

VI. CONCLUSION and FUTURE WORK

In this work, we demonstrated the extensive computing power of GPUs and harnessed this power by utilizing NVIDIA's Compute Unified Device Architecture (CUDA). To the best of our knowledge, this work is the first one to use CUDA paradigm in order to implement a real-time non-photorealistic filtering effect to the real-world video image in the AR context. Another difference of this work from the previous ones is that virtual objects are overlaid on top of the NPR filtered camera image after passing through custom shading process. We claim that, by applying non-photorealistic based filters to both real world video image and virtual objects, it is possible to increase the level of immersion in an augmented reality application. A near-term goal in this work is to move the entire vision based tracking system onto the GPU, in order to overcome a possible bottleneck in the current implementation of the pipeline. Also, we hope to implement other non-photorealistic filtering effects like technical illustration to experiment with possible usage areas like maintenance assistance.

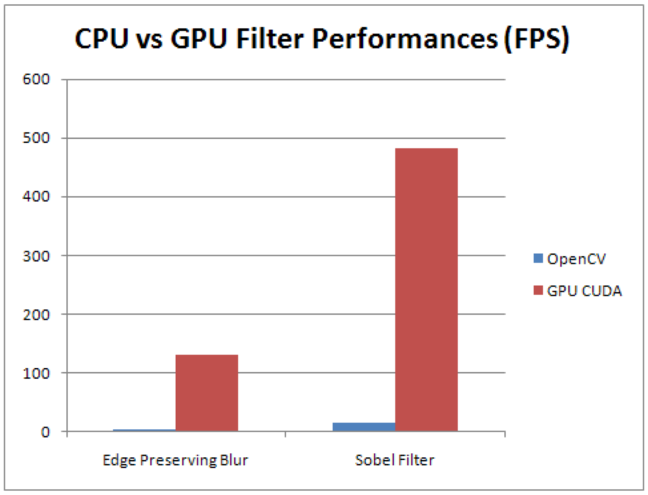


Fig. 7. OpenCV (CPU) vs GPU implementation performance comparison.

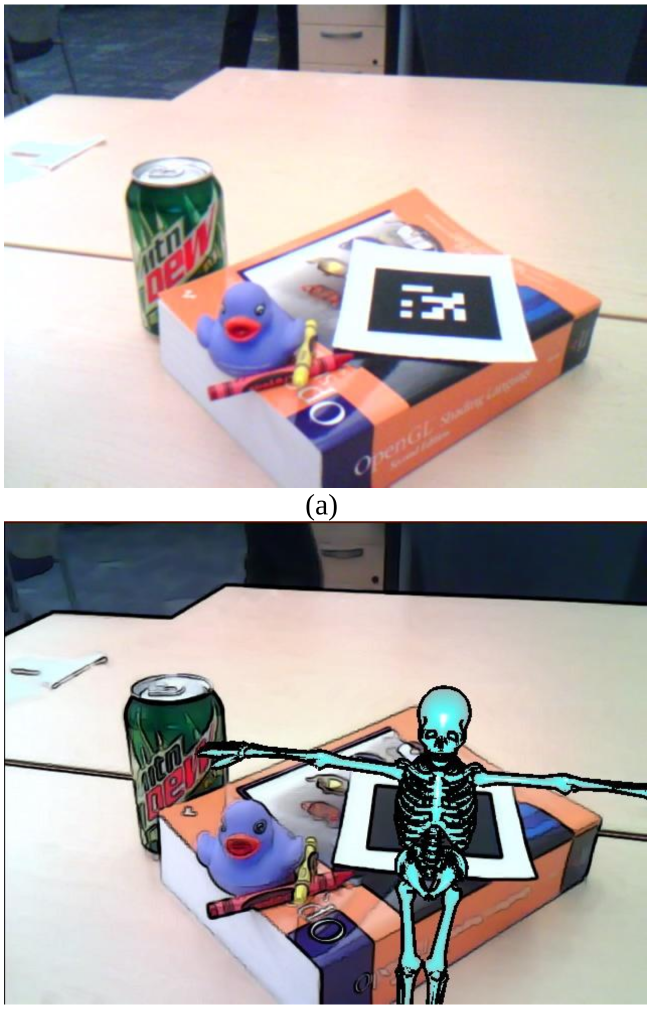


Fig. 8. Result of the proposed algorithm applied to an AR scene. (a) The original AR scene, (b) the AR scene after passed through the proposed pipeline.

REFERENCES

[1] R. T. Azuma, "A Survey of Augmented Reality," *Presence*, vol. 6, pp. 355-385, 1997.

[2] S. Feiner, B. MacIntyre, and D. Seligmann, "Knowledge-based augmented reality," *Communications of the ACM*, vol. 36, pp. 52-62, July 1993.

[3] Layar, *Augmented Reality Browser - Layar*, 2010. [Online]. Available: http://www.layar.com. [Accessed: Feb. 18, 2010].

[4] J. Fischer and D. Bartz, "Real-Time Cartoon-Like Stylization of AR Video Streams on the GPU," Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, 2005.

[5] H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana, "Virtual Object Manipulation on a Table-Top AR Environment," in *International Symposium on Augmented Reality*, Munich, Germany, 2000, pp. 111-119.

[6] T. R. Halfhill, "Parallel Processing with CUDA," Microprocessor Report, Jan. 28 2008.

[7] NVIDIA, "CUDA Programming Guide," Apr. 2 2009.

[8] M. Kuwahara, K. Hachimura, S. Eiho, and M. Kinoshita, "Processing of RI-angiocardiographic images," *Digital Processing of Biomedical Images*, pp. 187-203, 1976.

[9] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images," in *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, Washington, DC, USA, 1998, p. 839.

[10] H. Winnemöller, S. C. Olsen, and B. Gooch, "Real-Time Video Abstraction," *ACM Transactions on Graphics*, vol. 25, pp. 1221-1226, Jul. 2006.

[11] J. Chen, G. Turk, and B. MacIntyre, "Watercolor inspired non-photorealistic rendering for augmented reality," in *VRST '08: Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, Bordeaux, France, 2008.

[12] K. Fatahalian and M. Houston, "GPUs: A Closer Look," *Queue*, vol. 6, pp. 18-28, 2008.

[13] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*: Addison-Wesley Professional, 2005.

[14] GPGPU.org, "General-Purpose Computation on Graphics Hardware," *http://gpgpu.org*, 2009.

[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40-53, 2008.

[16] NVIDIA, "CUDA C Programming Best Practices Guide," 2009.

[17] NVIDIA, "CUDA Reference Manual," 2009.

[18] NVIDIA, *CUDA Zone*, 2009. [Online]. Available: http://www.nvidia.com/object/cuda_home.html. [Accessed: Feb. 19, 2010].

[19] D. Wagner and D. Schmalstieg, "ARToolKitPlus for Pose Tracking on Mobile Devices," in *12th Computer Vision Winter Workshop*, 2007.

[20] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*: Addison-Wesley Professional, 2005.

[21] A. Gooch, B. Gooch, P. Shirley, and E. Cohen, "A non-photorealistic lighting model for automatic technical illustration," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998, pp. 447-452.

[22] G. R. Bradski and A. Kaehler, *Learning opencv, 1st edition*: O'Reilly Media, Inc., 2008.