

Fuzzy Motion Detector based on 2D-Convolutions for Deinterlacing using Graphics Cards

A.S. Montemayor* ESCET-URJC 28933 Mostoles, Madrid SPAIN	F. Fernández† DTF-UPM 28860 Madrid SPAIN	J. Gutiérrez‡ DTF-UPM 28860 Madrid SPAIN	R. Cabido§ ESCET-URJC 28933 Madrid SPAIN	A. Sánchez¶ ESCET-URJC 28933 Madrid SPAIN
--	--	--	--	---

Abstract

In the last decade, consumer graphics cards have increased their computing power and capabilities mainly due to the computer games industry. These cards are programmable and capable of processing huge amounts of data in a Streaming Pipelined Architecture. In this situation, 2D convolutions are highly efficient and can be easily mapped on graphics hardware. In this work, we adapt a fuzzy motion detector to the hardware graphics computation framework in order to achieve a successful motion-adaptive video deinterlacing solution.

Keywords: Motion Detection, Deinterlacing, Graphics Processing Units, Real-Time Video Processing.

1 Introduction

Real-time video processing is a complex and demanding task that involves the use of high-tech systems. There is an increasing migration from analog to digital video because of the interest on smart technologies in many commercial, traffic, military, and law-enforcement applications [9, 4].

Now, multimedia market is including mobile and handheld visualization devices and in near future, those tools will demand more low-cost hardware solutions. These challenges lead to real-time video processing capabilities and an acceptable trade-off between system performance and involving costs. Many real-time video processing applications in the literature need from dedicated and expensive hardware.

On the other hand, multimedia and computer games industry have encouraged graphics hardware to improve their processing power to unprecedented limits. Their processing power should not be underestimated and many authors have demonstrated that these consumer Graphics Processing Units (GPU) have a huge raw performance, even superior to the most common and powerful CPUs [15, 8]. Also, these GPUs can be programmed to customize their rendering pipeline and thus generating personalized special effects.

Developers can also take advantage of these programmable capabilities even with applications far beyond rendering purposes. In this way, the GPU becomes a co-processor for the Central Processing Unit (CPU) with the aim that they can be encountered in most off-the-shelf desktop computers. Examples that demonstrate this fact are found in applications which exploit the power of the GPU for linear algebra calculations [11, 10], physically-based simulations [8], image processing [19, 2] or motion estimation and visualization [14] among many others [6].

Efficient video processing can be achieved

*antonio.sanz@urjc.es

†felipe.fernandez@es.bosch.com

‡jgr@dtf.fi.upm.es

§rcabido@gmail.com

¶angel.sanchez@urjc.es

using commodity graphics hardware as an alternative to specific high performance hardware. Moreover, this hardware is not only very affordable but also there are some reliable tools for making easy the programming task, such as commercial high level shading languages that tend to improve the abstraction layer between the hardware and the coding. NVIDIA's Cg ("C for Graphics"), Microsoft's HLSL for DirectX9.0 SDK and the OpenGL Shading Language are the most well-known shading languages.

On the other side, fuzzy logic has enhanced to manage uncertainty, and has been mainly applied to automatic control. It is based on Fuzzy Sets Theory where inputs do not simply belong or not to a set, but allows degrees of membership. In general terms a fuzzy controller is composed by three processing stages [3]: input fuzzification, fuzzy rule base evaluation and an output defuzzification stage. Fuzzy logic has been applied to a variety of fields, including Image and Video Processing, and Computer Vision. A detailed survey of applications of fuzzy techniques to these two fields can be found in [12].

In this work we explore a graphics hardware solution for a previous fuzzy video deinterlacing problem, described in [13]. These kind of applications have special interest in the graphics hardware community because commodity graphics cards usually include dedicated hardware for video deinterlacing to provide cinematic-quality and high-definition video playback. We take into account a fuzzy framework implemented by means of a composition of linear filters and fuzzy saturation functions, which have highly efficient computation on the GPU.

2 Video Deinterlacing

Nowadays, analog video coding standards are still based on the interlaced video scan format. Such approach was provided to reduce the required signal bandwidth transmission. In an interlaced scan format, video frames are split into odd and even line fields which are consecutively transferred in order to approximate

the whole frame. However, this process gives incomplete images and, in some devices, poor visual quality results.

Video deinterlacing is the necessary reconstruction task for obtaining progressive video from an interlaced format. Video deinterlacing methods are usually categorized in: Motion Compensated (MC) and Non-Motion Compensated (non-MC) algorithms. MC deinterlacing algorithms provide the highest reconstruction quality although they are computationally more expensive. They are based on a 2D velocity estimation and pixel shifting calculations.

On the other hand, non-MC techniques are cheaper and can achieve a good compromise between performance and quality. That is why many small visualization devices use them. An extensive review of deinterlacing technology is presented by Gerard de Haan in [1].

Simplest non-MC deinterlacing methods are based on time or space line replication. This way, missing lines of the actual field replicate those lines from the previous field (temporal replication, I_t) or from the known lines of the actual one (spatial replication, I_s).

Temporal or inter-field techniques are also named as weave methods, and spatial or intra-field techniques are also named as bob methods. Weave methods are quite effective in static scenes, while bob methods work better for dynamic ones.

Many spatio-temporal hybrid-deinterlacing techniques have been proposed to exploit the spatial and temporal correlation of video pictures and to overcome the artifacts associated with simple deinterlacers. The corresponding techniques called Motion-Adaptive (MA) algorithms, usually compute a motion-weighted combination of a temporal interpolation function $I_t(\cdot)$ and a spatial interpolation one $I_s(\cdot)$, according to the following equation:

$$I_{ts}(i, j, t) = \alpha I_s(i, j, t) + (1 - \alpha) I_t(i, j, t) \quad (1)$$

where $I_{ts}(i, j, t)$ is the obtained luminance or a RGB component on the column i , line j and time t of the corresponding field, and

$\alpha(i, j, t) \in [0,1]$ is the involved motion value per pixel. To compute this weighting parameter, most of these techniques are based on the computation of the absolute difference function $h(\cdot)$ between the luminance of two adjacent fields with the same parity:

$$h(i, j, k) = |I(i, j, t + 1) - I(i, j, t - 1)| \quad (2)$$

Unfortunately, due to several noise sources, the luminance difference does not become zero in all picture parts without motion. This implies that the corresponding motion detector should include some kind of additional spatio-temporal filtering in order to avoid some undesirable noise effects. This motion filter must be designed by taking into account the following two main assumptions: the noise level is usually small in comparison to the signal level, and the moving objects are large compared to the pixel size.

Thus, there is a need to balance the algorithm motion sensitivity with the ability to provide a good resolution. To accomplish this task, a fuzzy motion detector was developed by Van de Ville et al. [16, 17] based on a set of five fuzzy rules (FMD1). An improved version of this approach is shown in [7], where authors propose an alternative fuzzy motion detector (FMD2) that simplifies the corresponding computation and provides a good picture quality in both moving and still image areas. In fact, the core of the computation is based on fuzzy saturation functions and spatio-temporal filtering. This work was more version in [13] where a second saturation function is added and 2D FIR kernels are decomposed into 1D convolutions. Now, we implement the algorithm described in [13] into a graphics hardware platform.

The used saturation functions capture the nonlinearities of the corresponding fuzzy filter. The saturation function $sat_{x_1, x_2}(x)$ has been initially specified by the set of fuzzy rules:

$$\text{if } (x \text{ is } LOW) \text{ then } sat = 0 \quad (3)$$

$$\text{if } (x \text{ is } HIGH) \text{ then } sat = 1 \quad (4)$$

where the fuzzy labels *LOW* and *HIGH* belong to the corresponding trapezoidal fuzzy

partition defined by the coordinates $(x_{min}, x_1, x_2, x_{max})$. Parameters x_1 and x_2 simultaneously specify the threshold, gain and saturating regions of the corresponding variable. The equivalent fuzzy filter obtained preserves the interpretability property of the original system, and is easily understandable for a fuzzy or classical system designer.

3 Graphics Hardware

Commodity graphics hardware has drastically evolved since the mid 90's. With the aid of the rapid expansion of computer games and multimedia technologies these consumer GPUs have also become very powerful and inexpensive hardware. Nowadays, GPUs achieve more transistors than common CPUs, and more important, they have a number of simultaneous processing pipelines (from 4 to 24) to provide very high computational throughputs.

Traditionally, these 3D graphics cards implemented a fixed pipeline for the processing of primitive descriptions tuned as a state machine from an API such as OpenGL. But their previously fixed graphics pipeline stages were replaced with programmable components, the transform and lighting (T&L) and the multi-texturing ones, providing more versatility and power to the developer. The basic CPU/GPU architecture model is outlined in Fig. 1a.

The hardware accelerated programmability of GPUs has been exposed to programmers for the development of specialized programs called shaders. These shaders are loaded into the graphics card for replacing the fixed functionality. There are two kinds of shaders, respectively called vertex and fragment shaders. Originally, they had to be coded in assembler, but as the graphics hardware increased in functionality and programmability, these shaders were more difficult to implement. Even more, the rapid evolution of GPUs forced to rewrite previous shaders to get maximum performance when a new family of graphics hardware were released. As pointed out in the introduction, the solution came with the apparition of commercial high level shading lan-

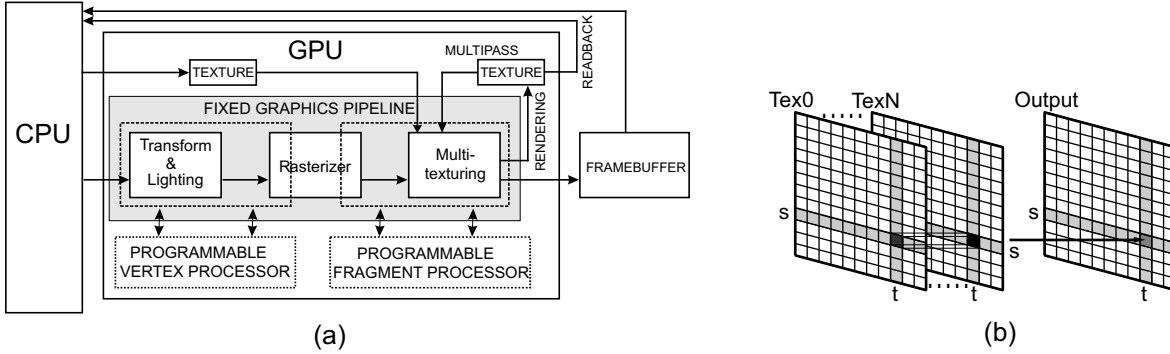


Fig. 1: (a) Basic CPU/GPU programming model. When enabled, programmable vertex and fragment execution paths replace their corresponding stages of the fixed graphics pipeline (represented in dot-lines). Also note the possibility of direct rendering to framebuffer or rendering to another texture (pbuffer), that can be used again as input data in the multipass approach. (b) Simple fragment program computation. A common fragment program will be executed over every position of the input textures ($Tex0-TexN$).

guages and their compilers, which helped in portability and legibility, thus improving the software development process. These shaders are primarily used for rendering complex special effects and realistic 3D scenes in real-time.

The programmability of the GPU at the fragment level is very well suited for stream computations. In its simplest form a kernel operation is executed over a large number of elements in a streaming single-instruction multiple-data (SIMD) fashion [18, 11]. In this sense, emerges the idea of a GPU as a stream computer where the spatial parallelism occurs at the level of a fragment. In the context of computer graphics, a texture is an image that can be mapped to a polygonal structure to provide realism to the model. Basically, as an image, it can represent four values (R, G, B, A) as color and transparency components in every accessible location, called fragment or texel (texture element).

Programmers are responsible for organizing their data in a grid to convert them into textures, so creating textures in which texels keep numerical values of interest. In order to achieve maximum performance it is desirable to fill the RGBA channels of those textures. The main reason for this behavior is because, in the fragment program, the processing cost

of a single channel in comparison to the processing cost of the entire quadruple (RGBA) is quite similar.

Textures are fixed to a well determined grid with the aim to operate on their texels. Then, a custom fragment shader is enabled and the operation kernel is executed over every fragment by simply rendering. An schematic view of this process is shown in Fig. 1b. Note, that the output result can be redirected to the input in a multipass approach for continuing the processing task. At this point, it is important to remark that data readback from video memory to host memory is a well-known computational bottleneck. This bottleneck has been reduced with the apparition of new ports like PCI-Express (PCIe) x16.

4 Hardware-Accelerated Video Deinterlacing

Using the high computational throughput of the GPU and the optimized model because of the use of linear convolutions, this kind of application is very efficient to be computed on the GPU. Moreover, and as stated in the introduction, GPUs usually integrate spatio-temporal video deinterlacing hardcoded algorithms and they will become the core of many

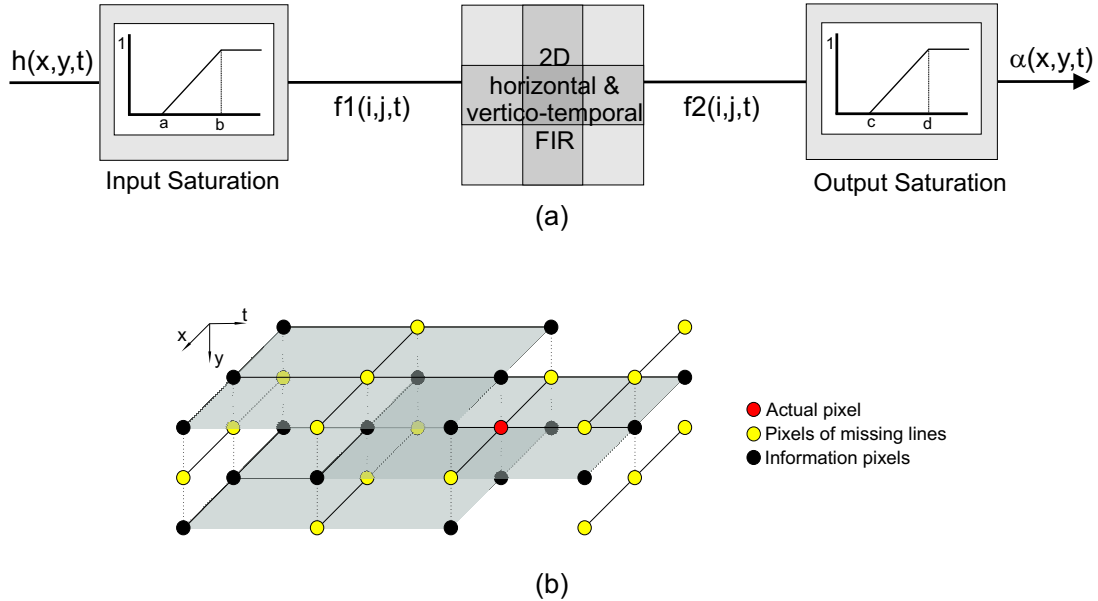


Fig. 2: a) General scheme of the proposed Fuzzy Motion Detector (FMD) for the motion adaptive deinterlacer. b) Spatial Data Dependence Graph (DDG) of the proposed FMD.

low-cost deinterlacers.

The considered approach taken in this work is an hybrid architecture between Gutierrez et al. [7] and Sanz et al. [13] previous works. In the first one, authors used 2D convolutions to accelerate the fuzzy rule evaluation. In the second one, authors integrated a second saturation function and decomposed those 2D FIR kernels into 1D convolutions enabling real-time processing in a desktop PC. Now, the hardware accelerated video deinterlacing is mapped into the GPU, where 2D computations are more efficient in terms of number of rendering passes. In addition, this approach adds the second saturation function to provide more versatility as proposed in [13].

The proposed fuzzy motion detector is based on the fuzzy motion-adaptive deinterlacing methods explained in Section 2. Figure 2a shows the schematic diagram of the proposed FMD and Fig. 2b, its spatial data dependence graph. The proposed fuzzy motion detector operates on four consecutive fields, which are transferred to video memory as four different textures ($\text{Tex0}.. \text{Tex3}$). First of all, the absolute differences ($h(t)$ and $h(t-1)$) be-

tween non-consecutive fields are computed as expressed by Eq. 2. Then, an input saturation function $sat_{a,b}(\cdot)$, a 2D smoothing kernel operation and an output saturation function $sat_{c,d}(\cdot)$ are successfully applied. The resulting factor is defined as $\alpha(i,j,t) \in [0, 1]$ which evaluates the spatial or temporal contribution per pixel to I_{ts} as described in Eq. 1. Therefore, the value of I_{ts} needs from the calculation of I_t and I_s in different fragment shaders.

The final task is to achieve a motion matrix for the missing lines of the actual field. This motion factor is the linear combination weight between the temporal interpolation reconstruction I_t and the spatial one I_s , following Eq. 1. This is computed by saturating the input, spreading the signal by a 2D convolution and saturating the output. These process is tuned by saturation parameters (a, b, c, d), which, in general, depend on the dynamism of the video sequence.

5 Experimental Results

Experiments have been performed using four graphics cards, in a 2.8 GHz Pentium 4 host

processor, 512 MB RAM, AGPx8 and a 3.2 GHz Pentium 4, 1GB RAM, PCI-Express x16, under Windows XP Professional SP2. For the AGP-PC we used a 2004 Nvidia GeForce6800 Ultra (NV45) and GeForce6800 GT (NV40), and a 2003 low performance Nvidia FX 5200 (NV34). For the PCIe x16 platform we used a 2005 Nvidia GeForce7800 GTX (G70). Applications were coded in C using OpenGL as rendering API, Cg 1.3 as shading language and Nvidia v71.84 drivers. We simulated an interlaced scan format from a progressive video taking only into account proper fields.

Figure 3 shows different stages of the processing for a dynamic (left) and a static (right column) scene for the Salesman test video sequence (QCIF format, 176x144). For this video resolution, we get 76 fps, 95 fps and 210 fps under the NV40, NV45 and G70 processors, respectively. Moreover, processing rate is only reduced to 65 fps, 88 fps and 198 fps for a 640x480 VGA video resolutions, which is much higher than a previous CPU solution (around 30 fps for a QCIF format in a 1.4 GHz Pentium 4, 128 MB RAM [13]). Qualitative results are provided in [13], as far as the algorithm proposed here is a direct implementation of the previous one under a new platform.

The huge performance of this application is mainly based on the fact that there is no readback from video to host memory and, once textures are uploaded, all the processing tasks are computed in the GPU. However, the frequent branching inside the needed fragment programs is virtually forcing the use of a modern graphics card. We have reported only 3 fps for the same 176x144 (QCIF) videos using a Nvidia GeForceFX5200 (NV34) with less flexibility in the conditional executions. Table 1 resumes these results.

6 Conclusions

Real-time video processing is a very demanding computational task. In this work we propose a fuzzy motion-adaptive video deinterlacing implementation on a common program-

Table 1: Experimental results. Performance measured in number of frames per second (fps) for two different video resolutions.

GPU	Year	QCIF	VGA
		176x144	640x480
NV34	2003	3 fps	-
NV40	2004	76 fps	65 fps
NV45	2004	95 fps	88 fps
G70	2005	210 fps	198 fps

mable graphics hardware architecture. In particular, high performance gains can be achieved for this kind of video processing activities mainly because deinterlacing execution is kept on video memory once data are uploaded. A major drawback is the conditional branching in the fragment program because of the simulation of an interlaced scan format from progressive video sequences. This effect can be reduced in a real application, where only known lines from each video field are provided as input.

Experimental results have shown a remarkable performance, and moreover, a very high throughput while using more recent hardware (more than a performance doubling from 2004 to 2005 GPU). As far as authors know, this is the first attempt to implement a fuzzy video deinterlacing algorithm on the Graphics Processing Unit (GPU).

Modern graphics mobile devices open a new applicability dimension of low-cost and high performance computing architectures. As a result, these kind of video processing can be pushed in a near future as a consumer solution.

References

- [1] E. B. Bellers, G. de Haan (2000). Deinterlacing. A Key Technology for Scan Rate Conversion. Elsevier, 2000.
- [2] P. Colantoni, N. Boukala, J. da Rugna (2003): Fast and Accurate Color Image Processing Using 3D Graphics Cards, In *Proc. of 8th Int. Workshop on Vision*,

- Modeling and Visualization, Germany, 2003.*
- [3] E. Cox (1992). Fuzzy Fundamentals, *IEEE Spectrum*, vol. 29, number 10, pages 58-61, 1992.
 - [4] S. L. Dockstader, M. Tekalp (2001). On the Tracking of Articulated and Occluded Video Object Motion, *Real-Time Imaging*, vol. 7, pages 415-432, 2001.
 - [5] N. Goodnight, R. Wang, C. Woolley, G. Humphreys (2003). Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware, *Eurographics Symposium on Rendering*, pages 1-13, 2003.
 - [6] GPGPU (General Purpose computing using GPUs) Website, <http://www.gpgpu.org>
 - [7] J. Gutiérrez-Ríos, F. Fernández-Hernández, J. C. Crespo, G. Triviño (2004). Motion Adaptive Fuzzy Video De-interlacing Method Based on Convolution Techniques. *Proc. of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU)*, vol. 3, pages 1635-1642, Perugia, Italy, 2004.
 - [8] M. J. Harris (2003). Real-Time Cloud Simulation and Rendering, Ph. D Thesis, Univ. of North Carolina at Chapel Hill, 2003.
 - [9] A. Iketani, A. Nagai, Y. Kuno, Y. Shirai (2001). Real-Time Surveillance System Detecting Persons in Complex Scenes, *Real-Time Imaging*, vol. 7, pages 433-446, 2001.
 - [10] J. Kruger, R. Westermann (2003). Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Trans. on Graphics* pages 908-916, 2003.
 - [11] M. McCool, S. Du Toit, T. Popa, B. Chan, K. Moule (2004). Shader Algebra, *ACM Transactions on Graphics*, 2004.
 - [12] M. Nachtgaele, D. Van der Weken, D. Van de Ville, E. E. Kerre (Eds) (2003). Fuzzy Filters for Image Processing. Springer, 2003.
 - [13] A. Sanz, F. Fernández-Hernández, J. Gutiérrez, G. Triviño, A. Sánchez, J.C. Crespo, A. Mazadiego (2004). Motion Adaptive Video Deinterlacing Using One Dimensional Fuzzy FIR Filters. In *Proc. of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU)*, vol. 3, pages 1643-1650, Perugia, Italy, 2004.
 - [14] R. Strzodka, C. Garbe (2004). Real-Time Estimation and Visualization on Graphics Cards. In *Proc. IEEE Visualization*, pages 545-552, 2004.
 - [15] C. J. Thompson, S. Hahn, M. Oskin (2002). Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, In *Int. Symposium on Microarchitecture (MICRO)*, 2002.
 - [16] D. Van de Ville, W. Philips, I. Lemahieu (2000). Fuzzy-Based Motion Detection and its Applications to Deinterlacing, In *Fuzzy Techniques in Image Processing*, E. E. Kerre and M. N. Nachtgaele (Eds), Chap 13, pages 337-369, Physica-Verlag, 2000.
 - [17] D. Van de Ville, R. Van de Walle, W. Philips, I. Lemahieu (2002). Motion Adaptive De-interlacing using Fuzzy Logic, In *Proc. of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU)*, 2002.
 - [18] S. Venkatasubramanian (2003). The Graphics Card as a Stream Computer, *Workshop on Management and Processing of Data Streams*, San Diego, USA, 2003.
 - [19] R. Yang, G. Welch (2002): Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware, *Journal of Graphics Tools*, vol. 7, number 4, pages 91-100, 2002.



(a)



(b)



(c)



(d)

Figure 3: Deinterlacing quality comparison for both, dynamic (left) and static (right) scenes. a) Weave and b) Bob deinterlacing methods, c) $\alpha(i, j, t)$ before the output saturation function and d) proposed FMD video deinterlacer result with saturation parameters ($a=5$, $b=10$, $c=50$, $d=80$). Note that video deinterlacer based on the fuzzy motion detector results a trade-off between static and dynamic scenes.