Fuzzy Motion-Adaptive Video Deinterlacing using Consumer Graphics Hardware

Antonio S. Montemayor[1], Felipe Fernández[2], Julio Gutiérrez[2], Raúl Cabido[1†] and Ángel Sánchez[1]

[1] Universidad Rey Juan Carlos, C/Tulipán, S/N,
28933 Móstoles, Madrid, Spain
{antonio.sanz, angel.sanchez}@urjc.es
[†] rcabido@gmail.com
[2] Universidad Politécnica de Madrid,
Campus de Montegancedo, 28860 Madrid, Spain
felipe.fernandez@es.bosch.com
jgr@fi.upm.es

Abstract. In the last decade, consumer graphics cards have increased their power due to the computer games industry. These cards are now programmable and capable of processing huge amounts of data in a Streaming Pipelined Architecture. In this work, we adapt a fuzzy motion-adaptive video deinterlacing solution to the hardware graphics computation framework. Experimental results show a remarkable performance. As far as authors know this is the first attempt to implement a fuzzy video deinterlacing algorithm on the Graphics Processing Unit (GPU).

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 [1, 2].

But also multimedia market has grown now including mobile and small visualization devices. In near future, those tools will demand more low-cost hardware solutions. These challenges lead to real-time video processing capabilities and 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 unprecedent limits. Their processing power should not be underestimated and many authors have demonstrated that these consumer Graphics Processing Units (GPU) have a great raw performance, even superior to the most common and powerful CPUs [13–16]. Also, these GPUs can be programmed to customize their rendering pipeline and thus generating personalized special effects.

Besides, developers can take advantage of these programmable capabilities even with applications far beyond rendering purposes. In this way, the GPU becomes as a co-processor for the central processing unit (CPU) remaining the idea that they can be encountered in most off-the-shelf desktop computers. Examples that demonstrate this fact are found in applications that exploits the power of the GPU for linear algebra calculations [19–22], physically-based simulations [16], image and volume processing [14, 23–25, 31], neural network implementations [26, 27], motion estimation and visualization [31] or even acceleration of database operations [28] among others [29].

Efficient video processing can be achieved 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. A brief classification, chronology and explanation of them can be found in [30].

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 fields can be found in [4, 5].

In this work we explore a graphics hardware application to a fuzzy video deinterlacing problem. This 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 graphics processing unit.

The rest of the paper is organized as follows. In the next section a description of video deinterlacing techniques can be found. Section 3 offers a background about commodity graphics hardware and its basic architecture. Next, Section 4 provides a brief explanation of the implemented motion-adaptive video deinterlacer. Experimental results can be found in Section 5, and finally, Section 6 offers the conclusions of this work.

2 Video Deinterlacing

Nowadays, analog video coding standards are still based on the interlaced video scan format. Such approach was found to reduce the required signal bandwidth transmission. In an interlaced scan format, video frames are split into odd and even line fields which are transferred consecutively 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 is 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 [6, 7].

Simplest non-MC deinterlacing methods are based on time or space line replication. This way, the missing lines of the actual field replicate the 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 weave methods, and spatial or intra-field techniques are also named 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, compute a motion-weighted combination of a temporal interpolation function $I_t(.)$ and a spatial interpolation one $I_s(.)$:

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

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 $h(.)$ between the luminance of two adjacent fields with the same parity:

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

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 with the pixels size.

Thus, there is a need to balance the algorithm's motion sensitivity with the ability to provide a good resolution. To accomplish this task, a fuzzy motion detector was developed by Van de Ville [8, 9] based on a set of 5 fuzzy rules (FMD1). An improved version of this approach is shown in [10]. There, authors

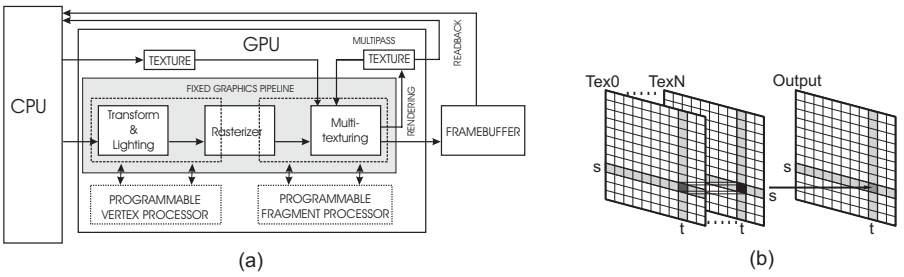

Fig. 1. (a) Basic CPU/GPU programming model. When enabled, programmable vertex and fragment execution paths replace their corresponding stages of the fixed graphics pipeline (represented in dot-lines). Also note the possibility of direct rendering to framebuffer or rendering to another texture (pbuffer), 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 ($Tex0$-$TexN$), for example returning a resulting value at (s,t) of the output texture for values at (s,t) of the input ones.

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. Moreover, this work was more developed in [11] adding a second saturation function and decomposing 2D FIR kernels into 1D convolutions.

The saturation functions used capture the nonlinearities of the corresponding fuzzy filter. The saturation function $sat_{x1,x2}(x)$ has been initially specified by the set of fuzzy rules:

$$if(x\ is\ LOW)\ then\ sat = 0;\ if\ (x\ is\ HIGH)\ then\ sat = 1 \tag{3}$$

where the fuzzy labels LOW and $HIGH$ belong to the corresponding trapezoidal type-1 fuzzy partition [12] 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 evolved drastically 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.

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 one, providing great versatility and power to the developer [17]. 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 languages and their compilers, which helped in portability and legibility, thus improving the 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, 19].

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 accesible location, called fragments or texels.

The programmer is responsible for organizing the data in a grid to convert them into a texture, 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 the textures. This 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.

The output result can be redirected to the input (by means of a pbuffer) in a multi-pass 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.

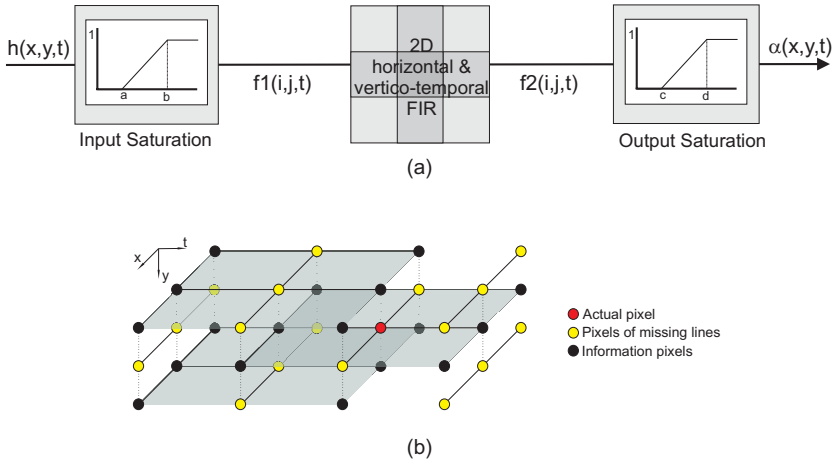

Fig. 2. a) General scheme of the proposed Fuzzy Motion Detector (FMD) for the motion adaptive deinterlacer. b) Spatial Data Dependence Graph (DDG) of the proposed FMD.

4 Hardware-Accelerated Video Deinterlacing

By means of the high computational throughput of the graphics card and the optimized model because of the use of linear convolutions, this kind of application is highly efficient to be computed on the GPU. Moreover, and as it is stated in the introduction, GPUs usually integrate spatial-temporal video deinterlacing hardcoded algorithms and they will become the core of many low-cost deinterlacers.

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 4 consecutive fields, which are transferred to video memory as 4 different textures (`Tex0..Tex3`). First it computes absolute differences (h(t) and h(t-1)) between non-consecutive fields as expressed by Eq. 2. Then it applies an input saturation function $sat_{a,b}(.)$, a 2D smoothing kernel operation and an output saturation function $sat_{c,d}(.)$. The resulting factor is defined as α(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.

Algorithm 1 shows the pseudocode of the proposed method. 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. It is calculated by satu-

rating the input, spreading the signal by a 2D convolution (for example the one written in the pseudocode) and saturating the output. These process is tuned by saturation parameters (a, b, c, d), which in general they depend on the dynamism of the video sequence.

Algorithm 1 System of Recurrence Equations of the Proposed FMD

for $t = 1$ to K **do**
  for $i = 1$ to N **do**
    for $j = 1$ to M **do**
      if (j and t are odd) or (j and t are even) **then**
        $h(i,j,t) = |I(i,j,t-1) - I(i,j,t+1)|$ {Motion input}
        $f_1(i,j,t) = sat_{a,b}(h(i,j,t))$ {Input saturation}
        {2D FIR low-pass filter: H and V-T}
        $f_2(i,j,t) = 1/8(f_1(i-1,j,t) + f_1(i,j-1,t-1) + 4f_1(i,j,t) + f_1(i+1,j,t) + f_1(i,j+1,t-1))$
        $\alpha(i,j,t) = sat_{c,d}(f_2(i,j,t))$ {Output saturation 0-1}
        {Output Luminance by S-T interpolation}
        $I_{ts}(i,j,t) = \alpha(i,j,t)I_s(i,j,t) + (1 - \alpha(i,j,t))I_t(i,j,t)$
      end if
    end for
  end for
end for
Function sat() $\{sat(.) : R \Rightarrow [0,1]\}$
$sat_{x1,x2}(x) = (x < x1) \Rightarrow 0; (x > x2) \Rightarrow 1; (x - x1)/(x2 - x1)$
End Function

5 Experimental Results

Experiments have been performed using the GPU Nvidia GeForce6800 Ultra (NV45) in a 2.8GHz Pentium 4 host processor, 512MB RAM, AGPx8, under Windows XP Professional SP2. The applications have been coded in C using OpenGL as rendering API, Cg 1.3 as shading language and Nvidia v71.84 drivers. The real-time video capture is done using a simple webcam and DsVideoLib project [32] which exposes, through DirectShow (DirectX9.0b), captured video frames as OpenGL textures in a synchronized way. We have simulated an interlaced scan format from a progressive video taking into account only 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 kind of video resolutions we get 95 fps. Moreover, processing rate is only reduced to 88 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 [11]).

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 stay 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.

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 programmable 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 fact can be made lighter in a real case, in which only known lines from each field are provided as input.

Also, 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. Iketani, A., Nagai, A., Kuno, Y., Shirai, Y.: Real-Time Surveillance System Detecting Persons in Complex Scenes, *Real-Time Imaging*, **7**: 433-446, (2001).
2. Dockstader, S.L., Tekalp, M.: On the Tracking of Articulated and Occluded Video Object Motion, *Real-Time Imaging*, **7**: 415-432, (2001).
3. Cox, E.: Fuzzy Fundamentals, *IEEE Spectrum*, **29(10)**: 58–61, (1992).
4. Nachtegael, M., Kerre, E. E. (Eds).: Fuzzy Techniques in Image Processing, Springer-Verlag, (2000).
5. Nachtegael, M., Van der Weken, D., Van de Ville, D., Kerre, E. E. (Eds).: Fuzzy Filters for Image Processing, Springer, (2003).
6. de Haan, G., Bellers, E. B.: De-interlacing. An Overview, *Proc. of the IEEE*, **86(9)**:1839–1857, (1998).
7. Bellers, E. B., de Haan, G.: Deinterlacing. A Key Technology for Scan Rate Conversion. Elsevier (2000).
8. Van de Ville, D., Philips, W., Lemahieu, I.: Fuzzy-Based Motion Detection and its Applications to Deinterlacing, *In Fuzzy Techniques in Image Processing*, E. E. Kerre and M. N. Nachtegael (Eds), Chap 13, pp. 337–369, Physica-Verlag (2000).
9. Van de Ville, D., Van de Walle, R., Philips, W., Lemahieu, I.: Motion Adaptive De-interlacing using Fuzzy Logic, *Proc. of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU), 2002.*
10. Gutiérrez-Ríos, J., Fernández-Hernández, F., Crespo, J. C., Triviño, G.: Motion Adaptive Fuzzy Video De-interlacing Method Based on Convolution Techniques. *Proc. of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU), 2004.*

11. Sanz A., Fernández-Hernández, F., Gutiérrez, J., Triviño, G., Sánchez, A., Crespo, J. C., Mazadiego, A.: Motion Adaptive Video Deinterlacing Using One Dimensional Fuzzy FIR Filters. *Proc. of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU), 2004.*
12. Klir, G. J., Yuang, B.: Fuzzy Sets and Fuzzy Logic. Prentice Hall (1995).
13. Thompson, C.J., Hahn, S., Oskin, M.: Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, *Int. Symposium on Microarchitecture (MICRO), 2002.*
14. Goodnight, N., Wang, R, Woolley, C., Humphreys, G.; Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware, *Eurographics Symposium on Rendering* , (2003) 1–13.
15. Purcell, T.: Ray Tracing on a Stream Processor, Ph. D Thesis, Univ. of Stanford (2004).
16. Harris, M. J.: Real-Time Cloud Simulation and Rendering, Ph. D Thesis, Univ. of North Carolina at Chapel Hill (2003).
17. Olano, M.: A Programmable Pipeline for Graphics Hardware. Ph.D. thesis, University of North Carolina at Chapel Hill (1998).
18. Venkatasubramanian, S.: The Graphics Card as a StreamComputer, *Workshop on Management and Processing of Data Streams, San Diego, California, USA* (2003).
19. McCool, M., Du Toit, S., Popa, T., Chan, B., Moule, K.: Shader Algebra, *ACM Transactions on Graphics* (2004).
20. Larsen, E. S., McAllister, D.: Fast Matrix Multiplies using Graphics Hardware, *In Proc. Supercomputing 2001.*
21. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers of the GPU: Conjugate gradients and multigrid, *ACM Trans. on Graphics*, (2003) 917–924.
22. Kruger J., Westermann R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. on Graphics* (2003) 908-916.
23. Yang, R., Welch, G.: Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware, *Journal of Graphics Tools*, (2002) **7(4)**:91–100.
24. Colantoni, P., Boukala, N., da Rugna, J.: Fast and Accurate Color Image Processing Using 3D Graphics Cards, *Proc. of 8th Int. Workshop on Vision, Modeling and Visualization, Germany* (2003).
25. Krueger, J., Westermann, R.: Acceleration Techniques for GPU-based Volume Rendering. *In Proc. IEEE Visualization 2003.*
26. Bohn, C.A.: Kohonen Feature Mapping Through Graphics Hardware. *In Proc. of 3rd Int. Conference on Computational Intelligence and Neurosciences 1998.*
27. Oh K.-S. and Jung K.: GPU implementation of neural networks, *Pattern Recognition*, (2004) **37**: 1311–1314.
28. Govindaraju, N.K., Lloyd, B., Wang, W., Lin M.C., Manocha, D.: Fast Computation of Database Operations using Graphics Processors, *In Proc. SIGMOD 2004, Paris, France* (2004).
29. GPGPU Website, http://www.gpgpu.org
30. Rost, R.J.: OpenGL Shading Language. Pearson Education (2004).
31. Strzodka, R., Garbe, C.: Real-Time Estimation and Visualization on Graphics Cards, *In Proc. IEEE Visualization 2004* : 545–552.
32. Pintaric T.: DsVideoLib: DirectShow Video Processing Library, http://sourceforge.net/projects/dsvideolib

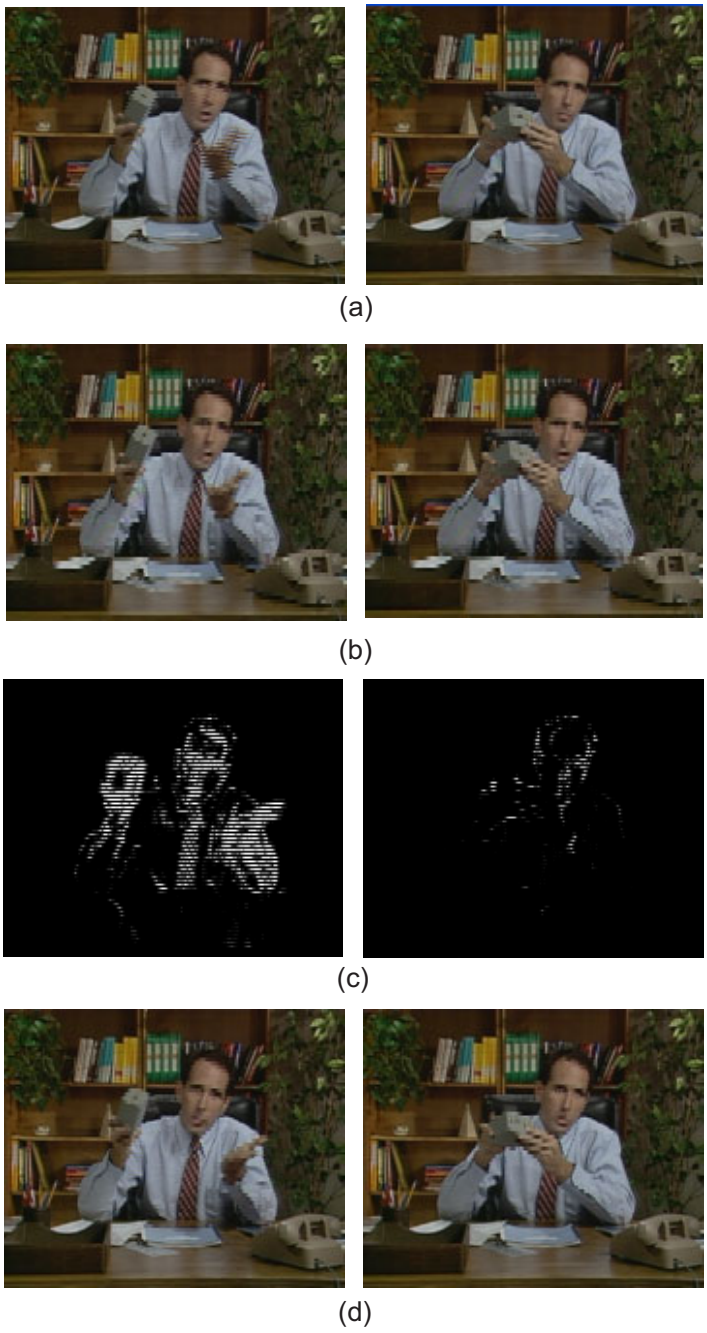

Fig. 3. Deinterlacing quality comparison for a dynamic (left) and static (right) scenes. a) Weave and b) Bob deinterlacing methods, c) $\alpha(i, j, t)$ before the output saturation function (called $f2(i, j, t)$ in Algorithm 1) and d) proposed FMD video deinterlacer result with saturation parameters (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.