(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
19 June 2008 (19.06.2008)

PCT

(10) International Publication Number
WO 2008/073416 A1

(51) International Patent Classification:
G06T 1/00 (2006.01) *G06F 15/76* (2006.01)

(21) International Application Number:
PCT/US2007/025305

(22) International Filing Date:
11 December 2007 (11.12.2007)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/869,516 11 December 2006 (11.12.2006) US
60/912,093 16 April 2007 (16.04.2007) US

(71) Applicant *(for all designated States except US)*: **CINNAFILM, INC.** [US/US]; 11204 Paris Avenue NE, Albuquerque, NM 87111 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **MAURER, Lance** [US/US]; 11024 Paris Avenue NE, Albuquerque, NM 87111 (US). **GORMAN, Chris** [US/US]; 11204 Paris Avenue NE, Albuquerque, NM 87111 (US). **SHARLET, Dillon** [US/US]; 11204 Paris Avenue NE, Albuquerque, NM 87111 (US).

(74) Agent: **MYERS, Jeffrey, D.**; Peacock Myers, P.C., P.O. Box 26927, Albuquerque, NM 87125-6927 (US).

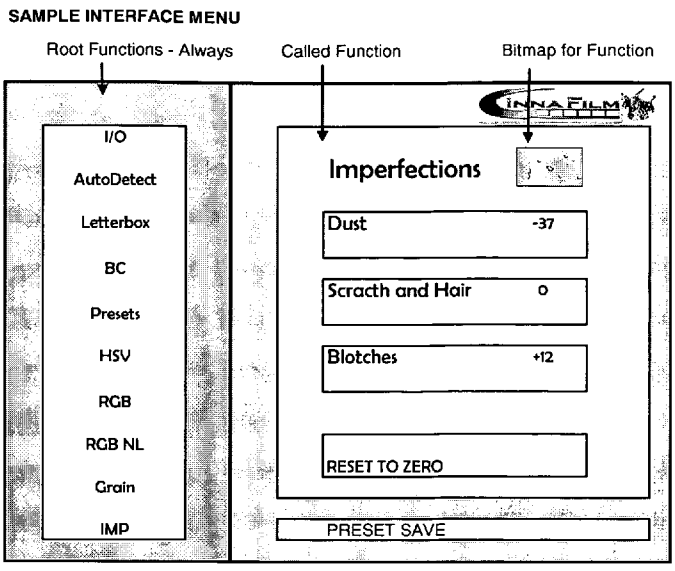
(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:
— *of inventorship (Rule 4.17(iv))*

Published:
— *with international search report*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

(54) **Title:** REAL-TIME FILM EFFECTS PROCESSING FOR DIGITAL VIDEO



(57) **Abstract:** A method, apparatus, and computer software for applying in real time imperfections to streaming video which causes the resulting digital video to resemble cinema film.

WO 2008/073416 A1

REAL-TIME FILM EFFECTS PROCESSING FOR DIGITAL VIDEO

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to and the benefit of the filing of U.S. Provisional Patent Application Serial No. 60/869,516, entitled "Cinnafilm: A Real-Time Film Effects Processing Solution for Digital Video", filed on December 11, 2006, and of U.S. Provisional Patent Application Serial No. 60/912,093, entitled "Advanced Deinterlacing and Framerate Re-Sampling Using True Motion Estimation Vector Fields", filed on April 16, 2007, and the specifications thereof are incorporated herein by reference.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0002] Not Applicable.

INCORPORATION BY REFERENCE OF MATERIAL SUBMITTED ON A COMPACT DISC

[0003] Not Applicable.

COPYRIGHTED MATERIAL

[0004]

BACKGROUND OF THE INVENTION

Field of the Invention (Technical Field):

[0005] The present invention relates to methods, apparatuses, and software for simulating film effects in digital images.

Description of Related Art:

[0006] Note that the following discussion refers to a publication, and that due to recent publication date it is not to be considered as prior art vis-a-vis the present invention. Discussion of such publication herein is given for more complete background and is not to be construed as an admission that such publication is prior art for patentability determination purposes.

[0007] The need and desire to make video look more like film is a considerable challenge due to high transfer costs and limitations of available technologies that are not only time consuming, but provide poor results.

[0008] U.S. Patent Application Serial No. 11/088,605, to Long et al. describes a system which modifies images contained on scan-only film to resemble that of an image captured on motion-picture film. This system, however, is limited to use in conjunction with special scan-only film and is not suitable for use in the now more-common digital images. Further, because the process of Long et al., is limited to scan-only film, the process of Long et al., cannot be used for streaming real-time or near real-time images. There is thus a present need for a method, apparatus, and system which can provide real-time or near real-time streaming digital video processing which alters the digital image to resemble images captured via motion picture film.

[0009] The present invention has approached the problem in unique ways, resulting in the creation of a method, apparatus, and software that not only changes the appearance of digital video footage to look like celluloid film, but performs this operation in real-time or near real-time. The invention (occasionally referred to as Cinnafilm™) streamlines current production processes for professional producers, editors, and filmmakers who use digital video to create their media projects. The invention permits independent filmmakers to add an affordable high quality film effect to their digital projects, provides a stand-alone film effects hardware platform capable of handling broadcast-level video signal, a technology currently unavailable in the digital media industry. The invention provides an instant film-look to digital video, eliminating the need for long rendering times associated with current technologies.

BRIEF SUMMARY OF THE INVENTION

[0010] Embodiments of the present invention relate to a digital video processing method, apparatus, and software stored on a computer-readable medium having and/or implementing the steps of receiving a digital video stream comprising a plurality of frames, adding a plurality of film effects to the video stream, outputting the video stream with the added film effects, and wherein for each frame the outputting occurs within less than approximately one second. The adding can

include adding at least two effects including but not limited to letterboxing, simulating film grain, adding imperfections simulating dust, fiber, hair, scratches, making simultaneous adjustments to hue, saturation, brightness, and contrast and simulating film saturation curves. The adding can also optionally include simulating film saturation curves via a non-linear color curve; simulating film grain by generating a plurality of film grain textures via a procedural noise function and by employing random transformations on the generated textures; adding imperfections generated from a texture atlas and softened to create ringing around edges; and/or adding imperfections simulating scratches via use of a start time, life time, and an equation controlling a path the scratch takes over subsequent frames. In one embodiment, the invention can employ a stream programming model and parallel processors to allow the adding for each frame to occur in a single pass through the parallel processors. Embodiments of the present invention can optionally include converting the digital video stream from 60 interlaced format to a deinterlaced format by loading odd and even fields from successive frames, blending using a linear interpolation factor, and, if necessary, offset sampling by a predetermined time to avoid stutter artifacts.

[0011] Objects, advantages and novel features, and further scope of applicability of the present invention will be set forth in part in the detailed description to follow, taken in conjunction with the accompanying drawings, and in part will become apparent to those skilled in the art upon examination of the following, or may be learned by practice of the invention. The objects and advantages of the invention may be realized and attained by means of the instrumentalities and combinations particularly pointed out in the appended claims.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

[0012] The accompanying drawings, which are incorporated into and form a part of the specification, illustrate one or more embodiments of the present invention and, together with the description, serve to explain the principles of the invention. The drawings are only for the purpose of illustrating one or more preferred embodiments of the invention and are not to be construed as limiting the invention. In the drawings:

[0013] Fig. 1 illustrates a preferred interface menu according to an embodiment of the invention;

[0014] Fig. 2 illustrates a preferred graphical user interface according to an embodiment of the invention;

[0015] Fig. 3 is a block diagram of a preferred apparatus according to an embodiment of the invention;

[0016] Fig. 4 is a block diagram of the preferred video processing module of an embodiment of the invention;

[0017] Fig. 5 is a block diagram of the preferred letterbox mask, deinterlacing and cadence resampling module of an embodiment of the invention; and

[0018] Fig. 6 is an illustrative texture atlas according to an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0019] Embodiments of the present invention relates to a methods, apparatuses, and software to enhance moving, digital video images at the coded level to appear like celluloid film in real time (processing speed equal to or greater than ~30 frames per second). Accordingly, with the invention processed digital video can be viewed "live" as the source digital video is fed in. So, for example, the invention is useful with video "streamed" from the Internet. The "film effects", added by an embodiment of the invention, include one and more preferably at least two of: letterboxing, adding film grain, adding imperfections simulating dust, fiber, hair, chemical burns, scratches, and the like, making simultaneous adjustments to hue, saturation, brightness, and contrast, and simulating film saturation curves.

[0020] Although the invention can be implemented on a variety of computer hardware / software platforms, including software stored in a computer-readable medium, one embodiment of hardware according to the invention is a stand-alone device, which is next described. Internal Video Processing Hardware preferably comprises a general purpose CPU (Pentium4®, Core2 Duo®, Core2 Quad® class), graphics card (DX9 PS3.0 or better capable), system board (with dual 1394/Firewire ports, USB ports, serial ports, SATA ports), system memory, power supply, and hard drive. A Front Panel User Interface preferably comprises a touchpad usable menu for access to image-modification features of the invention, along with three dials to assist in the fine tuning of the input levels. The touchscreen is most preferably an EZLCD 5" diagonal touchpad or equivalent, but of course virtually any touchscreen can be provided and will provide desirable results. With a touchscreen, the user can access at least some features and more preferably the entire set of features at any time, and can adjust subsets of those features in one or more of the following ways: (1) ON / OFF – adjusted with an on/off function on the touchpad; (2) Floating Point Adjustment (-100 to 100, 0 being no effect for example) – adjusted using the three dials; and/or (3) Direct Input– adjusted with a selection function on the touchpad. Fig. 1 illustrates a display provided by the preferred user interface.

[0021] The invention can also or alternatively be implemented with a panel display and user keyboard and/or mouse. The user interface illustrated in Fig. 2 allows quicker access to the multitude of features, including the ability to display to multiple monitors and the ability to manipulate high-definition movie files.

[0022] The apparatus of the invention is preferably built into a sturdy, thermally proficient mechanical chassis, and conforms to common industry rack-mount standards. The apparatus preferably has two sturdy handles for ease of installation. I/O ports are preferably located in the front of the device on opposite ends. Power on/off is preferably located in the front of the device, in addition to all user interfaces and removable storage devices (e.g., DVD drives, CD-ROM drives, USB inputs, Firewire inputs, and the like). The power cord preferably extrudes from the unit in the rear. An Ethernet port is preferably located anywhere on the box for convenience, but hidden using a removable panel. The box is preferably anodized black wherever possible, and constructed in such a manner as to cool itself via convection only. The apparatus of the invention is preferably locked down and secured to prevent tampering.

[0023] As illustrated in Fig. 3, an apparatus according to a non-limiting embodiment of the invention takes in a digital video/audio stream on a 1394 port and uses a Digital Video (DV) compression-decompression software module (CODEC) to decompress video frames and the audio buffers to separate paths (channels). The video is preferably decompressed to a two dimensional (2D) array of red, green, and blue color components (RGB image, 8-bits per component). Due to texture resource alignment requirements for some graphics cards, the RGB image is optionally converted to a red, green, blue, and alpha component (RGBA, 8-bits per component) buffer. The RGBA buffer is most preferably copied to the end of the input queue on the graphics card. The buffer is copied using direct memory access (DMA) hardware so that minimal CPU resources are used. On the graphics card, a video frame is preferably pulled from the front of the input queue and the video processing algorithms running on one or more processors, which can include hundreds of processors (128 in one implementation) to modify the RGBA data to achieve the film look. The processed frame is put on the end of the output queue. The processed video from the front of the output queue is then DMA'd back to system memory where it is compressed, along with the audio, using the software CODEC module. Finally, the compressed audio and video are then streamed back out to a second 1394 port to any compatible DV device.

[0024] Although other computer platforms can be used, one embodiment of the present invention preferably utilizes commodity x86 platform hardware, high end graphics hardware, and highly pipelined, buffered, and optimized software to achieve the process in realtime (or near realtime with advanced processing). This configuration is highly reconfigurable, can rapidly adopt new video standards, and leverages the rapid advances occurring in the graphics hardware industry.

[0025] Examples of supported video sources include, but are not limited to, the IEC 61834-2 standard (DV), the SMPTE 314M standard (DVCAM and DVCPRO-25, DVCPRO-50), and the SMPTE 370M (DVCPRO HD). In an embodiment of the present invention, the video processing methods can work with any uncompressed video frame (RGB 2D array) that is interlaced or non-interlaced and at any frame rate, although special features can require 60 fields per second interlaced (60i), 30 frames per second progressive (30p), or 24 frames per second progressive encoded in the 2:3 telecine (24p standard) or 2:3:3:2 telecine (24p advanced) formats. In addition to DV, there are numerous CODECs that exist to convert compressed video to uncompressed RGB 2D array frames. This embodiment of the present invention will work with any of these CODECs. Embodiments of the present invention can also provide desirable results when used in conjunction with high definition video.

[0026] The Frame Input Queue is implemented as a set of buffers, a front buffer pointer, and an end buffer pointer. When the front and end buffer pointers are incremented past the last buffer they preferably cycle back to the first buffer (i.e., they are circular or ring buffers). The Frame Output Queue is implemented in the same way. The Frame Input/Output Queues store uncompressed frames as buffers of uncompressed RGBA 2D arrays.

[0027] In a preferred embodiment of the present invention, a plurality of interface modules is preferably provided, which can be used together or separately. One user interface is preferably implemented primarily via software in conjunction with conventional hardware, and is preferably rendered on the primary display context of a graphics card attached to the system board, and uses keyboard/mouse input. The other user interface, which is preferably primarily a Hardware Interface, is preferably running on a microcontroller board that is attached to the USB or serial interfaces on the system board, is rendered onto an LCD display attached to microcontroller board, and uses a touch screen interface and hardware dials as input. Both interfaces display current state and allow the user to adjust settings. The settings are stored in the CFilmSettings object.

[0028] The CFilmSettings object is shared between the user interfaces and the video processing pipeline and is the main mechanism to effect changes in the video processing pipeline. Since this object is accessed by multiple independent processing threads, access can be protected using a mutual exclusion (mutex) object. When one thread needs to read or modify its properties, it must first obtain a pointer to it from the CSharedGraphicsDevice object. The CSharedGraphicsDevice preferably only allows one thread at a time to have access to the CFilmSettings object.

[0029] Fig. 4 shows details of the box labeled "Cinnafilm video processing algorithms" from Fig. 3. Uncompressed video frames enter the pipeline from the Frame Input Queue at the rate of 29.97 frames per second (NTSC implementation). On PAL implementations of the present invention, a rate of 25 frames per second is preferably provided. The video frame may contain temporal interlaced fields (60i), progressive frames (30p), or telecine interlaced fields (24p standard and 24p advanced). On PAL implementations, the video frame may contain temporal interlaced fields (50i) or progressive frames (25p).

[0030] In yet another embodiment of the present invention, the pipeline is a flexible pipeline that efficiently feeds video frames at a temporal frequency of 30 frames per second, handles one or more cadences (including but not limited to 24p or 30p), converts back to a predetermined number of frames per second, which can be 30 frames per second and preferably exhibits a high amount of reuse of software modules.

[0031] In a non-limiting embodiment, original video and film frames that have a temporal frequency of 24 frames per second are converted to 60 interlaced fields per second using the "forward telecine method". The telecine method repeats odd and even fields from the source frame in a 2:3 pattern for standard telecine or a 2:3:3:2 pattern for advanced telecine. For example, let F(an) be a function that returns the odd or even field of a frame n, where q=o indicates odd fields, q=e indicates even fields. The standard 2:3 telecine pattern would be:

F(0,o), F(0,e), F(1,o), F(1,e), F(1,o), F(2,e), F(2,o), F(3,e), F(3,o), F(3,e), ...

For better visualization of the pattern, let 0o stand for F(0,o), 0e stand for F(0,e), 1o stand for F(1,o), etc. Using this one can rewrite the 2:3 telecine pattern as:

{0o, 0e, 1o, 1e, 1o, 2e, 2o, 3e, 3o, 3e, ...}

One can group these to emphasize the 2:3 pattern:

{0o, 0e}, {1o, 1e, 1o}, {2e, 2o}, {3e, 3o, 3e}, ...

Now grouped to emphasis the resulting interlaced frames:

{0o, 0e}, {1o, 1e}, {1o, 2e}, {2o, 3e}, {3o, 3e}, ...

Notice that fields from frame 0 were used 2 times, frame 1 used 3 times, frame 2 used 2 times, and frame 3 used 3 times. One can reconstruct the original frames 0, 1, and 3 by selecting them from the sequence. To reconstruct original frame 2, one needs to build it from 2e and 2o fields in the {1o, 2e}, {2o, 3e} sequence.

[0032] The advanced 2:3:3:2 telecine pattern is:

{0o, 0e}, {1o, 1e, 1o}, {2e, 2o, 2o}, {3e, 3o}, ...

Now grouped to emphasis the resulting interlaced frames:

{0o, 0e}, {1o, 1e}, {1o, 2e}, {2o, 2e}, {3o, 3e}, ...

Notice that 4 out of 5 interlaced frames have fields from the same original frame number. Only the third frame contains fields from different original frames. Simply dropping this frame results in the original progressive frame sequence.

[0033] The Pipeline Selector reads the input format and the desired output format from the CFilmSettings object and selects one of six pipelines to send the input frame through.

[0034] The Letterbox mask, deinterlacing and cadence resampling module is selected when the user indicates that 60i input is to be converted to 24p or 30p formats. This module deinterlaces two frames and uses information from each frame for cadence resampling. This module also writes black in the letterbox region. Fig. 5 shows this module in detail.

[0035] The Letterbox mask, inverse telecine module is selected when the user indicates that 24p telecine standard or advanced is to be passed through or converted to 24p standard or advanced telecine formats. Even when conversion is not selected, the frames need to be inverse telecined in order for the film processing module to properly apply film grain and imperfections. This module also writes black in the letterbox region.

[0036] The Letterbox mask, frame copy module can be selected when the user indicates that 60i is to be passed through as 60i or when 30p is to be passed through as 30p. No conversion is possible with this module. This module also writes black in the letterbox region.

[0037] The Film process module, which is common to both the 24p and 30p/60i pipelines, transforms the RGB colors with a color transformation matrix. This transformation applies adjustments to hue, saturation, brightness, and contrast most preferably by using one matrix multiply. Midtones are preferably adjusted using a non-linear formula. Then imperfections (for example, dust, fiber, hair, chemical burns, scratches, etc.) are blended in. The final step applies the simulated film grain.

[0038] Interlace Using Forward Telecine takes processed frames that have a temporal frequency of 24 frames per second and interlaces fields using the forward telecine method. The user can select the standard telecine or advanced telecine pattern. This module produces interlaced frames, most preferably at a frequency 30 frames per second. The resulting frames are written to the Frame output queue.

[0039] The Frame Copy module can simply copy the processed frame, with a temporal frequency of 30 frames per second (or 60 interlaced fields), to the Frame output queue.

[0040] The following code (presented in C) is preferred to implement the Pipeline Selector of an embodiment of the invention:

```
// Process frame buffer in-place
void CGPU::ProcessFrame(BYTE* pInBuffer /*in*/, BYTE* pOutBuffer /*out*/, long
buffSize)
{
#ifdef ENABLE_FILTER
    HRESULT hr;

    CSharedGraphicsDevice* pSharedGraphicsDevice = GetSharedGraphicsDevice();
    IDirect3DDevice9* pD3DDevice = pSharedGraphicsDevice->LockDevice();
    CFilmSettings* pFilmSettings = pSharedGraphicsDevice->LockSettings();

    if (pFilmSettings->m_bypassOn)
    {
        // disable all effects
        pSharedGraphicsDevice->UnlockSettings();
        pSharedGraphicsDevice->UnlockDevice();
        memcpy(pOutBuffer, pInBuffer, buffSize);
        return;
    }

    if (pFilmSettings->m_resetPipeline)
    {
        ResetPipeline(pFilmSettings);
        pFilmSettings->m_resetPipeline = FALSE;
    }

    hr = m_pEffect->SetInt("g_motionAdaptiveOn", pFilmSettings-
>m_motionAdaptiveOn);

    // Begin scene drawing (queue commands to graphics card)
    pD3DDevice->BeginScene();

#if 0
    m_gpuUtil.DumpFrameTag(pInBuffer, L"Ref");
#endif

    //
    //  Render Stage A (Deinterlace/recadence, film effect)
    //
    if (pFilmSettings->m_inVideoCadence == IVC_I60)
    {
```

```
            if ((pFilmSettings->m_outVideoCadence == OVC_P24_STD) ||
(pFilmSettings->m_outVideoCadence == OVC_P24_ADV))
            {
                  ProcessStageA_Recadence24P(pD3DDevice, pFilmSettings,
pInBuffer);
            }
            else if (pFilmSettings->m_outVideoCadence == OVC_P30)
            {
                  // Deinterlace 60i to 30p
                  ProcessStageA_Simple(pD3DDevice, pFilmSettings, pInBuffer,
"ProcessField");
            }
            else
            {
                  // don't deinterlace, just copy frame as is
                  ProcessStageA_Simple(pD3DDevice, pFilmSettings, pInBuffer,
"CombineField");
            }
      }
      else if (pFilmSettings->m_inVideoCadence == IVC_P30)
      {
            // don't deinterlace, just copy frame as is
            ProcessStageA_Simple(pD3DDevice, pFilmSettings, pInBuffer,
"CombineField");
      }
      else if (pFilmSettings->m_inVideoCadence == IVC_P24)
      {
            ProcessStageA_UnTelecine(pD3DDevice, pFilmSettings, pInBuffer);
      }


      //
      //  Render Stage B (Interlace video)
      //
      if ((pFilmSettings->m_outVideoCadence == OVC_P24_STD) || (pFilmSettings-
>m_outVideoCadence == OVC_P24_ADV))
      {
            BOOL doAdvanced = (pFilmSettings->m_outVideoCadence == OVC_P24_ADV);
            ProcessStageB_Telecine(pD3DDevice, doAdvanced);
      }
      else
      {
            ProcessStageB_Simple(pD3DDevice);
      }

      // End scene drawing (submit commands to graphics card)
      hr = pD3DDevice->EndScene();

      // Read out the last processed frame into the output buffer.
      // We read an older frame so that we dont block on graphics card
      // which is rendering at GetEnd()->Prev()
      FrameIter* pFrameIter = m_resultQueue.GetFront();
      Frame* pFrame = pFrameIter->Get();
      m_gpuUtil.ReadFrame(pD3DDevice, pFrame->m_pRenderTarget, pOutBuffer);

#if 0
      m_gpuUtil.DumpFrame(pOutBuffer);
#endif

      pSharedGraphicsDevice->UnlockSettings();
      pSharedGraphicsDevice->UnlockDevice();
#endif
}
```

[0041] In a non-limiting embodiment, the invention preferably uses a Stream Programming Model (Stream Programming) to process the video frames. Stream Programming is a programming model that makes it much easier to develop highly parallel code. Common pitfalls in other forms of parallel programming occur when two threads of execution (threads) access the same data element, where one thread wants to write and the other wants to read. In this situation, one thread must be blocked while the other accesses the data element. This is highly inefficient and adds complexity. Stream Programming avoids this problem because the delivery of data elements to and from the threads is handled explicitly by the framework runtime. In Stream Programming, Kernels are programs that can only read values from their input streams and from global variables (which are read-only and called Uniforms). Kernels can only write values to their output stream. This rigidity of data flow is what allows the Kernels to be executed on hundreds of processing cores all at the same time without worry of corrupting data.

[0042] Direct3D 9 SDK is most preferably used to implement the Stream Programming Model and the video processing methods of the invention. However, the methods are not specific to Direct3D 9 SDK and can be implemented in any Stream Programming Model. In Direct3D a Kernel is called a Shader. In Direct3D 9 SDK, there are two different shader types: Vertex Shaders and Pixel Shaders. Most of the video processing preferably occurs in the Pixel Shaders. The Vertex Shaders can primarily be used to setup values that get interpolated across a quad (rectangle rendered using two adjacent triangles). In Pixel Shaders, the incoming interpolated data from a stream is called a Pixel Fragment.

[0043] In one embodiment, it is first preferred to set up the Direct3D runtime to render a quad that causes a Pixel Shader program to be executed for each pixel in the output video frame. Each Pixel Fragment in the quad gets added to one of many work task queues (Streams) that are streamed into Pixel Shaders (Kernels) running on each core in the graphics card. A Pixel Shader can be used for only producing the output color for the current pixel. The incoming stream contains information so that the Pixel Shader program can identify which pixel in the video output stream it is working on. The current odd and even video fields are stored as uniforms (read-only global variables) and can be read by the Pixel Shaders. The previous four deinterlaced/inverse telecined frames are also preferably stored as uniforms and are used by motion estimation algorithms.

[0044] The invention comprises preferred methods to convert 60 interlaced fields per second to 24 deinterlaced frames per second. The blending of 60i fields into full frames at a 24p sampling rate is most preferably done using a virtual machine that executes Recadence Field Loader Instructions. In this embodiment, one instruction is executed for every odd/even pair of 60i fields that are loaded into the Frame Input Queue. The instructions determine which even and odd

fields are loaded into the pipeline, when to resample to synthesize a new frame, and the blend factor (linear interpolation factor) used during the resampling.

```
struct RecadenceInst
{
        BOOL m_loadFieldOdd;        // load odd field into pipeline
        BOOL m_loadFieldEven;       // load even field into pipeline
        int m_processFrame;         // combine two fields from head of pipeline
        float m_blendFactor;        // factor to blend two fields from head of
pipeline
};

RecadenceInst g_recadenceInst[] =
{       //      load odd        load even       process             blend
        {       TRUE,           TRUE,           TRUE,           0.75f   },
        {       TRUE,           TRUE,           TRUE,           0.25f   },
        {       FALSE,          TRUE,           FALSE,          0.00f   },
        {       TRUE,           TRUE,           TRUE,           0.25f   },
        {       TRUE,           FALSE,          TRUE,           0.75f   },
};
```

[0045] The instruction also indicates when the two fields from the head of the queue are to be deinterlaced and resampled into a progressive frame. Since there 4/5 as many frames in 24p than in 30p, four of the five instructions will process fields to produce a full frame. The two fields at the head of the pipeline are preferably processed with the specified blend factor.

[0046] The following sequence shows 30 interlaced frames per second and 60 progressive fields per second on a timeline:

```
60i Frames:  F0      F1      F2      F3      F4
60i Fields:  o   e   o   e   o   e   o   e   o   e
Time(s):     0/30    1/30    2/30    3/30    4/30  ...
```

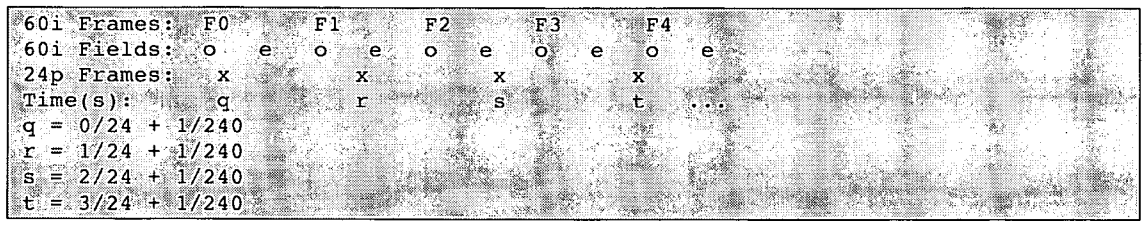
To convert to 24 frames per second, one needs to synthesize 4 new progressive frames from the original 5 frames. One approach is to start sampling at t = 0/30seconds(s):

```
60i Frames:  F0      F1      F2      F3      F4
60i Fields:  o   e   o   e   o   e   o   e   o   e
24p Frames:  x       x       x       x
Time(s):     0/24    1/24    2/24    3/24  ...
```

Notice that 0/24s and 2/24s samples, shown as an "x", line up perfectly with either an odd or even field. These 24p frames can be constructed using standard deinterlacing techniques. Samples 1/24s and 3/24s occur at a time that is halfway between the odd and even field sample times (1/24s = 2.5/60s). These samples are problematic because at t=1/24s there is no original field to sample

from. Since one is exactly halfway between an odd and even field sample, there is no bias towards any one field. The goal is to reconstruct a frame that renders objects in motion at their precise position at the desired sample time. One can synthesize a new frame by averaging the two 60i fields (blending 50% of from each pixel from the odd field with 50% from the even field). The resulting frame is less than ideal, but still looks good for areas of slow motion. But when the video is played at full speed, a temporal artifact is clearly visible. This is because half of the 24p frames contain motion artifacts and the other half does not. This is perceived as a 12 Hz stutter.

[0047] The invention preferably employs offset 24p sampling by (1/4 * 1/60) = 1/240 second to avoid 12Hz stutter artifact. The 12Hz stutter problem is solved by introducing a time offset of 1/240 sec., or one quarter of 1/60 sec., to the 24p sampling timeline.



[0048] Now each sampling point "x" is consistently 1/240 second away from a field sample time. One now synthesizes a new frame by averaging two deinterlaced 60i fields with blend factors of .25 (25%) for the closest field and .75 (75%) for the next closest field. These blend factors then preferably are stored in the Recadence Field Loader Instructions.

[0049] On a pixel by pixel basis, the deinterlaced color value is preferably chosen from one of two possibilities: a) a color value from the .25/.75 blending of the two nearest upsampled fields, or b) a color value from the odd field source (if we are rendering a pixel in the odd line in the destination) or even field source (if we are rendering a pixel in the even line). A motion metric is used to determine if color (a) or (b) is chosen.

[0050] An embodiment of the invention preferably uses bilinear sampling hardware, which is built into the graphics hardware and is highly optimized, to resize fields to full frame height. In this embodiment, multiple bilinear samples from different texture coordinates are averaged together to get an approximate Gaussian resizing filter. Odd fields are preferably sampled spatially one line higher than even fields. When upsampling even field images, it is preferred to use a slight texture coordinate offset (1/480 for standard definition) during sampling. This eliminates the bobbing effect that is apparent in other industry deinterlacers. Because of the special texture sampling hardware in graphics hardware, a bilinear sample takes the same amount of time as a point sampler. By using

bilinear samples, one reduces the number of overall samples required, thereby reducing the overall sampling time.

[0051] For motion adaptive deinterlacing, the motion metric is preferably computed as follows: a) for the both the odd and even fields, sum three separate bilinear samples with different (U,V) coordinates such that we sample the current texel, ½ texel up, and ½ texel down, b) scale the red, green, and blue components by well known luminance conversion factors, c) convert the odd and even sums to luminance values by summing the color components together, d) compute the absolute difference between the odd and even luminance value, and e) compare the resulting luminance difference with the threshold value of 0.15f (0.15f is empirical). By summing three different bilinear samples together, one is in effect blurring the source image. If one does not blur the source fields before computing the difference, one can mistakenly detect motion wherever there are horizontal features.

[0052] One embodiment of the invention preferably uses graphics interpolation hardware to interpolate the current row number. The row number is used to determine if the current pixel is in the letterbox black region. If in the black region, the pixel shader returns the black color and stops processing. This early out feature reduces computation resources. Next follows the preferred pixel shader code that computes motion adaptive deinterlacing, resamples at a 24p cadence, and applies letterbox masking. The "g_evenFieldOfs.y" is a constant value that adjusts a texture coordinate position by ½ texel:

```
float4 ProcessFieldPS(VS_OUTPUT VSOUT) : COLOR
{
    float4 outColor : register(r0);

    if ((VSOUT.m_rowScaled < g_letterBoxLow) || (VSOUT.m_rowScaled >
g_letterBoxHigh))
    {
        outColor = float4(0, 0, 0, 0);
    }
    else
    {
        float2 oddTexCoord = VSOUT.m_texCoord + g_oddFieldOfs;
        float2 evenTexCoord = VSOUT.m_texCoord + g_evenFieldOfs;

        float4 colA = tex2D(OddFieldLinearSampler, oddTexCoord);
        float4 colB = tex2D(EvenFieldLinearSampler, evenTexCoord);

        bool first = frac(VSOUT.m_rowScaled) < .25;

        // compute the blended sample
        outColor = lerp(colB, colA, g_fieldBlendFactor);

        if (g_motionAdaptiveOn)
        {
            // Move up 1/2 texel and sample
            colA += tex2D(OddFieldLinearSampler, oddTexCoord -
g_evenFieldOfs.y);
```

```
                colB += tex2D(EvenFieldLinearSampler, evenTexCoord -
g_evenFieldOfs.y);

                // Move down 1/2 texel and sample
                colA += tex2D(OddFieldLinearSampler, oddTexCoord +
g_evenFieldOfs.y);
                colB += tex2D(EvenFieldLinearSampler, evenTexCoord +
g_evenFieldOfs.y);

                // Compute difference
                float4 a = colA * float4(0.3086f, 0.6094f, 0.0820f, 0.0f);
                float lumA = a.r + a.g + a.b;

                float4 b = colB * float4(0.3086f, 0.6094f, 0.0820f, 0.0f);
                float lumB = b.r + b.g + b.b;

                lumA = abs(lumA - lumB);

                if (lumA < 0.15f)   // .15 is an empirical value
                {
                    // Area of low motion; switch to weave
                    if (first)
                    {
                        outColor = tex2D(EvenFieldPointSampler,
evenTexCoord);
                    }
                    else
                    {
                        outColor = tex2D(OddFieldPointSampler,
oddTexCoord);
                    }
                }
            }
        outColor = FilmProcess(VSOUT, outColor);
    }

    return outColor;
}
```

[0053] Next is discussed the preferred methods used to convert 60 interlaced fields per second to 30 deinterlaced frames per second. The resampling of 60i fields into 30 full deinterlaced frames per second is done by leveraging a portion of the 60i to 24p deinterlacing code. In the 60i to 24p method, the fields that are loaded into the deinterlacer are preferably specified by the Recadence Field Loader Instructions. In 60i to 30p, one simply loads the odd and even fields for every frame. The field blend constant is always set to 0.0 (or 1.0 is equally valid). This approach leverages complicated code for more than one purpose. This method results in motion adaptive deinterlaced frames.

[0054] The preferred methods to convert telecined (standard and advanced) video, encoded as 60 interlaced fields per second, to 24 deinterlaced frames per second are next discussed. The original frames recorded at 24p and encoded using the telecine method (standard 2:3 and advanced 2:3:3:2 repeat pattern) are recovered using a virtual machine that executes UnTelecine Field Loader

Instructions. One instruction is executed for every odd/even pair of 60i fields that are loaded into the Frame Input Queue. The following code shows the preferred UnTelecine Field Loader Instructions:

```
struct UnTelecineInst
{
    BOOL m_loadFieldOdd;        // load odd field into next full frame
    BOOL m_loadFieldEven;       // load even field into next full frame
};

UnTelecineInst g_stdUnTelecineInst[] =
{   //      load odd        load even
    {       TRUE,           TRUE    },
    {       TRUE,           TRUE    },
    {       FALSE,          TRUE    },
    {       TRUE,           FALSE   },
    {       TRUE,           TRUE    },
};

UnTelecineInst g_advUnTelecineInst[] =
{   //      load odd        load even
    {       TRUE,           TRUE    },
    {       TRUE,           TRUE    },
    {       FALSE,          FALSE   },
    {       TRUE,           TRUE    },
    {       TRUE,           TRUE    },
};
```

When an odd or even field is loaded, the m_oddFieldLoaded or m_evenFieldLoaded flag is set. When both flags are set, i.e. two fields have been loaded, the inverse telecine module combines the two fields into one full progressive 24p frame.

[0055] The virtual machine instruction pointer is preferably aligned with the encoded 2:3 (or 2:3:3:2) pattern. In order to do this reliably, the field difference history information is preferably stored for the last about 11 frames (10 even field deltas, 10 odd field deltas, 20 difference values in one example). In one embodiment, the TelecineDetector module performs this task. The TelecineDetector stores the variance between even fields or odd fields in adjacent frames. The variance is defined as the average of the difference between a channel in each pixel in consecutive even or odd fields squared. The TelecineDetector generates a score given the history, a telecine pattern, and an offset into the pattern. The score is generated by looking at what the pattern is supposed to be. If the fields are supposed to be the same, it adds the variance between those two fields to the score. The pattern and offset that attains the minimum score is most likely to be the telecine pattern the video was encoded with, and the offset is the stage in the pattern of the newest frame. The preferred code for the TelecineDetector is:

```
// We need to keep 10 frames of history
#define DIFF_HISTORY_LENGTH (10)

enum TELECINE_TYPE
{
```

```
        TT_STD_A = 0,           // standard 2:3 telecine
        TT_STD_B,               // standard 2:3 telecine
        TT_ADV_A,               // advanced 2:3:3:2 telecine
        TT_ADV_B,               // advanced 2:3:3:2 telecine
        TT_UNKNOWN,
};
//
// The field difference computed between the current field and the previous field is
// stored in the current field's obect.  Thus when detelecine the current frame, we
// can look at the past frame differences to determine which decode instruction we
// should be on.
//

// Standard Telecine Pattern
// Pattern repeats after 10 fields (5 frames)
//        !             !
//3    2  3   2  3    2  3   2  3
//x xx aa bb bc cd dd ee ff fg gh hh
//     dd dd sd dd ds dd dd sd dd ds
//     i0 i1 i2 i3 i4 i0 i1 i2 i3 i4
char* g_pStdTcn_A = "dd dd ds dd sd";
char* g_pStdTcn_B = "dd dd sd dd ds";

// Advanced Telecine Pattern
// Pattern repeats after 10 fields (5 frames)
//        !             !
//  2  2  3   3    2  2  3   3    2
//  xx aa bb bc cc dd ee ff fg gg hh
//     dd dd sd ds dd dd dd sd ds dd
//     i0 i1 i2 i3 i4 i0 i1 i2 i3 i4
char* g_pAdvTcn_A = "dd dd sd ds dd";
char* g_pAdvTcn_B = "dd dd ds sd dd";

char * g_ppTcnPatterns[] = { g_pStdTcn_A, g_pStdTcn_B, g_pAdvTcn_A, g_pAdvTcn_B };

const int g_TcnPatternCount = sizeof(g_ppTcnPatterns) / sizeof(g_ppTcnPatterns[0]);

class TelecineDetector
{
public:
        void Reset()
        {
                m_History.clear();
        }

        //  This function finds the minimum score for all the possible
        //  (telecine pattern, offset) pairs
        void DetectState(I32 frameIndex, float OddDiffSq /*in*/, float EvenDiffSq
/*in*/,
                        TELECINE_TYPE* pTelecineType /*out*/, int* pIndex /*out*/)
        {
                AddHistory(frameIndex, OddDiffSq, EvenDiffSq);

                float best = -1.0f;
                *pTelecineType = TT_UNKNOWN;

                for(int j = 0; j < g_TcnPatternCount; ++j)
                {
                        for(int i = 0; i < 5; ++i)
                        {
                                float s = Score(g_ppTcnPatterns[j], i);
                                if(s < best || (i == 0 && j == 0))
                                {
                                        best = s;
                                        *pTelecineType = (TELECINE_TYPE)j;
                                        *pIndex = i;
                                }
                        }
                }
        }
```

```
protected:
        //  One history sample
        struct Frame
        {
                I32 frameIndex;
                float OddDiffSq;
                float EvenDiffSq;

                Frame(I32  f,  float  o,  float  e)  :  frameIndex(f),  OddDiffSq(o),
EvenDiffSq(e) { }
        };

        //  A list of history samples
        std::list < Frame > m_History;

        //  Get the index'th pattern element
        void GetPatternElement(char * pattern, int index, bool & odd, bool & even)
        {
                while(index < 0)
                        index += 5;
                while(index >= 5)
                        index -= 5;
                odd = pattern[index * 3 + 1] == 's';
                even = pattern[index * 3 + 0] == 's';
        }

        //  Compute the score for a given pattern and offset
        float Score(char * pattern, int offset)
        {
                float s = 0.0f;
                I32 base = m_History.front().frameIndex;
                for(std::list  <  Frame  >::iterator  i  =  m_History.begin();  i  !=
m_History.end(); ++i)
                {
                        bool oddsame, evensame;
                        GetPatternElement(pattern,  offset  -  ((int)base  -  (int)i-
>frameIndex), oddsame, evensame);
                        float zodd = i->OddDiffSq;
                        float zeven = i->EvenDiffSq;

                        // If the fields are supposed to be the same, add the variance
to the score.
                        // If they are supposed to be different, it doesn't matter
whether they are the same or not
                        if(oddsame)
                                s += zodd;
                        if(evensame)
                                s += zeven;
                }
                return s;
        }

        void AddHistory(I32 frameIndex, float OddDiffSq, float EvenDiffSq)
        {
                Frame f(frameIndex, OddDiffSq, EvenDiffSq);
                m_History.push_front(f);
                while(m_History.size() > DIFF_HISTORY_LENGTH)
                        m_History.pop_back();
        }
};
```

[0056] Next follows the preferred Pixel Shader subroutine FilmProcess() that applies color adjustments, imperfections, and simulated film grain:

```
float4 FilmProcess(VS_OUTPUT VSOUT, float4 color)
{
```

```
    // Apply color matrix for hue, sat, bright, contrast
    // this compiles to 3 dot products:
    color.rgb = mul(float4(color.rgb, 1), (float4x3)colorMatrix);

    // Adjust midtone using formula:
    //    color + (ofs*4)*(color - color*color)
    // NOTE: output pixel format = RGBA
    float4 curve = 4.0f * (color - (color * color));
    color = color + (float4(midtoneRed, midtoneGreen, midtoneBlue, 0.0f) *
curve);

    // Apply imperfections/specks
    float4 c = tex2D(Specks, VSOUT.m_texCoord + g_frameOfs);
    color.rgb = ((1.0f - c.r) * color.rgb); // apply black specks
    color.rgb = ((1.0f - c.g) * color.rgb) + c.g; // apply white specks

    // Apply film grain effect
    // TODO:  confirm correct lum ratios
    c = color * float4(0.3086f, 0.6094f, 0.0820f, 0.0f);
    float lum = c.r + c.g + c.b;
    c = tex2D(FilmGrain, VSOUT.m_texCoord + g_frameOfs); // TODO: are we using
correct offsets here?
    lum = 1.0f - ((1.0f - lum) * c.a * grainPresence);
    color = color * lum;

    color = clamp(color, 0, 1);
    return color;
}
```

[0057] FilmProcess() takes as input a VSOUT structure (containing interpolated texture coordinate values used to access the corresponding pixel in input video frames) and an input color fragment represented as red, green, blue, and alpha components. The first line applies the color transform matrix which adjusts the hue, saturation, brightness, and contrast. Color transformation matrices are as commonly used. The next line computes a non-linear color curve tailored to mimic film saturation curves.

[0058] The invention preferably computes a non-linear color curve tailored to mimic film saturation curves. The curve is a function of the fragment color component. Three separate curves are preferably computed: red, green, and blue. The curve formula is chosen such that it is efficiently implemented on graphics hardware, preferably:

color = color + (adjustmentFactor * 4.0) * (color – color * color)

The amount of non-linear boost is modulated by the midtoneRed, midtoneGreen, and midtoneBlue uniforms (global read-only variables). These values are set once per frame and are based on the input from the user interface.

[0059] The invention preferably uses a procedural noise function, such as Perlin or random noise, to generate film grain textures (preferably eight) at initialization. Each film grain texture is unique and the textures are put into the texture queue. Textures are optionally used sequentially

from the queue, but random transformations on the texture coordinates can increase the randomness. Textures coordinates can be randomly mirrored or not mirrored horizontally; and/or rotated 0, 90, 180, 270 degrees. This turns, for example, 8 unique noise textures into 64 indistinguishable samples.

[0060] Film Grain Textures are preferably sampled using a magnification filter so that noise structures will span multiple pixels in the output frame. This mimics real-life film grain when film is scanned into digital images. Noise that varies at every pixel appears as electronic noise and not film grain.

[0061] A system of noise values (preferably seven) can be used to produce color grain where the correlation coefficient between each color channel is determined by a variable grainCorrelation. If 7 noise values are labeled as follows: R, G, B, RG, RB, GB, RGB, the first 3 of these values can be called the uncorrelated noise values, the next 4 can be called the correlated noise values. When sampling a noise value for a color channel, one preferably takes a linear combination of every noise value that contains that channel. For example, when sampling noise for the red channel, one could take the noise values R, RG, RB, and RGB. Let c = grainCorrelation. Now, three functions can be created that define the transition from uncorrelated noise to correlated noise, grain1(c), grain2(c), and grain3(c). These functions preferably have the property that 0 < grainX(c) < 1, grain1(c) + grain2(c) + grain3(c) = 1 for 0 < c < 1, grain1(0) = 1, and grain3(1) = 1. Now define the following linear combination of the noise channels, the sampling for R is shown below:

grain1(c) * R + 0.5f * grain2(c) * (RG + RB) + grain3(c) * RGB

This will result in a smooth transition between uncorrelated noise and fully correlated (R = G = B) noise. Preferred code follows:

```
float4 FilmProcess(VS_OUTPUT VSOUT, float4 color)
{
        // Apply color matrix for hue, sat, bright, contrast
        // this compiles to 3 dot products:
        color.rgb = mul(float4(color.rgb, 1), (float4x3)colorMatrix);

        // Adjust midtone using formula:
        //    color + (ofs*4)*(color - color*color)
        // NOTE: output pixel format = RGBA
        float4 curve = 4.0f * (color - (color * color));
        color = color + (float4(midtoneRed, midtoneGreen, midtoneBlue, 0.0f) * curve);

        // Apply imperfections/specks
        float4 c = tex2D(Specks, VSOUT.m_texCoord + g_frameOfs);
        color.rgb = ((1.0f - c.r) * color.rgb); // apply black specks
        color.rgb = ((1.0f - c.g) * color.rgb) + c.g; // apply white specks

        // Apply film grain effect
```

```
        // TODO:  confirm correct lum ratios
        // Y709 = 0.2126R + 0.7152G + 0.0722B
        // Uncorrelated noise R = c1.r
        // Uncorrelated noise G = c1.g
        // Uncorrelated noise B = c1.b
        // Correlated noise RG = c2.r
        // Correlated noise RB = c2.g
        // Correlated noise GB = c2.b
        // Correlated noise RGB = c2.a
        float2 texCoord = VSOUT.m_noiseTexCoord;
        float lum = dot(color.rgb, float3(0.3086f, 0.6094f, 0.0820f));
        float4 c1 = tex2D(FilmGrainA, texCoord);
        float4 c2 = tex2D(FilmGrainB, texCoord);

        c.r = grain3 * c2.a + grain2 * (c2.r + c2.g) + grain1 * c1.r;
        c.g = grain3 * c2.a + grain2 * (c2.r + c2.b) + grain1 * c1.g;
        c.b = grain3 * c2.a + grain2 * (c2.g + c2.b) + grain1 * c1.b;

        c -= 0.5f; // normalize noise
        c *= (1.0f - lum); // make noise magnitude inversely porportional tc brightness
        color += c * grainPresence * 2.0f;

        color = clamp(color, 0, 1);
        return color;
}
```

[0062] Film Grain Textures are preferably sampled using bilinear sampling graphics hardware to produce smooth magnification. The grain sample color is adjusted based on the brightness (lumen value) of the current color fragment and a user settable grain presence factor. The preferred formula is: grain = (1.0f - lum) * (grainSample - 0.5f) * grainPresence * 2. This makes film grain structures more noticeable in dark regions and less noticeable in brighter regions. The grain color is then added to the output color fragment by:

color = color + grain.

[0063] Imperfections (dust, fiber, scratches, etc.) are preferably rendered using graphics hardware into a separate frame sized buffer (Imperfection Frame). A unique Imperfection Frame can be generated for every video frame. Details of how the Imperfection Frame is created are discussed below. In one embodiment, the Imperfection Frame has a color channel that is used to modulate the color fragment before the Imperfection color fragment is added in.

[0064] In a non-limiting embodiment of the present invention, the pipeline preferably enables a fragment shader program to perform all the following operations on each pixel independently and in one pass: motion adaptive deinterlace, recadence sampling, inverse telecine, apply linear color adjustments, non-linear color adjustments, imperfections, and simulated film grain. Doing all these operations in one pass significantly reduces memory traffic on the graphics card and results in better utilization of graphics hardware. The second pass interlaces or forward telecines processed frames to produce the final output frames that are recompressed.

[0065] A texture atlas is preferably employed, such as shown in Fig. 6, to store imperfection subtextures for dust, fibers, hairs, blobs, chemical burns, and scratch patterns. The texture atlas is also used in the scratch imperfection module. Each subtexture is preferably 64x64 pixels. The texture atlas size can be adjustable with a typical value of about 10x10 (about 640x640 pixels). Using a texture atlas instead of individual textures improves performance on the graphics hardware (each texture has a fixed amount of overhead if swapped to/from system memory).

[0066] The texture atlas is preferably pre-processed at initialization time to soften and create subtle ringing around edges. This greatly increases the organic look of the imperfection subtextures. The method uses the following steps:

i. Ib = BlurrMore(Ia)
ii. Ic = Diff(Ib, GaussBlur(Ib, 2.5))
iii. Id = Ic + (Ic / 2)

Doing this once instead of during every frame improves performance.

[0067] Within a given category (dust, fiber, etc.) a subtexture can be randomly selected. The subtexture then preferably is applied to a quad that is rendered to the Imperfection Frame. In this embodiment, the quad is rendered with random position, rotation (about the X, Y, and Z axis), and scale. Rotation about the X and Y axis is optionally limited in order to prevent severe aliasing due to edge on rendering (in one instance it is preferred to limit this rotation to about +/- 22 degrees off the Z plane). Rotation values that create a flip about the X or Y can be allowed. Rotation about the Z axis is unrestricted. The subtexture can be rendered as black or white. The color can be randomized and the ratio of black to white is preferably controllable from the UI. Another channel is optionally used to store the modulation factor when the Imperfection Image is combined with the video frame. The subtextures are sampled using a bilinear minification filter, bilinear magnification filter, Linear MipFilter, and max anisotropy value of 1. These settings are used to prevent aliasing.

[0068] Many imperfection parameters are preferably randomized. Some parameters, such as frequency and size, are varied using a skewed random distribution. Random values are initially generated with an even distribution from 0.0 to 1.0. The random distribution is preferably then skewed using the exponential function in order to create a higher percentage of random samples to occur below a certain set point. Use of this skewed random function increases the realism of simulated imperfections.

[0069] The following code demonstrates an exponentially skewed random function:

```
// Exponential distribution skews results towards range_min.
// Good values for exponent are:
//    1.3 yields ~59% results in the lower half of range
//    1.5 yields ~64% results in the lower half of range
//    2.0 yields ~70% results in the lower half of range
//    2.5 yields ~75% results in the lower half of range
float Specks::RandomExpDist(const Range& r, float exponent)
{
      float ratio = float(rand()) / float(RAND_MAX);
      ratio = pow(ratio, exponent);
      return  (ratio * (r.m_max - r.m_min)) + r.m_min;
}
```

[0070] Scratch type imperfections can be different than dust or fiber type imperfections in that they can optionally span across multiple frames. In order to achieve this effect, every scratch deployed by the invention preferably has a simulated lifetime. When a scratch is created it preferably has a start time, life time, and coefficients to sine wave equations used to control the path the scratch takes over the frame. A simulation system preferably simulates film passing under a mechanical frame that traps a particle. As the simulation time step is incremented the simulated film is moved through the mechanical frame. When start time of the scratch equals the current simulation time, the scratch starts to render quads to the Imperfection Frame. The scratch continues to render until its life time is reached.

[0071] Scratch quads are preferably rendered stacked vertically on top of each other. Since the scratch path can vary from left to right as the scratch advances down the film frame, the scratch quads can be rotated by the slope of the path using the following formula:

roll = (pi / 2.0f) + atan2(ty1 - ty, tx1 - tx).

[0072] Scratch size is also a random property. Larger scratches are rendered with larger quads. Larger quads require larger time steps in the simulation. Each scratch particle requires a different time delta. The invention solves this problem by running a separate simulation for each scratch particle (multiple parallel simulations). This works for simulations that do not simulate particle interactions. When the particle size gets quite small, one does not typically want to have a large number of very small quads. Therefore, it is preferred to enforce a minimum quad size, and when the desired size goes below the minimum, one switches to the solid scratch size and scale only in the x scratch width dimension.

[0073] Scratch paths can be determined using a function that is the sum of three wave functions. Each wave function has frequency, phase, and magnitude parameters. These

parameters can be randomly determined for each scratch particle. Each wave contributes variations centered around a certain frequency: 6Hz, 120Hz, and 240Hz.

[0074] Preferred code for the imperfections module follows:

```
//
// class SpeckType
//

class Range
{
public:
    float m_min;
    float m_max;
};


class SpeckType
{
public:
    bool m_enable;
    float m_frequency;  // number of imperfections per second
    float m_probBlack;  // probability of black color (vs. white). 1.0 = all
black .5 = black/white, 0.0 = all white

    int m_atlasBaseIndex;     // offset to first texture in the texture atlas
    int m_atlasNumTextures;   // number of textures in the atlas

    float m_scaleProbExp;  // probability skew factor.  newProd = prob^^exp
    Range m_scaleX;
    Range m_scaleAspect;  // Y = aspect * X

    bool m_allowFlipAboutX;
    Range m_rotAboutX;
    bool m_allowFlipAboutY;
    Range m_rotAboutY;
    Range m_rotAboutZ;
};

enum SPEC_TYPE
{
    ST_FIBER = 0,
    ST_DUST,
    ST_PARTICLE,
    ST_BLOB
};


//
// class Particle
//

const int g_numPathTerms = 3;

class Particle
{
public:
    void Reset(); // initialize values
    void Seed();  // seed random values into simulation
    float ComputePosition(float time);
```

```
    float m_timeCurrent;       // current time
    float m_timeStart;  // time (secs) when particle is visible
    float m_timeStop;   // time (secs) when particle is gone

    float m_color;      // particle color
    float m_size; // particle width
    int m_textureIndex;

private:
    float m_startPos;   // starting x position

    float m_pFrequency[g_numPathTerms];
    float m_pPhase[g_numPathTerms];
    float m_pMagnitude[g_numPathTerms];

    float Random(float minVal, float maxVal);
    float RandomExpDist(float minVal, float maxVal, float exponent);
};

void Particle::Reset()
{
    memset(this, 0, sizeof(Particle));
}

void Particle::Seed()
{
    m_startPos = Random(-2.0f, 2.0f);

    // position modifier constants
    m_pFrequency[0] = Random(0.0f, 1.0f); // 1.0 = one cycle every 6.28 secs (2
* pi)
    m_pPhase[0] = Random(0.0f, 6.28f);
    m_pMagnitude[0] = RandomExpDist(0.0f, 2.0f, 3.0f);

    m_pFrequency[1] = Random(0.0f, 20.0f);
    m_pPhase[1] = Random(0.0f, 6.28f);
    m_pMagnitude[1] = RandomExpDist(0.0f, 0.25f, 3.0f);

    m_pFrequency[2] = Random(0.0f, 40.0f);
    m_pPhase[2] = Random(0.0f, 6.28f);
    m_pMagnitude[2] = Random(0.0f, .05f);
}

float Particle::ComputePosition(float time)
{
    int i;
    float time2 = time;

    float x = m_startPos;
    for (i = 0; i < g_numPathTerms-1; ++i)
    {
        x = x + sin(time2 * m_pFrequency[i] + m_pPhase[i]) *
m_pMagnitude[i];
    };
    return x;
}

float Particle::Random(float minVal, float maxVal)
{
    float ratio = float(rand()) / float(RAND_MAX);
    return  (ratio * (maxVal - minVal)) + minVal;
}
```

```
float Particle::RandomExpDist(float minVal, float maxVal, float exponent)
{
    float ratio = float(rand()) / float(RAND_MAX);
    ratio = pow(ratio, exponent);
    return  (ratio * (maxVal - minVal)) + minVal;
}

//
// class ParticleSet
//

class ParticleSet
{
private:
    const static int ARRAY_SIZE = 15;

public:
    ParticleSet();

    // reset iterator to beginning of particle list
    void ResetIterator();

    // get first/next particle
    Particle* GetNextParticle();

    // remove the particle returned in last call to GetNextParticle()
    void RemoveParticle();

    // allocate a new particle and add to the list.  returns NULL if list full.
    Particle* NewParticle();

private:
    Particle m_particleArray[ARRAY_SIZE];
    bool m_valid[ARRAY_SIZE];
    int m_index;
};

ParticleSet::ParticleSet()
{
    int i;
    for (i = 0; i < ARRAY_SIZE; ++i)
    {
        m_valid[i] = false;
    }
    m_index = 0;
}

// reset iterator to beginning of particle list
void ParticleSet::ResetIterator()
{
    m_index = 0;
}

// get first/next particle
Particle* ParticleSet::GetNextParticle()
{
    while (m_index < ARRAY_SIZE)
    {
        if (m_valid[m_index])
        {
            return &(m_particleArray[m_index++]);
        }
```

```
            ++m_index;
    }
    return 0;
}

// remove the particle returned in last call to GetNextParticle()
void ParticleSet::RemoveParticle()
{
    int lastIndex = m_index - 1;
    if ((lastIndex >= 0) && (lastIndex < ARRAY_SIZE))
            m_valid[lastIndex] = false;
}

// allocate a new particle and add to the list.  returns NULL if list full.
Particle* ParticleSet::NewParticle()
{
    int i;
    for (i = 0; i < ARRAY_SIZE; ++i)
    {
            if (!m_valid[i])
            {
                    m_valid[i] = true;
                    Particle& particle = m_particleArray[i];
                    particle.Reset();
                    return &particle;
            }
    }
    return 0;
}


//
// class QuadStack
//

struct Position
{
    float x;
    float y;
    float z;
};

struct TexCoord
{
    float u;
    float v;
};

struct TexCoordPair
{
    TexCoord t1;
    TexCoord t2;
};

struct SpeckVertex
{
    Position pos;
    TexCoord texCoord;
    float material;
};

D3DVERTEXELEMENT9 g_pSpeckVertexDecl[] =
{
```

```
  {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
  {0, 20, D3DDECLTYPE_FLOAT1, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR, 0},
  D3DDECL_END()
};

class QuadStack
{
public:
    QuadStack();
    ~QuadStack();

    void Initialize(IDirect3DDevice9* pD3DDevice);  // allocate buffer with
default size
    void Reset(); // reset stack

    // Add a (-1,-1):(1,1) quad transformed by pMatrix to the quad stack.
    void Add(D3DXMATRIX* pMatrix, TexCoordPair* pTexCoordPair, float material);
    void GetTriList(IDirect3DVertexBuffer9** ppVertexBuffer,
IDirect3DVertexDeclaration9** ppVertexDecl, int* pStride, int* pNumPrims);

private:
    int m_buffSize;
    IDirect3DVertexDeclaration9* m_pVertexDecl;
    IDirect3DVertexBuffer9* m_pVertexBuffer;
    SpeckVertex* m_pStartVertex;
    SpeckVertex* m_pNextVertex;
    SpeckVertex* m_pEndVertex;
};


QuadStack::QuadStack() : m_pStartVertex(0), m_pNextVertex(0),
          m_pEndVertex(0), m_buffSize(0), m_pVertexBuffer(0)
{
}

QuadStack::~QuadStack()
{
    SAFE_RELEASE(m_pVertexBuffer);
    SAFE_RELEASE(m_pVertexDecl);
}

void QuadStack::Initialize(IDirect3DDevice9* pD3DDevice)
{
    m_buffSize = sizeof(SpeckVertex) * 6 * g_maxNumSpecksPerPass;      // 6
vertices in a quad when drawing with trilists
    HRESULT hr = pD3DDevice->CreateVertexBuffer( m_buffSize, D3DUSAGE_WRITEONLY
| D3DUSAGE_DYNAMIC,
                                                 0, D3DPOOL_DEFAULT,
&m_pVertexBuffer, 0 );

    hr = pD3DDevice->CreateVertexDeclaration(g_pSpeckVertexDecl,
&m_pVertexDecl);
}

void QuadStack::Reset()
{
    HRESULT hr = m_pVertexBuffer->Lock(0, m_buffSize, (void**)&m_pStartVertex,
D3DLOCK_DISCARD);
    m_pNextVertex = m_pStartVertex;
    m_pEndVertex = (SpeckVertex*)(m_buffSize + (BYTE*)m_pStartVertex);
}
```

```
// Add a quad transformed by pMatrix to the quad stack.
void QuadStack::Add(D3DXMATRIX* pMatrix, TexCoordPair* pTexCoordPair, float
material)
{
    if (m_pNextVertex == m_pEndVertex)
        return;             // stack full

    Position pSrcPos[] = {
        {-1.0f, -1.0f, 0.0f},
        { 1.0f, -1.0f, 0.0f},
        { 1.0f,  1.0f, 0.0f},
        {-1.0f,  1.0f, 0.0f}
    };

    Position pDstPos[4];
    D3DXVECTOR3* pOut = (D3DXVECTOR3*)&pDstPos[0].x;
    UINT outStride = sizeof(Position);
    D3DXVECTOR3* pV = (D3DXVECTOR3*)&pSrcPos[0].x;
    UINT vStride = sizeof(Position);

    D3DXVec3TransformCoordArray(pOut, outStride, pV, vStride, pMatrix, 4);

    // Upper triangle
    m_pNextVertex->pos = pDstPos[0];
    m_pNextVertex->texCoord.u = pTexCoordPair->t1.u;
    m_pNextVertex->texCoord.v = pTexCoordPair->t2.v;
    m_pNextVertex->material = material;
    ++m_pNextVertex;

    m_pNextVertex->pos = pDstPos[1];
    m_pNextVertex->texCoord.u = pTexCoordPair->t2.u;
    m_pNextVertex->texCoord.v = pTexCoordPair->t2.v;
    m_pNextVertex->material = material;
    ++m_pNextVertex;

    m_pNextVertex->pos = pDstPos[2];
    m_pNextVertex->texCoord.u = pTexCoordPair->t2.u;
    m_pNextVertex->texCoord.v = pTexCoordPair->t1.v;
    m_pNextVertex->material = material;
    ++m_pNextVertex;

    // Lower triangle
    m_pNextVertex->pos = pDstPos[0];
    m_pNextVertex->texCoord.u = pTexCoordPair->t1.u;
    m_pNextVertex->texCoord.v = pTexCoordPair->t2.v;
    m_pNextVertex->material = material;
    ++m_pNextVertex;

    m_pNextVertex->pos = pDstPos[2];
    m_pNextVertex->texCoord.u = pTexCoordPair->t2.u;
    m_pNextVertex->texCoord.v = pTexCoordPair->t1.v;
    m_pNextVertex->material = material;
    ++m_pNextVertex;

    m_pNextVertex->pos = pDstPos[3];
    m_pNextVertex->texCoord.u = pTexCoordPair->t1.u;
    m_pNextVertex->texCoord.v = pTexCoordPair->t1.v;
    m_pNextVertex->material = material;
    ++m_pNextVertex;
}
```

```
void QuadStack::GetTriList(IDirect3DVertexBuffer9** ppVertexBuffer,
IDirect3DVertexDeclaration9** ppVertexDecl, int* pStride, int* pNumPrims)
{
    m_pVertexBuffer->Unlock();

    *ppVertexBuffer = m_pVertexBuffer;
    m_pVertexBuffer->AddRef();

    *ppVertexDecl = m_pVertexDecl;
    m_pVertexDecl->AddRef();

    *pStride = sizeof(SpeckVertex);
    *pNumPrims = (m_pNextVertex - m_pStartVertex) / 3;  // three vertices per
triangle
}


//
// Texture Atlas
//

class TexAtlasCoord
{
public:
    void Initialize(int texWidth, int subTexWidth);
    void ComputeCoord(int index, TexCoordPair* pTexCoordPair);

private:
    int m_texWidth;
    int m_subTexWidth;
    int m_numCols;
    float m_fNumCols;
    float m_fNumRows;
};

void TexAtlasCoord::Initialize(int texWidth, int subTexWidth)
{
    m_texWidth = texWidth;
    m_subTexWidth = subTexWidth;
    m_numCols = texWidth / subTexWidth;
    m_fNumRows = m_fNumCols = float(m_numCols);
}

void TexAtlasCoord::ComputeCoord(int index, TexCoordPair* pTexCoordPair)
{
    float row = float(index / m_numCols);
    float col = float(index % m_numCols);
    pTexCoordPair->t1.u = col / m_fNumCols;
    pTexCoordPair->t1.v = row / m_fNumRows;
    pTexCoordPair->t2.u = (col + 1.0f) / m_fNumCols;
    pTexCoordPair->t2.v = (row + 1.0f) / m_fNumRows;
}


//
// Specks
//

Specks::Specks() : m_pEffect(0), m_pSpecksTexture(0), m_pSpecksRenderTarget(0),
    m_pTextureAtlas(0), m_pTexAtlasCoord(0), m_pQuadStack(0)
{
    memset(&m_pSpeckType, 0, sizeof(m_pSpeckType));
}
```

```
Specks::~Specks()
{
    Cleanup();
}


// Allocate GPU resources
void Specks::Initialize(IDirect3DDevice9* pD3DDevice, int videoWidth, int
videoHeight, D3DXMATRIX* viewProj)
{
    // Store of view X projection matrix
    m_viewProj = *viewProj;

    // Create surface for rendering imperfections to
    HRESULT hr;
    hr = pD3DDevice->CreateTexture(videoWidth, videoHeight, 1,
D3DUSAGE_RENDERTARGET,
                                   D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT,
&m_pSpecksTexture, NULL);

    hr = m_pSpecksTexture->GetSurfaceLevel(0, &m_pSpecksRenderTarget);

    m_pQuadStack = new QuadStack;
    m_pQuadStack->Initialize(pD3DDevice);

    m_pParticleSet = new ParticleSet;

    //
    // Load the Specks effect
    //
    WCHAR pPath[MAX_PATH];
    DXUTFindDXSDKMediaFileCch(pPath, MAX_PATH, SPECKS_FX);

    LPD3DXBUFFER  pErrors;
#ifdef DEBUG_SHADER
    hr = D3DXCreateEffectFromFile(pD3DDevice, pPath, NULL, NULL,
                D3DXSHADER_DEBUG | D3DXSHADER_SKIPOPTIMIZATION,
                NULL, &m_pEffect, &pErrors);
#else
    hr = D3DXCreateEffectFromFile(pD3DDevice, pPath, NULL, NULL,
                NULL, NULL, &m_pEffect, &pErrors);
#endif

    // Load the texture atlas
    DXUTFindDXSDKMediaFileCch(pPath, MAX_PATH, TEXTURE_ATLAS);
    hr = D3DXCreateTextureFromFile(pD3DDevice, pPath, &m_pTextureAtlas);
    m_pTexAtlasCoord = new TexAtlasCoord;
    m_pTexAtlasCoord->Initialize(TEXTURE_ATLAS_WIDTH, TEXTURE_ATLAS_SUB_WIDTH);

    //
    // Imperfections
    //
    m_pSpeckType = new SpeckType[g_numSpeckTypes];

    // Hair/Fiber
    const float pi = 3.14159f;
    SpeckType& fiber = m_pSpeckType[ST_FIBER];
    fiber.m_enable = false;
    fiber.m_probBlack = 0.5f; // 50% black, 50% white
    fiber.m_atlasBaseIndex = 0;
    fiber.m_atlasNumTextures = 50;
    fiber.m_scaleProbExp = 10.0f; // linear size distribution
```

```
    fiber.m_scaleX.m_min = 0.25f;
    fiber.m_scaleX.m_max = 4.0f;
    fiber.m_scaleAspect.m_min = 0.1f;
    fiber.m_scaleAspect.m_max = 1.0f;
    fiber.m_allowFlipAboutX = true;
    fiber.m_rotAboutX.m_min = -pi/4.0f; // -45 degrees
    fiber.m_rotAboutX.m_max = pi/4.0f; // +45 degrees
    fiber.m_allowFlipAboutY = true;
    fiber.m_rotAboutY.m_min = -pi/4.0f; // -45 degrees
    fiber.m_rotAboutY.m_max = pi/4.0f; // +45 degrees
    fiber.m_rotAboutZ.m_min = 0.0f;
    fiber.m_rotAboutZ.m_max = 2.0f * pi; // +360 degrees

    // Dust
    SpeckType& dust = m_pSpeckType[ST_DUST];
    dust.m_enable = false;
    dust.m_probBlack = 0.5f; // 50% black, 50% white
    dust.m_atlasBaseIndex = 70;
    dust.m_atlasNumTextures = 30;
    dust.m_scaleProbExp = 10.0f; // linear size distribution
    dust.m_scaleX.m_min = 0.1f;
    dust.m_scaleX.m_max = 1.0f;
    dust.m_scaleAspect.m_min = 0.75f;
    dust.m_scaleAspect.m_max = 1.0f;
    dust.m_allowFlipAboutX = true;
    dust.m_rotAboutX.m_min = -pi/8.0f; // -22 degrees
    dust.m_rotAboutX.m_max = pi/8.0f; // +22 degrees
    dust.m_allowFlipAboutY = true;
    dust.m_rotAboutY.m_min = -pi/8.0f; // -22 degrees
    dust.m_rotAboutY.m_max = pi/8.0f; // +22 degrees
    dust.m_rotAboutZ.m_min = 0.0f;
    dust.m_rotAboutZ.m_max = 2.0f * pi; // +360 degrees

    // Particle
    SpeckType& particle = m_pSpeckType[ST_PARTICLE];
    particle.m_enable = false;
    particle.m_probBlack = 0.5f; // 50% black, 50% white
    particle.m_atlasBaseIndex = 50;
    particle.m_atlasNumTextures = 20;
    particle.m_scaleProbExp = 2.0f; // linear size distribution
    particle.m_scaleX.m_min = 0.0f;
    particle.m_scaleX.m_max = 1.0f;
    particle.m_scaleAspect.m_min = 1.0f;    // N/A
    particle.m_scaleAspect.m_max = 1.0f;    // N/A
    particle.m_allowFlipAboutX = false;     // N/A
    particle.m_rotAboutX.m_min = 0.0f;      // N/A
    particle.m_rotAboutX.m_max = 0.0f;      // N/A
    particle.m_allowFlipAboutY = false;     // N/A
    particle.m_rotAboutY.m_min = 0.0f;      // N/A
    particle.m_rotAboutY.m_max = 0.0f;      // N/A
    particle.m_rotAboutZ.m_min = 0.0f;      // N/A
    particle.m_rotAboutZ.m_max = 0.0f;      // N/A
}


// Delete resources
void Specks::Cleanup()
{
    SAFE_RELEASE(m_pSpecksRenderTarget);
    SAFE_RELEASE(m_pSpecksTexture);
    SAFE_DELETE(m_pQuadStack);
    SAFE_DELETE(m_pParticleSet);
    SAFE_RELEASE(m_pEffect);
```

```
    SAFE_RELEASE(m_pTextureAtlas);
    SAFE_DELETE(m_pTexAtlasCoord);
    SAFE_DELETE(m_pSpeckType);
}

void Specks::GenerateFrame(IDirect3DDevice9* pD3DDevice, IDirect3DTexture9**
ppSpecksTexture, CFilmSettings* pFilmSettings)
{
    SpeckType& fiber = m_pSpeckType[ST_FIBER];
    if (pFilmSettings->m_fiberFrequency > 0.0f)
    {
        fiber.m_enable = TRUE;
        fiber.m_frequency = pow(pFilmSettings->m_fiberFrequency, 2.5f) *
16.0f;
        fiber.m_probBlack = pFilmSettings->m_fiberColor;
        fiber.m_scaleX.m_max = .5f + (2.0f * pFilmSettings->m_fiberSize);
    }
    else
    {
        fiber.m_enable = FALSE;
    }

    SpeckType& dust = m_pSpeckType[ST_DUST];
    if (pFilmSettings->m_dustFrequency > 0.0f)
    {
        dust.m_enable = TRUE;
        dust.m_frequency = pow(pFilmSettings->m_dustFrequency, 2.5f) *
280.0f;
        dust.m_probBlack = pFilmSettings->m_dustColor;
        dust.m_scaleX.m_max = .1f + (.9f * pFilmSettings->m_dustSize);
    }
    else
    {
        dust.m_enable = FALSE;
    }

    SpeckType& particle = m_pSpeckType[ST_PARTICLE];
    if (pFilmSettings->m_scratchFrequency > 0.0f)
    {
        particle.m_enable = TRUE;
        particle.m_frequency = pow(pFilmSettings->m_scratchFrequency, 2.5f)
* 6.0f;
        particle.m_probBlack = pFilmSettings->m_scratchColor;
        particle.m_scaleX.m_min = 0.0f;
        particle.m_scaleX.m_max = pFilmSettings->m_scratchSize;
    }
    else
    {
        particle.m_enable = FALSE;
    }

    m_pQuadStack->Reset();

    for (int speckTypeIdx = 0; speckTypeIdx < g_numSpeckTypes; ++speckTypeIdx)
    {
        SpeckType& speckType = m_pSpeckType[speckTypeIdx];
        if (!speckType.m_enable)
            continue;    // skip if not enabled

        int numRolls = 1;

        // compute probability of speck this frame.  can be > 1.0.
```

```
            float probOfSpeck = (speckType.m_frequency) / 24.0f;
            // Even though probability of speck this frame can be 1.0, we
            // don't want to use this because this will result in exactly
            // one speck per frame.  Instead we want to increase number of
            // "rolls of the dice" while decreasing the odds proportionately.
            const float probLimit = .5f;      // TODO: fine tune this number
            if (probOfSpeck > probLimit)
            {
                  float rolls = ceil(probOfSpeck / probLimit);
                  numRolls = int(rolls);
                  probOfSpeck = probOfSpeck / rolls;
            }

            // now roll the dice numRolls times to determine if we draw
            for (int rollIdx = 0; rollIdx < numRolls; ++rollIdx)
            {
                  if (!Probability(probOfSpeck))
                        continue;

                  if (ST_PARTICLE == speckTypeIdx)
                  {
                        // multi-frame scratch type speck
                        Particle* pNewParticle = m_pParticleSet->NewParticle();
                        if (0 == pNewParticle)
                        {
                              continue;
                        }

                        Range tStart;
                        tStart.m_min = 0.0f;
                        tStart.m_max = 1.0f / 24.0f;
                        pNewParticle->m_timeStart = Random(tStart);  // secs

                        Range tStop;
                        tStop.m_min = (1.0f / 16.0f); // live for 1/16 sec
minimum
                        tStop.m_max = 4.0f;       // maximum of 4.0 seconds
                        pNewParticle->m_timeStop = pNewParticle->m_timeStart +
RandomExpDist(tStop, 10.0f); // secs

                        pNewParticle->m_color = 1.0f;
                        if (Probability(speckType.m_probBlack))
                              pNewParticle->m_color = 0.0f;

                        pNewParticle->m_size =
RandomExpDist(speckType.m_scaleX, speckType.m_scaleProbExp);
                        pNewParticle->Seed();
                        pNewParticle->m_textureIndex = RandomIndex(0,
speckType.m_atlasNumTextures);
                  }
                  else
                  {
                        // regular one-frame-only speck

                        // Determine random scale

                        float sx = RandomExpDist(speckType.m_scaleX,
speckType.m_scaleProbExp);
                        float sy = Random(speckType.m_scaleAspect); // compute
aspect ratio
                        sy = sx * sy;  // using aspect ration, compute scale
for Y
                        D3DXMATRIX mat1;
```

```
                        D3DXMatrixScaling(&mat1, sx, sy, 1.0f); // range 0.05 -
0.5

                        // Determine random rotations

                        // The flip concept is designed to allow limiting
rotation and thus
                        // prevent aliasing when rendering edge on.
                        const float pi = 3.14159f;
                        float yaw = 0.0f;
                        if (speckType.m_allowFlipAboutX && Probability(.5f))
                              yaw = pi;
                        yaw += Random(speckType.m_rotAboutX);

                        float pitch = 0.0f;
                        if (speckType.m_allowFlipAboutY && Probability(.5f))
                              pitch = pi;
                        pitch += Random(speckType.m_rotAboutY);

                        // rotate about Z (roll) doesn't cause edge-on cases
                        float roll = Random(speckType.m_rotAboutZ);
                        D3DXMATRIX mat2;
                        D3DXMatrixRotationYawPitchRoll(&mat2, yaw, pitch,
roll);

                        // Determine random translation
                        const Range trans = {-2.2f, 2.2f};
                        float tx = Random(trans);       // range is -2 to 2;
we add a little to allow hanging off edge
                        float ty = Random(trans);
                        D3DXMATRIX mat3;
                        D3DXMatrixTranslation(&mat3, tx, ty, 0.0f);

                        D3DXMATRIX mat4;
                        D3DXMatrixMultiply(&mat4, &mat1, &mat2);
                        D3DXMatrixMultiply(&mat1, &mat4, &mat3);

                        TexCoordPair texCoordPair;
                        int textureIndex =
RandomIndex(speckType.m_atlasBaseIndex, speckType.m_atlasNumTextures);
                        m_pTexAtlasCoord->ComputeCoord(textureIndex,
&texCoordPair);

                        float material = 1.0f;
                        if (Probability(speckType.m_probBlack))
                              material = 0.0f;

                        m_pQuadStack->Add(&mat1, &texCoordPair, material);
                  }
            }
      }


      //
      // Generate scratches
      //
      m_pParticleSet->ResetIterator();
      Particle* pParticle = m_pParticleSet->GetNextParticle();
      while (pParticle)
      {
            float scaleX = 1.0f;
            int isChangingTexture = 1;  // standard particle changes texture
```

```
            if (pParticle->m_size < .20)
            {
                  // When the particle size gets really small, we don't really
want to
                  // have small square textures approaching infinite in number.
Instead
                  // we want to have a maximum number of particle steps and as
we approach
                  // the maximum, we want to switch to the solid scratch size
and scale
                  // just the x dimension.

                  pParticle->m_textureIndex = 0;    // use the solid line
texture
                  isChangingTexture = 0;     // don't let the texture change
                  scaleX = .5 + (2.0f * pParticle->m_size);           //
scaleX will range from .5 to 1.0
            }
            // determine number of particle steps based on particle size
            float factor = 1.0f - pParticle->m_size;       // number of particle
steps is inversely proportional to particle size
            float delta = float(g_maxParticleSteps - g_minParticleSteps);
            int numParticleSteps = g_minParticleSteps + int(factor * delta);

            // compute the timestep based on the number of particle steps.
            // we assume that the frame is 1/24 sec.  should work just fine for
30 fps too.
            float timeStep = (1.0f / 24.0f) / numParticleSteps;

            // determine final x and y scale of particle
            // The height of the frame is 4.0f.  The Quad model Y values range
from -1 to 1
            // Thus when we scale by 2.0f, this results in a 4.0f high model
            float scaleY = 2.0f / numParticleSteps;
            scaleX = scaleX * scaleY;

            // Determine scale transform
            D3DXMATRIX mat1;
            D3DXMatrixScaling(&mat1, scaleX, scaleY, 1.0f);

            // determine random verticle offset to make it look more like a
            // particle is bouncing around while scratching.
            Range offsetRange;
            offsetRange.m_min = 0.0f;
            offsetRange.m_max = 4.0f / float(numParticleSteps);
            float ofsY = Random(offsetRange);

            // Compute initial texture coordinates
            TexCoordPair texCoordPair;
            m_pTexAtlasCoord-
>ComputeCoord(m_pSpeckType[ST_PARTICLE].m_atlasBaseIndex + pParticle-
>m_textureIndex, &texCoordPair);

            // we always render one extra particle because we start rendering
off-frame by ofsY
            int step = 0;
            for (step = 0; step < (numParticleSteps + 1); ++step)
            {
                  if (pParticle->m_timeCurrent > pParticle->m_timeStart)
                  {
                        // Determine translation
```

```
                        // TOOD: should 2.0f - (scaleY) actually be 2.0f *
(scaleY)?
                        // TODO: optimized below
                        float ty = (float(step) * (-4.0f /
float(numParticleSteps))) + 2.0f - (scaleY);       // y goes from 2 to -2
                        float tx = pParticle->ComputePosition(pParticle-
>m_timeCurrent);
                        float ty1 = (float(step+1) * (-4.0f /
float(numParticleSteps))) + 2.0f - (scaleY);       // y goes from 2 to -2
                        float tx1 = pParticle->ComputePosition(pParticle-
>m_timeCurrent + timeStep);
                        D3DXMATRIX mat3;
                        D3DXMatrixTranslation(&mat3, tx, ofsY + ty, 0.0f);

                        // Determine rotation
                        const float pi = 3.14159f;
                        float roll = (pi / 2.0f) + atan2(ty1 - ty, tx1 - tx);
// rotate about Z (roll)
                        D3DXMATRIX mat2;
                        D3DXMatrixRotationYawPitchRoll(&mat2, 0.0f/*yaw*/,
0.0f/*pitch*/, roll);

                        D3DXMATRIX mat4;
                        D3DXMatrixMultiply(&mat4, &mat1, &mat2);
                        D3DXMatrixMultiply(&mat2, &mat4, &mat3);

                        // Determine texture index
                        if (isChangingTexture && Probability(.20))
                        {
                              int index = pParticle->m_textureIndex;
                              index += (Probability(.5)) ? 1 : -1;
                              int atlasNumTextures =
m_pSpeckType[ST_PARTICLE].m_atlasNumTextures;
                              index = (index + atlasNumTextures) %
atlasNumTextures;
                              pParticle->m_textureIndex = index;
                              m_pTexAtlasCoord-
>ComputeCoord(m_pSpeckType[ST_PARTICLE].m_atlasBaseIndex + index,
&texCoordPair);
                        }

                        // Add rectangle to the stack
                        m_pQuadStack->Add(&mat2, &texCoordPair, pParticle-
>m_color);
                  }
                  pParticle->m_timeCurrent += timeStep;

                  if (pParticle->m_timeCurrent > pParticle->m_timeStop)
                  {
                        m_pParticleSet->RemoveParticle();
                        break;
                  }
            }

            pParticle = m_pParticleSet->GetNextParticle();
      }


      HRESULT hr;
      hr = pD3DDevice->SetRenderTarget(0, m_pSpecksRenderTarget);
      pD3DDevice->Clear(0, 0, D3DCLEAR_TARGET, 0x00000000, 0, 0); // clear render
target to black
```

```
    IDirect3DVertexBuffer9* pVertexBuffer;
    IDirect3DVertexDeclaration9* pVertexDecl;
    int stride;
    int numPrims;
    m_pQuadStack->GetTriList(&pVertexBuffer, &pVertexDecl, &stride, &numPrims);

    if (0 < numPrims)
    {
        hr = pD3DDevice->SetStreamSource(0, pVertexBuffer, 0, stride);
        hr = pD3DDevice->SetVertexDeclaration(pVertexDecl);
        hr = m_pEffect->SetTechnique("BlendImperfection");
        hr = m_pEffect->SetTexture("ImperfectionTex", m_pTextureAtlas);

        UINT uPasses;
        hr = m_pEffect->Begin(&uPasses, 0);
        hr = m_pEffect->SetMatrix("WorldViewProj", &m_viewProj);
        hr = m_pEffect->BeginPass(0);
        hr = pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, numPrims);
        hr = m_pEffect->EndPass();
        hr = m_pEffect->End();
    }

    pVertexBuffer->Release();
    pVertexDecl->Release();

    // Return the completed specks texture
    *ppSpecksTexture = m_pSpecksTexture;
    m_pSpecksTexture->AddRef();
}


float Specks::Random(const Range& r)
{
    float ratio = float(rand()) / float(RAND_MAX);
    return  (ratio * (r.m_max - r.m_min)) + r.m_min;
}

int Specks::RandomIndex(int base, int number)
{
    float ratio = float(rand()) / float(RAND_MAX);
    return (base + int((float)number * ratio));
}

#if 0
// Returns true 1 in range_max times
bool Specks::Probability(int range_max)
{
    float ratio = float(rand()) / float(RAND_MAX);
    return  (0 == int(ratio * float(range_max)));
}
#endif

// Returns true based on probability fn P(n). 0 <= n <= 1
bool Specks::Probability(float n)
{
    // prevent divide by zero
    if (n < 0.00001) // if 1 in 100,000
        return false;

    float range_max = 1.0f / n;
    if (range_max > float(RAND_MAX))
        range_max = float(RAND_MAX);
```

```
    float ratio = float(rand()) / float(RAND_MAX);
    return (0 == int(ratio * float(range_max)));
}

// Exponential distribution skews results towards range_min.
// Good values for exponent are:
//   1.3 yields ~59% results in the lower half of range
//   1.5 yields ~64% results in the lower half of range
//   2.0 yields ~70% results in the lower half of range
//   2.5 yields ~75% results in the lower half of range
float Specks::RandomExpDist(const Range& r, float exponent)
{
    float ratio = float(rand()) / float(RAND_MAX);
    ratio = pow(ratio, exponent);
    return  (ratio * (r.m_max - r.m_min)) + r.m_min;
}
```

[0075] An embodiment of the invention also preferably employs advanced deinterlacing and framerate re-sampling using true motion estimation vector fields. The preferred True Motion Estimator (TME) of an embodiment of the invention is a hierarchical and multipass method. It preferably takes as input an interlaced video stream. The images are typically sampled at regular forward progressing time intervals (e.g., 60Hz). The output of the TME preferably comprises a motion vector field (MVF). This is optionally a 2D array of 2-element vectors of pixel offsets that describe the motion of pixels from one video frame (or field) image to the next. The application of motion offsets to a video frame, where time=n-1, will produce a close approximation of the video frame at time=n. The motion offsets can be scaled by a blendFactor to achieve a predicted frame between the frames n-1 and n. For example if the blendFactor is .25, and the motion vectors in the field are multiplied by this factor, then the resulting predicted frame is 25% away from frame n-1 toward n. Varying the blend factor from 0 to 1 can cause the image to morph from frame n-1 to the approximate frame n.

[0076] Framerate resampling is the process of producing a new sequence of images that are sampled at a different frequency. For example, if the original video stream was sampled at 60Hz and you want to resample to 24Hz, then every other frame in the new sequence lies halfway between two fields in the original sequence (in the temporal domain). You can use a TME MVF and a blend factor to generate a frame at the precisely desired moment in the time sequence.

[0077] An embodiment of the present invention optionally uses a slight temporal offset of ¼ of 1/24 of a second in its resampling from 60 interlaced to 24 progressive. This generates a new sampling pattern where the blendfactor is always .25 or .75. In this embodiment, the present invention preferably generates reverse motion vectors (i.e., one runs the TME process backwards as well as forwards). When the sampling is .75 between two fields, use the reverse motion vectors and a blend factor of .25. The advantage of this approach is that one is never morphing more than

25% away from an original image. This results in less distortion. An excellent background in true motion estimation and deinterlacing is given by E.B. Bellers and G. de Haan, *De-interlacing: A Key Technology for Scan Rate Conversion* (2000).

[0078] Field offsetting and smoothing is preferably done as follows. A video field image contains the odd or even lines of a video frame. Before an odd video field images can be compared to an even field image, it must be shifted up or down by a slight amount (usually a ½ pixel or ¼ pixel shift) to account for difference in spatial sampling. The invention shifts both fields by an equal amount to align spatial sampling and to degrade both images by the same amount (resampling changes the frequency characteristics of the resulting image).

[0079] Near horizontal lines in the original field usually exhibit quite noticeable aliasing artifacts. These artifacts may cause problems with the motion finding process and may produce false motion vectors. At the same time or at substantially the same time that the video fields are re-sampled to fix spatial alignment, high-frequency smoothing is preferably also applied to reduce the effect of aliasing.

[0080] In addition to the color channels of the image, it is preferred to add a fourth channel that is the edge map of the image. The edge map values can be computed from the sum of the horizontal and vertical gradients (sum of dx and dy) across about three pixels. Any edge image processing, such as sobel edge detector, will work. The addition of this edge map improves the motion vectors by adding an additional cost when edges don't align during the motion finding. This extra penalty helps assure that the resulting motion vectors will map edges to edges.

[0081] In computing the TME for one image pair, denoted I(n-1) for image at time=n-1 and I(n), the motion estimation algorithm is performed on different sized levels of the image pair. The first step in the algorithm is to resize the interlaced image I(n-1) to one-half size in each dimension. The process is repeated until one has a final image that is only a pixel in size. This is sometimes called an image pyramid. In the current instance of the preferred method, one gets excellent results with only the first four levels.

[0082] It is preferred to perform the motion estimation on smaller sizes because it more efficiently detects large scale motion, or global motion, such as camera panning, rotations, zoom, and large objects moving fast. The motion that is estimated on a smaller image is then used to seed the algorithm for the next sized image. The motion estimation is repeated for the larger sized images and each step adds finer grain detail to the motion vector field. The process is repeated until the motion vector field for the full size images is computed.

[0083] The actual motion finding is preferably done using blocks of pixels (this is a configurable parameter, in one instance of the invention it is set to 8x8 pixel blocks). In this embodiment, the algorithm sweeps over all the blocks in the previous image I(n-1) and searches for a matching block in the current image I(n). The search for a block can be done by applying a small offset to the block of pixels and computing the Sum of the Absolute Differences (SAD) metric to evaluate the match. The offsets are selected from a set of candidate vectors. Candidate vectors can be chosen from neighboring motion vectors in the previous iteration (spatial candidate), from the smaller image motion vectors (global motion candidate), from the previous motion vector (temporal candidate). The candidate set is further extended by applying a random offset to each of the candidate vectors in the set. Each offset vector in the final candidate set preferably has a cost penalty associated with it. This is done to shape the characteristics of the resulting motion vector field. For example, if we want a smoother motion field we lower the penalty for using spatial candidates. If one wants smoother motion over time, lower the penalty for temporal candidates.

[0084] Preferred code for the advanced deinterlacing and framerate re-sampling using true motion estimation vector fields method of the invention next follows:

```
class MotionEstimator
{
public:
    MotionEstimator();
    virtual ~MotionEstimator();

    // Allocate resources
    void Initialize(IDirect3DDevice9* pD3DDevice, int videoWidth, int videoHeight,
D3DXMATRIX* pWorldToProjection);

    void GetMotionImageSize(int* pMotionWidth, int* pMotionHeight);

    void BuildMotionVectors(IDirect3DDevice9* pD3DDevice, Frame* pField);

    Frame* GetMotionVectors(IDirect3DDevice9* pD3DDevice);

    IDirect3DTexture9* GetDebugMotionTexture(IDirect3DDevice9* pD3DDevice, float
blendFactor);

    // Delete resources
    void Cleanup();
…
};


const int g_numLevels = 4;
const int g_queueLength = 4;
const int g_blockWidth = 8;
const int g_blockHeight = 8;

// Dimenion (width & height) of the texture of random update vectors
// Each texel contains two update vectors.  Dimension should always
// be >= max(image dimension) / blocksize; otherwise adjacent pixels
// will get the same 'random' update vectors
int g_randomTextureDim = 90;  // 90 == 720 / 8

…

…
```

```
void MotionEstimator::BuildMotionVectors(IDirect3DDevice9* pD3DDevice, Frame* pField)
{
    HRESULT hr;
    hr = pD3DDevice->SetStreamSource(0, m_pVertexBuffer, 0, sizeof(MotionEstimatorVertex));
    assert(hr == S_OK);
    hr = pD3DDevice->SetVertexDeclaration(m_pVertexDecl);
    assert(hr == S_OK);

    Frame** ppImages = new Frame*[m_numLevels];
    Frame** ppPrevImages = new Frame*[m_numLevels];
    Frame** ppMotions = new Frame*[m_numLevels];
    Frame** ppPrevMotions = new Frame*[m_numLevels];

    FrameIter* pFrameIter;
    for (int levelIndex = 0; levelIndex < m_numLevels; ++levelIndex)
    {
        Level& level = m_pLevels[levelIndex];
        pFrameIter = level.m_imageQueue.GetFront();
        pFrameIter->Prev();    // backup to n-1 frame
        ppPrevImages[levelIndex] = pFrameIter->Get();
        ppImages[levelIndex] = pFrameIter->Get();

        pFrameIter = level.m_motionQueue.GetFront();
        pFrameIter->Prev();    // backup to n-1 frame
        ppPrevMotions[levelIndex] = pFrameIter->Get();
        ppMotions[levelIndex] = pFrameIter->Get();
    }

    // store the incoming frame number and odd/even attribute in the level 0 motion image
    ppMotions[0]->m_frameNumber = pField->m_frameNumber;
    ppMotions[0]->m_imageType = pField->m_imageType;

    // Convert the incoming field to a full sized frame and store it in level 0.
    BOOL isOddField = (pField->m_imageType == IT_ODD_FIELD);
    Doubler(pD3DDevice, pField->m_pTexture, isOddField, ppImages[0]->m_pRenderTarget);
    //CGPUUtil::DumpFrameTag(ppImages[0]->m_pTexture, L"C:\\temp\\TME\\level0Image_");

    // Compute the edges of level 0 image
    int srcWidth = m_pLevels[0].m_imageWidth;
    int srcHeight = m_pLevels[0].m_imageHeight;
    ComputeEdges(pD3DDevice, ppImages[0]->m_pTexture, srcWidth, srcHeight, ppImages[0]-
>m_pRenderTarget);
    //CGPUUtil::DumpFrameTag(ppImages[0]->m_pTexture, L"C:\\temp\\TME\\level0Edge_");

    // Compute half sizes for each level
    for (int levelIndex = 1; levelIndex < m_numLevels; ++levelIndex)
    {
        // downsize image
        srcWidth = m_pLevels[levelIndex-1].m_imageWidth;
        srcHeight = m_pLevels[levelIndex-1].m_imageHeight;
        HalfSize(pD3DDevice, ppImages[levelIndex-1]->m_pTexture, srcWidth, srcHeight,
ppImages[levelIndex]->m_pRenderTarget);
        //CGPUUtil::DumpFrameTag(ppImages[levelIndex]->m_pTexture,
L"C:\\temp\\TME\\levelImage_");

        // compute edges for this level's image
        srcWidth = m_pLevels[levelIndex].m_imageWidth;
        srcHeight = m_pLevels[levelIndex].m_imageHeight;
        ComputeEdges(pD3DDevice, ppImages[levelIndex]->m_pTexture, srcWidth, srcHeight,
ppImages[levelIndex]->m_pRenderTarget);
        //CGPUUtil::DumpFrameTag(ppImages[levelIndex]->m_pTexture,
L"C:\\temp\\TME\\levelEdge_");

        // downsize the previous frames motion vector field
        srcWidth = m_pLevels[levelIndex].m_motionWidth;
        srcHeight = m_pLevels[levelIndex].m_motionHeight;
        HalfSize(pD3DDevice, ppPrevMotions[levelIndex-1]->m_pTexture, srcWidth, srcHeight,
            ppPrevMotions[levelIndex]->m_pRenderTarget, 0.5f);    // scale values by 1/2
        //CGPUUtil::DumpFrameTag(ppPrevMotions[levelIndex]->m_pTexture,
L"C:\\temp\\TME\\levelMotion_");
```

```
    }


    // Compute motion vectors for each image level
    for (int levelIndex = m_numLevels-1; levelIndex >= 0; --levelIndex)
    {
        Level& level = m_pLevels[levelIndex];

        // Set the current image
        hr = m_pEffect->SetTexture("g_imageTexture", ppImages[levelIndex]->m_pTexture);
        assert(hr == S_OK);

        // Set the previous image
        hr = m_pEffect->SetTexture("g_prevImageTexture", ppPrevImages[levelIndex]-
>m_pTexture);
        assert(hr == S_OK);

        // Set the previous motion
        hr = m_pEffect->SetTexture("g_prevMotionTexture", ppPrevMotions[levelIndex]-
>m_pTexture);
        assert(hr == S_OK);

        // Set the global motion
        if (levelIndex == (m_numLevels - 1))
        {
            hr = m_pEffect->SetBool("g_globalMotionValid", false);
            assert(hr == S_OK);
        }
        else
        {
            hr = m_pEffect->SetBool("g_globalMotionValid", true);
            assert(hr == S_OK);
            hr = m_pEffect->SetTexture("g_globalMotionTexture", ppMotions[levelIndex+1]-
>m_pTexture);
            assert(hr == S_OK);
        }

        // Apply multiple passes to find optimal solution
        //
        bool toggle = true;
        for (int pass = 0; pass < 9; ++pass)
        {
            // Setup motion image render target and current motion image
            if (toggle)
            {
                toggle = false;
                // For even numbered passes the current motion image is the render target.
                // Final pass must go through here; otherwise need to copy the results
                V( pD3DDevice->SetRenderTarget(0, ppMotions[levelIndex]->m_pRenderTarget) );

                // Set the current motion texture
                if ((0 == levelIndex) && (pass > 0))
                {
                    V( m_pEffect->SetTexture("g_motionTexture", m_pMotionTexture) );
                }
                else
                {
                    V( m_pEffect->SetTexture("g_motionTexture", ppPrevMotions[levelIndex]-
>m_pTexture) );
                }
            }
            else
            {
                toggle = true;
                if (0 == levelIndex)
                {
                    V( pD3DDevice->SetRenderTarget(0, m_pMotionRenderTarget) );
                }
                else
                {
                    // For odd number passes the previous motion image is the render target.
```

```
                    // This is safe because the previous motion image is used only in Pass 0
                    V( pD3DDevice->SetRenderTarget(0, ppPrevMotions[levelIndex]-
>m_pRenderTarget) );
                }
                // Set the current motion
                V( m_pEffect->SetTexture("g_motionTexture", ppMotions[levelIndex]-
>m_pTexture) );
            }

            // Set globals
            float ofsX = 1.0f / level.m_imageWidth;
            float ofsY = 1.0f / level.m_imageHeight;

            D3DXVECTOR4 imageTexelBox1UV(0.0f, 0.0f, ofsX, 0.0f);
            D3DXVECTOR4 imageTexelBox2UV(0.0f, ofsY, 0.0f, ofsY);
            ofsX += ofsX; ofsY += ofsY;
            D3DXVECTOR4 imageTexelRightUV(ofsX, 0.0f, ofsX, 0.0f);
            D3DXVECTOR4 imageTexelDownUV(0.0f, ofsY, 0.0f, ofsY);

            hr = m_pEffect->SetVector("g_imageTexelBox1UV", &imageTexelBox1UV);
            assert(hr == S_OK);
            hr = m_pEffect->SetVector("g_imageTexelBox2UV", &imageTexelBox2UV);
            assert(hr == S_OK);
            hr = m_pEffect->SetVector("g_imageTexelRightUV", &imageTexelRightUV);
            assert(hr == S_OK);
            hr = m_pEffect->SetVector("g_imageTexelDownUV", &imageTexelDownUV);
            assert(hr == S_OK);

            // g_texelCenterUV:
            //   (x,y) = (.5 / imageWidth), (.5 / imageHeight)
            //   (z,w) = (.5 / motionWidth), (.5 / motionHeight)
            D3DXVECTOR4 texelCenterUV((0.5f / level.m_imageWidth), (0.5f /
level.m_imageHeight),
                    (0.5f / level.m_motionWidth), (0.5f / level.m_motionHeight));
            hr = m_pEffect->SetVector("g_texelCenterUV", &texelCenterUV);
            assert(hr == S_OK);

            if (0 == pass)
            {
                hr = m_pEffect->SetTechnique("ComputeBlockTMEPass0");
                assert(hr == S_OK);
            }
            else
            {
                // Set the random offsets to the random update vector texture
                SetRandomSeed();

                hr = m_pEffect->SetTechnique("ComputeBlockTMEPass1");
                assert(hr == S_OK);
            }

            UINT uPasses;
            hr = m_pEffect->Begin(&uPasses, 0);
            assert(hr == S_OK);
            hr = m_pEffect->BeginPass(0);
            assert(hr == S_OK);
            hr = pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 2);
            assert(hr == S_OK);
            hr = m_pEffect->EndPass();
            assert(hr == S_OK);
            hr = m_pEffect->End();
            assert(hr == S_OK);
        }
    }

    //CGPUUtil::DumpFrameTag(ppMotions[0]->m_pTexture, L"C:\\temp\\TME\\level0Motion_");

    SAFE_DELETE_ARRAY(ppImages);
    SAFE_DELETE_ARRAY(ppPrevImages);
    SAFE_DELETE_ARRAY(ppMotions);
    SAFE_DELETE_ARRAY(ppPrevMotions);
```

```
}

Frame* MotionEstimator::GetMotionVectors(IDirect3DDevice9* pD3DDevice)
{
    assert(m_pLevels);
    FrameIter* pFrameIter = m_pLevels[0].m_motionQueue.GetEnd();
    return pFrameIter ->Get();
}



// In: color fields
// Out: line doubled, converted to grayscale
void MotionEstimator::Doubler(IDirect3DDevice9* pD3DDevice, IDirect3DTexture9* pSrcTexture,
            BOOL isOddField, IDirect3DSurface9* pDstRenderTarget)
{
    HRESULT hr = pD3DDevice->SetRenderTarget(0, pDstRenderTarget);
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetTechnique("Doubler");
    assert(hr == S_OK);

    D3DXVECTOR4* pTexCoordOfsUV;
    if (isOddField)
        pTexCoordOfsUV = &m_oddTexCoordOfsUV;
    else
        pTexCoordOfsUV = &m_evenTexCoordOfsUV;
    hr = m_pEffectUtil->SetVector("g_texCoordOfsUV", pTexCoordOfsUV);
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetTexture("g_imageTexture", pSrcTexture);
    assert(hr == S_OK);

    UINT uPasses;
    hr = m_pEffectUtil->Begin(&uPasses, 0);
    assert(hr == S_OK);
    hr = m_pEffectUtil->BeginPass(0);
    assert(hr == S_OK);
    hr = pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 2);
    assert(hr == S_OK);
    hr = m_pEffectUtil->EndPass();
    assert(hr == S_OK);
    hr = m_pEffectUtil->End();
    assert(hr == S_OK);
}

void MotionEstimator::HalfSize(IDirect3DDevice9* pD3DDevice, IDirect3DTexture9* pSrcTexture,
            int srcWidth, int srcHeight, IDirect3DSurface9* pDstRenderTarget, float
scaleFactor)
{
    HRESULT hr = pD3DDevice->SetRenderTarget(0, pDstRenderTarget);
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetTechnique("HalfSize");
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetFloat("g_halfSizeScaleFactor", scaleFactor);
    assert(hr == S_OK);

    // Compute offset in the middle of four texels
    float fullTexelU = 1.0f / float(srcWidth);
    float fullTexelV = 1.0f / float(srcHeight);
    D3DXVECTOR4 texCoordOfsUV(fullTexelU, fullTexelV, 0.0f, 0.0f);
    hr = m_pEffectUtil->SetVector("g_texCoordOfsUV", &texCoordOfsUV);
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetTexture("g_imageTexture", pSrcTexture);
    assert(hr == S_OK);

    UINT uPasses;
    hr = m_pEffectUtil->Begin(&uPasses, 0);
```

```
    assert(hr == S_OK);
    hr = m_pEffectUtil->BeginPass(0);
    assert(hr == S_OK);
    hr = pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 2);
    assert(hr == S_OK);
    hr = m_pEffectUtil->EndPass();
    assert(hr == S_OK);
    hr = m_pEffectUtil->End();
    assert(hr == S_OK);
}

void MotionEstimator::ComputeEdges(IDirect3DDevice9* pD3DDevice, IDirect3DTexture9*
pSrcTexture,
            int srcWidth, int srcHeight, IDirect3DSurface9* pDstRenderTarget)
{
    HRESULT hr = pD3DDevice->SetRenderTarget(0, pDstRenderTarget);
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetTechnique("ComputeEdges");
    assert(hr == S_OK);

    // Compute offset for the texel center
    float halfTexelU = 0.5f / float(srcWidth);
    float halfTexelV = 0.5f / float(srcHeight);
    D3DXVECTOR4 texCoordOfsUV(halfTexelU, halfTexelV, 0.0f, 0.0f);
    hr = m_pEffectUtil->SetVector("g_texCoordOfsUV", &texCoordOfsUV);
    assert(hr == S_OK);

    // Compute offset for filter sampling
    float fullTexelU = 1.0f / float(srcWidth);
    float fullTexelV = 1.0f / float(srcHeight);
    D3DXVECTOR4 computeEdgesOfsUV(fullTexelU, 0.0f, 0.0f, fullTexelV);
    hr = m_pEffectUtil->SetVector("g_computeEdgesOfsUV", &computeEdgesOfsUV);
    assert(hr == S_OK);

    hr = m_pEffectUtil->SetTexture("g_imageTexture", pSrcTexture);
    assert(hr == S_OK);

    UINT uPasses;
    hr = m_pEffectUtil->Begin(&uPasses, 0);
    assert(hr == S_OK);
    hr = m_pEffectUtil->BeginPass(0);
    assert(hr == S_OK);
    hr = pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 2);
    assert(hr == S_OK);
    hr = m_pEffectUtil->EndPass();
    assert(hr == S_OK);
    hr = m_pEffectUtil->End();
    assert(hr == S_OK);
}
```

--HLSL Code--

```
...

PixelShaderOutput DoublerPixelShader(VertexShaderOutput inPixel)
{
    PixelShaderOutput outPixel;

    // Approximate a gaussian filter
    float4 c;
    c = tex2D(DoublerSampler, inPixel.m_texCoordUV.xy - g_filterOfsUV.xy);
    c += 2.0f * tex2D(DoublerSampler, inPixel.m_texCoordUV);
    c += tex2D(DoublerSampler, inPixel.m_texCoordUV.xy + g_filterOfsUV.xy);
    c *= 0.25f;

    // Convert to grayscale
    c *= float4(0.3086f, 0.6094f, 0.0820f, 0.0f);
    outPixel.m_color = 0;
    outPixel.m_color.x = c.r + c.g + c.b;
```

```
    return outPixel;
}

Technique Doubler
{
    Pass P0
    {
        AlphaBlendEnable = False;
        VertexShader = compile vs_3_0 StandardVertexShader();
        PixelShader  = compile ps_3_0 DoublerPixelShader();
    }
}


//
//  HalfSize - Bilinear image reduction by 2x
//
//  The HalfSize technique uses carefully aligned texture coordinates
//  to use the bilinear filter hardware to averages 4x4 pixels into 2x2.
//
...

PixelShaderOutput HalfSizePixelShader(VertexShaderOutput inPixel)
{
    PixelShaderOutput outPixel;
    outPixel.m_color = g_halfSizeScaleFactor * tex2D(HalfSizeSampler, inPixel.m_texCoordUV);
    return outPixel;
}

Technique HalfSize
{
    Pass P0
    {
        AlphaBlendEnable = False;
        VertexShader = compile vs_3_0 StandardVertexShader();
        PixelShader  = compile ps_3_0 HalfSizePixelShader();
    }
}


//
//  ComputeEdges
//
//  The HalfSize technique uses carefully aligned texture coordinates
//  to use the bilinear filter hardware to averages 4x4 pixels into 2x2.
//
...

PixelShaderOutput ComputeEdgesPixelShader(VertexShaderOutput inPixel)
{
    PixelShaderOutput outPixel;

    float4 texCoord = inPixel.m_texCoordUV.xyxy;
    texCoord += g_computeEdgesOfsUV;    //xy: move right; zw: down
    float4 cx1 = tex2D(ComputeEdgesSampler, texCoord.xy);
    float4 cy1 = tex2D(ComputeEdgesSampler, texCoord.zw);

    texCoord += g_computeEdgesOfsUV;    //xy: move right; zw: down
    float4 cx2 = tex2D(ComputeEdgesSampler, texCoord.xy);
    float4 cy2 = tex2D(ComputeEdgesSampler, texCoord.zw);

    texCoord = inPixel.m_texCoordUV.xyxy;
    texCoord -= g_computeEdgesOfsUV;    //xy: move left; zw: up
    float4 cx3 = tex2D(ComputeEdgesSampler, texCoord.xy);
    float4 cy3 = tex2D(ComputeEdgesSampler, texCoord.zw);

    texCoord -= g_computeEdgesOfsUV;    //xy: move left; zw: up
    float4 cx4 = tex2D(ComputeEdgesSampler, texCoord.xy);
    float4 cy4 = tex2D(ComputeEdgesSampler, texCoord.zw);
```

```
    float dx = abs((cx1.x + cx2.x) - (cx3.x + cx4.x));
    float dy = abs((cy1.x + cy2.x) - (cy3.x + cy4.x));

    outPixel.m_color = tex2D(ComputeEdgesSampler, inPixel.m_texCoordUV);
    outPixel.m_color.y = (dx + dy) * 0.25f;

    return outPixel;
}

Technique ComputeEdges
{
    Pass P0
    {
        AlphaBlendEnable = False;

        VertexShader = compile vs_3_0 StandardVertexShader();
        PixelShader  = compile ps_3_0 ComputeEdgesPixelShader();
    }
}
```

--HLSL CODE—

```
//-----------------------------------------------------------------
// Vertex shader I/O structures
//-----------------------------------------------------------------

...

BlockTMEPass0Vertex BlockTMEPass0VertexShader(VertexShaderInput inVertex)
{
...
}


struct BlockTMEPass1Vertex
{
    float4 m_positionPS  : POSITION;
    float4 m_motion1UV : TEXCOORD0;        // xy:sample right, zw:sample down
    float4 m_motion2UV : TEXCOORD1;        // xy:sample left, zw:sample up
    float4 m_randomUV : TEXCOORD2;        // xy:sample from random texture
    float2 m_imageUV : TEXCOORD3;        // xy:sample from image texture
};

BlockTMEPass1Vertex BlockTMEPass1VertexShader(VertexShaderInput inVertex)
{
...
}

...


// 'base' is in image UV space
// 'offset' is in image pixel space
float ComputeSAD(in float2 base, in float2 offset)
{
    // convert from pixel space to image UV space
    // g_texelCenterUV.xy contains image 1/2 texel dimension
    offset = offset * 2.0f * g_texelCenterUV.xy;

    float4 r1 = base.xyxy + g_imageTexelBox1UV;
    float4 tr1;
    float2 sum = 0;
    for (int y = 0; y < 8; y++)
    {
        tr1 = r1;
        for (int x = 0; x < 4; x++)
        {
            sum += abs(tex2D(TMEImageSampler, tr1.xy).xy - tex2D(TMEPrevImageSampler, tr1.xy
- offset).xy);
```

```
            sum += abs(tex2D(TMEImageSampler, tr1.zw).xy - tex2D(TMEPrevImageSampler, tr1.zw
- offset).xy);
            tr1 += g_imageTexelRightUV;
        }
        r1 += g_imageTexelDownUV;
    }
    return sum.x + sum.y;
}


void BlockTMEOptimize(inout float bestSum, inout float2 bestMotionVect, in float s, in
float2 mv)
{
    if (s < bestSum)
    {
        bestSum = s;
        bestMotionVect = mv;
    }
}

// Compute penalties based on the the block width x height x number of channels
float g_penaltyTemporal = (8.0f * 8.0f * 2.0f / 256.0f) * 0.5f;
float g_penaltyRandom = (8.0f * 8.0f * 2.0f / 256.0f) * 2.0f;
bool g_globalMotionValid;

MotionShaderOutput BlockTMEPass0PixelShader(BlockTMEPass0Vertex inPixel)
{
    MotionShaderOutput outPixel;
    outPixel.m_color = float4(0.0f, 0.0f, 0.0f, 0.0f);

    float2 bestMotionVect, mv;
    float bestSum, s;

    // Get a temporal motion candidate
    bestMotionVect = GetPrevMotion(inPixel.m_motion1UV.xy);
    bestSum = ComputeSAD(inPixel.m_imageUV, bestMotionVect) + g_penaltyTemporal;

    mv = GetPrevMotion(inPixel.m_motion1UV.zw);
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyTemporal;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    mv = GetPrevMotion(inPixel.m_motion2UV.xy);
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyTemporal;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    mv = GetPrevMotion(inPixel.m_motion2UV.zw);
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyTemporal;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Get global motion candidate
    if (g_globalMotionValid)
    {
        mv = 2.0f * GetGlobalMotion(inPixel.m_globalUV);
        s = ComputeSAD(inPixel.m_imageUV, mv);
        BlockTMEOptimize(bestSum, bestMotionVect, s, mv);
    }

    // zero candidate
    mv = float2(0.0f, 0.0f);
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyTemporal;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    outPixel.m_color.rg = bestMotionVect;
    return outPixel;
}


MotionShaderOutput BlockTMEPass1PixelShader(BlockTMEPass1Vertex inPixel)
{
    MotionShaderOutput outPixel;
```

```
    outPixel.m_color = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Lookup random values
    float4 randomOfs1 = GetRandomMotion(inPixel.m_randomUV.xy);
    float4 randomOfs2 = GetRandomMotion(inPixel.m_randomUV.zw);

    float2 bestMotionVect, mv;
    float bestSum, s;

    // Get spatial motion candidate 1
    bestMotionVect = GetCurrMotion(inPixel.m_motion1UV.xy);
    bestSum = ComputeSAD(inPixel.m_imageUV, bestMotionVect);

    // Random candidate
    mv = bestMotionVect + randomOfs1.xy;
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyRandom;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Get spatial motion candidate 2
    mv = GetCurrMotion(inPixel.m_motion1UV.zw);
    s = ComputeSAD(inPixel.m_imageUV, mv);
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Random candidate
    mv = mv + randomOfs1.zw;
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyRandom;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Get spatial motion candidate 3
    mv = GetCurrMotion(inPixel.m_motion2UV.xy);
    s = ComputeSAD(inPixel.m_imageUV, mv);
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Random candidate
    mv = mv + randomOfs2.xy;
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyRandom;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Get spatial motion candidate 4
    mv = GetCurrMotion(inPixel.m_motion2UV.zw);
    s = ComputeSAD(inPixel.m_imageUV, mv);
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    // Random candidate
    mv = mv + randomOfs2.zw;
    s = ComputeSAD(inPixel.m_imageUV, mv) + g_penaltyRandom;
    BlockTMEOptimize(bestSum, bestMotionVect, s, mv);

    outPixel.m_color.rg = bestMotionVect;
    return outPixel;
}

Technique ComputeBlockTMEPass0
{
    Pass P0
    {
        AlphaBlendEnable = False;

        VertexShader = compile vs_3_0 BlockTMEPass0VertexShader();
        PixelShader  = compile ps_3_0 BlockTMEPass0PixelShader();
    }
}

Technique ComputeBlockTMEPass1
{
    Pass P0
    {
        AlphaBlendEnable = False;

        VertexShader = compile vs_3_0 BlockTMEPass1VertexShader();
        PixelShader  = compile ps_3_0 BlockTMEPass1PixelShader();
```

```
    }
}
```

[0085] Although the invention has been described in detail with particular reference to these preferred embodiments, other embodiments can achieve the same results. Variations and modifications of the present invention will be obvious to those skilled in the art and it is intended to cover in the appended claims all such modifications and equivalents. The entire disclosures of all references, applications, patents, and publications cited above are hereby incorporated by reference.

CLAIMS

What is claimed is:

1. A digital video processing method comprising the steps of:
receiving a digital video stream comprising a plurality of frames;
adding a plurality of film effects to the video stream; and
outputting the video stream with the added film effects; and
wherein for each frame the outputting step occurs within less than approximately one second.

2. The method of claim 1 wherein the adding step comprises adding at least two effects selected from the group consisting of letterboxing, simulating film grain, adding imperfections simulating dust, fiber, hair, scratches, making simultaneous adjustments to hue, saturation, brightness, and contrast and simulating film saturation curves.

3. The method of claim 2 wherein the adding step comprises simulating film saturation curves via a non-linear color curve.

4. The method of claim 2 wherein the adding step comprises simulating film grain by generating a plurality of film grain textures via a procedural noise function and by employing random transformations on the generated textures.

5. The method of claim 2 wherein the adding step comprises adding imperfections generated from a texture atlas and softened to create ringing around edges.

6. The method of claim 2 wherein the adding step comprises adding imperfections simulating scratches via use of a start time, life time, and an equation controlling a path the scratch takes over subsequent frames.

7. The method of claim 2 wherein the adding step comprises employing a stream programming model and parallel processors causing the adding step for each frame to occur in a single pass through the parallel processors.

8. The method of claim 1 additionally comprising the step of converting the digital video stream from 60 interlaced format to a deinterlaced format by loading odd and even fields from successive frames, blending using a linear interpolation factor, and, if necessary, offset sampling by a predetermined time to avoid stutter artifacts.

9. An apparatus for altering a digital image, said apparatus comprising:

an input receiving a digital image;

software embodied on a computer-readable meadium adding a plurality of film effects to the digital image;

one or more processors performing operations of the software and thus producing a resulting digital image; and

an output sending the resulting digital image within less than approximately one second from receipt of the digital image by said input.

10. The apparatus of claim 9 wherein said plurality of film effects comprises two or more elements selected from the group consisting of letterboxing, simulating film grain, adding imperfections simulating dust, fiber, hair, scratches, making simultaneous adjustments to hue, saturation, brightness, and contrast, and simulating film saturation curves.

11. The apparatus of claim 10 wherein said film saturation curves are added via a non-linear color curve.

12. The apparatus of claim 9 wherein one of said film effects comprises film grain generated a plurality of film grain textures via a procedural noise function and by employing random transformations on the generated textures.

13. The apparatus of claim 9 wherein one of said film effects comprises imperfections generated from a texture atlas of said software to create ringing around edges.

14. The apparatus of claim 9 wherein one of said film effects comprises simulation of scratches via use of a start time, life time, and an equation controlling a patch the scratch takes over subsequent frames.

15. The apparatus of claim 9 wherein said software and processors comprise a stream programming model and parallel processors causing said plurality of film effects to be added in a single pass through said parallel processors.

16. The apparatus of claim 9 wherein at least one of said processors converts said resulting digital image from 60 interlaced format to a deinterlaced format by loading odd and even fields from successive frames, blending using a linear interpolation factor, and, if necessary, offset sampling by a predetermined time to avoid stutter artifacts.

17. Computer software stored on a computer-readable medium for manipulating a digital video stream, said software comprising:

software accessing an input buffer into which at least a portion of said digital video stream is at least temporarily stored; and

software adding a plurality of film effects to at least a portion of said digital video stream within less than approximately one second.

18. The computer software of claim 17 wherein said adding software adds at least two effects selected from the group consisting of letterboxing, simulating film grain, adding imperfections simulating dust, fiber, hair, scratches, making simultaneous adjustments to hue, saturation, brightness, and contrast and simulating film saturation curves.

19. The computer software of claim 17 wherein said adding software simulates film saturation curves via a non-linear color curve.

20. The computer software of claim 17 wherein said adding software simulates film grain by generating a plurality of film grain textures via a procedural noise function and by employing random transformations on the generated textures.

21. The computer software of claim 17 wherein said adding software adds imperfections to at least a portion of said digital video stream by accessing a texture atlas to create ringing around edges.

22. The computer software of claim 17 wherein said adding software adds imperfections simulating scratches having a start time, a life time, and an equation controlling a path the scratch takes over subsequent frames.

23. The computer software of claim 17 wherein said adding software employs a stream programming model for implementation on parallel processors to allow the plurality of effects to occur in a single pass through the parallel processors.

5

24. The computer software of claim 17 additionally comprising software converting the digital video stream from 60 interlaced format to a deinterlaced format by loading odd and even fields from successive frames, blending using a linear interpolation factor, and, if necessary, offset sampling by a predetermined time to avoid stutter artifacts.

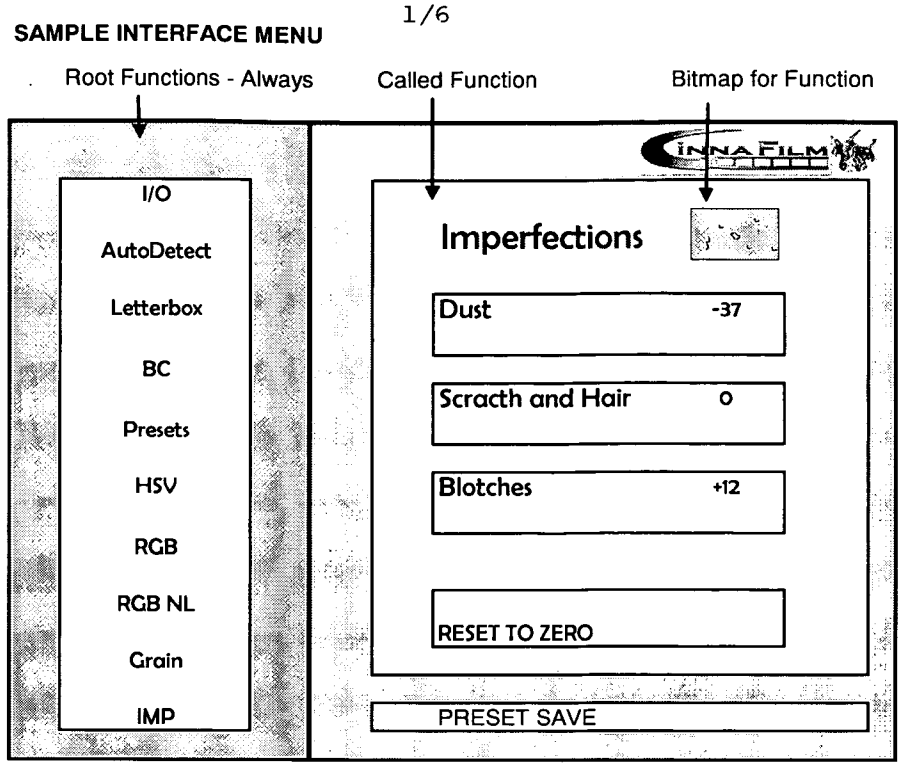

SAMPLE INTERFACE MENU
Root Functions - Always
Called Function
Bitmap for Function
I/O
AutoDetect
Letterbox
BC
Presets
HSV
RGB
RGB NL
Grain
IMP
Imperfections
Dust
-37
Scracth and Hair
0
Blotches
+12
RESET TO ZERO
PRESET SAVE

Figure 1


Cinnafilm Editor
Color Adjust
Red
Green
Blue
Brightness
Contrast
Hue
Saturate
Film Settings
Lock RGB
NL Red
NL Green
NL Blue
Grain
Correlation
Size
Video In
Video Out
Control
Dust Settings
Freq
Color
Size
Fiber Settings
Scratch Settings
Capture Control
Port 0
Port 1
File
Enter Filename here
Browse...
Start
Stop
Decode all frames
Enable TME processing
Render Control
Deck Control
Play
Pause
Resume
Frame
Max
TME
Markers
Set
Delete
Prev
Next
Output
Refresh
Pre Film
Post Film
Close
Mute
Settings
Reset
Copy
Paste
Debug Settings
Off
Show Vb
Blend Factor
Display Frames
Resolution
Cinnafilm

Figure 2

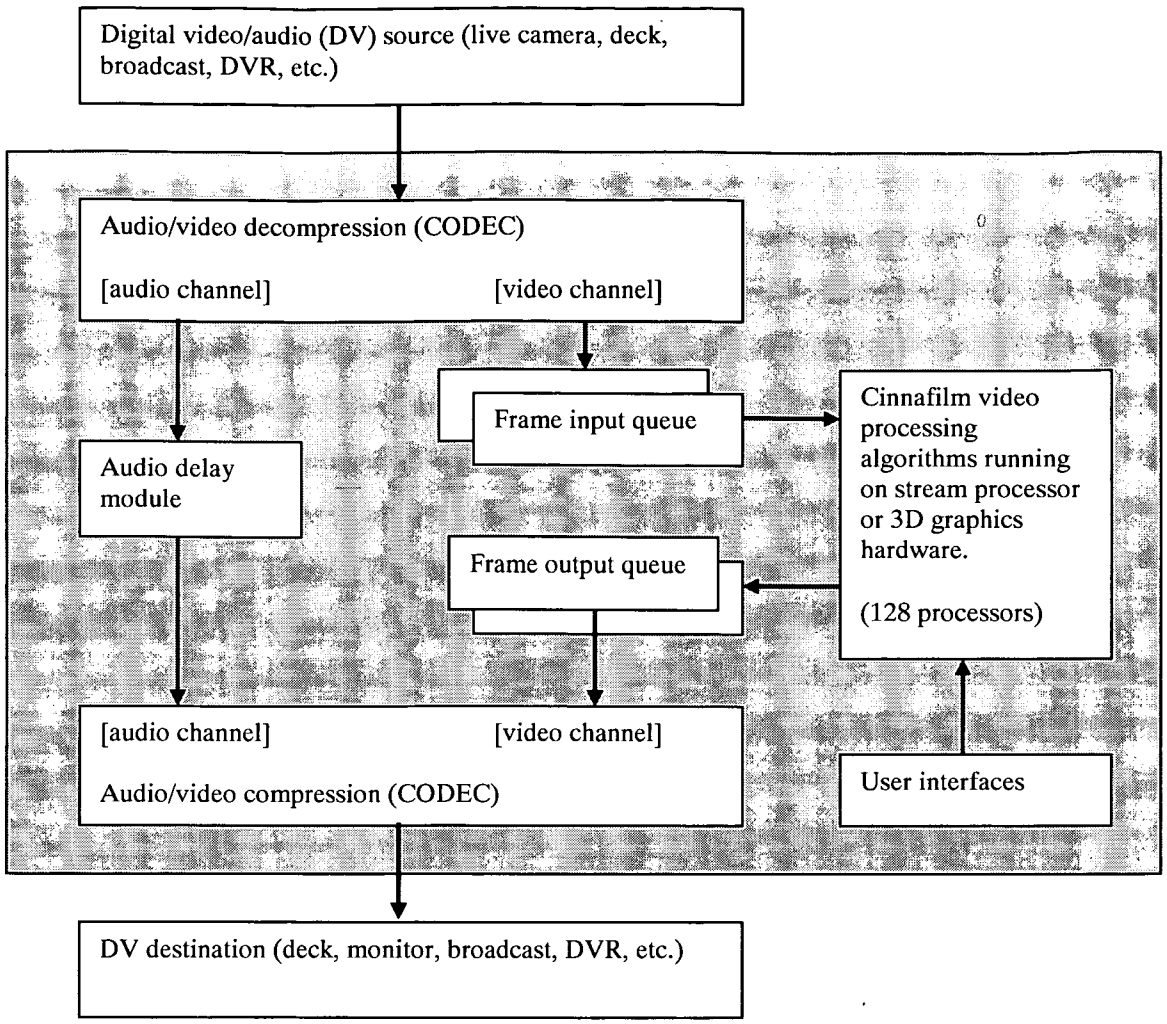

Digital video/audio (DV) source (live camera, deck, broadcast, DVR, etc.)
Audio/video decompression (CODEC)
[audio channel]
[video channel]
Frame input queue
Cinnafilm video processing algorithms running on stream processor or 3D graphics hardware.
(128 processors)
Audio delay module
Frame output queue
[audio channel]
[video channel]
Audio/video compression (CODEC)
User interfaces
DV destination (deck, monitor, broadcast, DVR, etc.)

Figure 3

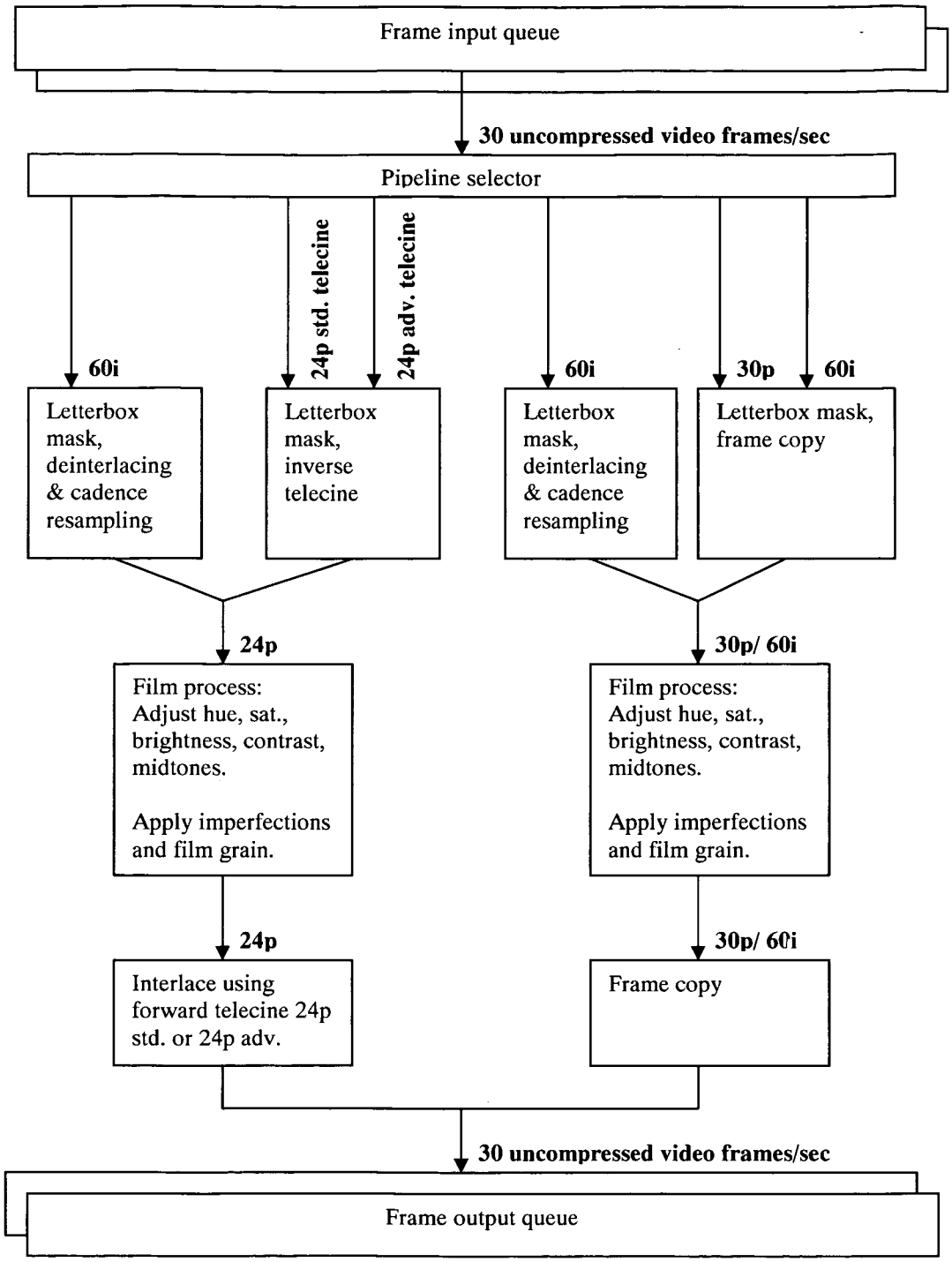

Frame input queue
30 uncompressed video frames/sec
Pipeline selector
60i
24p std. telecine
24p adv. telecine
60i
30p
60i
Letterbox mask, deinterlacing & cadence resampling
Letterbox mask, inverse telecine
Letterbox mask, deinterlacing & cadence resampling
Letterbox mask, frame copy
24p
30p/ 60i
Film process: Adjust hue, sat., brightness, contrast, midtones.
Apply imperfections and film grain.
Film process: Adjust hue, sat., brightness, contrast, midtones.
Apply imperfections and film grain.
24p
30p/ 60i
Interlace using forward telecine 24p std. or 24p adv.
Frame copy
30 uncompressed video frames/sec
Frame output queue

Figure 4

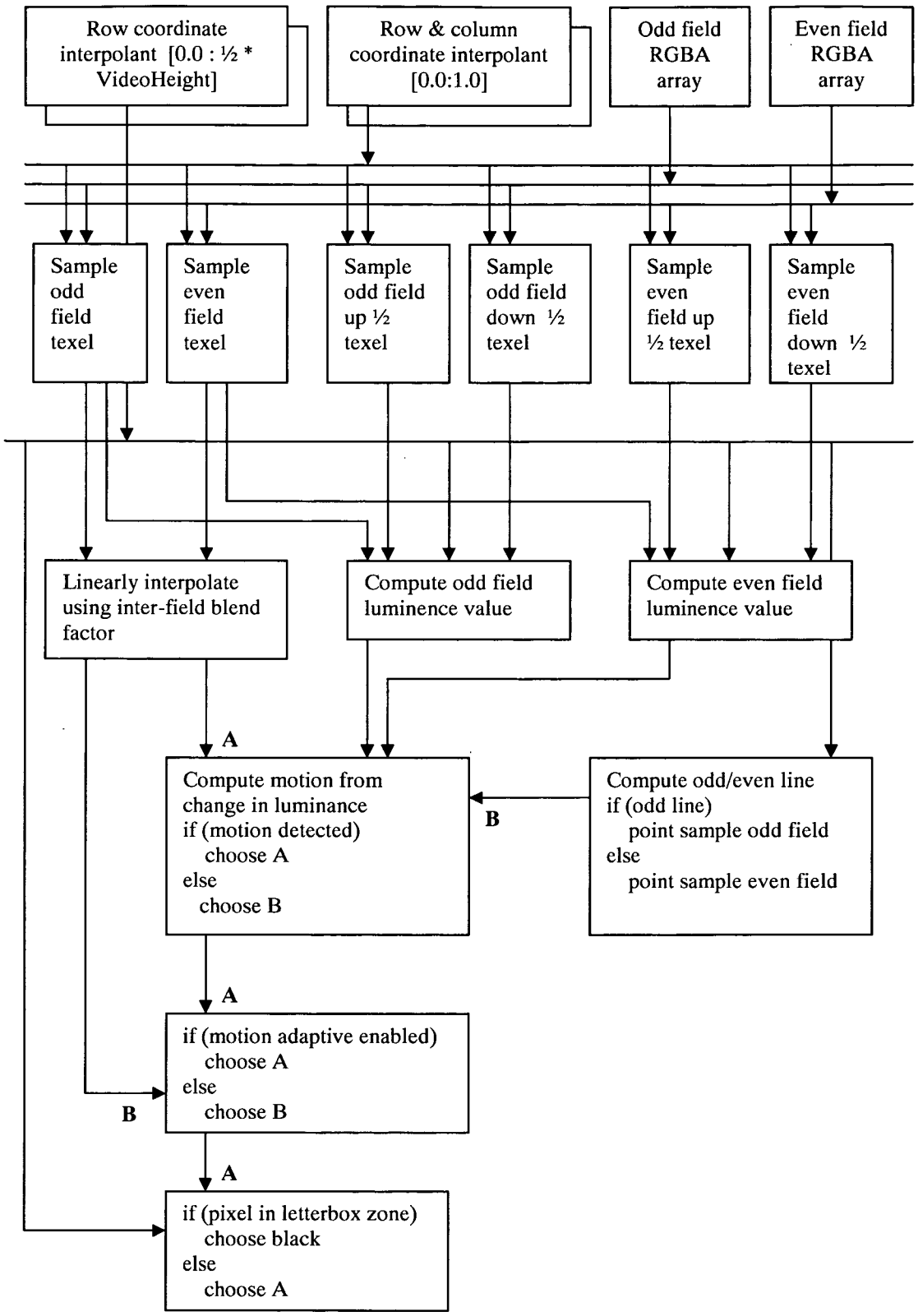

Row coordinate interpolant [0.0 : ½ * VideoHeight]
Row & column coordinate interpolant [0.0:1.0]
Odd field RGBA array
Even field RGBA array
Sample odd field texel
Sample even field texel
Sample odd field up ½ texel
Sample odd field down ½ texel
Sample even field up ½ texel
Sample even field down ½ texel
Linearly interpolate using inter-field blend factor
Compute odd field luminence value
Compute even field luminence value
A
Compute motion from change in luminance
if (motion detected)
choose A
else
choose B
B
Compute odd/even line
if (odd line)
point sample odd field
else
point sample even field
A
if (motion adaptive enabled)
choose A
else
choose B
B
A
if (pixel in letterbox zone)
choose black
else
choose A

Figure 5

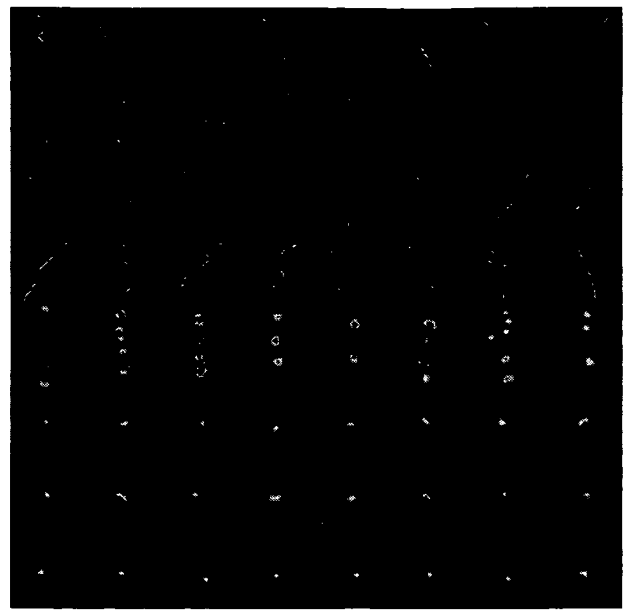


Figure 6

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2007/025305

A. CLASSIFICATION OF SUBJECT MATTER

G06T 1/00(2006.01)i, G06F 15/76(2006.01)i

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 8 G06T

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean Utility models and applications for Utility Models: IPC as above

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKIPASS(KIPO internal): "computer graphic, film effect"

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 6,268,863 B1 (MARE RIOUX) 31. Jul. 2001. See abstract, column 1, line 5 - column 3, line 37.	1-24
A	US 2005/0168463 A1 (MIGUEL A. SEPULVEDA) 4. Aug. 2005. See abstract, page 3, [0036] - page 3, [0038].	1-24
A	US 2001/0000779 A1 (YOSHIFUSA HAYAMA et al.) 3. May 2001. See abstract, page 14, [0284] - page 15, [0298].	1-24

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
14 MAY 2008 (14.05.2008)	**15 MAY 2008 (15.05.2008)**

Name and mailing address of the ISA/KR
Korean Intellectual Property Office
Government Complex-Daejeon, 139 Seonsa-ro, Seo-gu, Daejeon 302-701, Republic of Korea
Facsimile No. 82-42-472-7140

Authorized officer
CHO, Woo Yeon
Telephone No. 82-42-481-8524

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2007/025305

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US06268863	31.07.2001	CA2249132AA	02.04.1999
		CA2249132C	18.09.2007
		EP01019874A1	19.07.2000
		JP13519575	23.10.2001
		W09918542A1	15.04.1999
US20050168463A1	04.08.2005	EP01560163A2	03.08.2005
		EP01560163A3	19.07.2006
		EP01808814A1	18.07.2007
		KR1020060042886	15.05.2006
		KR2005084981A	29.08.2005
		US07236170	26.06.2007
US20010000779A1	03.05.2001	JP10198821A2	31.07.1998
		KR1019980042193	17.08.1998
		US06343987	05.02.2002
		US06738067	18.05.2004
		US20010033282A1	25.10.2001
		US20010034255A1	25.10.2001
		US6343987BA	05.02.2002