Eurographics Italian Chapter Conference (2006)
G. Gallo and S. Battiato and F. Stanco (Editors)

Adaptive Frame Rate Up-conversion with Motion Extraction from 3D Space for 3D Graphic Pipelines

M. Falchetto, M. Barone, D. Pau
AST- STMicroelectronics, Via C. Olivetti, 2, Agrate Brianza, Italy

Abstract

A novel 3D Graphic Pipeline for mobile handheld devices is introduced. It's able to achieve increased frame-rate at its output, high picture quality, improved temporal anti-aliasing, thanks to frame-rate up-conversion algorithm based on primitive's motion extraction and efficient compensation.
Classic rendering process doesn't exploit spatial and temporal coherence of the motion: each frame is rendered by using a lot of the pipeline resources inside geometry, rasteriser and fragment processing stages. Moreover each frame is computed independently from the temporal adjacent one with a brute force approach.
Video based algorithms instead (e.g. MPEG type of processing), heavily exploit temporal, spatial and statistical coherency of the content being coded, relying on motion estimation to extract motion vectors from pixel domain during coding process and apply them during motion compensation decoding process.
Previous works on 3D graphics were based on the use of impostors, sprite or coherent layers. Those approaches are too much complex and requires high computational power. Other ones based on frame averaging are too much simple: they achieve poor spatial and temporal quality and artifacts become clearly visible as the motion dynamic increases.
Proposed approach instead exploits motion temporal coherency by extracting motion vectors from motions of visible primitives in screen space. Temporal coherence has been exploited between key frames using an adaptive motion compensated stage.
This method is able to reconstruct the frames without introducing noticeable artifacts on final pictures.
Results achieved are 10-15 dB better quality than simple temporal frame average and show a natural motion with no annoying artifacts like real video content shows.

Categories and Subject Descriptors (according to ACM CCS): I.3.2, C.3 [Computer Graphics]: Real-time and embedded systems

1. Introduction

The field of real-time computer graphics is constantly pushing hardware capabilities to their limits. There is demand to increase the complexity of the models being rendered and also the resolution of the images. To provide enough computational power for interactive graphics, fully parallelized systems have been developed which are capable to render many polygons at all major stages of the graphics pipeline. Researchers of the University of North Carolina have devised a taxonomy of such architectures, naming the classes "sort-first", "sort-middle" and "sort-last" [MOL91, MCE*94]. While the latter two have been well explored and developed also into commercial products, sort-first has not, despite the fact that it offers great promise for real-time rendering.
Sort-first offers an advantage over sort-middle and sort-last since it can take advantage of the existing frame-to-frame motion coherence that is inherent in applications that try to generate natural motion for their characters; this can help to reduce the memory bandwidth and workload needed to display a series of *n* consecutive frames. Sort-first also offers an advantage over sort-last, since it does not require huge amounts of communication bandwidth to deal with pixel traffic.
Considering consumer and mobile interactive/gaming real-time graphic applications, the viewpoint usually changes very little from frame to frame, thus the on-screen distribution of primitives does not change appreciably either. As a consequence there is a high temporal coherence of the content between displayed frames. The algorithm proposed in this paper exploits this coherence, minimizing memory bandwidth, computational complexity and power consumption needed to achieve the target frame rate.

2. Traditional Graphics Pipeline

Before describing the proposed method, a short review of a traditional "pipelined" rendering process will be exposed. Nowadays, there are many different approaches that can be

used to render images, but there is typically only one choice that is applicable to commercial consumer and mobile products for real-time image generation. This is usually referred to as the "standard" or traditional graphics pipeline, which has the following major steps:

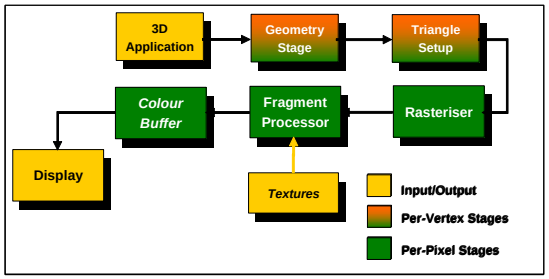

Figure 2.1: Traditional Graphics Pipeline

At the beginning of the pipeline, there is the 3D application that defines a geometric model consisting of hundreds of thousands of geometric primitives per frame represented in a model coordinate system. Primitives traditionally considered are points, lines and triangles. The model may include a set of texture images to apply to the primitives' surfaces. Usually the process of texturing is done on a per-pixel basis, with one or more specialized dedicated hardware module(s) called "texture units" [HAS93, HEC86] inside the fragment processor stage (see figure 2.1). In addition, there are information about the choice of the viewing location and its direction as well as other information that affects the generated image (such as lighting and fog information). At the end of the pipeline, millions of pixels are generated to compose the final displayed image. For a detailed discussion on the various steps, please refer to [FVF*90]. For the purposes of this description, the pipeline has been simplified into only the following steps: traversal, geometric transformation, rasterization.

Traversal is typically considered the first stage of the pipeline. It begins with the process of examining the entire graphics database and deciding which parts must be submitted to the remaining part of the pipeline. Thus, some high-level culling of primitives can be done here. Traversal may also involve converting primitives from a specific model-format to a format that is convenient to the pipeline processing.

The traversal process may either be embedded inside the 3D application stage, or it may be considered a part of the graphics subsystem itself. In the former case, the application keeps track of the model database and feeds a stream of primitives to the graphics system. This is referred to as "immediate-mode" operation. In the latter case, the graphics system itself stores the model database, allowing the application to change or edit it. This is referred as "retained-mode" operation, which is the operation mode to which proposed work is related to.

Geometric transformation involves a set of floating-point calculations that, among other things, transforms the model-space coordinates of the vertices of the primitives into a set of viewing-space coordinates. In addition, several other calculations may be performed in order to prepare the primitives for rasterization. These may include computations such as normal-vector transformation, texture-coordinate transformation, or lighting-vector calculation.

Rasterization is the process that takes a primitive defined by a set of viewing-space coordinates and raw surface features and generates a set of fragments in screen space to represent that primitive. The source data may also include a texture image that will be mapped over the surface of the primitive and used to modulate the color or other surface properties of the primitive. The amount of work required to rasterize a given primitive depends on the on-screen size of the primitive. A given primitive may be smaller than a single pixel or it may occupy the entire screen. The average size tends to vary accordingly to the overall model complexity (larger models have "smaller" primitives). Sizes from two to seventy pixels (on average) are quite common in gaming applications for PC and console graphics.

The amount of rasterization workload is also sensibly affected by the amount of anti-aliasing [FVF*90, LK00] effect desired.

Anti-aliasing refers to the correct filtering of a rendered image by taking into consideration the color contribution of each primitive at the sub-pixel level. Anti-aliasing is often done by super-sampling, where one computes multiple color samples at different places within each pixel and then correctly averages these samples together to produce the final pixel color. If four samples are computed per pixel, then four times as much rasterization workload is required, eight samples require eight times workload, etc.

In order to reach the target level of performance for the applications mentioned in the introduction, a graphics system must offer very high performance at all these major stages of the pipeline. Given the increasing processing demands of the applications and the level of performance of current processors, this implies that parallel processing must be used for both the transformation and rasterization pipeline stages.

3. Proposed Graphics Pipeline

Previous paragraph introduces, at a very high level, a traditional 3D immediate-mode pipeline.

The pipeline hereafter proposed, exploits temporal motion's coherence between adjacent frames; hence, according to the introduction, it could be defined as a sort-first retained-mode pipeline. But, at the same time, it has several stages that are usually implemented in a traditional pipeline.

As it will be shown, this pipeline still need a geometry stage (because the motion's temporal coherency is exploited in the screen space coordinates system, generating a motion vectors field). Moreover, a rasteriser and a fragment processor are needed, since a variable percentage (10% up to 50%) of the frames displayed are rendered following the traditional way. These frames, called from now on "*keyFrames*", are used as inputs of the frame rate up-conversion motion-compensation module. This module produces, using the motion field extract from the enhanced geometry stage of the pipeline the remaining frames (90% to 50%) using a low complex filtering process. Frames that are produced using the motion-

compensation stage are named, from now on, "*intFrames*".
Figure 3.1 shows the proposed pipeline; dark yellow stages are the input (3D application) and the output (Display) of the entire pipeline. Green stages are stages that a typical real-time pipeline encloses. Light grey stages are introduced to support and exploit the temporal coherence of the motion. Four new stages are defined. The first one is the "Motion Field Extraction" stage. Paragraph 3.1 describes this stage, defining its inputs and outputs. Its main goal is to extract the per-vertex motion field of the 3D Application. Second Stage (that is an optional stage), analyzes the motion field and calculate, according to the global motion of the scene, the frame rate up-conversion ratio. Third stage introduced, in paragraph 3.2 is the "Motion Field Interpolation" Stage. This stage transforms the per-vertex motion field into a per-pixel motion field to properly feed the latter stage introduced that is the "Motion Compensation Stage" (par. 3.3). Its goal is to generate high quality frames using per-pixel motion field and one or more *keyFrames* rendered using the traditional stages of the pipeline (Rasteriser+Fragment Processor). Motion of the primitives could be extracted assuming that the pipeline processes only static vertex data. In fact static vertex data persist over several frames, thus allowing the motion field extraction stage to effectively associate to each vertex its positions over several adjacent frames. For example, the "Vertex Buffer Objects" (*VBO*) as defined per openGL|ES 1.1 standard should be used. See [OGL04] for further details on *VBO*.

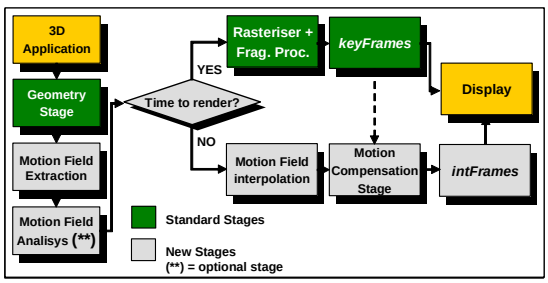

Figure 3.1: overview of the proposed pipeline

3.1. Per-vertex Motion Extraction

As briefly introduced, proposed method exploits the temporal coherence analyzing the motion of the primitives between adjacent frames. Hence, its first step consists on extracting the motion field from the scene. The scene is defined as the entire set of visible objects at any given time. The motion extraction can be done, for example, in two ways.
First one is valid when a fundamental assumption holds: when an object moves from a position A to a position B, and the distance between A and B is small enough, then the segment AB is a good approximation of the real motion (i.e. this is an assumption that every MPEG encoder uses to exploit the spatial and temporal coherency of the motion); when this assumption holds, then every point moving from position A to B moves always along straight lines. Hence only the vertices positions A and B, relative to the *keyFrames* have to be tracked: other positions are linearly interpolated onto the segment AB.
A second approach consists on tracking all the positions of the visible vertices of the entire scene, for every given frame. This approach has the advantage of knowing, at each given time, the exact position of each vertex. Hence, it is able to achieve high quality since the motion compensation stage is driven with the exact vertex position.
By using any of the two methods for motion extraction, the motion field of the scene is composed by the (x, y, z) coordinates of the visible vertices relative to a number *n* of adjacent frames. The x, y coordinates are the quantized screen-space coordinates. The z coordinate is normalized into the [0, 1] range, where 0 correspond to the *nearplane*, and 1 is the *farplane*. We used a fixed-point representation to store these data in a compact form in order to optimize the memory footprint need to store that information.

3.2. Per-pixel Motion Interpolation

The motion extraction module is in charge to compute the per-vertex motion field over a set of n consecutive frames. This motion field needs to be further processed to properly feed the motion-compensation stage that generates the motion-compensated pictures, called *intFrames*, starting from 1 or more reference previously rendered pictures, called *keyFrames*. Since the motion-compensation stage works on per-pixel basis, it is necessary to compute per-pixel motion field from the incoming per-vertex motion one. This stage is in charge to remove the vertices motion vectors associate to occluded primitives as well.
In fact the motion compensation stage works on only the visible pixels, so only per-pixel "visible" motion-vectors will feed it. The choice of dealing only with "visible" motion vectors is crucial to lower at minimum the work of the motion-compensation stage: once a pixel is colored, it will be never updated (i.e. blended) again with others: that means that the depth complexity of the motion-stage is always one: the theoretical minimum.
The motion interpolation stage produces, as output, a per-pixel motion field that is stored in a *motionBuffer*. This buffer, together with one or more reference images, will be the inputs of the next motion-compensation stage.
An efficient implementation of this stage could partially reuse part of traditional pipeline hardware: i.e. the rasteriser generates per-pixel motion field using perspective correction, to interpolate the (x, y, z) attributes. The occluded primitives are similarly discarded using an occlusion culling algorithm coupled with a Z-buffer algorithm [FVF*90].

3.3. Motion-compensation stage

This stage of the proposed pipeline produces at the output the interpolated frames or *intFrames*.
Its inputs are:

- per-pixel motion field of visible objects (stored inside the *motionBuffer*)
- One or more reference images (*keyFrames*)

Using these input data, the stage is able to create a motion-compensated frame which should replaces the frame a traditional pipeline would render at same time

location but with a much lower computational cost since rasterization renders only *keyFrames*. This stage basically computes the final color of each pixel of the *intFrame* using the information stored inside the *keyFrames* (colors in RGB format) and into the *motionBuffer* (x, y, z coordinates of the motion vectors). The *motionBuffer* itself could be thought as a grid of *voxels* associated to the reference *keyFrames*. The *(x, y, z)* coordinates, in fact, points to a 3D location *where* the pixel was at time the reference *keyFrame* was rendered.
The motion-compensation stage shows several interesting features:

1) Each pixel of the *intFrame* is processed only once while the fragment processor of a traditional pipeline, in general, processes every pixel several times before deciding the final color to be displayed.
2) Every pixel of the *intFrame* is colored once, using a non-linear, motion compensated, sub-pixel precision filtering process that computes adaptively a weighted sum of RGBA colors extract from the reference *keyFrames*.
3) Due to the type of temporal color filtering adopted, the interpolated frame is well smoothed, producing a noticeable anti-aliased effect, similar to the one obtained from an expensive full scene AntiAliasing method.

The general color filtering function [RGW92] (formula 1.0) is the weighted sum of *N* pixels of the reference *keyFrames*, selected using the per-pixel motion vectors.
The *(R,G,B)* color of a generic pixel *(x,y)* of the interpolated frame at time t, is computed as follows:

$$(R,G,B)_{(x,y)} = \sum_{i=1}^{N} \left[w_i \cdot (R,G,B)_i \right] \qquad \textbf{(1.0)}$$

Where:

$$\begin{cases} 0 \le w_i \le 1 \ \forall i = 1..N \\ \sum_{i=1}^{N} w_i = 1 \end{cases} \qquad \textbf{(1.1)}$$

And the $(R,G,B)_i$ are colors extracted from the *keyFrames* using the quantized motion vectors.

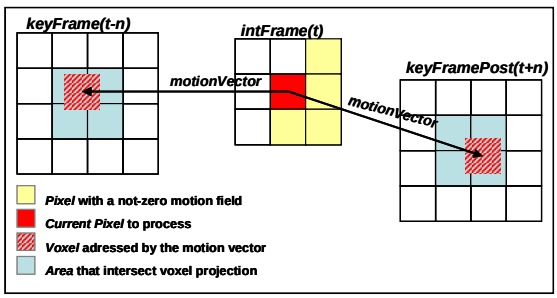

Figure 3.3.1: the filtering process

The core of the algorithm is the correct determination of the number *N* of pixels involved in the coloring phase and the correct determination of the associated weights w_i (see formula 1.0).
The number *N* of pixels depends on how many pixels intersect the projection (onto the reference keyFrame) of the *voxel* pointed by the motion vector associated to the generic *(x,y)* pixel.
The weight w_i is either zero if the pixel *i-esimo* haven't got any intersection with the *voxel* considered; or it has a value greater than zero that is inversely proportional to the distance of the *voxel* from the considered pixel, and proportional to the intersection area.
Roughly speaking, the number N characterizes the area of the reference *keyFrame* that is involved in the coloring process, whereas the weights w_i determine the exact color contribution of each pixel inside this area.

4. Experimental Results

This paragraph shows the results, in term of quality, that the proposed pipeline is able to achieve.
Throughout the simulations done, both visual and objective assessments (based on PSNR measure) were performed.
The PSNR (see [RIC03] for details), acronym of "*peak signal-to-noise ratio*", is the ratio between the maximum dynamic of the signal, squared, and the power of the noise that affects the fidelity of its representation. *PSNR* is usually expressed in terms of the decibel logarithmic scale (dB). It is widely used to numerically evaluate the "quality" of a picture.
Besides, *PSNR* is often used to compare two pictures: a reference pictures against one obtained from a filtering process.
In this case, the "reference" image is a *frame* that is produced through the traditional rendering process, during a generic time *t*. This "reference" is compared to the frame, obtained at same time *t*, but using the *motion-compensated* stage of the proposed pipeline.
In particular, in all the simulations considered, the $PSNR_Y$ of the luminance is used as a reliable measure of similarity, since the human-eye is more sensible to this channel.
Since the luminance *Y* is calculated from the *RGB* components of an image, for example, with the formula:

$$\boldsymbol{Y} = 0.299*\boldsymbol{R} + 0.587*\boldsymbol{G} + 0.114*\boldsymbol{B} \qquad \textbf{(1.2)}$$

The $PSNR_Y$ is computed as:

$$\boldsymbol{PSNR_Y} = 0.299*\boldsymbol{PSNR_R} + 0.587*\boldsymbol{PSNR_G} + 0.114*\boldsymbol{PSNR_B} \qquad \textbf{(1.3)}$$

Moreover, two criteria are used: 1st $PSNR_Y$ (formula 1.3) is computed over the entire image, 2nd on each 4x4 macroblock of the image itself.
The first criterion (named *"global PSNR"*) gives an idea of the quality over the entire images, whereas the second one (named *"macroblock PSNR"*) highlights the local artifacts eventually present on the image. The 4x4 size was chosen as the best compromise to highlight the local artifacts, still preserving global properties. Moreover, to resume the *macroblock PSNR* into a meaningful index, a proper statistic was defined: for each macroblock *m* of the image, we computed its $PSNR_Y$, and a "Quality index" was assigned following this classification:

$$\begin{cases} VERYBAD & if\ 0dB \le PSNR_Y \le 20dB \\ BAD & if\ 20db < PSNR_Y \le 25db \\ QUITEGOOD & if\ 25db < PSNR_Y \le 30db \\ GOOD & if\ 30db < PSNR_Y \le 35db \\ VERYGOOD & if\ 35db < PSNR_Y \le 40db \\ EXCELLENT & if\ 40db < PSNR_Y \le 50db \end{cases}$$

Starting from this indexing, the number of *macroblocks* for each class was counted, and the percentage of each class was organized in a 100% stacked column chart.
Several 3D Animations were tested, simulating from a single triangle, up to models composed of 9000 and more triangles.
Also the motion of the objects into the scene could vary, ranging from a simple constant, slow rotation, up to a complex series of camera movements.
The "reference" sequence is the one produced by a standard pipeline, that doesn't exploit any temporal coherence. The results obtained using the proposed motion-compensated pipeline, were compared with a simple frame averaging algorithm that doesn't use the motion-compensation. This simple algorithm is obtained setting the per-pixel motion-field always to zero, and the coloring function parameters are set to N=2 and $W_1 = W_2 =$ 0.5.
All the simulations presented hereafter are obtained using a frame rate up conversion ratio of 4:1. That means that there are 4 frames of the animation produced using the motion-compensation stage every 1 frame obtained using the standard rendering process.

Table below shows a summary of the average *global PSNR* obtained for each simulation considered.
Remarkably, on average, the gain is around 11dB. Moreover, every simulation, except the "Falling Ring" and the "Cube (Fast Rotation)" ones, has an average global PSNR that is greater than 30dB: it means that the frames are "visually" equal. Considering that the target frame rate is 30 frames per second and that consequently every frame is displayed for 1/30 sec., the difference between two animations are unnoticeable.

	Method		
Simulation	Temporal Avg.	Motion Compensated	Gain
TieFighter	22.96	36.17	+ 13.21
F14 Tomcat	23.13	34.54	+ 11.41
Cube (Fast Zoom In-Zoom Out)	20.12	33.53	+ 13.41
Cube (Fast Rotation)	16.92	29.97	+ 13.06
Cube (Slow Rotation)	22.75	31.05	+ 8.30
Pyramid (Fast Rotation)	15.77	33.56	+ 17.80
Pyramid (Slow Rotation)	22.57	31.54	+ 8.97
Rotating Wheel (In-Out Motion)	30.25	39.91	+ 9.66
Rotating Wheel (Rotation)	23.35	34.05	+ 10.69
Falling Ring	19.33	26.64	+ 7.30
Average Global PSNR:	21.72	33.10	11.38

Figure 4.1: global PSNR results

Next, the results obtained considering the "*macroblock PSNR*" measure will be shown; they will highlight the presence of local artifacts on the images considered. The charts proposed are relative to the "*TieFighter*" simulation. This simulation is the most complex considering both the motion, and the 3D model displayed onto the screen. Figure 4.2 shows the results obtained using the simple temporal averaging method. The presence of several blocks with a "*verybad*" quality is evident, on average they are 12% of the total, with 5 peaks ranging from 25% up to 40%. The peaks are relative to a period characterized by a fast motion in the scene (as shown in Figure 4.5). On average it could be noted that, most of the time, over 50% of the blocks have a quality that is at best "good"; 40% of the blocks are indexed as "greater than good". Following visual assessments, this is not enough to avoid annoying local artifacts.
Figure 4.3 is about the same simulation, but produced using the proposed motion-compensated approach.
It is self evident how the blocks with a "*verybad*" quality disappear. The blocks with a "bad" quality are still present, but they are on average less than 7% and no evident peaks appear. Moreover, there are, on average, 18% of blocks that have a quality that is at best "*good*". 82% of the blocks have a quality at least "*good*".
Figures 4.4 and 4.5 are visual assessments on two frames grabbed from the "*TieFighter*" animation. Picture on the left side is the "reference" frame, rendered using a traditional pipeline. Pictures in the center and on right side are obtained as a biased (by 128) difference between the reference image on the left and the two frames obtained using the simple non motion-compensated method (center), and the proposed motion-compensated one (on the right). Figure 4.4 shows the model near the camera, it's moving out of the screen, with a fast vertical motion. On figure 4.5, the *TieFighter* is entering from left with a fast horizontal motion. The presence of several noticeable artifacts on the pictures on center is evident.
Pictures on the right show slightly differences, most of which are located near the borders of the primitives. These differences are partly caused by the quantization of the per-pixel motion field, but also are a consequence of the particular adaptive non-linear filtering adopted in the motion-stage; this filtering shows also a desired effect: a nice anti-aliasing behavior on the final image.

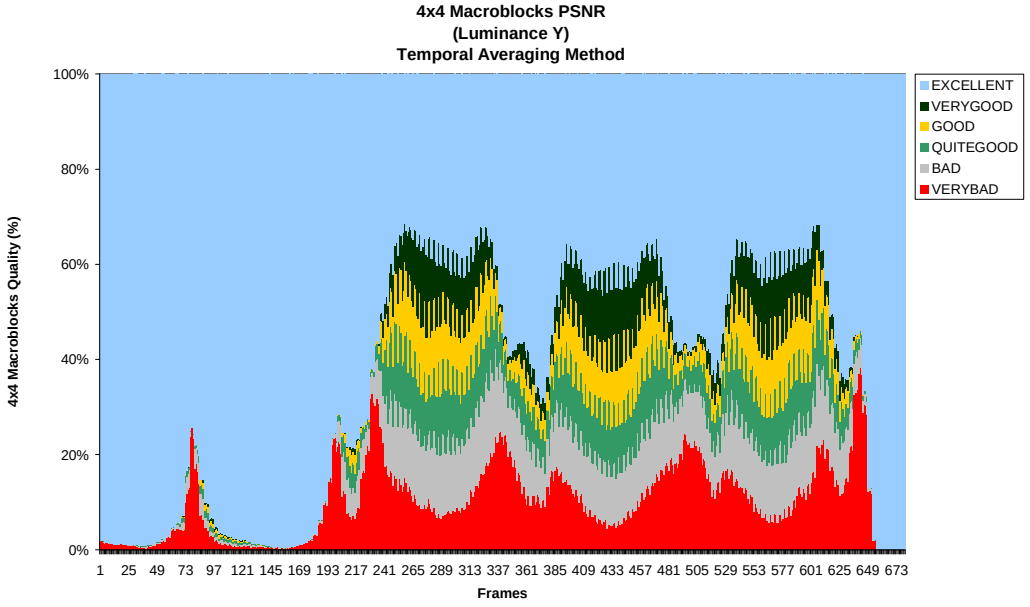

Figure 4.2: *macroBlock PSNR* results ("*TieFighter*" simulation): Temporal Averaging Method

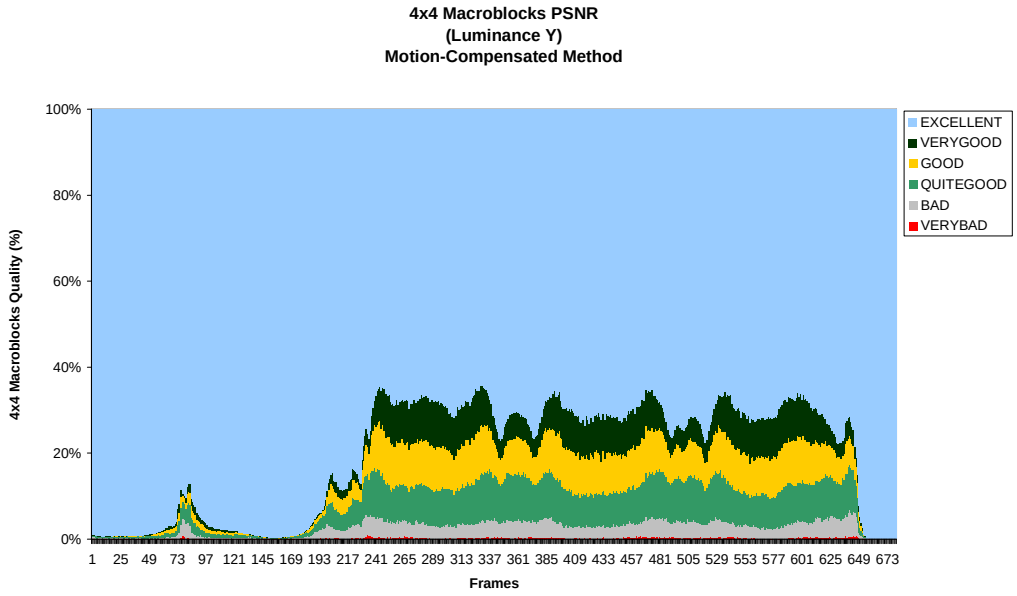

Figure 4.3: *macroBlock PSNR* results ("*TieFighter*" simulation): proposed Motion Compensated Method

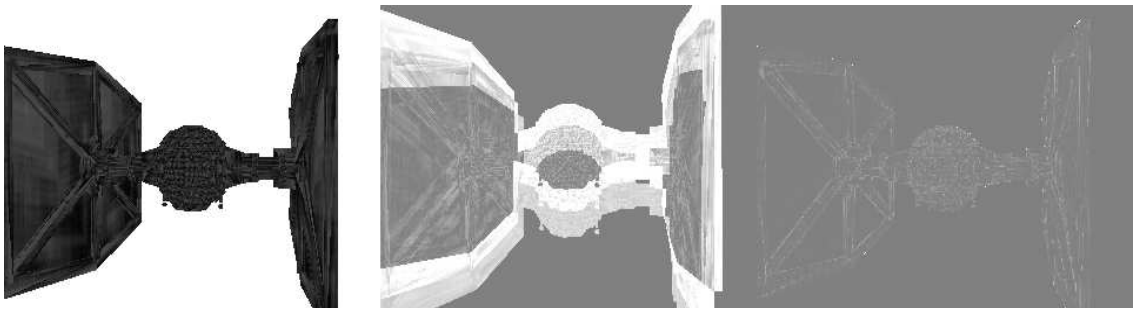

Figure 4.4: left to right: Reference image, temporal average method error, motion-compensated method error (Frame 640)

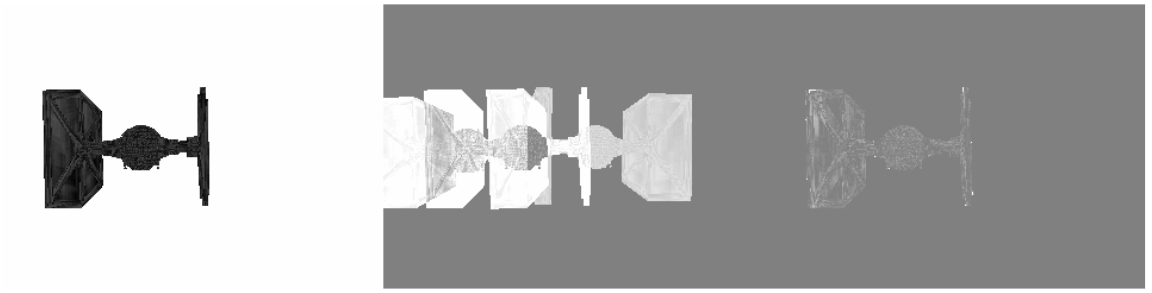

Figure 4.5: left to right: Reference image, temporal average method error, motion-compensated method error (Frame 78)

5. Conclusions

This paper describes a novel pipeline that exploits the temporal coherence between adjacent frames.
A traditional pipeline was described, at a very high level, and then the proposed pipeline was introduced, describing the functionalities of the innovative stages introduced. Thus, some criteria to evaluate the results obtained were defined and a series of simulations were introduced as well. Visual assessments and objective measures (based on PSNR) were used to analyze the capability of the proposed system.
Envisaged application of proposed method is for consumer and mobile graphics pipeline where is critical the need to lower the power consumption and the overall workload and complexity of the underlying architecture.

References

[FVF*90] FOLEY J. D., VAN DAM A., FEINER S. K, HUGHES J. F.: Computer Graphics: Principles and Practice, 2nd Ed., *Addison-Wesley Publishing Co.*, Inc., Reading, Mass., 1990.
[HAS93] HAEBERLI P., SEGAL M., Texture Mapping as a Fundamental Drawing Primitive, Silicon Graphics Computer Systems, ed. Paris, France, (Jun. 1993), pp. 259-266
[HEC86] HECKBERT P. S., Survey of Texture Mapping, IEEE Computer Graphics and Applications, (Nov. 1986) pp. 56-67
[KAH96] KAO K.R., HWANG J.J, Techniques & Standards for Image Video & Audio Coding, *ed. Prentice Hall PTR*, 1996
[LK00] LEE Jin-Aeon, KIM Lee-Sup, Single-Pass Full-Screen Hardware Accelerated Antialiasing, *Eurographics Workshop on Graphics Hardware*, (Aug. 2000), pages 67-75
[MOL91] MOLNAR S.: Image-Composition Architectures for Real-Time Image Generation, *University of North Carolina at Chapel Hill*, (Oct. 1991), doctoral dissertation, TR 91-046
[MCE*94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H. : A Sorting Classification of Parallel Rendering, *IEEE Computer Graphics & Applications*, (Jul 1994), Vol. 14, No. 4, pp. 23-32.
[MUE01] MUELLER A.S.: The Sort-First Architecture for Real-Time Image Generation, *University of North Carolina at Chapel Hill*, (Jun. 2001), doctoral dissertation
[OGL04] OpenGL|ES Common/Common-Lite Profile Specification *Version 1.1.04 (Annotated)*, www.khronos.org, Par. 2.9 Buffer Objects, pp. 11-12
[RGW92] GONZALES R. C., WOOD R. E., Digital Image Processing, *Addison-Wesley Publishing Company*, 1992
[RIC03] RICHARDSON I.E.G., H.264 and MPEG-4 Video Compression, *John Wiley & Sons Ltd., 2003, pp. 23-24*