Software Shaders in Interactive Environments Using Relief Impostors

ESTEBAN WALTER GONZALEZ CLUA[1]
BRUNO FEIJÓ[1]
MARCELO DREUX[2,1]
FRANCISCO FONSECA[1]

[1] ICAD/IGames/VisionLab, Department of Computer Science, PUC-Rio
{esteban, bruno, ffonseca}@inf.puc-rio.br

[2] Department of Mechanical Engineering, PUC-Rio
dreux@mec.puc-rio.br

Abstract. This work presents an architecture for real-time visualization, which is able to render a set of 3D objects by using the CPU idle time. The objects being rendered are sent to the graphics pipeline as relief impostors. Their depth maps are used to apply 3D image-warping operations in order to prolong the life cycle of the synthesized images. While an image is within the graphics pipeline the CPU idle time is used to generate additional necessary images. With this method, it is possible to render images with special illumination models and effects that are impossible or unsuitable to be implemented with the shaders of the available graphics cards.

Keywords: relief impostors**,** real-time rendering, image-based rendering, relief textures, 3D games.

1 Introduction

Graphics hardware is now programmable at the vertex and at the pixel level. Several illumination effects that in the past could only be implemented in the CPU are now possible to be generated in real-time by the new generation of graphics cards [Fernando, 03] [Mitchell, 02].

The graphics pipeline is increasingly focused in the GPU (Graphics Processing Unit), reducing the CPU visualization tasks. Figure 1 shows a comparison between the CPU workload taken to run a typical game application in computers with and without a graphics card. In figure 1a, without GPU, the processor is busy 95% of the time on average. In figure 1b, with GPU, only 3.5% of the CPU time is being used. In many applications, such as simulators and 3D games, some idle CPU time is being allocated to complex tasks of artificial intelligence or physical simulation, but there is still idle time available.

Nevertheless, the number of illumination models that can be integrally implemented in the graphics cards shaders is still limited. The shaders are very specialized processors, designed to deal with data streams, but limited to be modified during processing time. Furthermore, they also have some restrictions regarding memory access and limitations concerning data input and output.

Ray-tracing [Glassner 89] is, for instance, a model that cannot be implemented in graphics cards shaders, since their available programming languages do not support recursive calls. Volumetric visualization is another technique that cannot be fully implemented in GPU's. It requires access to an enormous amount of information during rendering time and these data cannot be entirely loaded in the graphics card memory.

This work presents an approach to take advantage of the CPU idle time by using it to render some objects of the scene, through illumination models that are inadequate to the GPU. With this approach, the result obtained by the CPU rendering is stored as an impostor [Maciel et al. 95].

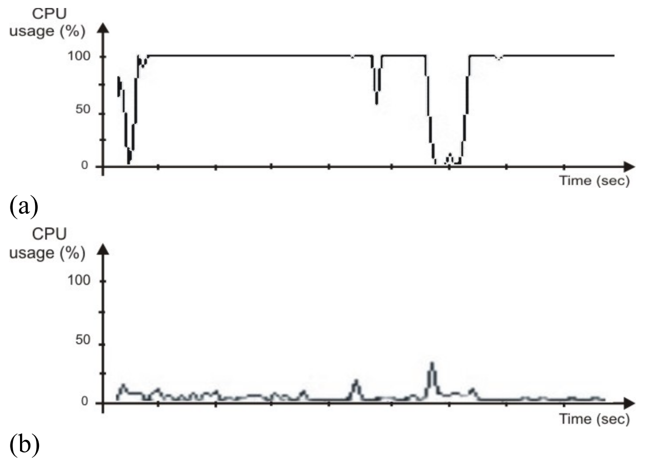

Figure 1 – CPU workload of a graphics application in a computer with same camera movements, without GPU (a) and with GPU (b).

As the frame rate of the objects rendered by the CPU is lower than the rest of the scene, rendered by the GPU, it is important to prolong, as much as possible, the life time of an impostor. In order to achieve this, image-based rendering techniques are used, more specifically the 3D warping algorithm [McMillan, 97], with the improvements described in [Oliveira, 00a]. Moreover, relief impostors store the geometry and recompute the image and the depth map in real time, when needed, and then they are sent as input to the relief texture algorithm.

Although there are many approaches for mutual cooperations in rendering processes between CPU and GPU, like [Cebenoyan, 04] and [Denny 03], most of them describe methods for GPU assuming processes of the CPU. This paper proposes an original manner of the CPU running tasks of the GPU.

2 Relief Impostors

Impostors are an efficient way to represent objects by the use of images. The idea behind it is to represent a 3D object by a sprite but, differently from the standard approach, those objects are defined as geometric models, rendered and finally projected in a plane as a texture with transparency.

Impostors are real-time generated billboards which distort the image in a way similar to what happens with the real object geometry. It is used the axial billboarding for their representation, which consists of a billboard that rotates around some fixed world space axis and aligns itself so as to face the viewer as much as possible within this range. The impostor texture can also be treated in real-time to simulate certain effects (e.g. out of focus images to simulate depth of field).

In practice, an impostor should be re-used by some, sufficiently close, point of views (exploring frame-by-frame coherence). Traditional impostors are adequate to small static objects (or sufficiently distant), while the proposed technique should also be applied to slow, distant objects. Tests to determine if a certain impostor is still valid to the current point of view are of key importance to the proposed approach.

[Schaufler, 95] and [Schaufler, 97] describe impostors as an adequate method to minimize the number of times a complex object need to be rendered, as it is computationally expensive to be generated [Forsyth, 01]. Generally speaking, impostors are a cache to those complex objects, since their visualization can be re-used while it is still valid to the current observer position.

The generated image of the objects can be proportional to their size on the screen. Thus, if they are too distant they can be rendered in low resolution. As they come closer to the observer the images should be rendered in finer resolutions. In [Damon, 03], it is described an approach to implement impostors by using video memory rendering techniques. It increases the number of objects that can be rendered, as long as enough video memory is available.

Impostors can also store the depth associated to each pixel. In that case, some authors name them as nailboards [Shaufler, 97]. Nevertheless, in this article no distinction is made between those two terms. The stored depth can be used in the pipeline z-buffer, so that an object may be inter-penetrated by other objects, and the occlusion problems are correctly treated. The depth can be stored in the alpha channel, leaving one bit to indicate if the texel is transparent or not.

Relief impostors, firstly described in [Clua, 04], make use of the 3D image warping equation, defined by [McMillan, 97]. It consists of a geometric transformation function $w{:}U' \rightarrow W \subset \boldsymbol{R}^2$ capable to map a source image I_s onto a target image i_t. The source image should also contain, besides the pixel colors, the correspondent pixel depth. Furthermore, camera information of the target image has be to known (its position $\dot{C}_s$ as well as the projection plane).

The above-mentioned equation is obtained by applying an equivalence between the two camera systems (source and target) and a point in common. [P_s , C_t] represents the source camera and [P_t , C_t] represents the target camera. The equation is given by:

$$\vec{x} \doteq P_t^{-1}P_s.\vec{x}_s + P_t^{-1}(\dot{C}_s - \dot{C}_t).\partial_s(u_s,v_s) \qquad (1)$$

where $\partial_s(u_s, v_s) = 1/t_s(u_s, v_s)$ is called generalized disparity of pixel (u_s, v_s) from the source image.

Equation 1 can be seen as a composition of two bidimensional transformations: the first term represents a homographic planar perspective transformation to the source image (can be interpreted as a texture projection) and the second term is equivalent to a per-pixel transformation, proportional to the generalized disparity (given by the term $\partial_s(u_s, v_s)$) into the direction of the epipole of the target image.

The perspective transformation term can be solved by standard implementations of hardware texture projections. The pixel-per-pixel transformation can be re-written as a one-dimensional simple structure and hence allows an efficient software implementation.

An important property related to this equation is that the pixels depth information of the new image being generated are not required when the observer moves to a different position. It is sufficient to know the pixels depth of the source image only.

Relief textures [Oliveira et al. 00b] consist of a factorization of the 3D image-warping equation into two distinct phases. The first, called pre-warping, corresponds to a per-pixel transformation proportional to the generalized disparity. The second is simply a conventional texture mapping, responsible for the perspective transformation.

The pre-warping is applied to images with depth information to each texel and is responsible for the movement (warping) of those texels. The movement is realized in a way to minimize or to correct the parallax effect, due to the observer change of position. During the pre-warping problems related to holes and texel conflicts are resolved. The second phase, texturing, performs scaling, rotation filtering and perspective deformations, which are necessary for the correct relief texture mapping

As already presented, the 3D warping equation, which describes the movement of a pixel in relation to the observer position, depends only on his final position. Therefore, by choosing an adequate orthogonal camera model, [Oliveira et al. 00b] demonstrates that several simplifications can be done to the McMillan equation. The pre-warping equation can be simplified to:

$$u_i = \frac{u_s - k_1\partial(u_s, v_s)}{1 + k_3\partial(u_s, v_s)} \qquad (2)$$

and

$$v_i = \frac{v_s - k_2\partial(u_s, v_s)}{1 + k_3\partial(u_s, v_s)} \qquad (3)$$

where $k_1 = \frac{\vec{f}.(\vec{b}\times\vec{c})}{\vec{a}.(\vec{b}\times\vec{c})}$, $k_2 = \frac{\vec{f}.(\vec{c}\times\vec{a})}{\vec{b}.(\vec{c}\times\vec{a})}$, $k_3 = \frac{\vec{f}.(\vec{a}\times\vec{b})}{\vec{c}.(\vec{a}\times\vec{b})}$, are constants that determine the amount of change in the coordinates of corresponding pixels in the images of the orthogonal and perspective projection cameras (figure 2). Formulas (2) and (3) are called pre-warping relief texture equations.

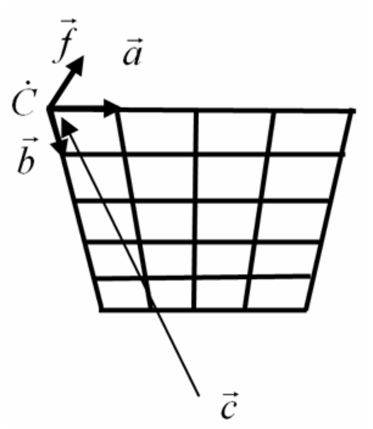

Figure 2 – Orthogonal camera model used to generate a relief impostor. The reference point $\dot{C}$ is coincident with the origin of the image plane and the unit vector $\vec{f}$ is orthogonal to that plane. $\vec{c}$ is the vector from the Center of Projection of the perspective projection camera.

Those factorizations show that, although the pre-warping phase is bidimensional, it can be treated as a unidimensional process, once the coordinates u_i and v_i are completely independent. To evaluate u_i is not necessary to know the value of v_s, and vice-versa. Thus, it is possible to produce the horizontal warping independently of the vertical warping. The warping can be initially applied to rows and later to columns. This independency allows the implementation of different versions of the pre-warping phase, as described in [Oliveira, 00a]. The more efficient one is then chosen to be part of the proposed framework. It can also be noticed that when $\partial(u_s, v_s)$ = 0 the pre-warping phase does not deform the original image and thus it becomes a standard texturing process (second phase of the relief texture mapping).

3 Visual coherence in impostors swapping

Problems related to pixel conflicts appear during the warping process, when more than one pixel is moved to the same position of the warped image. It happens because a texel can have a displacement greater than its neighbor during the pre-warping process (it can be seen in McMillan equation through the $\partial(u_s, v_s)$ parameter). This texel conflict can be solved by an adaptation of the painter algorithm, as presented in [Oliveira 00b]

Furthermore, sometimes a texel moves to a new position and no other texel occupies that position, thus holes are created. Those regions need to be filled with an appropriate color. This problem is solved by interpolating two neighbor texels. [Oliveira 00a] shows different approaches for this interpolation.

There are many ways to monitor the resultant error from that interpolation. A possible manner is to count the number of interpolated texels. It can be simultaneously done with the sampling. The ratio between the interpolated texels and the valid texels can give an idea if an impostor has too many accumulated errors. When the ratio is greater than a pre-defined value it means that the impostor is becoming obsolete and that a new impostor need to be generated to replace the current one. Figure 3 shows a region where the observer can move in order to keep the number of interpolated texels lower than 20% of the number of valid texels.

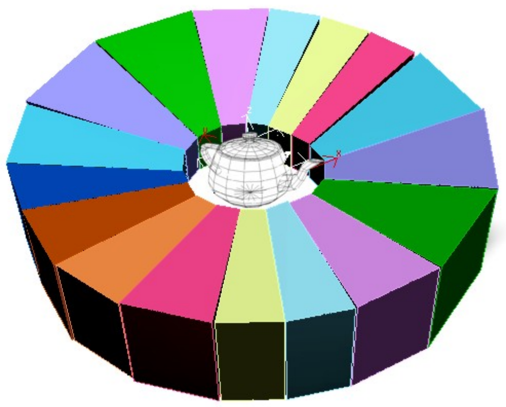

Figure 3 – Each colored area indicates a region where the observer can move in order to keep the number of interpolated texels lower than 20% of the valid texels.

This method of impostor monitoring could be called a brute-force technique, because it verifies the validity of every texel of the destination image. Since the number of texels is high, it could considerably reduce a real time system performance. Therefore, in order to optimize it, this work makes use of the critical point heuristic.

This heuristic is based on the following proposition:

Let T_1 and T_2 be two texels which belong to the same relief impostors, such that T_1 and T_2 are neighbors and $|\partial(u_1,v_1)-\partial(u_2,v_2)|$ is maximum for any pair in such situation. Let Δu_i be the displacement to be applied to the texel u_i from the source image to obtain the destination image. Then, there is a camera position C' where

$$|(u_{s_1}+\Delta u_1)-(u_{s_2}+\Delta u_2)| \tag{4}$$

is maximum to every texture [Clua, 04]. In those conditions, T_{i1} and T_{i2} define the critical point of a relief impostor for the ith region, as can be seen in the figure 4.

The heuristic test consists in an inference for the value of equation (4) for the set of critical points, one for each region of the image. If the average of those errors is greater than a pre-defined value, then it indicates that the impostor has to be updated.

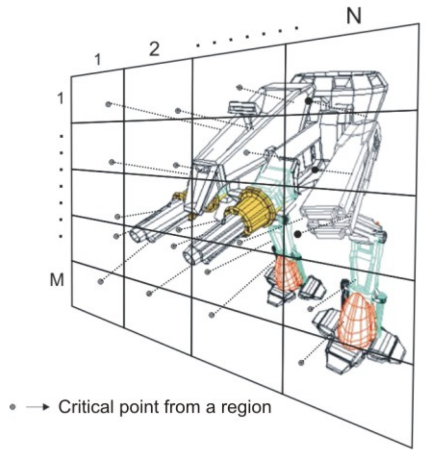

Figure 4 – An impostor divided in several regions, each one has its own critical point.

The horizontal error can be obtained by combining equations (2) and (4):

$$Error_u = \left|1+\frac{\partial(u_1,v_1)(u_e-u_{s_1})}{h+\partial(u_1,v_1)}-\frac{\partial(u_2,v_2)(u_e-u_{s2})}{h+\partial(u_2,v_2)}\right| \tag{5}$$

It should be noticed that the horizontal error is only related to the horizontal warping. Also it is necessary to evaluate the vertical error:

$$Error_v = \left|1+\frac{\partial(u_1,v_1)(v_e-v_{s_1})}{h+\partial(u_1,v_1)}-\frac{\partial(u_2,v_2)(v_e-v_{s2})}{h+\partial(u_2,v_2)}\right| \tag{6}$$

The real distance is obtained from equation (5) and (6):

$$Error = \sqrt{Error_u^2 + Error_v^2} \tag{7}$$

4 CPU rendering of impostors

The proposed system is basically composed of two threads. The first thread is called warping thread. Its function is to process the pre-warping phase of a relief impostor. This processing needs to be done in the RAM memory, since it is performed by the CPU. [Popescu 00], [Popescu 01] and [Clua 04] discuss the drawbacks of doing the warping by the graphics hardware. As soon as a pre-warping is done, the resulting image is copied to the GPU video memory, in order to perform the texturing phase of the 3D warping equation. It is important to notice that while the pre-warping is being processed the video memory has an old impostor image. Thus, if there is short available time to the warping thread, the system should abort the strategy of software rendering and perform the rendering through the standard graphics pipeline. This process management is performed by a state machine: when the new impostor is not available yet. It tells the system that, if the warping is not appropriate for a correct image, the GPU must re-render the object. Every time the rendering thread finishes a new relief impostor, the state machine sets a flag to indicate that a valid impostor is available. It is also responsible for a synchronization between the CPU and the GPU: When a camera or object movement happens, the state machine tells the warping thread to start a new pre-warping into the impostor. When it is complete, the GPU copies the result to its video buffer. Figure 5 illustrates these components.

The second thread is called rendering thread. Its function is to constantly generate more updated impostors than the one being used by the warping thread. A better impostor is obtained with a camera model closer to the current camera position.

In order to avoid generating similar relief impostors (what occurs when the observer moves slowly or the object is far from the camera), the proposed system applies the critical point heuristic to the image, before asking for a new warping. If the error is small it is not necessary to generate a new impostor and hence only a warping is performed. On the other hand, if the error is large the rendering thread process can be activated. If there is more than one object being rendered as relief impostors, it is necessary to prioritize the ones with larger errors.

The critical point heuristics may indicate that there is a large error related to the current impostor and the generation of more warpings to the image could affect the object appearance. In that case, it is necessary to change the current impostor to a new one. Nevertheless, sometimes the rendering thread has not finished synthesizing it yet. A solution to that could be to perform its rendering through the GPU. Another possibility could be a progressive rendering: an image with a lower resolution would be sent to the warping thread. As soon as a higher resolution rendering is finished the memory can be updated with this impostor.

Empirically, it can be stated that the number of necessary impostors updates is small, to keep the error relatively low. In fact, as presented in figure 3, a single relief impostor is valid to represent a relatively large region, when compared with a standard impostor [Maciel et al. 95]. In practice, in a typical graphics application, it is not usual to stay orbiting around an object. It allows that later impostors could be stored in a Database, as long as space is available. Therefore, before a rendered thread is triggered, a database search is done to check it there is a stored impostor related to the current camera position.

Either the warping thread or the rendering thread are processed during the system idle time. However, in a 3D game or in a interactive virtual environment the idle time can be useful to other types of non graphics operations and hence that time can be shorter. Thus, it is convenient to use a multi-processing system, where processors can be dedicated to both threads. This solution is economically feasible with the use of hyper-threading processors [INTEL 01]. This technology is already available in many popular computers. New game consoles, such as PlayStation 3, are also being built to support multi-threading.

As the evaluation of the second thread is entirely done by software, there is no limitation regarding the available shader programs. Therefore, the shader thread can be considered as a software shader for a specific object. In the present work, a ray-tracing software shader has been implemented, as presented in section 6.

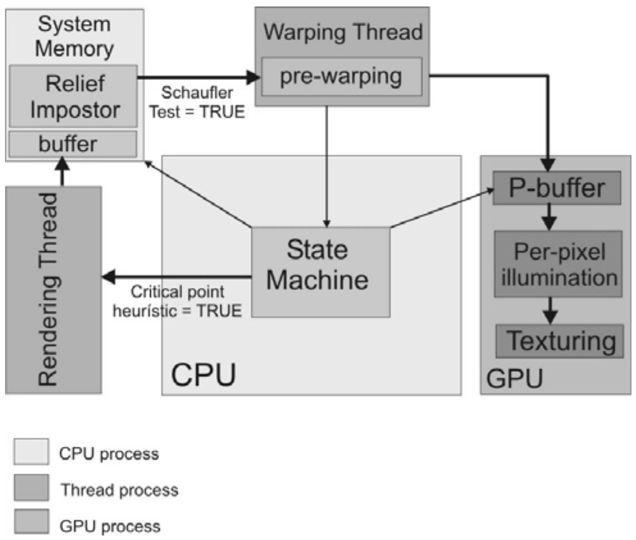

Figure 5 – Architecture of the visualization framework.

5 Optimizations

As already mentioned, an objective of this work is to make use of the CPU idle time to perform operations of the graphics pipeline. Nevertheless, this time should not be used to unnecessary tasks, such as to render impostors that can be obtained by a warping (which is cheaper) or to perform an impostor pre-warping without need (for instance, when the camera is not moving). The first situation is partially solved by the critical point heuristic, since new impostors are only generated when camera positions already have a large accumulated warping error. The second situation is handled in this work through an adaptation of the Schaufler´s approximation [Schaufler 95]. Before requesting a pre-warping in a new frame, the observer movement is analyzed to check if it was large enough to make pixels of the previous image to represent incorrect points of the new image. In the simplest situation, if the camera remains still the pre-warping would generate identical images, unnecessarily. [Clua 04a] discuss in detail how that adaptation is done.

The pre-warping process consists in evaluating, for each texel, equations (2) and (3). [Oliveira 00b] suggests using two lookup tables updated to every new camera position. This can be done because the texel depth is stored in a single byte. Thus, a texel can only have 255 depth levels. A table is, then, built for each depth level *Depth:*

```
For $Depth$ = 1 to 255 do
    $d = Depth \, . \, MAX\_Depth \, / \, 255;$
    $Coef_1 [Depth] = K_1 \, . \, d$
    $Coef_2 [Depth] = K_2 \, . \, d$
    $Coef_3 [Depth] = (1 + K_3 \, . \, d)^{-1}$
```

Where *MAX_Depth* is the maximum displacement of the relief impostor. The pre-warping is evaluated by performing the following operations:

$$u_{next} = (u + Coef_1[Depth\_Texel]) \, . \, Coef_3[Depth\_Texel]$$
$$v_{next} = (v + Coef_2[Depth\_Texel]) \, . \, Coef_3[Depth\_Texel]$$

As shown in the expressions above, this phase consists of only four elementary operations.

Another important optimization is achieved by sending data from RAM memory to GPU video memory when an impostor generation is completely done. It is important to notice that every time the GPU video memory is accessed it causes a graphics hardware processing interruption.

Finally, it is convenient to test if the rendering thread is coping with the number of impostors requested by the critical point heuristics. If not, it will be necessary to send the object rendering to the GPU, and thus losing the advantages of the software shader.

6 Results

A real time rendering framework has been developed which is able to fulfill the requirements described in sections 4 and 5. The implemented software shader is a ray-tracer for polygonal objects. These objects can be created by commercial 3D modelers.

A hyper-threading processor has been utilized, so the rendering thread and the hyper thread not only make use of the idle time but also get the benefit of a dedicated processor. There is a priority mechanism which guarantees that the warping thread is hierarchically superior to the rendering thread. This is necessary because the warping updating rate is greater than the creation rate of new impostors.

Figure 6 shows an object being rendered by the hardware and by the ray-tracing software shader. As the illumination of the first is done in a pixel-per-pixel basis, it is possible to notice an increase in its realism.

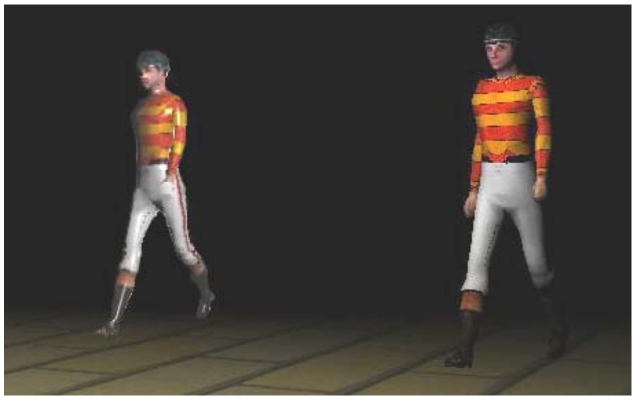
Figure 6 – The character at the left side is being rendered by the software shader and the character at the right is entirely rendered by GPU.

The time taken to obtain the warping of a 256 x 256 pixels image, in a Pentium IV 2.6GHz is 11 ms, and therefore has a rate of 91 warpings per second. On the other hand, the time taken to generate a new relief impostor is about 228 ms, for a one-level ray-traced object composed by 1500 poligons. In order to keep negligible the accumulated warping error, the heuristic value produces 17 images for a complete walk around the object.

The scene presented in figure 6 has a frame rate of 124 frames per second when completely rendered by the

GPU. On the other hand, if the software shader is used the rate is 131 frames per second. Although the difference is small, it is due to the time taken to transfer the image from RAM memory to the GPU video memory.

Figure 7 shows the CPU time for both situations.

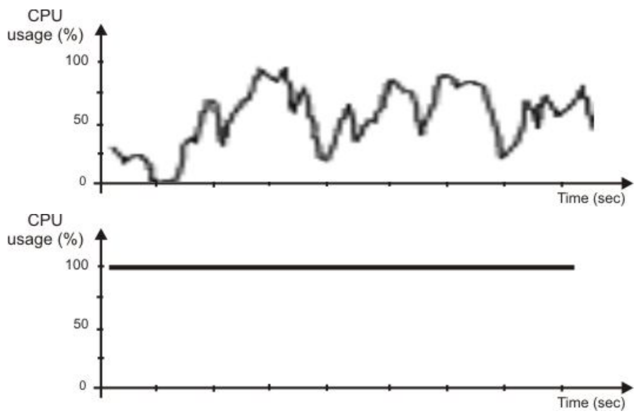

Figure 7 – CPU time for a system without (a) and with (b) software shaders.

7 Conclusion and future works

This work presents the concept of a software shader, which is able to perform rendering operations not suitable to be implemented in graphics hardware. It has been shown that shaders can take advantage of the CPU idle time, without affecting the application performance. The only additional spent time is the time of data transferring from RAM memory to GPU video memory.

In the present work, impostors are treated as static objects. Nevertheless, the described system appropriately deals with simple impostors transformations, since a camera translation could be interpreted as an object movement. However, it is necessary a deeper research in order to deal with geometric transformations of objects represented by impostors. A possible approach to tackle that problem could be the use of the view morphing algorithm [Seitz 96].

The rendering thread could be improved in the case that the critical point heuristic does not require the generation of a new impostor. In that case it should be wise to predict, and render, an impostor that could be necessary in the near future. Thus, an inference should be taken based on the object movement and/or on the observer trajectory. Also it is possible to create an efficient storage mechanism to keep impostors already used, in such a way that they could be reused.

In order to increase the processing power of the rendering and warping threads, parallel processing could be used. [Fonseca 04] explores an efficient manner to increase the performance of relief texture generation, by using more than one processor.

Finally, to corroborate the efficiency of the proposed technique, it is interesting to develop other software shaders, besides the implemented ray-tracer.

8 Acknowledgement

The authors are grateful to the important contributions given by Manuel Menezes Oliveira Neto, Waldemar Celes, Edilberto Strauss, Luis Eduardo Sauerbronn, Hélio Lopes, Cesar Tadeu Pozzer and Fábio Policarpo. The authors are also grateful to ICAD/IGames - the Intelligent CAD and Games Laboratory of PUC-Rio, to CNPq, the Brazilian government research council, and FINEP (through the VisionLab project), which partially sponsored this research.

9 References

[Cebenoyan, 04] Cebenoyan, C. Graphics Pipeline Performance. In GPU Gems – Programming Techiques, Tips and Tricks for Real-Time Graphics, p. 473-486, Addison-Wesley, March, 2004.

[Clua 04a] Clua, Esteban. Relief Impostors. Department of Computer Science, PUC-Rio, PhD thesis. April, 2004. Available from: www.icad.puc-rio/esteban/phd_thesis.pdf. (in Portuguese).

[Damon, 03] Damon, W. Impostors Made Easy. Intel Technical Report, 2003. Available from: http://www.intel.com/cd/ids/developer/asmo-na/eng/segments/20219.htm [accessed 20 May 2004].

[Denny 03] Denny, Markus. Solving Geometric Optimization Problems using Graphics Hardware. Proceedings of Eurographics 2003, p. 441-451, 2003.

[Fernando, 03] Fernando, R. and Kilgard, M. The Cg Tutorial - The definitive guide to programmable Real-Time Graphics. Addison Wesley and NVidia, Boston. 2003.

[Fonseca 04] Fonseca, Francisco. Texturas com Relevo utilizando Iluminação por Pixel e Processamento Paralelo. Department of Computer Science, Puc-Rio, Master thesis. January, 2004 (in Portuguese).

[Forsyth, 01] Forsyth, Tom. Impostors: Adding Clutter. In Mark DeLoura, ed., Game Programming Gems 2, Charles River Media, p. 488-496. 2001.

[Glassner 89] Glassner, A., S. An Introduction to Ray Tracing. Academic Press. 1989.

[INTEL 01] INTEL Corporation. Introduction to Hyper-Threading Technologies. White paper from INTEL Data Research. Document Number 250008-002. 2001.

[Maciel et al. 95] Maciel, Paulo W. and Peter Shirley, Visual Navigation of Large Environments Using Textured

Clusters, Symposium on Interactive 3D Graphics pp 95-102. April 1995.

[McMillan 97] McMillan, L. An Image-Based Approach to Three Dimensional Computer Graphics. Department of Computer Science, University of North Carolina at Chapel Hill, Ph. D. thesis. 1997.

[Mitchell 02] Mitchell, J.L. RadeonTM 9700 Shading, SIGGRAPH 2002 – State of the Art in Hardware Shading Couse Notes, 2002. Available from: www.ati.com/developer/SIGGRAPH02/ATIHardwareShading_2002_Chapter3-1.pdf [accessed 20 May 2004].

[Oliveira 00a] Oliveira, M. Relief Texture Mapping. Department of Computer Science , University of North Carolina, PhD thesis. 2000.

[Oliveira 00b] Oliveira, M., Bishop, G. and McAllister, D. Relief texture mapping. ACM SIGGRAPH Computer Graphics Proceedings, p. 359–368, November 2000.

[Popescu 00] Popescu, V., Eyles, J., et al. The WarpEngine: An Architecture for the Post-Polygonal Age. ACM SIGGRAPH Computer Graphics Proceedings, p. 433-442, July 2000.

[Popescu 01] Popescu, V. Forward Rasterization: A Reconstruction Algorithm for Image-based Rendering. Department of Computer Science, University of North Carolina, PhD thesis. 2001.

[Schaufler 95] Schaufler, G. Dynamically Generated Impostors. In: Workshop on Modeling – Virtual Worlds – Distributed Graphics, D. W. Fellner, ed., Infix Verlag, p. 129-135, November 1995.

[Schaufler 97] Schaufler, G. Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. In: Proceedings of the 8th Eurographics Workshop on Rendering. St. Ettiene, France, Springer-Verlag, p. 151-162, June 1997.

[Schaufler 98] Schaufler, G. Per-Object Image Warping with layered Impostors. In: Proceedings of the 9th Eurographics Workshop on Rendering. Vienna, Austria, p. 145-156, June 1998.

[Seitz 96] Seitz, S. and Dyer, C. View Morphing. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 21-30, August 1996.