A Parallel Approach for Visualization of Relief Textures

Francisco Fonseca[1]   Bruno Feijó[1]   Marcelo Dreux[2]   Esteban Clua[1]

Catholic Univ. of Rio de Janeiro
R. Marquês de São Vicente, 225
Gávea, Rio de Janeiro, RJ
Brazil 22231-000

[1]{ffonseca,bruno,esteban}@inf.puc-rio.br
[2]dreux@mec.puc-rio.br

ABSTRACT

With the continuous increase of processing power, the graphic hardware – also called Graphic Processor Unit (GPU) – is naturally assuming most part of the rendering pipeline, leaving the Central Processor Unit (CPU) with more idle time. In order to take advantage of this when rendering relief textures, the present work proposes two approaches for the mapping of relief textures. Both methods are fully implemented on the CPU leaving the GPU responsible for the per-pixel shading effects. These approaches allow the use of CPU idle time and/or multi-processed systems for the increase of real-time rendering quality and the inclusion of image-based representations.

Keywords
Relief Textures, Image-Based Rendering, Parallel Processing, Real-Time Rendering.

1   INTRODUCTION

In [Oli00], Oliveira introduces the concept of relief texture mapping as a technique to represent details of three-dimensional surfaces. While the traditional texture mapping technique [Cat74] does not consider view-motion parallax and, consequently, only reveals the two-dimensional nature of the texture, Oliveira's approach supports the parallax mechanism and permits the user to have a 3D texture perception. However, as it does not store sufficient geometric information about the details that are being simulated, the technique proposed by Oliveira does not allow the correct representation of non-diffuse surfaces. Moreover, the overhead introduced by the pre-warping step [Oli00] makes it difficult to be used in real-time rendering applications that require high

frame rates, such as games.

Some works have been proposed in order to improve the relief texture mapping technique. For instance, Fujita and Kanai [Fuj02] use the capability of the GPU programming to extend the relief texture mapping technique so that it can support per-pixel shading effects, such as normal mapping and reflection mapping. Although this approach achieves successful results by using shading effects, it does not appropriately work with high frame rate requirements.

Policarpo *et al.* [Pol05] implement the original relief texture mapping on the GPU. In their approach, the view direction is transformed to the texture space and a linear search is performed in order to find an intersection between the view direction and the virtual surface (represented by a depth map). Moreover, this process is improved through a binary search. Their approach complies with real-time requirements and supports per-pixel shading effects.

As the power of GPUs are rapidly increasing – in a faster pace than the power of CPUs – there is an inclination for them to assume almost the entire rendering pipeline processing work, leaving the CPU with more and more idle time. The objective of this work is to investigate the possibility of taking advantage of the CPU's idle time when rendering

relief textures in order to obtain results as good as those obtained in the approach applied by Policarpo *et al* in [Pol05].

The contribution presented in this paper consists in two parallel approaches that optimize the processes involved in relief texture mapping. In some cases, the parallelization of the relief texture mapping algorithm represents an acceleration of up to 300% compared to the conventional technique.

2 RELIEF TEXTURE MAPPING

A relief texture is an image, obtained by an orthographic projection camera, which contains depth information. In a more formal way, a relief texture is a pair $\{i, K\}$, where i is a digital image and K is an orthographic projection camera model associated with i. Each color element of i is augmented in order to include a scalar value that represents the distance (depth), in the Euclidean space, between the sample and a reference entity. Since K is an orthographic projection camera model, the reference entity is the projection plane of K.
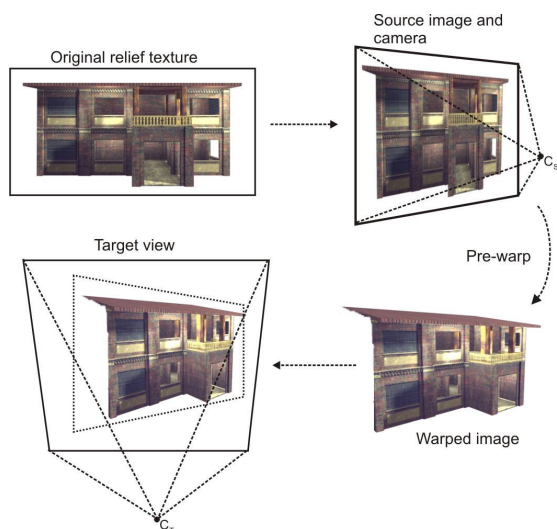

Figure 1. relief texture mapping process [Oli00].

At an implementation point of view, each element of i from a relief texture may be represented as a RGBA information, where RGB channels store color values while alpha channel stores depth values. The information about the camera model K may be represented as a 4x4 matrix.

In 1997, Leonard Mcmillan introduced the concept of three-dimensional image warping [Mcm97], which is the basis for the relief texture mapping technique. 3D image warping consists of geometric information that maps a $\{i_1, K\}$ source image with depth onto a i_2 target image, allowing a correct visualization of the i_1 image from several view points.

It is important to notice that 3D image warping may be interpreted as a composition of two two-dimensional transformations: a planar perspective transformation and a per-pixel shift in the direction of the epipole of the target view. Hence, the relief texture mapping can be seen as a factorization of a 3D image warping into these two transformations. That factorization proposed by Oliveira [Oli00] allows the planar perspective transformation, which consists essentially in a texture mapping operation, to be applied after the per-pixel shift (pre-warping). So, through the factorization, it is possible to benefit from the texture mapping implemented in hardware in order to make the final transformation (Figure 1). More details about that process may be obtained in Oliveira's Ph.D. thesis [Oli00].

2.1 Implementation

As defined in Section 2, a relief texture is composed of color, depth and camera information. However, with only such information it is not possible to capture effects that depend on the view point and illumination direction. A feasible solution to represent such effects is to use normal maps in conjunction with relief texture mapping.

Thus, normal information is merged with depth information in order to generate a normal map with depth, which is represented by an RGBA image, where the alpha channel stores a depth value and the color channels store the normal vector. The color information is stored in a conventional texture map, represented by an RGB image.

The implementation approach used to perform the relief texture mapping consists of a five-step algorithm, which may be represented by the diagram of Figure 2.
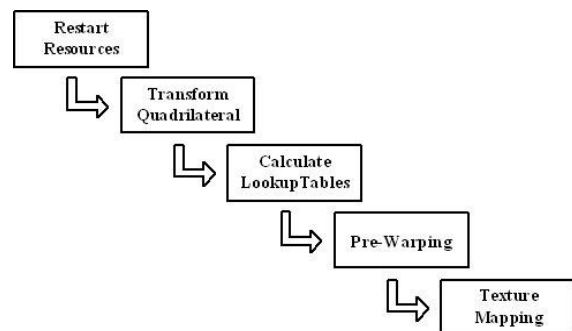

Figure 2. Diagram of the relief texture mapping algorithm.

The algorithm input data are: the current observer position p, a quadrilateral q and a relief texture $\{i_1, K_1\}$, where i_1 represents a normal map with depth information and a conventional texture map. Firstly, it is necessary to initialize image buffers and lookup tables. As the graphics application works in a loop fashion, in which the resultant image is updated

every frame, this step becomes necessary in order to avoid reading the information generated during previous executions.

When this step has finished, the parameters of the quadrilateral q are transformed according to the current viewing configuration. After that, some lookup tables are computed in order to avoid repetitive operations, hence optimizing part of the process.

The next step, called pre-warping, is the core of the algorithm and is the step that consumes more time and computational resources. This step is responsible for creating an output image i_2 that represents a partial visualization of the mapping from $\{i_1, K_1\}$ onto q viewed from p.

Thus, the content of the image i_2 may be stored into the GPU texture memory and, finally, it may be mapped onto quadrilateral q in order to produce a correct visualization. More details about this implementation may be obtained at Fonseca's MSc. dissertation [Fon04].

3   PARALLEL PROCESSING
In the context of real-time graphics applications, the rendering pipeline is usually divided into three conceptual stages: application, geometry and raster [Möl02]. Nowadays, both second and third stages are fully implemented in the graphics hardware, while the first stage is implemented on the CPU.

This division may also be used to represent the whole relief texture mapping computation. The diagram in Figure 3 illustrates the relief texture mapping pipeline stages according to the approach described in Section 2.1.
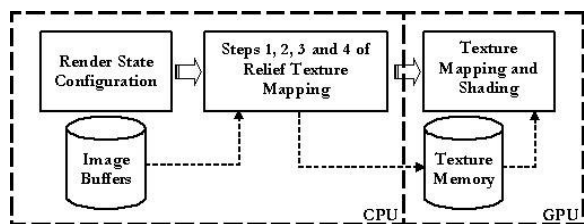

Figure 3. Conventional process for relief texture mapping.

In this diagram, the application stage is executed in a sequential fashion and comprehends all the steps performed on the CPU. The geometry and raster stages are implicitly represented as *texture mapping* and *shading* steps, both executed on the GPU.

By definition, the speed of a pipeline is determined by the slowest stage, independently how fast are other stages. In general, the slowest stage is known as the bottleneck.

According to Akenine-Möller & Haines [Möl02], the first step of a pipeline optimization process consists in locating the bottleneck. This localization is accomplished by a set of tests, such as those described in [Möl02].

Analyzing the algorithm proposed in the Section 2.1, the strongest candidate to be the bottleneck of the relief texture mapping process is the application stage, since all texels of the normal and color maps are processed every frame. For this reason, it is decided to make the referred tests to the application stage.

One way to verify if the application stage limits the rendering speed is to send data through the pipeline in such a way that the other stages perform little or no work. In OpenGL this can be achieved substituting every call to `glVertex3f` and `glNormal3f` by the call to `glColor3f`. By doing this, the work of sending data from the CPU remains unchanged, while the work of sending and receiving data on the geometry and raster stages are drastically reduced. If the performance does not improve, it is possible to affirm that the bottleneck is the application stage. The authors of the present paper applied this test to some input textures and verified that the application stage is actually the bottleneck.

In the light of the above mentioned considerations, the present authors propose the implementation of two parallel approaches for the relief texture mapping computation. The next section describes, for each approach, the rationale of the implementation and the proposed methodology.

3.1   Parallel Approach
With the intention of optimizing the CPU process, a CPU thread is created. This thread has the capability of running the four first steps of the relief texture mapping algorithm. With the aid of the Hyper-Threading technology [Mar03], this thread can be executed in parallel with the conventional CPU process and hence allowing a considerable time saving. The diagram of Figure 4 illustrates the new approach.
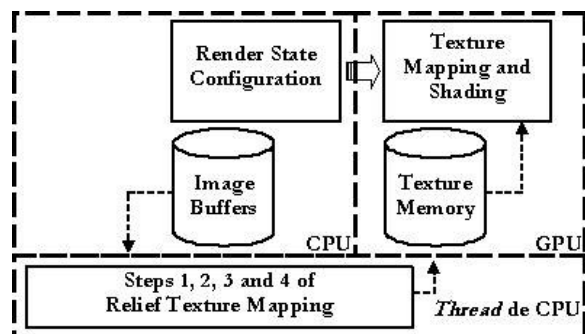
In order to guarantee that the final result is correct, besides creating a CPU thread, it is necessary to synchronize it with the rest of the processes that are being executed. This synchronization is represented by the state machine illustrated in Figure 5.
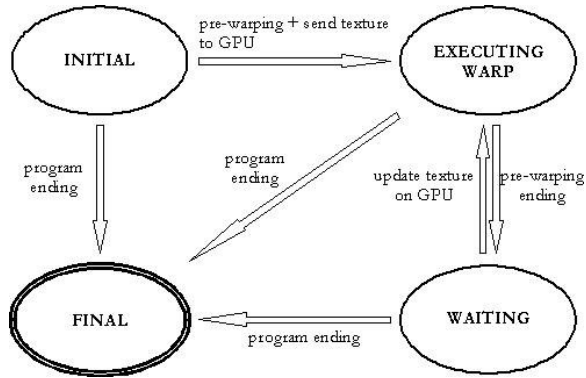

Figure 5. CPU thread state machine.

At the moment of the thread creation, the thread state is set to *initial*. The transition between the *initial* state and *executing warp* state is done by the first warping operation followed by the transmission of the resultant image to the GPU texture memory. During the rest of the execution, the state machine stays in a loop between the *execution warp* and *waiting states*, except at the end of the program, when the CPU thread goes to the *final* state. The pre-warping operation is performed during the *execution warp* state. When it finishes, it changes to *waiting* state. In this state, the resultant new image is sent to the texture memory and its content is refreshed. As soon as the texture memory is updated, the state is set to *executing warp* and it indicates that a new warping operation must be performed.

With the Hyper-Threading or similar technology, a processor may be exclusively assigned to the CPU thread, without affecting the rendering pipeline performance.

The transition from *initial* state to the *executing warp* state occurs during the execution of the main draw function, performed once a frame. An algorithm that describes this process is presented below.

```
algorithm draw()
 1 Configure the OpenGL API rendering state;
 2 Activate vertex program;
 3 Activate fragment program;
 4 If it is the first execution of draw then
 5    Start resources;
 6    Transform the quadrilateral;
 7    Initialize lookup tables;
 8    Perform pre-warping;
 9    Send processed texture to the GPU;
```

```
10    stateThread ← EXECUTING_WARP;
11 else
12    If stateThread = WAITING then
13       Update texture on the GPU;
14       stateThread ← EXECUTING_WARP;
15    end-if
16 end-if
17 Draw quadrilateral with the processed texture;
18 Deactivate fragment program;
19 Deactivate vertex program;
end
```

At line 1, the OpenGL API rendering state is configured. This configuration consists in enabling operations such as blending, culling, and depth tests. In lines 2 and 3 the vertex and fragment programs are enabled in order to calculate per-pixel illumination. The first time that the `draw` function is executed, the four first steps of the relief texture mapping algorithm (lines 5 to 8) are performed and the resultant texture from the warping is sent to the graphics card, through the OpenGL `glTexImage2D` function. Once the texture is sent to the graphics card, the state changes from *initial* to *executing warp* (line 10). After the second execution of the `draw` function, it is necessary to verify (in each cycle) if the current state of the CPU thread is *waiting*. In that case, it means that the thread has executed a warping operation and, consequently, the texture in the graphics card must be updated (line 13). So, the update is achieved by the OpenGL `glTexSubImage2D` function. After that, the CPU thread changes from *waiting* to *executing warp* state and a new warping operation must be performed. Finally, at line 17, the quadrilateral with the mapped texture is drawn and, in lines 18 and 19 the fragment and vertex programs are deactivated. It is important to notice that during every execution of the *draw* function, the CPU thread is executed in parallel.

The remaining state transitions are described by the `threadCPU` algorithm.

```
algorithm threadCPU()
 1 While true do
 2    If stateThread = EXECUTING_WARP then
 3       Start resources;
 4       Transform the quadrilateral;
 5       Initialize lookup tables;
 6       Perform pre-warping;
 7       stateThread ← WAITING;
 8    else if stateThread = FINAL then
 9       return;
10    end-if
11 end-while
end
```

The algorithm is basically a loop: line 1 guarantees that the thread is being executed until its current state is modified to the *final* state. During this loop, if the current state is *executing warp*, the four first steps of the relief texture mapping algorithm are executed (lines 3 to 6) and the state is assigned to waiting (line 7), which indicates that a texture updating operation must be performed. When the state is equals to *final*, the `threadCPU` function ends.

3.2 Multi-Threaded Approach

Besides the previous approach, the use of Hyper-Threading technology allows the elaboration of a parallel process for different parts of input data. More specifically, it is possible to divide the $\{i_1, K_1\}$ input texture into two parts and to simultaneously execute the four first steps of the relief texture mapping algorithm for each part. Consequently, a post-processing becomes necessary in order to unify the resultant textures into one unique texture that will be mapped into quadrilateral *q*. The diagram in Figure 6 illustrates that process.
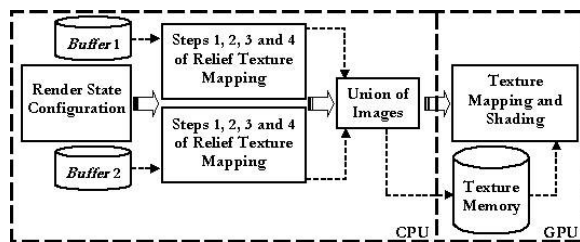

Figure 6. Multi-Threaded process diagram.

Firstly, it is necessary to divide the image buffer into two different parts. It is important to notice that the image buffer consists of a normal map, a depth map and a color map used as input data to the algorithm. Although it is not obvious, both resultant images of the division process must have the same dimension as the original one. For instance, the left half of the first resultant image will be filled with the left half of the original image content, while the right part of the resultant image will be filled with invalid values (i.e. values that represent background information, as illustrated in Figure 7). This is necessary because during the pre-warping process some texels could exceed the limits of the plane that supports the image. Consequently, the resultant images of the pre-warping process would contain an incomplete vision of the representation (see Figure 8).
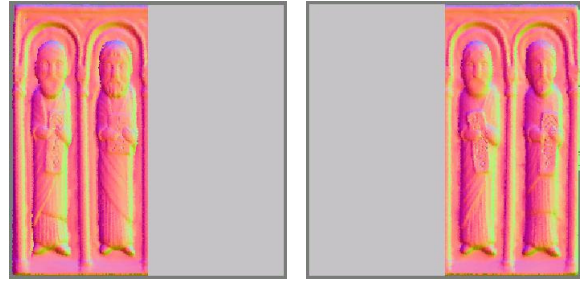

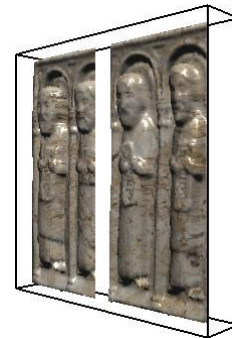**Figure 7. Illustration of the division process of the normal map.**

Figure 8. The incomplete vision of the represented surface occurs because texels that exceed the limits of the support plane during the warping are not considered.

During the union process of both resultant images, some parts of the result may have overlapping texels in regions where the limits of the support plane of the image were exceeded. The algorithm will correct this problem discarding the texels that have invalid values in the places where this overlapping occurs.

The `draw` function, described in Section 3.1, must be modified in order to correspond to that new approach:

```
algorithm draw()
 1 Configure the OpenGL API rendering state;
 2 Activate vertex program;
 3 Activate fragment program;
 4 If it is the first execution of draw then
 5    Execute the four first steps for left half;
 6    Execute the four first steps for right half;
 7    Unify the resultant images;
 8    Send processed texture to the GPU;
 9    stateThread1 ← EXECUTING_WARP;
10    stateThread2 ← EXECUTING_WARP;
11 else
12    If stateThread1 = WAITING and
           stateThread2 = WAITING then
13       Unify the resultant images;
14       Update texture on the GPU;
```

```
15      stateThread1 ← EXECUTING_WARP;
16      stateThread2 ← EXECUTING_WARP;
17    end-if
18  end-if
19  Draw quadrilateral with the processed texture;
20  Deactivate fragment program;
21  Deactivate vertex program;
end
```

The differences from the function described in Section 3.1 are basically located between lines 5 and 16. The first time that the `draw` function is executed the four first steps of the relief texture mapping algorithm (lines 5 and 6) are performed for both halves of the original image. Once the warping process is concluded, the union of the resultant images may be executed. After this, the resultant texture from that union is sent to the graphics card. As soon as the texture is sent, the transition from the *initial* state to the *executing warp* state can be made (lines 9 and 10) for each thread. In cases where the `draw` function is executed again, it is necessary to test if the current state of the CPU, for both halves, is *waiting*. It means that the tread completed a full warping operation and, consequently, the result of each warping may be unified so that the texture in the graphics card may be updated (line 14). Following it, the state of each CPU thread changes from *waiting* to *executing warp* and a new warping operation must be performed.

4   RESULTS

This section presents statistics of elapsed time of the implemented algorithms as well as the obtained visual results. These measures were made with an Intel Pentium IV PC with 2.66 GHz and 512 Mb of memory RAM and a graphics card GeForce FX 5600 with 256 Mb of video memory.

Three samples have been used, as shown in Figure 9 at the end of the section. Table 1 presents some specifications for the used samples such as, resolution (in pixels) and the number of texels with invalid depth values (i.e. texels that represent background information).

Table 2 presents the frame rate for each one of the implemented approaches. The sequential approach (S) represents the conventional technique of relief texture mapping, while the parallel (P) and the multi-threaded (M) approaches are the algorithms proposed in this work.

Sample	Resolution	Invalid Texels
1	256x256	3800 (5,80%)
2	256x256	3223 (4,92%)
3	256x256	30484 (46,51%)

Table 1. Samples information.

	Sample1		Sample2		Sample3	
	FPS	SD	FPS	SD	FPS	SD
S	24.18	0.86	24.93	1.54	38.24	1.86
P	97.97	4.50	96.45	5.01	103.64	4.88
M	33.14	6.02	31.09	7.67	34.51	8.21

Table 2. Frames per second average (FPS) and standard deviation (SD) for sequential (S), parallel (P) and multi-threaded (M) approaches.

Analyzing Table 2, it is possible to conclude that the parallel approach is better than the sequential one. However, the sequential approach is the more stable method in relation to the frame rate average. It may be verified by the small dispersion of the data presented by the obtained standard deviation.

Only in the case of sample 3 the multi-threaded approach is worse than the sequential one. There are two points that must be considered in order to explain that fact. The first one refers to the sample 3, which is a special kind of sample, where the number of invalid texels is large (see Table 1) and, therefore, the pre-warping step effort is inferior in relation to samples 1 and 2, since only valid texels need to be processed. The second point refers to the multi-threaded approach, which has an overhead of three processes being executed in a two processor system. So, it is possible to conclude that the inherent overhead related to the multi-threaded approach was superior to the gain obtained by the optimization of the pre-warping step in the case of the sample 3, demonstrating the inferiority of the multi-threaded approach in relation to the sequential one in this specific case.

It is important to notice that high frame rates per second obtained by the parallel approach do not reflect the number of times that a warping operation is being performed. These rates refer to the number of times that the images are being rendering per second, independently if a new warping operation is being executed or not. It occurs because the warping process is being executed in parallel to the rest of the pipeline and, consequently, sometimes it is possible to notice a progressive update in the rendered image when there are many camera movements.

The superiority of the parallel approach in relation to the multi-threaded one was already expected, since the latter consumes a reasonable time during the realization of the texture union operation.

The obtained visual results are presented in Figure 10, which are the same for all approaches.

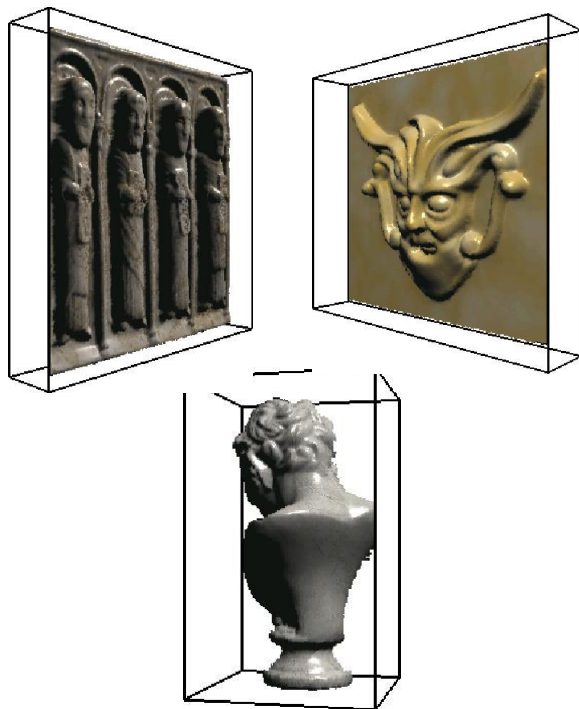**Figure 9. Samples 1 (left image), 2 (right image) and 3 (bellow image).**

Figure 10. Visual results for sample 1, sample 2

and sample 3.

5   CONCLUSIONS AND FUTURE WORKS

In order to optimize the relief texture mapping process, two new approaches have been implemented:

- **Parallel**. In the parallel approach a CPU thread was created with the objective of resolving the first four steps of the relief texture mapping algorithm. Such thread is executed in parallel with the main CPU process, using for this propose the Hyper-Threading technology.

- **Multi-Threaded**: In this method, the input texture is divided into two parts with the objective of simultaneously running the four first steps of the relief texture mapping algorithm for each part. It becomes necessary a post-processing to unify both results into a single resultant image.

It is possible to conclude that the parallelization of the relief texture mapping considerably speeds up the process in comparison to the conventional methods (up to 300%), i.e., with the parallelization the relief texture mapping may be implemented in real-time on the CPU and thus allowing the GPU to be used only for shading calculations. The performance and image quality of the proposed approach are similar to the ones obtained by Policarpo et al [Pol05].

In relation to future works, two possible optimizations could be incorporated: utilization of a pipeline of multi-processors in the relief texture mapping and a variant to the multi-threaded approach that processes rows and columns in parallel.

Furthermore, a better analysis of the warping per second rate could be done in relation to the frame per second rate (where wps corresponds to the number of warpings that are being performed per second). This analysis could allow a more efficient implementation of the process (with less warping operations), in the case of the wps to be larger than the fps rate.

Finally, an implementation that uses many CPU processors could be developed in order to take more advantages of the described techniques.

6   ACKNOWLEDGEMENTS

7   REFERENCES

[Cat74] Catmull, E. *A Subdivision for Computer Display of Curved Surfaces*. December 1974. Thesis (Ph.D in Computer Science). Department of Computer Science, University of Utha, Utha, 1974.

[Fon04] Fonseca, F.M. A. *Relief Textures using Per-Pixel Illumination and Parallel Processing*. January 2004. Dissertation (MSc in Computer Science). Department of Computer Science,

Catholic University of Rio de Janeiro, Rio de Janeiro, 2004 (in portuguese).

[Fuj02] Fujita, M and Kanai, T. Hardware-Assisted Relief Texture Mapping. In: European Association for Computer Graphics (EUROGRAPHICS). 2002. *Proceedings of the annual conference of the European Association for Computer Graphics* Saarbrücken, Germany, 2002.

[Mar03] Marr, D. T. *et al*. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*. v. 6, n. 1, p. 4-152, fev. 2003.

[Mcm97] Mcmillan, L. *An Image-Based Approach toThree-Dimensional Computer Graphic*s. April 1997. Thesis (Ph.D in Computer Science).

Department of Computer Science, University of North Carolina, Chapel Hill, 1997.

[Möl02] Akenine-Möller, T.; Haines, E. *Real-Time Renderin*g. 2. ed. Massachusetts: A K Peters, 2002. 482 p.

[Oli00] Oliveira, M. M. de. *Relief Texture Mappin*g. March 2000. Thesis (Ph.D in Computer Science). Department of Computer Science, University of North Carolina, Chapel Hill, 1997.

[Pol05] Policarpo, F.; Oliveira, M. M.; Comba, J. L. D. *Real-time relief mapping on arbitrary polygonal surface*s. Proceedings of the Symposium on Interactive 3D Graphics and Games, Washington, District of Columbia, 2005.