A Distributed Texture Mapping Algorithm

Sergio A. S. de Moraes

EMBRAPA - Dep. de Informática
SAIN - Final da W3 Norte
Brasilia - DF
e-mail: sergiom@sede.embrapa.br

UNIVERSIDADE CATÓLICA DE BRASÍLIA
FCT - Dept. de Computação
Campus de Águas Claras
Taguatinga - DF

ABSTRACT

This paper describes our work on texture mapping algorithms adapted to run efficiently on a distributed programming environment, which is based on a virtually shared distributed memory paradigm. The texture mapping task is broken up into subtasks which are dynamically assigned to processors. The processors are collected from available nodes of a network of workstations. The tasks assignment and control is done using the master/worker model of computation. The results show a good speed up on the texturing process.

Key words: Texture mapping, distributed computing, computer graphics.

1 Introduction

The search for more realism in computer generated images has seen the development of several techniques such as ray-tracing, radiosity, and composite rendering, that have added a more realistic look to synthetic images. However, none of these are capable of adding fine detail to complex imagery without rendering themselves computationally over-expensive[Crow88]. Texture mapping, on the other hand, is computationally inexpensive compared to other techniques, and the level of details that it can add to an image is almost unlimited. However, this technique is not a substitute to the others, but rather an add on, which makes the whole rendering process a slower one. Therefore, in this paper we address the issue of parallelising the texture mapping process as a way to keep down the rendering time of realistic images. In fact, the process of rendering complex and high definition images, with hundreds of objects and texture maps, is still a slow one even when specialised hardware is used.

The basic idea was the divide and conquer approach. The texture mapping task was broken into small tasks. These tasks were then assigned to idle processors on a network of workstations. The novelty aspects of our work are the flexibility on the granularity level of the subtasks and the type/model of parallelism that can be chosen by the programmer/user in order to suit the application in question.

In this work we introduce a novel method to speedup the texture mapping process through distributed processing. First we will make a brief review of texture mapping theory. We shall present some texture mapping task partitioning methods and how they perform on our system. Our task distribution model can be said to be adaptive, allowing us to adapt the level of parallel granularity to suit both the application problem and the number of processors available at each run of the program. All this is accomplished without major changes to the program code.

* This work has been sponsored by CNPq/ProTeM-CC, SMM Project
2 Texture Mapping Theory

Realism lies in the complexity of details that a surface presents. However, the level of details required to render a photo-realistic image is far too complicated to be modelled through geometric primitives. As a solution to this problem, Catmull [Catm74] introduced in 1974 a method that he called texture mapping. Since then, this method has been refined by Blinn and Newell [Blin82], Crow [Crow84], and others. Nowadays it plays a great role in the process of generating realistic images.

Since its first use in computer graphics by Catmull in 1974, texture-mapping has increased its role in adding realism in computer generated imagery, alongside other techniques such as ray-tracing, radiosity, and composite rendering. In a survey review paper, Heckbert [Heck86] identifies all those parameters that have already been modulated to provide a textural impression:

- surface colour - diffuse reflection coefficients;
- specular and diffuse reflection - environment mapping;
- normal vector perturbation - bump mapping;
- specularity - glossiness coefficients;
- transparency;
- shadows, surface displacement, and mixing coefficients;
- local co-ordinate system - frame mapping.

Nevertheless, the most popular use of texturing has been to modify the colour of an object surface. In general the colour which is applied to points on an objects surface are taken from a stored image, known as the texture image. Figure 1 summarises the possibilities of texture mapping methods. As it is shown in the figure, the texture domain can be one, two, or three dimensional. The most common method is two-dimensional, which can take the format of a frame-grabbed image. This possibility expands immensely the source of texture pattern.

The texture effects generated by one and three dimensional domains are usually very restricted, although the mapping process is considerably easier: They are generated procedurally. For example, one-dimensional texture can simulate thin film interference, which are the cause of the colour effects that appear on soup film on a bubble or oil film on the surface of water [Watt89]. Three dimensional texture can represents, for example, wood [Peac85], marble [Perl85], or clouds [Gard85].
In general, a pixel may be considered to be a small window looking onto a surface, as it is shown in Figure 2. When texture is applied to that surface, the inverse viewing transform, inverse mapping [Feib80], is invoked to find the projection of the pixel onto the surface. Usually only the four corners of a rectangular pixel are needed to map the texture surface covered by the pixel. This mapping results in an irregular quadrilateral area in the texture map. Often, the axes of the 2D-space textures are referred to as \((u,v)\). The quadrilateral area mapped into texture space is defined by the four \((u,v)\) pairs, one for each pixel corner. Then the average texture value inside that quadrilateral is found. This average value is returned to the renderer as the average particular colour of that surface seen from that pixel.

3 **Distributed Texture Mapping Method**

In this section we will describe a novel approach to texture mapping algorithms adapted to run efficiently on a distributed programming environment[Mora93], which is based on the LINDA parallel programming paradigms[Carr89].
The whole task of texturing was broken up into small subtasks which were assigned dynamically and anonymously to processors. The processors were collected from available nodes of a network of workstations. We used the Master/Worker model. The results have shown a good speed up of the texturing process. As the same solution can be extended to other graphic rendering processes we foresee a significantly gain on the rendering time of highly realistic and complex images without the use of expensive dedicated graphics hardware.

Before we describe how the texture-mapping distribution methods was achieved, it is worthwhile to make some other considerations. First, texture-mapping will be done at object basis. This choice has been made because the alternative, let say texture-mapping on a raster basis, may be more time consuming if we consider that a given screen line might cover several objects surfaces. In this case, for just one scan line we need to have as many texture-maps available as there are surfaces that are intercepted by that specific line. So, the cost of this approach tends to be unacceptable. Second, we considered that only the relevant (visible) object surfaces of a image would undergo the distributed texturing process, which requires objects to be analysed for visibility. Finally, the actual maximum number of worker processes in the experiments shown in this paper were limited to 5 due the lack of workstation availability when this research was done.

Texturing Mapping Distribution Algorithm

The basic texture mapping algorithm we used in this work included the following steps:

- Objects surfaces are defined by geometric primitives (triangles).
- Each vertex of a triangle has its related texture parameters.
- The triangle primitives are subject to perspective transformation.
- Triangles are scanconverted to screen co-ordinates.
- For each pixel the texture map is indexed by texture co-ordinates and a value is retrieved.
- This value is displayed as the pixel intensity.

To distribute this among the nodes of a network, we have considered as the basic task the rendering of a triangle primitive (their scan-converted screen pixels). Therefore, each parallel process will do all the above steps on each of the triangular primitives defined in the database of the scene to be rendered. The computation task was defined, then, as a triangle primitive, the texture index at each vertex of the triangle, and a file descriptor of the texture map to be used on the mapping process of the triangle.

Overview of the Distributed Texture Mapping Algorithm

A master process reads the scene data. For each triangle primitives, with a texture index for each triangle vertex, the master assigns a texture map. This information constitutes a task description, which is deposited as a task tuple into tuple space. The master repeats this procedure for all triangle primitives in the scene database. It then waits for the tuples with the pixel values. Those pixels values are written into a frame buffer for display (in our implementation, they were written to a file).

The workers, the texture mapping renderers, are activated in each one of the participating nodes. Each one of them takes a task tuple from tuple space, accesses the relevant texture map file, does the scan conversion and perspective transformation of the triangle, and then does the mapping. The resulting pixels values are then returned to tuple space as a result tuple. The mapping renderer workers continue this process until no more task tuples are available in tuple space. They then stop processing.

4 The Distributed Texture Mapping Algorithm

The algorithm for distributing the texture mapping computation among a number of processors is designed to achieve the maximum performance. The algorithm is based on a job control tuple, which is used as the distributed data structure to hold the number of tasks, on an object/map basis, that remain present in tuple space. The tuple data field that contains the number of tasks left in tuple space is an array of the pair
(object/map, n_triangles). The n_triang`es variable contains the number of triangles still left to be mapped with the corresponding texture map described by the variable object/map. Another field (next_pair) in the job_control tuple holds a pointer to next (object/map, n_triangles) pair that will be the first searched pair for task availability. Its aim is to make processes work in parallel on as many different texture maps as possible.

The algorithm works as follows:

The master process:

- The master process reads the scene database, containing object primitives and their related texture map names.
- The master process starts all worker processes taking part in the experiment. Each worker is assigned an initial texture map and its position in the (object/map, n_triangles) data structure (this means that for the first task each worker does not need to access the job_control tuple).
- The master generates the job_control tuple, the task tuples and then starts retrieving tuples containing the resulting pixels. When no more results are left in tuple space the master stops processing. The job_control tuple is adjusted to reflect the tasks done initially by each worker, which does not require an access to this tuple.

The worker:

- For the first task, the worker loads its assigned texture map and, then, immediately accesses a tuple task.
- The worker does the texture mapping, and outputs in tuple space an array containing the pixel values.
- For each subsequent task left in tuple space, the worker accesses the job_control tuple. If no more tasks are available for the object/map it already holds in memory, it starts checking the (object/map, n_triangles) array from the pair pointed by the next_pair variable. When an available task is detected, the worker subtracts 1 from the relevant n_triangles variable and sets the next_pair variable to point to following cell in the (object/map, n_triangles) array. The job_control tuple is put back into tuple space and the new task tuple is read for the texture mapping process.
- When all the n_triangles in the job_control tuple variables are set to zero this means that there is no more tuple task available in tuple space. This is the condition that makes a worker stop processing.

This algorithm is quite effective because it speeds up the slower part of the texture mapping process, which is the acquisition of the texture map and loading it into memory. Another advantage is that each worker works over a texture map until no more tasks for this particular map need to be done. Then, this worker acquires another map that still has tasks available. This helps to maintain a balanced load among the nodes of the network.

Nevertheless, the above algorithm does not avoid a worker getting a task for a map that has only one more task left and has many workers working already with this map. However, a small modification to the algorithm eliminates that side effect and can be more effective in dynamically load balancing the overall system. In this modification the next_pair variable is not used. However, a n_workers variable is included in the (object/map, n_triangles) array. This variable is used to hold the number of workers using the corresponding texture map. Thus, with this new information a worker can make a more intelligent decision: which of the remaining texture maps that still has tasks to be done and has the highest ratio of outstanding tasks to engaged workers.
5 Algorithm Implementation and Results

We used the master-worker model to parallelise the texture mapping algorithm. Using eval() the master starts one worker process for each of the other participating workstation. It then uses out() to create a task tuple for each computational task (a triangle primitive, its texture indexes, and its related map descriptor). Each worker looks for a task tuple and grabs one with in(). After finishing the computation required by the task, the worker uses out() to dump a tuple containing the completed textured pixels for the image sub-area defined by the triangle primitive, and looks for more task tuples. Meanwhile the master collects the texture mapped objects and send them to a file (or a frame-buffer). When no more tasks are left to be done, the workers and the master processes stop running. A complete description of operations eval(), out() and in() can be found in [Mora93].

The following pieces of code show the main procedures of the master and the worker program:

```
Master()
{
    structure = {
        char *texture;
        int n_triangles;
        int n_workers; } job_control;
    read_Database(map_names, n_triangles, triangles_primitives);
    for(i = 0; nmaps > 0 && n_workers > 0; i++, nmaps--, n_workers--) {
        set_job_control_variables();
        eval(workers(map_name, i));
    }
    if (nmaps != 0) {
        for(; nmaps > 0 && n_workers > 0; i++, nmaps--) {
            set_job_control_variables();
        }
    }
    else {
        if(n_workers != 0) {
            for(i = 0; n_workers > 0; n_workers--, i++) {
                eval(workers(job_control.texture[i], i));
                set_job_control_variables();
                i = (i % max_maps);
            }
        }
    }
    out("job_control", job_control, number_of_maps);
    for(all_triangle_primitives) {
        out("task", map_name, triangle);
        all_triangle_primitives--;
    }
    for(n_tasks) {
        in("results", &rgb_array, &xy_array);
    }
```

Worker(map_name, i) {
structure = {
    char *texture;
    int n_triangles;
    int n_workers; } job_control;
read_to_memory(map_name)
while(ok) {
in("task", map_name, &triangle);
do_texture_mapping();
out("result", rgb_array, xy_array);
in("job_control", &job_control, &n_maps);
if(job_control.n_triangles[i] == 0)
    index = search_the_smaller_rate(job_control);
if(index < 0)
    break; /* No more task left in tuple space */
map_name = job_control[index].texture;
}

The complete program code of the master and worker processes for the texturing experiments is presented in [Mora94]. In the following paragraphs we describe these experiments and present their main results.

The images generated in these experiments are shown in Plate 1 and Plate 2. These scenes were used to evaluate the performance gain of the distributed texture mapping algorithm. Plate 1 shows a room with mapped walls, ceiling, and floor. All the objects are planes. Each plane is defined as a pair of triangle primitives. Although the result are good, the simplicity of the scene has made the workers respond quite uniformly, despite the different sizes of the images. This can be explained by the fact that for each image size the amount of tuple space access has remained constant: the same number of primitives, which is equivalent to the number of tasks. What has increased, however, was the number of pixels rendered by each worker (the "task results" tuples), which generated more traffic on the network. Nevertheless, this traffic was spread over a bigger span of time, since each task required more time to be finished. Therefore, the communication cost had its influence on the results reduced when the image size was increased. Considering that each task (triangle primitive) covered an equivalent sub-area of the image, the time each worker spent on them was also equivalent. This accounts for the uniformity of the performance gain results.

Therefore, we have set up the scene shown in Plate 2 and have undertaken the same type of measurement as with the simple image. Not surprisingly the results on this setting were even better than the first one. This is because the complexity of the scene has made the uniprocessor version run more slowly, while in the parallel version more workers have worked on the more demanding objects once the simple objects had been done. Figure 3 depicts the speedup curves for the image shown in Plate 1, and Figure 4 depicts the speedup curves for the image shown in Plate 2. Table 1 and Table 2 show the results, respectively, for both experiments. Table 3 gives the execution times of the uniprocessor version of the algorithm. All the results presented in this work are the average values of the times collected in 9 runs of each program experiment.

The master is excluded from the machine count in the plot, since for this application the demands on it are quite modest. The times shown are wallclock times. This took less then between 4% and 10% for the best and worst case in the more simple experiment, and between 4% and 15% for the best and worst case in the more complex experiment, of the time of the 5 workstations/worker involved in this experiment.
Table 1 Execution Time (secs) and speedup results for the Plate 1 image experiment.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>128x128 Image</th>
<th>256x256 Image</th>
<th>512x512 Image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exec. time</td>
<td>Speedup</td>
<td>Exec. time</td>
</tr>
<tr>
<td>1</td>
<td>35.77</td>
<td>0.73</td>
<td>44.08</td>
</tr>
<tr>
<td>2</td>
<td>19.55</td>
<td>1.34</td>
<td>23.74</td>
</tr>
<tr>
<td>3</td>
<td>14.13</td>
<td>1.86</td>
<td>16.93</td>
</tr>
<tr>
<td>4</td>
<td>11.45</td>
<td>2.30</td>
<td>13.54</td>
</tr>
<tr>
<td>5</td>
<td>9.81</td>
<td>2.68</td>
<td>11.49</td>
</tr>
</tbody>
</table>

Table 2 Execution Time (secs) and speedup results for the Plate 2 image experiment

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>128x128 Image</th>
<th>256x256 Image</th>
<th>512x512 Image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exec. time</td>
<td>Speedup</td>
<td>Exec. time</td>
</tr>
<tr>
<td>1</td>
<td>56.86</td>
<td>0.72</td>
<td>81.83</td>
</tr>
<tr>
<td>2</td>
<td>33.85</td>
<td>1.19</td>
<td>42.48</td>
</tr>
<tr>
<td>3</td>
<td>25.00</td>
<td>1.61</td>
<td>31.34</td>
</tr>
<tr>
<td>4</td>
<td>21.01</td>
<td>1.92</td>
<td>25.77</td>
</tr>
<tr>
<td>5</td>
<td>18.62</td>
<td>2.17</td>
<td>22.43</td>
</tr>
</tbody>
</table>
Table 3. Uniprocessor execution time (secs)

<table>
<thead>
<tr>
<th>Texturing Experiment</th>
<th>128x128 Image</th>
<th>256x256 Image</th>
<th>512x512 Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate 1</td>
<td>26.37</td>
<td>32.18</td>
<td>63.75</td>
</tr>
<tr>
<td>Plate 2</td>
<td>40.42</td>
<td>66.82</td>
<td>171.87</td>
</tr>
</tbody>
</table>

Figure 3. Speed-up curves for the texture mapped scene shown in Plate 1.

Figure 4. The speed-up curves for mapping the scene shown in Plate 2.
The uniprocessor (one worker process only) time of the parallel version of the algorithm shown in the result tables has performed worse than the sequential version of the same algorithm. This is due to the fact that the master/worker model has an inherent basic overhead cost which tends to be less and less influential as the number of workers grows.

Another important consideration to be made at this point is relative to the maximum number of workers that would be meaningful for performance improvement in this network setting environment. This must be considered separately for each set of experiments. In the case of the 512x512 image size Plate1 scene, the reduced number of tasks is the upper limit on the number of workers. The communication cost and tuple access time was negligible compared to the task completion time, which was in the order of seconds. Thus, we could have 10 workers and a speed up close to the ideal one.

In the case of the Plate 2 scene, again for the biggest size image, the complexity of the scene has introduced a major variable to the equation. Here the tasks size are very small and in much more number. For most of them the completion time was in the order of milliseconds. Again the amount of traffic generated is the same for each image size whether we have one or 5 workers. So, if we increase the number of workers to match the number of tasks, the traffic generated would be concentrated in a very small time interval. That would put the network under such a pressure that the communication cost and tuple space access time would be the dominant factors. Therefore, the actual number of workers that would lead to the maximum gain in performance is dependent on the complexity of the image and the size of the tasks.

6 Conclusion

The quest for realism in computer graphics has always pushed computer power to its edge, no matter how fast a CPU is. Most of the graphics systems on the market have tackled this problem through the use of dedicated hardware. In this work we have demonstrated that it is possible to harness all the latent computer power capacity of a modern computer network in order to deliver a desirable level of user/computer interaction on the task of rendering complex and realistic images. Of course our approach is not in opposition to dedicated hardware. Indeed, if specialized hardware is available on the network, no doubts the our system will deliver even better results.

Our experiments have demonstrated that it is possible to parallelize a big and slow task by breaking it up into small subtasks, and making them run in parallel on the nodes of a network. The distributed processing environment based on the LINDA paradigm has all the necessary tools to make such solution very simple on programming and powerful on its results.

However, processing over a network implies communication overhead which must be kept low. We used several methods to keep such drawback under control, such as replicating the texture maps on each participating node, or transferring only the relevant part of the map. As our results have shown, there is a limit on the size of the subtasks to be processed in parallel, which is a trade-off on the communication/computation ratio. As it is ease in our system to play around on this matter, the user can rapidly stabliah a good solution.

As a future work on this subject, we intend to extend our research to parallelize other components of the graphics pipeline in order to analyze how is the impact of several groups of parallel processes competing for resources on a network of workstations.

References


Plate 1 - A scene with a totally mapped room.

Plate 2 - A Scene with differently shaped objects.