Parallel Culling and Sorting based on Adaptive Static Balancing

Lucas Machado Bruno Feijó
VisionLab/ICAD/IGames, Dept. of Informatics, PUC-Rio
{lmachado, bfeijo}@inf.puc-rio.br

Abstract

This paper presents a new and effective method for parallel octree culling and sorting for multicore systems using counting sort and based on a new balancing algorithm, called adaptive delayed static balancing. The adaptive nature of the method is governed by a dynamic split level that can adjust the algorithm to new camera positions keeping a well-balanced workload amongst the processors. Also this paper introduces the concept of n-dimensional resource space as a discrete Euclidean space. This work presents a simple and effective thread management system called MinTMS.

1. Introduction

Octree culling is a classical algorithm for reducing the amount of data sent to the GPU for rendering. The technique consists of dividing the 3D space into eight cubes and repeating the process for each cube until a certain level of the octree is reached (usually, the leaves) and objects are stored. Rendering is done by testing the intersection of the view frustum with the octree nodes and sending to the GPU only the visible objects. In this case, if a certain node cannot be seen its entire subtree is pruned from the octree. This process can be easily parallelized, but the balance of the workload is not trivial. Another aspect of the rendering process is resource sorting (e.g. Textures, meshes, and pixel shading techniques). There is always a cost associated to resource changes. Therefore, these changes should be reduced by sorting and grouping objects with common resources. Most of the methods for parallel rendering is concentrated on PC clusters and grids, while the literature on parallel culling for multicore systems is scarce. This paper presents a new and effective method for parallel octree culling and sorting for multicore systems using counting sort (which is O(n) time) and based on a new balancing algorithm called adaptive delayed static balancing. Tests revealed a performance improvement of the culling process between 3 and 4 times in relation to the classical single threaded octree culling process. However, the most important performance analysis concerns the capability that the proposed method has to adapt itself to new camera positions, which are continuously changing over time.

This paper is organized as follows. In the next section, previous works are analyzed. In section 3, we present the concept of the adaptive delayed static balancing for parallel culling. The algorithm for node rendering is presented in section 4. Section 5 presents the adaptive nature of the proposed method. In section 6, we have the entire algorithm. Also this paper proposes a simple API for thread management called MinTMS in section 7. Section 8 presents the parallel sorting method that handles the sorting of multiple resources of the objects. Finally, some results are described in section 9 and section 10 presents some final remarks.

2. Related Work

A lot of research on parallel search and sorting algorithms has already been done and many techniques exist in the literature [Grama et al. 2003] [Wilkinson and Allen]. Also several works have been carried out in the area of parallel rendering using sorting techniques. Molnar et al. [1994] proposed a classification of parallel rendering system based on in which stage of the rendering pipeline the sorting is carried out (sort-last, sort-middle, and sort-first). Humphreys et al. [2002] present a sort-first method for distributed rendering using a cluster of common PCs. Abraham et al. [2004] propose a load-balancing strategy for sort-first distributed rendering using PC-based clusters. Baxter et al. [2002] present a parallel rendering architecture using two graphics pipeline and one processor, including occlusion culling, LOD and scene graph hierarchy. However, these works concentrate on distributed rendering using PC clusters and/or on global aspects of parallel rendering. The literature has few works concentrated on algorithms for parallel culling and sorting using multicore systems.

Octree is a classic data structure used in many computer graphics applications [Foley et al. 1995], [Dalmau 2003]. However, parallel occlusion algorithms using octrees are not usual. Greene et al. [1993] are the first authors to propose an octree hierarchy for visibility computation with some potential to parallelize. Their work had a great influence on graphics hardware design. Xiong et al. [2006] present an algorithm for parallel occlusion culling on GPUs clusters using the occlusion query function provided by current GPUs. As far as the authors of the present papers are aware, there is no previous work on parallel octree occlusion and sorting for multicore systems based on simple and efficient static balancing and $O(n)$ time sorting algorithm.

3. Initial Concepts

One of the main problems in parallel culling using octrees is how to balance the workload amongst the processors. The simple strategy of equally distributing the top level nodes between processors (called static balancing) may result in long idle times in some processors at certain camera positions. An alternative solution to the problems of static balancing is the use of a dynamic balancing strategy, where a processor asks another one for working when it becomes idle. The drawback of this solution is the addition of increasing communication overheads. In this paper, we propose a new and effective strategy called "adaptive delayed static balancing" that has the following characteristics:

1. Instead of distributing the nodes equally amongst the processors at the start of the processing, the algorithm waits until a certain level d (called "split level") in the octree is reached and only then it distributes the work as a static balancing procedure. This characterizes a "delayed" static balancing strategy.
2. Irrelevant nodes are pruned from the tree before the work is distributed amongst the processors.
3. The split level d is dynamically adapted to changes in the virtual environment. This characterizes an "adaptive" strategy.

The reason for the implementation of the above-mentioned delay is that the frustum usually interacts with the nodes in lower levels of the octree. In such lower levels there is a better chance for a more balanced distribution of work. In the proposed algorithm, before the split level d is reached, a sequence of nodes is visited in a breath-first way and a list of nodes (node_list) is prepared for the distribution stage of the algorithm. Irrelevant nodes (*i.e.* branches of the tree with no intersection with the frustum) are automatically pruned from node_list. The nodes from node_list are distributed amongst the processors by creating a list of nodes for each processor and storing it in a vector called working_list. The implementation of this strategy requires the following main tasks:

- To visit the nodes until the split-level is reached
- To set up the list of nodes to be distributed (pruning the octree adequately): node_list
- To expand nodes in node_list
- To render the leaf nodes of the octree that are intersected by the frustum

4. Rendering Nodes

The tasks presented in the previous section can be accomplished by the function RenderNodes(idx, n, node_list, **frustum**). In this paper, "to render nodes" from an octree means to add the objects from a leaf node to a data structure that should be processed by the processor idx considering resource optimizations and GPU communications. The function RenderNodes can transverse the octree completely or stop after n nodes (n = 0 means no limit to transverse the tree). In the case of having a limit (n > 0), this function returns a non-empty node_list containing the nodes to be distributed amongst the processors (including the main processor that is currently setting node_list up). Figure 1 presents the pseudo-code of the function RenderNodes, where frustum is a structure containing the coordinates and orientation of the frustum (which are constantly moving at each frame in time).

```
RenderNodes(idx, n, node_list, frustum)
  set node_count to 0
  while node_list is not empty
      node = first node of node_list
      eliminate first node of node_list
      if frustum intersects node
          if node is a leaf
              add all objects of node to idx data
          else
              put children of node at the end of node_list
      if n > 0          // i.e.: render must stop after n nodes
          increment node_count by 1
          if node_count is equal to n
              break the loop of while
  return node_list
```

Figure 1 The function to render nodes

The function RenderNodes can be executed by the main processor (*e.g.* P1) or one of the secondary processors (*e.g.* P2, P3, or P4). In the case of secondary processors, RenderNodes is executed by another function that is controlled by a thread management system. This later function is RenderNodesProcessorTask(idx, working_list[idx]), where idx is the processor index and working_list is the list of nodes to be processed, as shown in Figure 2. The global vectors startTime[idx] and endTime[idx] are used to calculate the idle time of the processors. The task RenderNodesProcessorTask is controlled by using a new and simple API for thread management proposed in the present paper.

```
RenderNodesProcessorTask (idx,working_list[idx])
  get current time and save it as startTime[idx]
  RenderNodes(idx,0,working_list[idx],frustum
  Get current time and save it as endTime[idx]
```

Figure 2 Calling RenderNodes for processor idx

5. Dynamic Adaptation of the Split Level

Before presenting the complete algorithm proposed in this paper, we should consider the dynamic adaptation of the split level. The best split level (d=0, d=1, d=2, …) is the one that minimizes the sum of the idle time of all processors. Our algorithm employs an adaptive strategy that constantly changes the split level. This strategy is based on the fact that deeper levels tend to reduce the total idle time. Therefore, we expect that each new time frame should increment the split level d. However, depending on the movement of the camera through the virtual environment, the tendency for decaying idle time is broken and decrements in the value of the split level should be tried until the normal trend is recovered (*i.e.* the increase of d causes the decay of total idle time). This is a process that searches

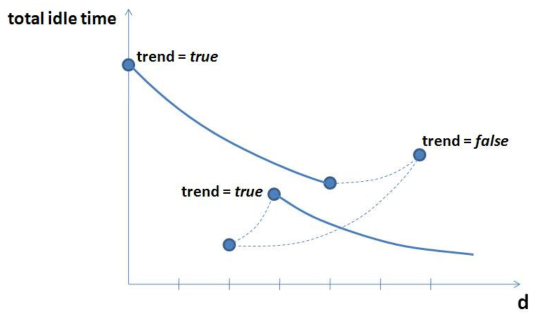

Figure 3 Trends (decaying idle time with increasing d) and adjustment periods

for the optimum value of d. There is no way of deducing a function relating d and total idle time. Experiments have suggested us that trends (solid lines in Figure 3) can be eventually disturbed by adjustment periods (dashed lines in Figure 3). This behavior inspires us to propose the function split_level to dynamically adapt d to changes in the virtual environment based on trends, as shown in Figure 4.

```
Last sum of the idle time of all processors as a global
variable: **last_total_idle_time** = 0
Current trend as a global variable: **trend** = *false*
Number of levels of the octree as a global variable (already
calculated): **num_levels**

split_Level(d, total_idle_time)
  if total_idle_time is greater than last_total_idle_time
      reverse trend      // e.g. trend = not(trend)
  if trend is true
      if d is less than (num_levels – 1)
          increment d by 1
  else
      if d is greater than zero
          decrement d by 1
  last_total_idle_time = total_idle_time
  return d
```

Figure 4 Function split_Level to dynamically adapt d to changes in the virtual environment

6. The Proposed Algorithm

Considering the explanation presented so far, the adaptive delayed static balancing algorithm can be described by the pseudo-code of the function ADStBalancingRender(d, octree, frustum) in Appendix A, where d is the split level (initially zero), octree is a structure containing the octree of the scene, and frustum is a structure containing the coordinates and orientation of the frustum (which are constantly moving at each frame in time). The function ADStBalancingRender is called by the main program at each time frame. This function calls CalculateTotalIdleTime() that is presented in Figure 5.

7. MinTMS

The proposed algorithm considers that the threads are initialized by the main program. This initialization procedure together with three other procedures (used in ADStBalancingRender, Appendix A) are proposed as a simple API for thread management that hides the difficulties of using low level system functions. This API, called MinTMS (for Minimum Thread Management System) (see Appendix A), is described as follows:

Init(n)
This method creates n threads that remain blocked until StartWorking() is called.
SetTask(idx,task,data)

This method sets a task and the data to be processed by the thread idx. The task is executed when StartWorking is called.

StartWorking(idx)

This method unblocks the thread idx. The thread idx returns to a blocked state after processing its task.

WaitUntilWorkFinished()

This method implements a barrier and blocks the main processor until all threads have finished their tasks.

The starting processing time of each processor as a global vector: **startTime[]**
The ending processing time of each processor as a global vector: **endTime[]**
Total number of processors as a global variable: **num_processors**

```
CalculateTotalIdleTime()
    min = stratTime[0]
    max = endTime[0]
    for i = 0 to (num_processors -1) stepping by 1
        if startTime is less than min
            min = startTime[i]
        if endTime[i] is greater than max
            max = endTime[i]
    idle = 0
    for i = 0 to (num_processors -1) stepping by 1
        increment idle by (startTime[i] – min)
        increment idle by (max – endTime[i])
    return idle
```

Figure 5 Function to calculate the total idle time

8. The Proposed Counting Sort Method

8.1 Sorting Algorithms

The main feature of a sorting algorithm [Cormen et al. 2001] is the amount of time required to reorder *n* given numbers into increasing order. However, there are other features to be considered. A sorting algorithm is called *in-place* if it uses no additional array storage (buffer) and is called *stable* if duplicate elements remain in the same relative position after sorting. Mergesort is a stable $O(n \log n)$ sorting algorithm but it is not in-place. Heapsort is an in-place $O(n \log n)$ sorting algorithm, but it is not stable. Quicksort is regarded as one of the fastest sorting algorithm, but it is not stable and, stickling speaking, it not in-place.

It is a well-known theorem that is not possible to sort faster than $O(n \log n)$ time for algorithms based on 2-way comparisons. Sorting numbers faster than this lower bound must be done without the use of comparisons, what is only possible under certain very restrictive circumstances. Under these special conditions, an entire class of linear time sorting algorithm arises. For instance, counting sort is a stable O(n) sorting algorithm, but not in-place, which can only be used in applications that sort small integers. In

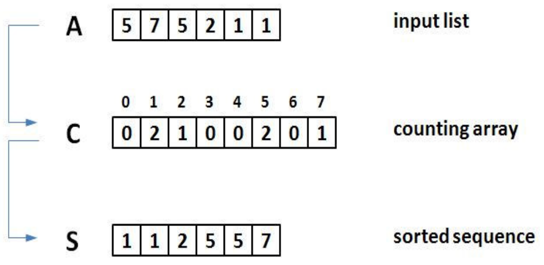

Figure 6 Example of counting sort

this algorithm, for each integer k found in the input list A, we increment the value of C[k] by 1 (the size of C is determined by the largest integer in A), as shown in Figure 6. C[k] is called *counting array*. In the next section, counting sort is presented as the best algorithm for resource sorting in parallel rendering.

8.2 Resource Sorting

Resources are data, properties, components, techniques, and programs used by the 3D objects in order to be rendered properly. Textures, meshes, and pixel shading techniques are common resources used in the rendering processes of real-time applications. Each type of resource defines a discrete axis (*i.e.* an axis with integer coordinate values) called *dimension* (*e.g.* textures are identified by the integer values 0, 1, 2, … in the texture axis). *Resource space* is a discrete Euclidean space defined by one or more dimensions. Therefore, the *texture* dimension and the *mesh* dimension form a two-dimensional resource space. An efficient rendering strategy is the one that groups objects sharing the same resources (*i.e.* it groups the objects in the same point of the resource space). This strategy minimizes the costs associated with every resource change during the rendering process (there is always a great cost associated to jumps within the resource space). In this paper, for each point (i,j,k,…) of the resource space, we define the *n-dimensional resource data array* R[i,j,k,…] containing the following data:

- The number ***c*** of objects sharing the same set of resources i,j,k, …;
- A list ***L*** of these objects.

We use the following notation to present this n-dimensional array:

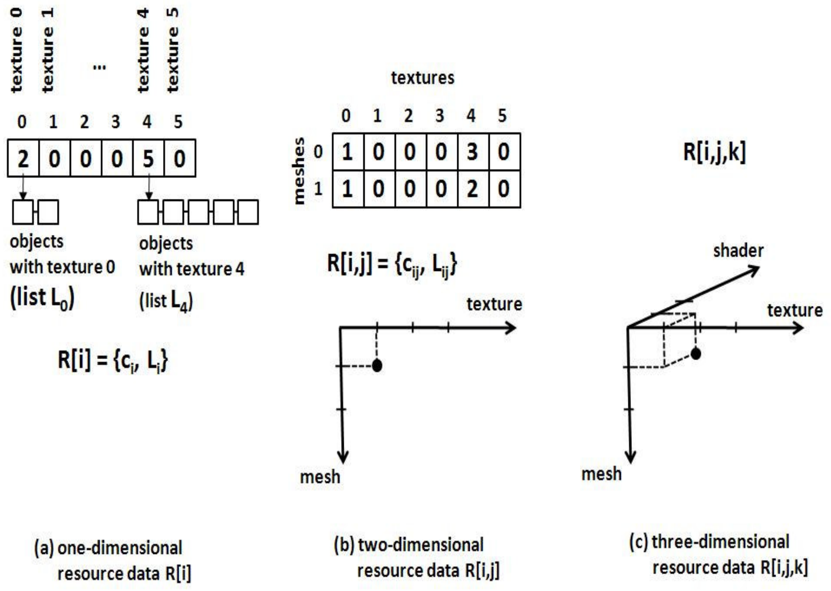
texture 0
texture 1
...
texture 4
texture 5
0 1 2 3 4 5
2 0 0 0 5 0
objects with texture 0 (list L0)
objects with texture 4 (list L4)
R[i] = {ci, Li}
textures
meshes
R[i,j] = {cij, Lij}
texture
mesh
R[i,j,k]
shader
(a) one-dimensional resource data R[i]
(b) two-dimensional resource data R[i,j]
(c) three-dimensional resource data R[i,j,k]

Figure 7 Simple cases of the n-dimensional resource data array R, where *c* is the number of objects and *L* is the list of the objects. The discrete resource spaces are also illustrated.

$$R[i, j, k, ...] = \{c_{i,j,k,...}, L_{i,j,k,...}\} \qquad \text{(Eq.01)}$$

Figure 7 illustrates the simplest cases for R[i,j,k,…]: one, two, and three-dimensional resource data arrays. In Figure 7(b) the two dimensions are texture and mesh. In this 2-dimension example, the rendering process can fix a mesh and render objects per texture (*e.g.* it fixes mesh 0 and renders 1 object with texture 0 and then 3 objects with texture 4).

In the case of one dimension represented by textures (Figure 7(a)), we can easily identify R[i] as being an extended version of the *counting array* C[k] in the counting sort algorithm (Figure 6). The main job of the function RenderNodes (Figure1) is to add objects to the resource data array R of each processor. Therefore, this job is a counting sort process. As resources can be represented by small integer numbers (complex 3D scenes hardly go beyond 300 different textures), the most appropriate sort algorithm for parallel rendering is counting sort. In this way, we have the fastest and convenient option: a *stable O*(*n*) sorting algorithm. We should notice that the *in-place* nature of counting sort (presented in the previous section) is not relevant in the present application, because we need a storage array to distribute work amongst the processors anyway.

8.3 The Sorting Process

The function RenderNodes (Figure1) builds the resource data array R of each processor Pi, in such a way that the objects are distributed amongst the processors and grouped according to the resources they use. In this paper, the proposed algorithm merges the

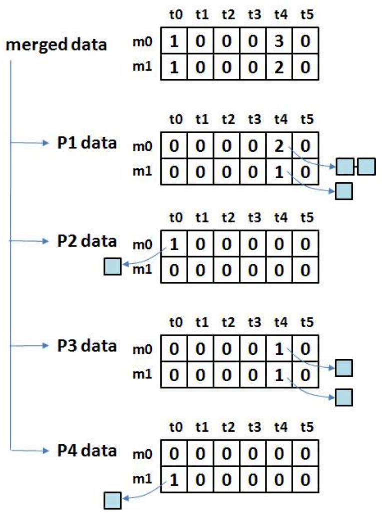

Figure 8 The merged resource data array M

arrays R into a single n-dimensional array M, called *merged resource data array*, by performing the sum of the corresponding $c_{i,j,k,...}$ and transferring the references to the lists $L_{i,j,k,...}$. Figure 8 illustrates the entire merging process for the two dimensional case and four processors. We should notice that Pi data is not inside each processor (in fact the sets of Pi data are in a common structure that each processor can freely access).

Once the merged data array M is completed we can scan it and whenever *c* is greater than zero the list of sorted objects *L* can be rendered using the resources identified by the integer coordinates (i,j,k,…).

9. Some Results

The computer used for tests is a quadcore machine (Intel Core 2 Extreme Q6850 3.00GHz). The GPU rendering performance should be isolated from the performance analysis of the proposed parallel culling method. Therefore, no fps figures are presented.

The first test compares the proposed method with a classical single thread octree culling for an octree with 8 levels (2.396.745 nodes). The result in Table 1 shows an improvement of 3.16.

Table 1 The proposed method (4 threads) *vs* standard Single Thread for an 8-level octree

ADStBalancing (4 Threads)	Single Thread
milliseconds	milliseconds
19	60

The second type of test analyses the adaptive nature of the proposed method by investigating its performance at several values of the split level (d) and the number of nodes processed by each processor. The tests use a camera with FOVy = 30 degrees with a 9-

Table 2 The proposed method with the camera at the centre of a 9-level octree and FOV=30 (intersecting all main nodes right below the root). GPU time is not included.

Split Level	number of nodes				Average Culling Time
d	P1	P2	P3	P4	microseconds
0	2019	482	482	2018	360
1	2025	480	480	2016	362
2	2073	464	464	2000	365
3	897	1880	1880	344	202
4	4761	80	80	80	304

Table 3 The proposed method with the camera at the corner of a 9-level octree and FOV=30 (intersecting only one main nodes right below the root). GPU time is not included.

Split Level	number of nodes				Average Culling Time
d	P1	P2	P3	P4	microseconds
0	4435	2	2	2	288
1	4435	2	2	2	288
2	1945	1888	304	304	214
3	1033	2512	448	448	260
4	4441	0	0	0	292

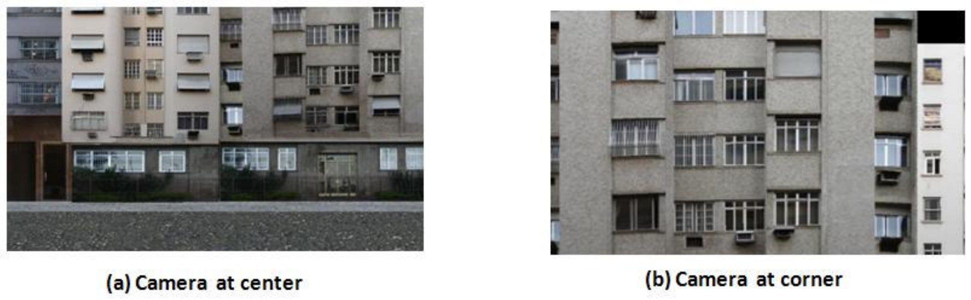

Figure 9 Two different camera positions used in the tests of Table 2 and Table 3

level octree, 4 processors, and 5 values of split levels.

Tables 2 and 3 show that the split level scheme adapts the algorithm for different camera positions. In Table 3, d = 0 is a bad start for both time (a culling time higher than the one for the camera at the center) and workload balancing (number of nodes). In both cases the system stabilizes around d=3 for case 1 (Table 2) and d = 2 in case 2 (Table 3). Figure 9 shows the final rendering for each camera position.

10. Final Remarks

This paper presents a new and effective method for parallel octree culling and sorting for multicore systems using counting sort (which is $O(n)$ time) and based on a new balancing algorithm called *adaptive delayed static balancing*. Tests revealed a time performance improvement of the culling process between 3 and 4 times in relation to the classical single threaded octree culling process. However, the most important result is the effectiveness of the adaptive mechanism based on the dynamic split level that can adjust the algorithm to new camera positions and keep a well-balanced workload amongst the processors. The tests do not consider GPU time and also avoid any connection with the number of resources (*i.e.* number of textures, meshes, …).

Also this paper introduces the concept of *n-dimensional resource space* as a discrete Euclidean space, in which the resource array is identified with the counting array of the counting sort algorithm. No other sorting algorithm can be faster than this $O(n)$ time algorithm for the culling process. The proposed *adaptive delayed static balancing* method naturally generates points in the n-dimensional resource space in a counting sort way.

Another important result is the proposed thread management system MinTMS, which reveals itself as a simple and effective API.

Future works should cover extensive statistics and comparisons, including plots of time *vs* number of nodes, time *vs* number of resources, total idle time *vs* split level, more complex scenes, and more points in the camera path. The comparison with related work is difficult because the literature is scarce on parallel octree culling for multicore machines and we have no access to the code of other authors to reproduce the same test situation. Another future work should be the investigation of other heuristics and statistics that can improve the adaptive performance of the method. Further work should also consider a parallel merging process (*i.e.* to mount the array M in parallel, Figure 8).

Acknowledgements

We would like to thank CNPq and FINEP for the financial support of scholarships and research projects.

References

Cormen, T. H. Leiserson and Charles, E. R. and Ronald, L., 2001. *Introduction to Algorithms Second Edition*. Massachusetts: The MIT Press.

Dalmau, D. S.-C., 2003. *Core Techniques and Algorithms in Game Programming*. Indiana: New Riders.

Foley, J. D. V. D. and Andries, F. and Steven, K., 1995. *Computer Graphics: Principles and Practice in C*. New York: Addison Wesley.

Grama, A. and Gupta, A. and Karypis, G. and Kumar, V., 2003. *Introduction to Parallel Computing*, New York: Addison Wesley.

Wilkinson, B. and Allen, M., 2004. *Parallel Programming Techniques and Applications using Networked Workstations and Parallel Computers*, New Jersey: Prentice Hall.

Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H., 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, 14(4), 1994, pp. 23-32.

Humphreys, G., Houston, M., Ng, R., Frank, R. Ahern, S., Kirchner, P.D., and Klosowski, J.T., 2002. Chromium: a stream-processing framework for interactive rendering on clusters. *Proceedings of ACM SIGGRAPH 2002, acm Transactions on Graphics*, 21(3), 2002, pp. 693-702.

Abraham, F., Celes, W., Cerqueira, R., and Campos, J.L., 2004. A load-balancing strategy for sort-first distributed rendering. *XVII Brazilian Symposium on Computer Graphics and Image Processing, Proceedings SIBGRAPI 2004*, 17-20 Oct 2004, Curitiba, PR, Brazil, IEEE Computer Society, 2004, pp. 292-299.

Baxter, W.V, Sud, A., Govindaraju, N.K., and Manocha D., 2002. Gigawalk: Interactive walkthrough of complex environments. *Proceedings of 13th Eurographics workshop on Rendering*, 2002, pp. 203-214.

Greene, N., Kass, M., and Miller, G., 1993. Hierarchical Z-buffer visibility. *Proceedings of ACM SIGGRAPH 1993, acm Transactions on Graphics*, 1993, pp. 231-238.

Xiong, H., Peng, H., Qin, a., and Shi, J., 2006. Parallel occlusion culling on GPUs cluster. *Proceedings of 2006 ACM International Conference on Virtual Reality Continuum and its Applications* (VRCIA 2006), Hong Kong, China, 14-17 June 2006, pp. 19-26.

APPENDIX A – Algorithm and MinTMS

The index of the main processor as a global constant: **MAIN_PROCESSOR_IDX** = 0
Last sum of the idle time of all processors as a global variable: **last_total_idle_time** = 0
Current trend as a global variable: **trend** = *false*
Number of levels of the octree as a global variable: **num_levels**
Total number of processors and processor id vector as global variables: **num_processors** and **p_idx[ ]**

```
ADStBalancingRender (d, octree, frustum)
    get current time and save it as startTime[MAIN_PROCESSOR_IDX]
    calculate[1] the number of nodes up to the current split level d: n = $(8^{d+1} -1)/7$
    node_count = 0
    clear node_list
    put the root node of octree in node_list
    node_list = RenderNodes(MAIN_PROCESSOR_IDX, n, node_list, frustum)   // n is greater than zero
    set work_size to the size of node_list divided by num_processors
    for  i = 0 to (num_processors-2) stepping by 1       // i is the secondary processor executing the rendering task
        transfer work_size nodes from node_list to working_list[i]
        eliminate the transferred nodes from node_list
        set the task RenderNodesProcessorTask(p_idx[i],working_list[i])  // done by the MinTMS method: SetTask
        make processor i to start the task RenderNodesProcessorTask   // this is done by calling StartWorking(i)[1]
    RenderNodes(MAIN_PROCESSOR_IDX,0,node_list,frustum)       // remaining nodes in the main processor
    get current time and save it as startTime[MAIN_PROCESSOR_IDX]
    If num_processors is greater than 1
        wait for the other processors finish working       // this is done by waitUntilWorkFinished()[1]
    total_idle_time = CalculateTotalIdleTime()
    d = split_Level(d, total_idle_time)
    return d
```

```
#include "ThreadManager.h"
#include <process.h>
ThreadManager* ThreadManager::instance;
ThreadManager::ThreadManager() {
    instance = NULL;
}
ThreadManager* ThreadManager::GetInstance() {
    if(instance == NULL)
       instance = new ThreadManager;
    return instance;
}
void ThreadManager::Init(int thread_count) {
    num_threads = thread_count;
    task_list = new ThreadTask[num_threads];
    data_list = new void*[num_threads];
    start_work = new HANDLE[num_threads];
    work_finished = new HANDLE[num_threads];
    for(int i = 0 ; i < num_threads ; i++)
    {
       start_work[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
       work_finished[i] = CreateEvent(NULL,TRUE,FALSE,NULL);
       int* id = new int;
       *id = i;
      _beginthread(ThreadManager::ProcessorThread,1024,id);
    }
}
void ThreadManager::ProcessorThread(void *data) {
    int processor_idx = *((int*) data);
    ThreadManager* manager = ThreadManager::GetInstance();
    while(true) {
       WaitForSingleObject(manager->start_work[processor_idx],
INFINITE);
       ResetEvent(manager->start_work[processor_idx]);
       manager->task_list[processor_idx](manager-
>data_list[processor_idx]);
       SetEvent(manager->work_finished[processor_idx]);
    }
    delete data;
}
void ThreadManager::SetTask(int thread_idx, ThreadTask task, void
*data) {
    task_list[thread_idx] = task;
    data_list[thread_idx] = data;
}
void ThreadManager::StartWorking(int thread_idx) {
    SetEvent(start_work[thread_idx]);
}
void ThreadManager::WaitUntilWorkFinished() {
    WaitForMultipleObjects(num_threads,     work_finished,     TRUE,
INFINITE);
    for(int i = 0 ; i < num_threads ; i++)
       ResetEvent(work_finished[i]);
}
```

The ThreadManager.h file is presented below:

```
#ifndef THREAD_MANAGER_H
#define THREAD_MANAGER_H
#include <windows.h>
typedef void (*ThreadTask)(void*);
class ThreadManager {
public:
    static ThreadManager* GetInstance();
    void Init(int num_threads);
    void SetTask(int thread_idx, ThreadTask task, void* data);
    void StartWorking(int thread_idx);
    void WaitUntilWorkFinished();
private:
    ThreadManager();
    static ThreadManager* instance;
    int num_threads;
    ThreadTask* task_list;
    void** data_list;
    HANDLE* start_work;
    HANDLE* work_finished;
    static void ProcessorThread(void* data);
};
#endif
```