3 Parallel Techniques for Computer Graphics

This chapter is about parallel techniques for culling and resource sorting. The literature has few works that concentrate on algorithms for parallel culling and sorting using multicore systems. This chapter[1] presents a new and effective method for parallel octree culling and sorting for multicore systems, using counting sort[Cormen01] and based on a new balancing algorithm, called Adaptive Two-Step Static Balancing (A2SSB). The adaptive nature of the balancing algorithm is governed by a dynamic split level that can adjust the algorithm to new camera positions, keeping a well-balanced workload among the processors. This chapter also introduces the concept of n-dimensional resource space as a discrete Euclidean space.

3.1 Introduction

The rendering processing can be organized in the following stages: culling - LOD processing - resource optimization - GPU communication (render calls, primitive transfers, etc). In the present work, we refer to "parallel rendering" as being the parallel algorithms for the first two stages.

Octree culling is a classical technique for reducing the amount of data sent to the GPU for rendering. The technique uses a tree data structure where each node has eight children. Each tree node represents a 3D axis aligned cube as shown on figure 3.1. The objects of the world can be stored on the leaves. 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. Figure 3.1 illustrates the case of a quadtree, where the nodes that are relevant for the frustum are marked by (*). We should notice that the dashed branches indicate irrelevant nodes in respect to the frustum and, therefore, must be pruned from the tree. Only the leaf nodes marked with (*) have their objects

[1]A preliminary version of this chapter was published by ACM(Journal of Computers in Entertainment) as one of the top 10 papers in SBGames 2008 [Machado09]

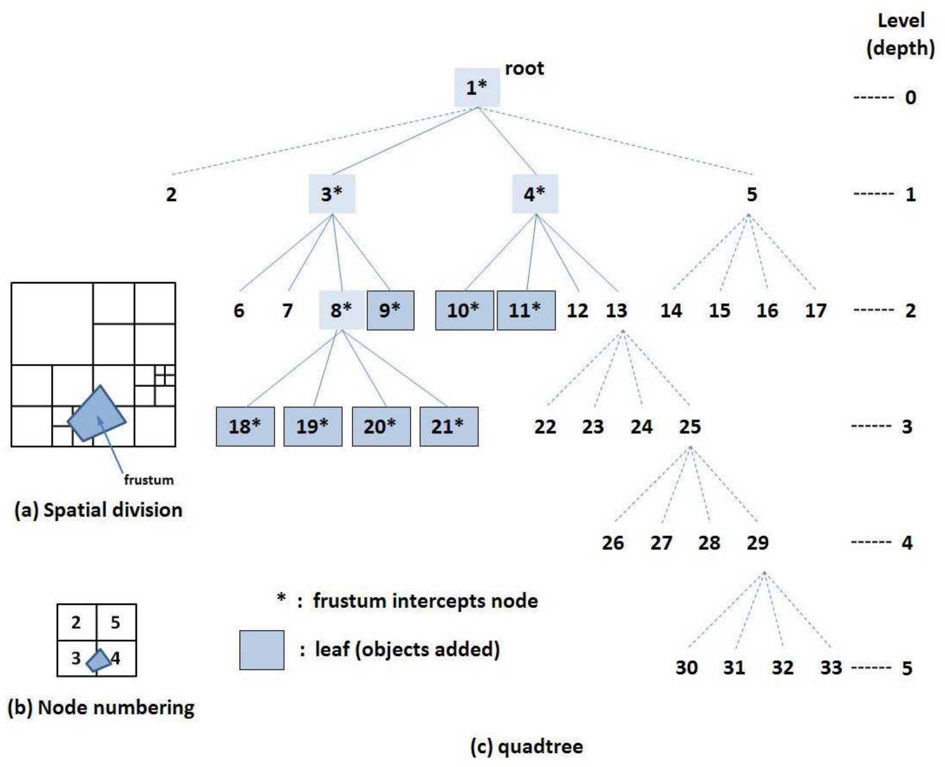


Figure 3.1: Example of spatial division with quadtree. Nodes that are relevant for the frustum are marked by (*) and irrelevant ones have dashed branches.

rendered. In the present work, the expression “to render objects at a node” means to add the objects from a leaf node (which intersects the frustum) to a data structure used for resource sorting that will be processed in a later rendering stage. We call “node processing” the process of visiting the tree’s nodes and rendering the objects from a node when necessary.

Node processing in octree culling can be easily parallelized, but the balance of the workload is not trivial. The main challenge is to find an efficient strategy to maintain the workload balanced while the camera moves.

Another aspect of the rendering process is resource sorting. Resources are data, properties, and shaders used by 3D objects in their rendering process, such as: textures, meshes, and shaders. There is always a cost associated to resource changes. Therefore, these changes should be reduced by sorting and grouping objects with common resources. The process of distributing the objects amongst several processors according to resource data is not trivial.

Most of the methods for parallel rendering are concentrated on PC clusters and grids, while the literature on parallel rendering for multicore systems is scarce. The solution for Parallel Octree Culling on a PC cluster is significantly different than the solution on a multicore system because the hardware architecture has a great influence on how a parallel solution is implemented. This work 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 Two-Step Static Balancing (A2SSB). Tests reveal a performance improvement of the culling process between three and four times on a quadcore computer in relation to the classical single threaded octree culling process. However, the most important performance analysis concerns the capability that the proposed algorithm has to adapt itself to new camera positions, which are continuously changing over time. Furthermore, the proposed balancing algorithm is greatly improved with the use of a cache memory strategy. We cannot find references on the use of cache memory strategies for parallel rendering in the literature. As was said before, we think that the reason for this lack of references is that works on parallel rendering concentrate on PC clusters and/or on global aspects of parallel rendering, rather than on multicore systems as required by the videogame industry.

3.2 Related Work

A lot of research on parallel search and sorting algorithms has already been done, and many techniques are available in the literature [Grama03, Wilkinson05]. Also, several works have been carried out in the area of parallel rendering using sorting techniques. Molnar et al. [Molnar94] propose a classification of parallel rendering system based on a stage of the rendering pipeline when the sorting is carried out (sort-last, sort-middle, and sort-first). Humphreys et al. [Humphreys02] present a sort-first method for distributed rendering using a cluster of common PCs. Abraham et al. [Abraham04] propose a load-balancing strategy for sort-first distributed rendering using PC-based clusters. Baxter et al. [Baxter02] present a parallel rendering architecture using two graphic pipelines 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; our focus is on multicore computers.

Octree is a classic data structure used in many computer graphics applications [Foley95, Dalmau03, Moller08] and a lot of research has been made to optimize its algorithms, for example, Castro et al. [Castro08] present a solution that uses Hashed Octrees and Statistical Optimization to improve the search for a leaf. However, parallel culling algorithms using octrees are not usual. Greene et al. [Greene93] are the first authors to propose an

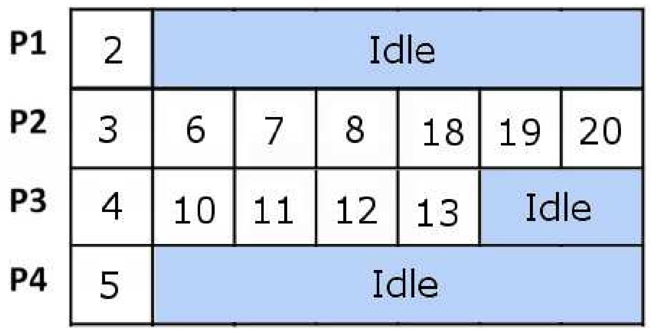


Figure 3.2: A possible work distribution for the quadtree of figure 3.1 at level 1.

octree hierarchy for visibility computation with some potential for parallel implementation. Their work had a great influence on graphics hardware design. Xiong et al. [Xiong06] present an algorithm for parallel occlusion culling on GPU clusters using the occlusion query function provided by current GPUs. As far as the author of the present work is aware, there is no previous work on parallel octree culling and sorting for multicore systems based on simple and efficient static balancing and O(n) time resource sorting algorithm.

3.3 An Adaptative Strategy in Two Steps

One of the main problems in parallel culling using octrees is how to balance the workload between the processors. The simple strategy of equally distributing the top level nodes between processors (a type of static balancing) may result in very long idle times in some processors at certain camera positions. Figure 3.2 illustrates a possible node distribution at level 1 of the tree in figure 3.1. As can be seen, this load balancing strategy results in processors P1 and P4 remaining idle after processing a single node and P2 and P3 doing most of the work.

An alternative to static balancing is the use of a dynamic balancing strategy. One example of such strategy is making a processor ask another one for work when it becomes idle. However, dynamic balancing algorithms have a serious drawback, which is the increasing communication overheads, especially for multicore systems with a large number of processors. The area of games and real-time simulation suffers from the lack of efficient solutions for parallel rendering in multicore systems. As we have mentioned before, most of the literature is devoted to PC clusters.

In the present work, we propose a static balancing algorithm for any number of processors that can be dynamically adjusted according to the

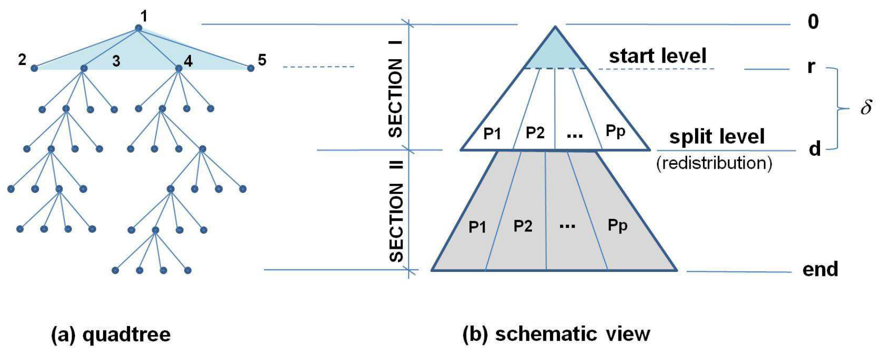


Figure 3.3: The split level of the A2SSB algorithm for p processors revealing the two-step nature of the proposed algorithm

camera's movements. Furthermore, we explore the multicore nature of the hardware. We call this new algorithm "Adaptive Two-Step Static Balancing" or A2SSB for short.

In the proposed algorithm, we observe that unbalanced workloads tend to become worst at deeper levels where nodes are becoming closer to the camera frustum. This suggests that we can define a special tree level from which we should adjust our initial balancing strategy. In the present work, this special level is called "split level". Therefore, we propose that the octree be divided into two sections by the "split level" (figure 3.3). In SECTION I, the nodes at the start level r are equally distributed amongst the p processors, which is an ordinary static balancing procedure. In this first step of the algorithm, the processors use the main memory (RAM). When the split level d is reached by all processors, the following adjustments are performed:

1. the workload is redistributed amongst the p processors;

2. a cache memory strategy is adopted (*i.e.*, RAM is not used).

These adjustments trigger the second step of the proposed algorithm (SECTION II in figure 3.3b), which is also a static balancing procedure. Irrelevant nodes are pruned from the octree before the workload is redistributed amongst the processors. In sections I and II, each processor performs a depth-first procedure. The solution proposed requires a balanced Octree that has all the leafs at the same level.

The algorithm is executed each time a new rendered frame is required, that is, each time the camera moves at time *t0, t1, t2,* ... . The split level d changes at each time step looking for an optimal level that reduces the idle time of the processors. The last task performed by the A2SSB algorithm is to update the value of the split level based on the history of the total idle time

- this important feature represents the adaptive nature of the algorithm. In section 3.6, a very simple method of updating the split level is proposed.

In the “second step” of the algorithm, where the nodes are closer to the frustum, the algorithm uses a special cache memory strategy, which we call “cache friendly strategy for octree node computing”. This strategy avoids memory bottlenecks and permits each processor to run at full speed. The use of cache memory is not an issue in parallel rendering algorithms using PC clusters. However, in multicore systems, this is mandatory. The use of a cache memory strategy in the “first step” of the algorithm is not worthy, because of the low number of nodes processed at the first levels and the higher cost of workload redistribution for such strategy.

The start level r is determined by the “start-level condition”, that is, a condition that ensures an equal distribution of the start level nodes such that all p processors have at least one node to process. The start-level condition is an important requirement to guarantee an adequate starting of the static balance in SECTION I. This condition is easily given by the following equation:

$$8^r \geq p \tag{1}$$

On equation 1, p is the number of processors. In the quadtree example of figure 3.3a, the start level is 1 for the case of a system with 4 processors, because $4^r \geq 4$ gives r = 1. If the situation is a quadtree using 8 processors, the start level would be level 2, because $4^r \geq 8$ gives r = 2. In this latter case, 16 nodes would be equally distributed amongst 8 processors (2 nodes per processor).

The values of the split level are calculated at the end of the time intervals *t0, t1, t2, ...*. The split level begins with a value that is equal to the start level, *i.e., d(t0) = r*. This means that the children of the visible start level nodes will be redistributed between the cores after the first step of execution. Then the split level is incremented by 1 at the end of *t0*.

At the end of the first step (*i.e.* the end of Section I in the octree) redistribution is performed. When the entire octree is traversed by all processors (*i.e.* the end of Section II is reached), a new split level is calculated.

3.4 Node Processing in Step One

The first step of the algorithm is the parallel depth-first computation of the octree until the split level is reached. This is done by the following recursive function $StepOneNodeProcessing$(Algorithm 1), where $nodeList$ is a dynamic buffer that stores the nodes of the split level (image 3.4 shows how $nodeList$ is used by steps one and two of the node processing):

Algorithm 1 StepOneNodeProcessing(node, level)

```
if frustum intersects $node$ then
  if $node$ is a leaf then
    render objects at $node$
  else
    if $level$ is equal to the split level then
      add children of $node$ to $nodeList$
    else
      for all $i$ from 0 to 7 do
        StepOneNodeProcessing($node \rightarrow children[i]$, $level + 1$)
      end for
    end if
  end if
end if
```

The function $StepOneNodeProcessing$ is called by each node at the start level r, that is, the variable *level* is equal to the start level when this function is called by the first time.

3.5 Node Processing in Step Two

In Step Two of the algorithm (corresponding to SECTION II in Figure 3.3), we adopt a node processing strategy that uses cache memory only - that is: the main memory (RAM) is not used. This is not an easy task. Programmers have very little control over cache memories (even through assembly language) and must mainly rely on their knowledge about the general characteristics of a particular multiprocessor architecture to write programs that avoids cash load misses. If few variables are defined and their values are constantly updated, there will be a great chance for small algorithms to run on cache memories. We can control this behavior by tracking the number of cache misses. This is more a strategy than a precise programming technique and we call it a "cache friendly strategy for octree node computing". In the present work, the tests are made on an Intel Core 2 Extreme Quad-core Processor with dual 4 MB of L2 Cache. The architecture of this quad-core system is shown in Figure 3.5a [Intel09].

The nature of the octrees permits the implementation of a cache friendly strategy. In fact, octrees can be traversed node-by-node with no need for information concerning the rest of the data structure - in other words, we can operate locally. This is possible if the following assumption is made: the octree is built through a uniform partition, as illustrated in Figure 3.6. In this case, all leaves are at the same level, and any level i has exactly 8^i nodes. This assumption can also be used to simplify calculations concerning the start level

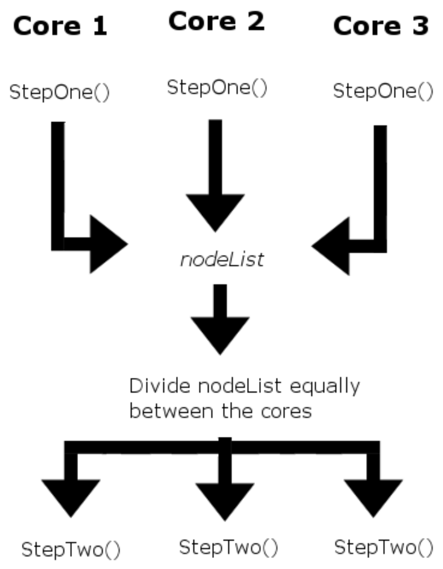


Figure 3.4: *nodeList* being used by the Node Processing Step One and Two

r, even before getting into the first and second step of the algorithm. The nodes at the start level r can be equally distributed amongst p processors, because 8^r nodes are divided by an even number (since the number of processors is normally a power of 2).

In a uniform octree, the implementation of a depth-first strategy is merely a matter of navigating node-by-node until the last level is reached. This navigation is reduced to a geometric calculation of the octree cells - called “boxes” in the present work. The octree nodes are directly represented by their boxes. At any level, the first child of a box can be reached by simply dividing the dimensions of the box by 2. Furthermore, the brothers of this first child can be accessed by an adequate translation of the box dimensions. The parent of a box (that is, a backtracking operation) can also be easily defined. All these calculations can be implemented by a constant updating operation on a limited number of variables, that keeps the node processing within the cache memory - that is: no repeated query to a RAM data structure is required for navigating the Octree. Figure 3.7 facilitates the explanation of these calculations. In the equations below, box.max$_0$, box.max$_1$, and box.max$_2$ mean x$_{\mathrm{max}}$, y$_{\mathrm{max}}$, and z$_{\mathrm{max}}$ of the box respectively. The same type of notation is used for x_{min}, y_{min}, and z_{min}. In this case, the first child of a box can be defined by:

$$box.max_i = \frac{box.min_i + box.max_i}{2}, i = 0, 1, 2 \qquad (2)$$

For the equation 2, the values of i represent the axis x, y, and z. This calculates the centroid of the box.

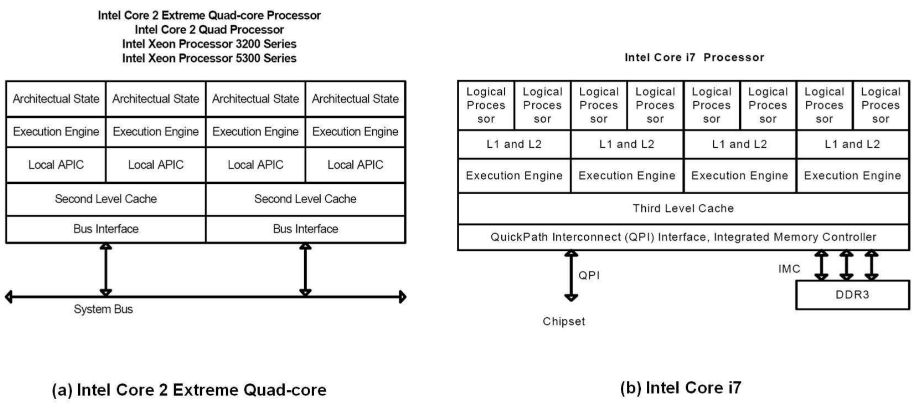

Intel Core 2 Extreme Quad-core Processor
Intel Core 2 Quad Processor
Intel Xeon Processor 3200 Series
Intel Xeon Processor 5300 Series
Architectual State
Execution Engine
Local APIC
Second Level Cache
Bus Interface
System Bus
Intel Core i7 Processor
Logical Processor
L1 and L2
Execution Engine
Third Level Cache
QuickPath Interconnect (QPI) Interface, Integrated Memory Controller
QPI
IMC
DDR3
Chipset
(a) Intel Core 2 Extreme Quad-core
(b) Intel Core i7

Figure 3.5: Intel quad-core processors. In the present work, tests are made on the Intel Core 2 Extreme Quad-core processor (a) [Intel09]

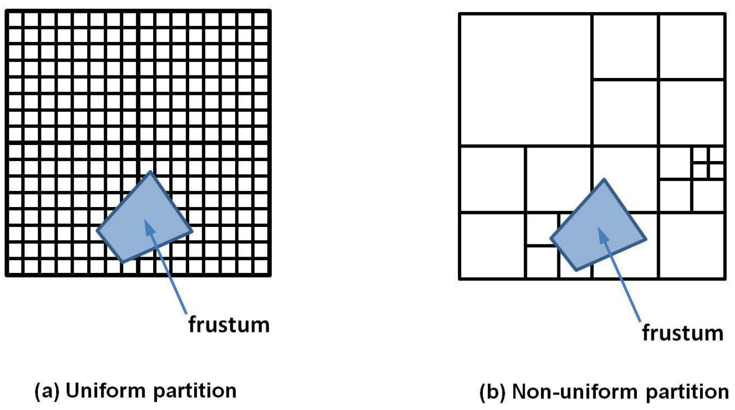


Figure 3.6: Only an uniform octree partition (a) is allowed by the proposed algorithm at step two.

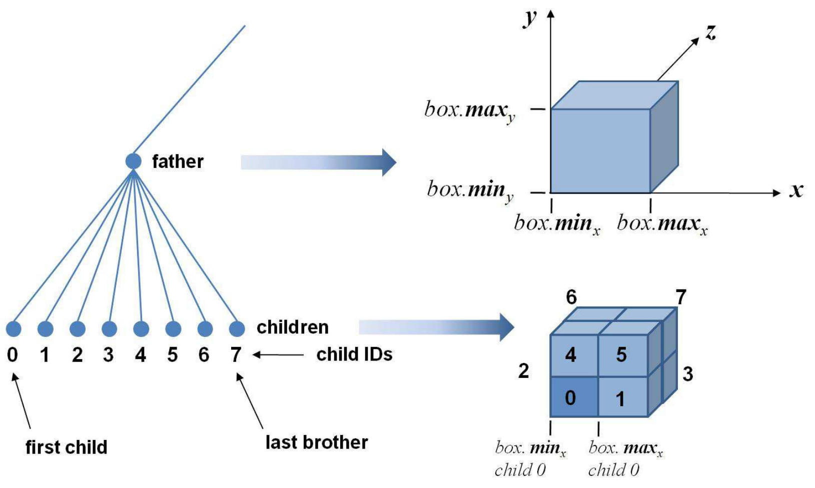


Figure 3.7: Octree nodes as boxes. The numbers are the child IDs.

$$trans[ID][i] = \begin{array}{ccc|c} x & y & z & \\ \downarrow & \downarrow & \downarrow & \\ \hline 1 & 0 & 0 & \leftarrow 1 \\ -1 & 0 & 1 & \leftarrow 2 \\ 1 & 0 & 0 & \leftarrow 3 \\ -1 & 1 & -1 & \leftarrow 4 \\ 1 & 0 & 0 & \leftarrow 5 \\ -1 & 0 & 1 & \leftarrow 6 \\ 1 & 0 & 0 & \leftarrow 7 \end{array}$$

Figure 3.8: The translation matrix that is used for finding all brothers of an octree node

When we are at child 7 as shown in figure 3.7, the father of a box can be calculated as follows:

$$box.min_i = box.min_i - (box.max_i - box.min_i) \tag{3}$$

The navigation amongst the brothers of a node needs a more creative implementation. Firstly, we should notice that the first brother in the x direction (ID = 1, in Figure 3.7) can be defined as a translation in x, that is:

$$box.min_i = box.min_i + delta_i, i = 0, 1, 2 \tag{4}$$

$$box.max_i = box.max_i + delta_i, i = 0, 1, 2 \tag{5}$$

where:

1. $delta_0 = box.max_0 - box.min_0$
2. $delta_1 = 0$
3. $delta_2 = 0$

We can generalize the definition of brothers as translations in x, y and z by using the translation matrix defined in figure 3.8:

For example, if we are the second child (ID=1) and go to the next child (ID=2), we should translate backwards in x(-1), make no movement in y(0), and translate fowards in z(1), that is:

1. $trans[2][0] = -1$

2. $trans[2][1] = 0$
3. $trans[2][2] = 1$

Therefore, we rewrite Eq.4 for the brother ID as:

$$box.min_i = box.min_i + delta_i, i = 0, 1, 2 \tag{6}$$

$$box.max_i = box.max_i + delta_i, i = 0, 1, 2 \tag{7}$$

where:

$$delta_i = (box.max_i - box.min_i) \times trans[ID][i] \tag{8}$$

The proposed algorithm reuses the same variable *box* while navigating the octree, a part of the strategy that greatly reduces cache misses. In the Algorithm 2, we should notice the following facts:

1. *child_stack* is a buffer with size equal to the number of levels in the octree;
2. each element of *child_stack* has a size of a single byte;
3. *child_id* represents the child IDs (0,1,2,3,4,5,6 and 7) in figure 3.7 and the value of -1 is used as a flag;
4. The last level of the octree is a known value;
5. There are two internal loops inside an infinite loop. The first internal loop executes until a child is generated that is not intersected by the frustum or we arrive at a leaf. The second internal loop will keep going up one level on the octree (*i.e.* generating fathers), if the child ID is 7;
6. At the end of the infinite loop code, if *child_id* is equal to -1, we are back to the first node and the function should end. If we are not at the first node, we generate the next brother, increment *child_id*, and we save the new *child_id* in *child_stack* at the current octree level.

The only occasion the algorithm escapes from the cache memory and accesses the RAM is when the objects of a box should be rendered. All the performance tests are made without this part of the algorithm, because we only want to evaluate the octree node processing. A further investigation could be the search for the best strategy to add objects to the rendering data structure.

Algorithm 2 StepTwoNodeProcessing(box, level, child_stack)

```
 1: $curr\_level = level$ {current level is set up}
 2: $child\_id = -1$
 3: $child\_stack[curr\_level] = child\_id$
 4: while true do {infinite loop}
 5:   while frustum intersects box do
 6:     if $curr\_level$ is equal to the last level of the octree then
 7:       render objects in box {this is the only point where RAM is accessed}
 8:       break the loop
 9:     else {generate first child}
10:       for each i from 0 to 2 do
11:         $box.max[i] = (box.min[i] + box.max[i]) \times 0.5$
12:       end for
13:       $child\_id = 0$
14:       increment $curr\_level$ by 1
15:       $child\_stack[curr\_level] = child\_id$
16:     end if
17:   end while
18:   while $child\_id$ is equal to the last brother's ID ($i.e.$ 7) do
19:     {generate the father and get the $child\_id$ of the father}
20:     for each $i$ from 0 to 2 do
21:       $box.min[i] -= (box.max[i] - box.min[i])$
22:     end for
23:     decrease $curr\_level$ by 1 {that is, go up on level}
24:     $child\_id = child\_stack[curr\_level]$
25:   end while
26:   if $child\_id$ is not equal to -1 then
27:     for each $i$ from 0 to 2 do {generate next brother}
28:       $delta = (box.max[i] - box.min[i]) \times trans[child\_id][i]$
29:       $box.min[i] += delta$
30:       $box.max[i] += delta$
31:     end for
32:     increment $child\_id$ by 1
33:     $child\_stack[curr\_level] = child\_id$
34:   else
35:     exit the function
36:   end if
37: end while
```

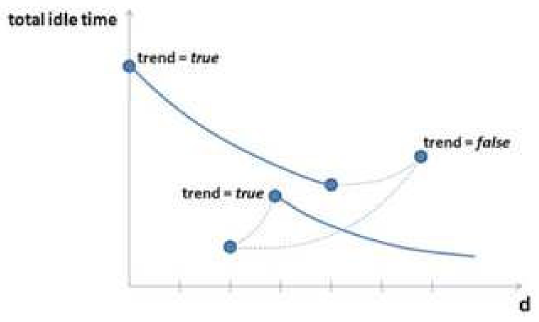


Figure 3.9: Examples of trends affecting the split level d and the total idle time. At first the trend is set as true and is constantly increasing d. This reduces the total idle time until we arrive at a moment where the increase of d also increased the total idle time, this results in the trend being set to false and we repeat the process, but now the trend constantly decreases d.

3.6 Dynamic adaptation of the Split Level

Before presenting the complete algorithm proposed in this work, 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 times of all processors. Our algorithm employs an adaptive strategy that constantly changes the split level according to the tendency of the total idle time. There is no way of deducing a function relating d and total idle time. Experiments have suggested that trends (solid lines in Figure 3.9) can exist and eventually be disturbed by adjustment periods (dashed lines in Figure 3.9).

This behavior stimulates us to propose the function *AdaptativeSplitLevel* to dynamically adapt d to changes caused by the travelling camera. Figure 3.9 is a mere illustration of a possible trend being adjusted.

In the proposed *AdaptativeSplitLevel* function, if an increment on the split level, at time t_i reduces the amount of idle time between the processors, we repeat the increment for the next time frame t_{i+1}. Otherwise, if the increment increases the total idle time, we revert the operation decrementing the split level at each new frame. The pseudocode of *splitLevel* is presented below, where *trend* (initially equal to *true*) and $LastIdleTime$ are global variables:

The *AdaptativeSplitLevel* function never allows the split level to go before the start level r. The value of idle time is calculated by the $CalculateIdleTime$ function shown at algorithm 4:

In algorithm 4, $startTime$ and $endTime$ are global variables that store the time of the execution start and end, from each processor.

Algorithm 3 AdaptativeSplitLevel(f, r, idleTime)

```
1: if $idleTime$ is greater than $LastIdleTime$ then
2:     reverse trend {e.g. trend = not(trend)}
3: else if $trend$ is true then
4:     if $d$ is less than the last level of the octree then
5:         increment $d$ by 1
6:     end if
7: else
8:     if $d$ is greater than the start level $r$ then
9:         decrement $d$ by 1
10:    end if
11: end if
12: $LastIdleTime = idleTime$
13: return $d$
```

Algorithm 4 CalculateIdleTime()

```
1: $min = startTime[0]$ {0 is the main processor}
2: $max = endTime[0]$
3: for each secondary processor i do {there are p-1 secondary processors}
4:     if $startTime[i]$ is less than $min$ then
5:         $min = startTime[i]$
6:     end if
7:     if $endTime[i]$ is greater than $max$ then
8:         $max = endTime[i]$
9:     end if
10: end for
11: $idleTime = 0$
12: for each processor $i$ do
13:     increment $idleTime$ by $(startTime[i] - min) + (max - endTime[i])$
14: end for
15: return $idleTime$
```

3.7 The full Algorithm

Before the full A2SSB algorithm is invoked, the nodes at the start level r are distributed and stored in a list belonging to each processor, which we call "starting node list". The processors that are different from the main processor are called secondary processors (therefore, there are one main processor and $p-1$ secondary processors). In the present work, the prefix Main indicates a node list that belongs to the main processor ($MainStartingNodeList$ and $MainNodeList$) and the prefix Second indicates a node lists on a secondary processor ($SecondStartingNodeList$ and $SecondNodeList$).

The main processor is always in charge of the entire process and it is part of its job to request that the other processors start processing their node lists. In terms of a real code, this request is made by invoking functions defined within a Thread Manager. The pseudocode that will be presented does not mention the usage of these Thread Manager functions neither mention other tasks of the secondary processors (such as saving their startTime and endTime values). Also, we should notice that the main processor must wait until all other processors finish their work. In a well-balanced parallel algorithm, these waiting states have short durations because the idle time is minimized.

The pseudocode for the Adaptive Two-Step Static Balancing (A2SSB) can be seen in algorithm 5.

Algorithm 5 A2SSB(frustum)

```
1: save the current time in $startTime[0]$ {0 is the main processor}
2: for each secondary processor k do
3:   for each i from 0 to the size of $SecondStartingNodeList$ do
4:     StepOneNodeProcessing($SecondStartingNodeList[i]$, $r$) {r is the start level}
5:   end for
6: end for
7: for each i from 0 to the size of $MainStartingNodeList$ do
8:   StepOneNodeProcessing($MainStartingNodeList[i]$, $r$)
9: end for
10: wait until secondary processors finish their work
11: {All nodes at split level were stored in the $MainNodeList$ by the Step One of the algorithm. Now theses nodes should be equally divided amongst the $p$ processors}
12: $np = \frac{\text{size of } MainNodeList}{p}$ {np = number of nodes for each processor}
13: for each secondary processor k do
14:   transfer $np$ nodes from $MainNodeList$ to the $SecondNodeList$ of processor k
15:   for each i from 0 to $np - 1$ do
16:     StepTwoNodeProcessing(SecondNodeList[i]$\rightarrow$box,$d$+1, $child\_stack[k]$)
17:   end for
18: end for
19: for each i from 0 to $np - 1$ do
20:   StepTwoNodeProcessing(MainNodeList[i]$\rightarrow$box,$d + 1, child\_stack[0]$)
21: end for
22: save the current time in $endTime[0]$
23: wait until secondary processors finish their work
24: {update the split level}
25: $idleTime = CalculateIdleTime()$
26: $d = AdaptativeSplitLevel(d, r, idleTime)$
```

As mentioned before, some parts of the pseudocode do not reveal typical tasks of a thread manager, such as to set specific tasks, trigger working orders, and initialize/finish time monitoring. Therefore, in the above pseudocode, when $CalculateIdleTime()$ is evoked the vectors $startTime$ and $endTime$ are supposed to be properly set up.

3.8 Performance Results

The performance results are analyzed for the octree node processing only - that is: no other processing (*e.g.* rendering processing) is considered. This performance analysis is carried out for five variants of the algorithm:

1. ALGO1 - Full A2SSB algorithm
2. ALGO2 - A2SSB without Cache Friendly Strategy

   In this version of A2SSB, after the nodes at the split level are distributed amongst the p processors, the Step Two of the algorithm uses an ordinary depth-first technique, where each processor accesses the octree in the main memory (RAM) - that is, the cache friendly strategy is not used.
3. ALGO3 - Breadth-first A2SSB without Cache Friendly Strategy

   This version is similar to the previous one, except that a breadth-first technique is used instead of a depth-first one.
4. ALGO4 - Single Processor with a Cache Friendly Strategy

   This is a single processor version where the cache friendly strategy is used from the octree root.
5. ALGO5 - Single Processor with Depth-first Strategy

   This is the canonical version of octree navigation in a depth-first way and using one processor only.

These five variants are used to process 299,593 octree nodes (based on a octree of 7 levels) with four processors (p = 4) in an Intel Core 2 Extreme Quad-core computer. The results are presented in the table of figure 3.10.

Although navigating an octree without accessing the RAM seems to be a good technique for a single processor computer, ALGO4 reveals that the impact on performance is almost none in relation to the canonical algorithm (ALGO5). ALGO3 shows that breadth-first strategy in multicore systems is a bad choice, presenting a performance similar to the canonical algorithm running on a single processor (ALGO5). The power of a cache friendly strategy in multicore systems is evident when we compare the results of ALGO1 and ALGO2.

Algorithm	Time (milliseconds)
ALGO1 - Full A2SSB algorithm	4.0
ALGO2 - A2SSB without Cache Friendly Strategy	7.1
ALGO3 - Breadth-first A2SSB without Cache Friendly Strategy	16.6
ALGO4 - Single Processor with a Cache Friendly Strategy	15.0
ALGO5 - Single Processor with Depth-first Strategy	15.5

Figure 3.10: Performance analysis of octree node processing algorithms for an octree with 299,593 nodes and running on an Intel Core 2 Extreme Quad-core computer.

Function	Cache Misses	Percentage
	(millions)	(Cache Misses/TotalCM)
Testing intersection between frustum and box	4	36.36
StepTwoNodeProcessing	3	27.27
Testing intersection between frustum and box	3	27.27
Testing intersection between frustum and box	1	9.09
TotalCM	11	

Figure 3.11: L2 cache misses for the proposed algorithm A2SSB (ALGO1 in Table of figure 3.10)

The quality of the cache friendly strategy can be evaluated by inspecting the following main evaluation parameters: the total amount of cache misses and the number of functions where cache misses take place. The proposed algorithm A2SSB (with cache friendly strategy) presents much lower values for those main evaluation parameters (Table of figure 3.11) than the ones presented by A2SSB without a cache friendly strategy (Table of figure 3.12).

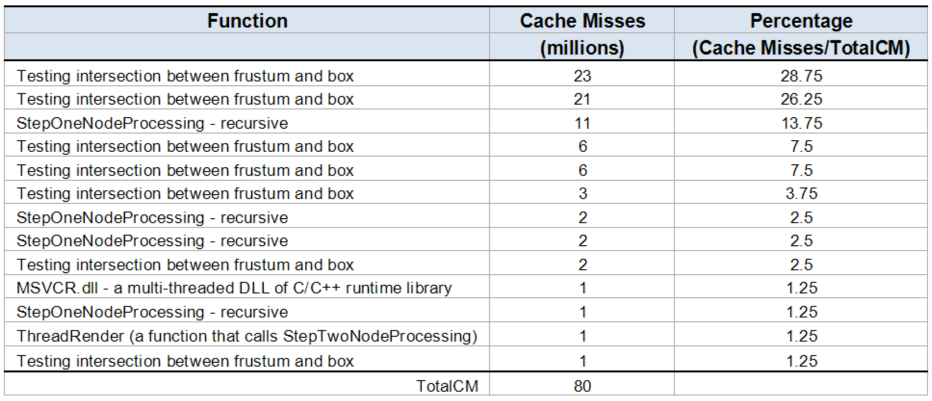
3.9 Algorithm Analysis

The analysis of the A2SSB algorithm starts by the identification of its three main stages (see Figure 3.3), which are responsible for the three parts of the execution time cost:

1. Firstly, the octree is processed by all p processors from the start level r to the split level d. This causes a time cost we call $S1$ - the cost of the Step One of the algorithm.

2. Secondly, the nodes intersecting the view frustum so far are distributed amongst the p processors. This is the transfer cost T.

3. Thirdly, similarly to the first stage, all p processors compute their portions of the octree until they arrive at the leaves. This is the cost of the Step Two of the algorithm: $S2$.



Function	Cache Misses (millions)	Percentage (Cache Misses/TotalCM)
Testing intersection between frustum and box	23	28.75
Testing intersection between frustum and box	21	26.25
StepOneNodeProcessing - recursive	11	13.75
Testing intersection between frustum and box	6	7.5
Testing intersection between frustum and box	6	7.5
Testing intersection between frustum and box	3	3.75
StepOneNodeProcessing - recursive	2	2.5
StepOneNodeProcessing - recursive	2	2.5
Testing intersection between frustum and box	2	2.5
MSVCR.dll - a multi-threaded DLL of C/C++ runtime library	1	1.25
StepOneNodeProcessing - recursive	1	1.25
ThreadRender (a function that calls StepTwoNodeProcessing)	1	1.25
Testing intersection between frustum and box	1	1.25
TotalCM	80	

Figure 3.12: L2 cache misses for the version of A2SSB without a cache friendly strategy (ALGO2 in Table of figure 3.10)

Therefore, the total cost is given by:

$$C = (S1 + S2) + T \tag{9}$$

In the worst case, the whole tree is visible and intersects the frustum. This means that the whole octree will be visited. Since each node should be visited only once, this is an O(n) algorithm where n is the number of octree nodes.

(a) The S1 cost

The cost of the first stage is given by:

$$S1 = \frac{S_\delta}{p} \tag{10}$$

and

$$S_\delta = n_d - n_r \tag{11}$$

where n_d in the number of nodes up to the split level d and n_r is the number of nodes up to the start level r. These number of nodes n_d can be calculated as follows:

$$n_d = 8^0 + 8^1 + ... + 8^d \tag{12}$$

If we multiply Eq.12 by 8 and substitute $8^1 + 8^2 + ... + 8^d$ by $(n_d - 1)$, we have:

$$8n_d = (n_d - 1) + 8^{d+1} \tag{13}$$

and:

$$n_d = \frac{8^{d+1} - 1}{7} \tag{14}$$

Eq.14 can also be deduced from Eq.12 using the canonical formula for the sum of a geometric progression.

Similarly, we have:

$$n_r = \frac{8^{r+1} - 1}{7} \tag{15}$$

From Eq.10, Eq.14, and Eq.15, we have:

$$S1 = \frac{8^{d+1} - 8^{r+1}}{7p} \tag{16}$$

(b) The T cost

The cost of transferring the nodes to the p processors at the second stage of the algorithm is given by:

$$T = n_p(p - 1) \tag{17}$$

where n_p is the number of nodes that each processor should compute. We should notice that n_p nodes remain on the main processor. The value of n_p is calculated by dividing the number of children of the split level (*i.e.* 8^{d+1} nodes) by p. Therefore, we have:

$$n_p = \frac{8^{d+1}}{p} \tag{18}$$

From Eq.17 and Eq.18, we get:

$$T = \frac{8^{d+1}}{p}(p - 1) \tag{19}$$

(c) The S2 cost

The third stage is where the bulk of the computation takes place. This stage computes the lower levels of the octree in parallel. Each node that a processor has to handle represents an octree with a height h_{S2} given by:

$$h_{S2} = h - (d + 1) \tag{20}$$

where h is the height of the full octree.

Therefore, the equation for the $S2$ cost is given by:

$$S2 = n_p n_{S2} \tag{21}$$

where n_p is the number of octrees to be processed by each processor (given by Eq.18) and n_{S2} is the number of nodes of each octree of height h_{S2}. The equation of $S2$ is similar to Eq. 14, that is:

$$n_{S2} = \frac{8^{h_{S2}+1} - 1}{7} \tag{22}$$

From Eq.20, Eq.22, and Eq.18, we transform the Eq.21 into the final form of the $S2$ cost equation. Firstly we get:

$$S2 = (\frac{8^{d+1}}{p})(\frac{8^{h-d}-1}{7}) \tag{23}$$

and then, by multiplication, we have:

$$S2 = \frac{8^{h+1} - 8^{d+1}}{7p} \tag{24}$$

(d) The total cost C

The total cost of the algorithm in terms of execution time is given by substituting Eq.16, Eq.19, and Eq.24 into Eq.9:

$$C = \frac{8^{h+1} - 8^{r+1}}{7p} + (p-1)\frac{8^{d+1}}{p} \tag{25}$$

The two terms of Eq.25 have clear interpretations. The first term is the cost of the parallel octree navigation, which is constant and depends on the number of processors p and the depth of the whole octree h (since r is calculated from p by Eq.1). The second term is the transfer cost of the distribution at the split level d. This second term depends on p and varies over time (because d is a dynamic value varying over time as the camera moves). However, the value of the second term is much smaller than the value of the first term, as shown in the following deduction. Considering that r is usually a small number (*e.g.* $r = 1$ for $p = 4$ or $p = 8$, as determined by Eq.1), we can discard the term 8^{r+1} from Eq.25 and get the following relation between the two terms:

$$\frac{transferCost}{navigationCost} = 7(p-1)8^{d-h} \tag{26}$$

The relation in Eq.26 is usually very small, because d is much smaller than h. For example, in a Quad-core computer ($p = 4$) and the split level being half of the full height of the octree (*i.e.* $d = h/2$), we have:

$$\frac{transferCost}{navigationCost} = \frac{21}{\sqrt{8^h}}, \text{for p = 4 and d} = \frac{h}{2} \tag{27}$$

As a numerical reference, for a large octree with $h = 8$ (corresponding to 19,173,961 nodes), we would have:

$$\frac{transferCost}{navigationCost} \approx 0.0051 \tag{28}$$

that is, the transfer cost is less than 1% of the navigation cost. For practical purposes, we can discard the transfer cost and say that the total cost is roughly given by:

$$C \approx \frac{8^{h+1}}{7p} \tag{29}$$

and for a fixed computer architecture, we can say that C is proportional to 8^h:

$$C \propto 8^h \tag{30}$$

3.10 Parallel Counting Sort

Rendering a scene can be optimized if the rendering is done after sorting the scene objects by some type of resource such as texture, mesh and shader. For example, we could render the scene using sorting by texture by using a code similar to the code below:

```
set texture 0
render all objects with texture 0
set texture 1
render all objects with texture 1
...
```

By reducing the amount of graphics device state changes, the rendering is accelerated [DirectXDocumentation], [ResourceChangeCost].

(a) Sorting Algorithms

The main feature of a sorting algorithm [Cormen01] is the amount of time required to reorder n given numbers into increasing order [2]. 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 a 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, strictly speaking, it is not in-place [3]. 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, something that 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 this

[2]For instance, bubblesort is a slow algorithm, that has a complexity of O(n^2), and mergesort is a fast algorithm that has a complexity of O($n \log n$)

[3]Quicksort may be considered an in-place sorting algorithm, if we consider that it only needs a small stack of size O($\log n$) for keeping track of the recursion

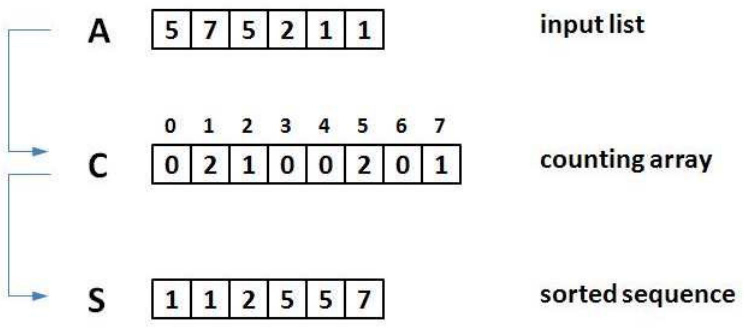


Figure 3.13: Example of counting sort

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 3.13. C[k] is called counting array. In the next section, counting sort is presented as the best algorithm for resource sorting in parallel rendering.

(b) Resource Sorting

Resources are data, properties, components, techniques, and programs used by the 3D objects in order to be rendered properly. 3D objects are elements of the scene such as cars, houses, people. 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 thesis, 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:

$$R[i,j,k,...] = [C[i,j,k], L[i,j,k]] \qquad (31)$$

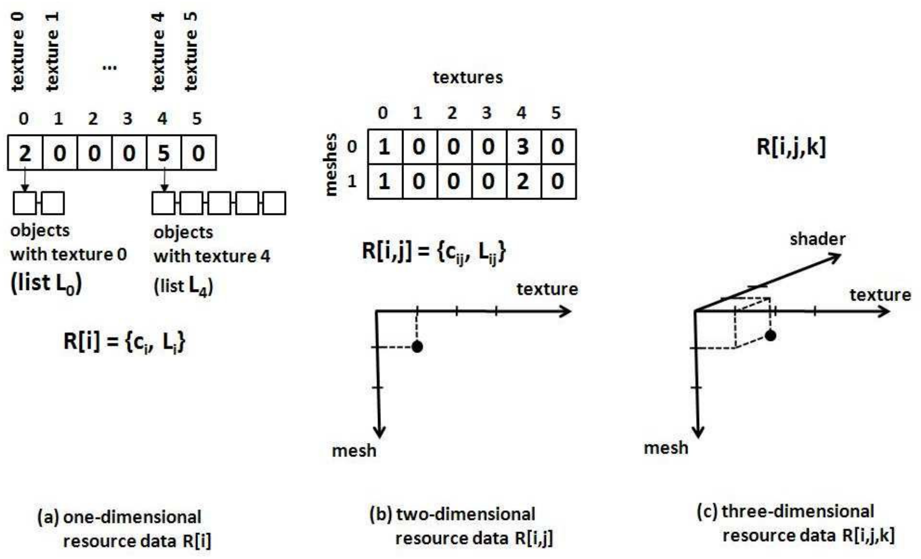

texture 0
texture 1
...
texture 4
texture 5
0 1 2 3 4 5
2 0 0 0 5 0
objects with texture 0 (list L_0)
objects with texture 4 (list L_4)
$R[i] = \{c_i, L_i\}$
(a) one-dimensional resource data R[i]
textures
0 1 2 3 4 5
meshes
0 1 0 0 0 3 0
1 1 0 0 0 2 0
$R[i,j] = \{c_{ij}, L_{ij}\}$
texture
mesh
(b) two-dimensional resource data R[i,j]
R[i,j,k]
shader
texture
mesh
(c) three-dimensional resource data R[i,j,k]

Figure 3.14: Simple cases of the n-dimensional resource data array R.

Figure 3.14 illustrates the simplest cases for R[i,j,k,...]: one, two, and three-dimensional resource data arrays. In Figure 3.15 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). This way, when rendering the first line of objects we only need a single mesh resource change (setting the render device to use mesh 0) reducing the total cost of resource changes for rendering the scene.

In the case of one dimension represented by textures (3.14(a)), we can easily identify R[i] as being an extended version of the counting array C[k] in the counting sort algorithm (Figure 3.13). The main job of the functions StepOneNodeProcessing and StepTwoNodeProcessing on the previous section 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 among the processors anyway.

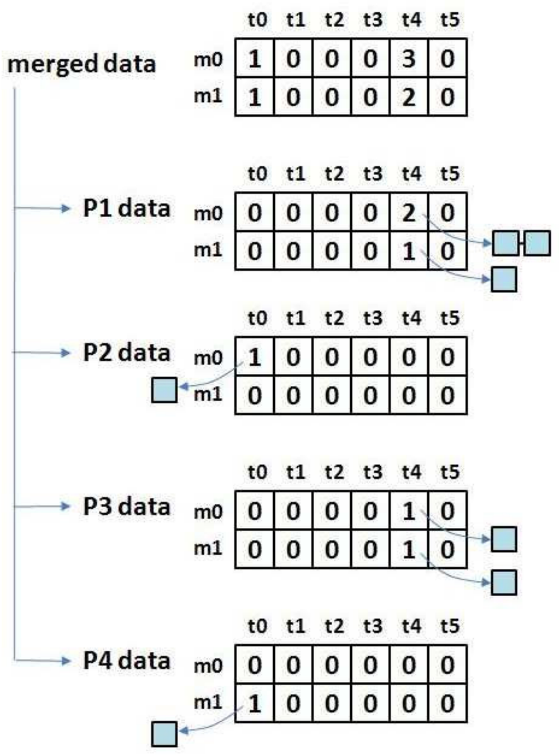


Figure 3.15: The merged resource data array M from $P_1..P_4$

(c) The Sorting Process

The functions StepOneNodeProcessing(Algorithm 1) and StepTwoNodeProcessing(Algorithm 2) builds the resource data array R of each processor P_i, in such a way that the objects are distributed amongst the processors and grouped according to the resources they use. In this thesis, the proposed algorithm merges the 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 3.15 illustrates the entire merging process for the two dimensional case and four processors. We should notice that P_i data is not inside each processor (in fact the sets of P_i 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,...).