Load balancing for parallel raytracer on Virtual Walls

Jiří Žára, *(zara@cs.felk.cvut.cz)**
Aleš Holeček, *(holecek@sgi.felk.cvut.cz)**
Jan Přikryl, *(prikryl@sgi.felk.cvut.cz)**
Jan Buriánek, *(xburiane@sgi.felk.cvut.cz)**
Knut Menzel, *(knut@uni-paderborn.de)*†

Abstract

Dynamic load balancing of parallel ray–tracing algorithm based on spacial subdivision is discused. We attempt to find the load balancing method with the fastes respond on the load extreams in the system. A optimal architecture of the computational system based on three level process hierarchy is proposed. The scene partitioning is based on Virtual Walls.

Keywords

computer graphics, rendering, parallel algorithm, ray–tracing, load balancing,

1 Introduction

Fast visualisation of 3D scene became one of the main research topics in modern computer graphics. Many new algorithms for visualisation and global illumination were introduced in this field during the last several years. Some of them where already implemented in hardware of modern graphic workstations which increases the speed of the rendering and makes these algorithms more suitable for practicle use.

Sometimes to achieve high speed of 3D scene rendering, the quality of the output image has to be partialy sacrificed. The extreme case is complete suppresion of some attributes of the visualised objects. For example it is impossible to simulate optical effects on lenses using hardware Z– buffer.

Another problem arise with the amount of computer memory needed for storing the visualized scene. As a example of this phenomenon, visualisation of NURBS faces or other analytical objects can be used. While the specification of NURBS face is given by several equation coefficients describing the face analytically, for above mentioned hardware Z–buffer, it has to be approximated by large number of polygons. These are only two of many reasons, why our research team decided to investigate the utilization of computationally expensive rendering algorithms for scientific and CAD/CAM visualization.

The price of computer hardware drops down drastically every year. It becomes more reasonable to employ massively parallel system for generating photorealistic images fully reflecting the complexity of light behaviour in a non trivial scene. One of those algorithms is ray–tracing, powerful yet simple approach to realistic image generation. Our purpose in studying ray–tracing is to develop visualization softaware producing images of the highest possible quality and handeling scenes consisting of hunderts of millions of objects in reasonable time.

*CTU, Dept. of Computer Science Karlovo n m. 13, 121 35 Praha 2.
†University of Paderborn, Center for Parallel Computing (PC^2), Germany

This paper is follow up on the paper *Parallelization of ray-tracing algorithm* published in proceedings of Winter School of Computer Graphics and CAD systems that took place in January 1994 in Plzeň, Czech Republic. There we have discussed the possibilities and ways of ray-tracing parallelization. In this contribution we want to introduce load balancing methods for dataflow oriented parallelization based on Virtual Walls programming frame. We expect a basic knowledge of ray-tracing algorithm. [1, 2]

2 Parallelization of ray-tracing algorithm

2.1 Reasons for parallelization

The primary computational burden of ray-tracing algorithm is the calculation of the intesection of ray and surface. Typically for complicated scenes modeling complex lighting effects, this calculation has to be performed millions of times. Rubin and Whitted [3] determined that due to this effect, the performance of ray-tracing algorithm worsens linearly as the number of objects increases. Our research shows [4, 5, 6] that also other factors influence the computational time. The most significant besides the number of objects are:

- resolution of resulting image
- allowed depth of ray recursion
- complexity of the objects
- optical attributes of the objects
- number of light sources

To decrease the computational time **algorithmical speedup utilities** can be used. For example simple hulls can be placed around each object. If given ray fails to intersect the bounding hull for particular object, the object needs no further consideration in the testing for intersection with the ray.

Another approach to the problem of reducing the computation of ray- objects intersection is based on **object space subdivision**. In this case the object space is devided into cells, each holding the informtion about objects intersecting the space partition bounded by the cell. A fast method is then given for tracing the rays through the cells, only performing intersections with those objects which are contained in the intersected cells. Deffinition of the bounding hulls can be used within each cell. The usual data structures used for objects space partitioning are *Octree* [7] and *Binary Space Partitioning tree* (BSP) [8].

However these methods significanly speed up the ray-tracing algorithm, memory demands for construction of the data strucuture (Octree, BSP tree) appears to be a serious problem. Extensive tree structure provides the best speedup, but requires huge amount of computer memory exceeding the one availble on nowadays computers.

2.2 Ways of parallelization

Our approach to solve most of these syndroms is to implement the ray- tracing algorithm into massively parallel system. There are two different methods of parallelization, which can be considered.

- screen subdivision
- spatial subdivision

Using the first approach, during the initiation phase of the computation, each process receives information about the entire scene and part of the screen (set of pixels on screen) which must be illuminated. After the initiation ends, the processes work independently from each other. This means there is no interprocess communication necessary at the time of the computation; the parallelization is non dataflow. [1, 5, 6]. The drawback of this method is that the large scene containing complex objects and their attributes, has to be copied to each processor's memory. This approach accelerate the computation, but does not solve the problem of memory limits.

As we have mentioned above, the effort for decreasing the time of illumination using the ray–tracing algorithm, corresponds very often with decreasing the number of tested intersections between rays and objects in the rendered scene. This means that better parallelization of rendering 3D scenes is based on the *object space subdivision* into 3D regions, hence called *sptial subdivision method.* This method is very similar to the algorithmical approach. The difference is, that each of the space cells created by the partitioning of the object space is assigned to a different processor. We decided to use the Virtual Walls concept proposing a solution for this class of problems.

3 Virtual Walls and ray–tracing

Virtual Walls stands for a general concept for solving three dimensional problems on distributed memory architectures. Within the concept the space is partitioned into volume cells. Neighbouring cells are divided by so called *virtual walls*. The objects from the space are located in the appropriate cell due to their position. The virtual walls are used only for the transport of necessary information from one cell to neighbouring cells. During the transport, the information is not changed or altered. Iterating local computation and neighbouring exchange of information leads to global solution of given problem.

The concept of Virtual Walls is being intensively developed at the Paderborn Centre for Parallel Computing and Paderborn Technical University. The porting of the ray–tracing algorithm onto this programming frame is a result of common project between the Czech Technical University and both Paderborn institutes mentioned above.

The mapping of the ray–tracing algorithm onto Virtual Walls can be described as follows. First of all a preprocessing has to be done; the viewport, through which the *initial rays* are being shot to the scene, has to be projected onto the cluster of cells representing the scene. This is necessary to determine which cells become the generator of initial rays for rendering of the given scene. The ideal case is, when the viewport projection covers one side of the cell cluster (Fig. 1).

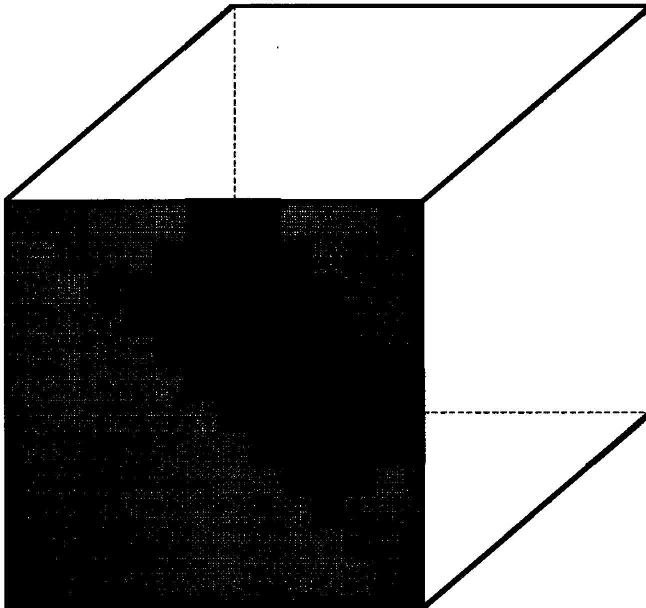

Figure 1: Ideal case of viewport mapping

In the second step each cell with its set of objects (*local scene*) is assigned to a process called **Black Box** (*BBox*). The BBox process is then responsible for tracing the rays in the assigned cell and computing the ray × objects intersection within the local scene of the cell .

The ray–tracing is started in the scene local to each BBox. Due to that the BBoxes start to exchange messages representing the rays which could not be solved within one cell. There is large

amount of exchanged messages through out the computation, which appears to be one of the main drawbacks of this method of parallelization. In order to decrease the communication, we have implemented so called *rays grouping*. The rays with the same destination are packed into groups which are then sent to a receiving process at once.

For another improvemet of the efficiency we have integrated so called *distributed pixel processing* [5, 9]. This allows to calculate only part of the resulting intensity of a pixel during the corresponding ray is traced through the cell. This strategy generates only forward rays until the recursion depth is reached or the ray leaves the scene. Backtracking the rays to its original sending cell is therefore unnecessary. Partly evaluated pixel intensities are collected at the node where they have been calculated and then again using the grouping principles sent to the *output process* (screen). The efficiancy of the distributed raytracer can be more improved by implmenting **load balancing** mechanism.

3.1 Load balancing on Virtual Walls

The distributed system consisting of communicating cells has to be controlled in such a way that each BBox is kept working and does not become idle. Control mechanisms of this kind are called *load balancing strategies* and are well known in the area of parallel computing [9, 10]. The load balancing strategies used within the concept of Virtual Walls are especially design for geometric load balancing and are based on *optimizing the cell shape* by moving the walls, *rotatiting the scene* and/or *optimizing by cell division.*

In case of moving a wall, the shape of its associated cells is changed. Due to that effect, some of the objects previously contained in one cell belong to the neighbouring cell afterwards (Fig. 2). The method of rotating the scene is based on computation of the gravity centres in the particular cell. Afterwards a line given by weight approximation of the local centres of gravity is calculated and then moved to the centre of the scene. Finally the scene is rotated until the line is horizontal or vertical (Fig. 3).

Optimizig by cell division is based on dynamic process generation. In case a BBox is overloaded, the space cell the BBox is responsible for is divided into smaller subcells. For the new subcells new BBox processes are spawn. (Fig. 4).

The detail description of all these load balancing methods can be found in [9, 10]. In the next section we will focus only on the load balancing strategy useful for the implementation of ray–tracing on Virtual Walls.

4 Load balancing for distributed raytracer

To make the distributed raytracer efficient, the algorithmical speedup utility should be used. This requires creating a tree–like structure in local scene of every BBox. It accelerates the calculation of the ray $\times$ objects intersections in all Black Boxes. For our implementation we decided to base the speedup algorithm on Binary Space Partitioning tree.

4.1 Problems

The majority of the problems with the load balancing for the distributed raytracer is connected with the data structure necessary for the speedup algorithmical methods. Building this data structure is very time consuming and to change the data structure is as time consuming as to build a new one. This has to be done every time the geometrical shape of the cell assigned to a BBox is changed.

The load balancing strategies can be divided into two groups:

- *static* - executed only once as a part of preprocessing phase
- *dynamic* - executed during the computation run whenever necessary

For 3D scene rendering, the dynamic load balancing methods seem to be adequate, since the objects of the scene may be moved, new objects may be generated, or the possition of the observer can

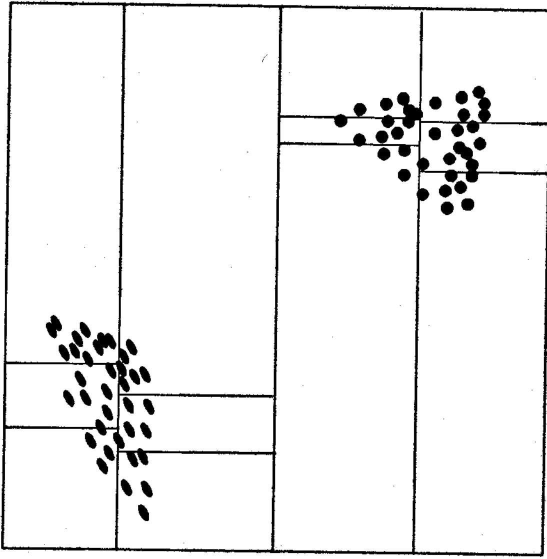

Figure 2: Load balancing by moving the walls

Figure 3: Load balancing by rotating the scene

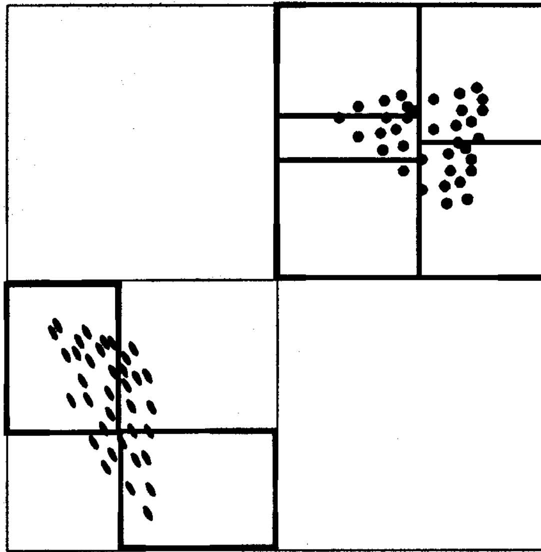

Figure 4: Load balancing by dividing the cell

be changed during the program run. Every attempt to change the load of the BBox by changing its geometry implies rebuiting the BSP tree in all BBoxes attached to the moved wall. For the time of BSP tree rebuilding the processors is fully occupied, and can not process any rays. The tests showed that the time is too long which makes the distributed raytracer inefficient. The load balancing strategy based on moving the walls, rotating the scene as well as the one based on cell division gives in the case of ray–tracing algorithm worse results, than computation without load balancing.

4.2 Solution

To avoid the problems with rebuilding the tree structure, we have proposed different method of load balancing. It is based on presumption, that there are more processors available in the system than there are cells in the partitioned scene. The BBox process then can be divided into two different processes:

1. *Virtual Box process*
2. *Ray–Tracing Core*

The **Virual Box process** (VBox) is designed as a ligh wieghted process. For each space cell created by the scene partitioning there is exactly one VBox. It is designated to generate the initial rays, to receive rays form neighbouring cells and maintain their queue and to calculate a load factor needed for the load balancing (will be discused later).

The **Ray–Tracing Core** is a process performing the acctual ray– tracing algorithm. It is independent form the Virtual Box process untill it is assigned local scene of a cell represented by one of the VBoxes. From that time the RTCore is bound to the particular VBox and requests rays to evaluate from the VBox.

All VBoxes have assigned at least one active RTCore during the entire computation time. These RTCores are called *native*. If $\aleph(VBox)$ is the number of VBoxes and $\aleph(RTCore)$ the number of RTCores which also reflects the number of available processors in the system then:

$$\aleph(VBox) \leq \aleph(RTCore)$$

Satisfaction of this constrain guarantees the proper function of the raytracer. To ensure the possibility of load balancing the constrain has to be slightly modified to:

$$\aleph(VBox) < \aleph(RTCore)$$

This means that there are *spare* RTCores in the system which are not assigned to any VBox. If $\aleph(RTCoreN)$ is a number of native RTCores and $\aleph(RTCoreS)$ number of spare RTCores, then

$$\aleph(RTCore) = \aleph(RTCoreN) + \aleph(RTCoreS)$$

Since the native RTCores are bound with the VBox only the spare RTCores can be used as a computional resource for the load balancing. This method of load balancing is called *process farming*. Larger number of spare RTCores gives better possibility to keep the system balanced. The number of spare RTCores is determined by the number of processors in the system and by the number of space partitions the given scene was divided into.

In frame of this strategy, to change the load of particular VBox equals to binding or releasing a RTCore. To avoid the construction of the BSP tree every time a new RTCore is assigned to the VBox, the native RTCore creates *local scene description* consisting of the BSP tree description. This is another result of the preprocessing phase. If a new RTCore is bound to a overloaded VBox, it has to read the VBox's local scene description. When it is finished with the reading, it starts to request rays for evaluation from the particular VBox.

At the time of reading the local scene description by the spare RTCore, the native RTCore keeps calculating the rays of the VBox. No delay is necessary due to the load balancing. The time of reading the local scene description T_r is recorded to help avoid a trashing effect at the end of the computation.

4.3 Local Load Factor

For the full specification of the load balancing strategy, the load factor definition is necessary. The load of a VBox depends on the number and complexity of the objects enclosed in local scene, the number of pixels that has to be illuminated by the particular VBox due to the viewport projection and the number of messages received from the neighbours.

In our implementation the load factor computed in every VBox is called **Local Load Factor** (LLF) and it is defined as follows:

$$LLF = L/R$$

where: L is the length of the rays queue waiting to be evaluated in the particular VBox and R is the number of rays, which where processed in Δt. Δt is a constant. The LLF reveal how many Δt are needed to process the current queue of the incoming rays.

5 Realization

The parallel raytracer and the load balancing strategy was implemented on the Parallel Virtual Machine (PVM). The PVM is a public domain software package, which permits the network of heterogeneous UNIX computers to be used as a single large parallel machine. The PVM software was developed and is maintained in Oak Ridge National Laboratory in Tennessee. Receantly the PVM is ported into operating systems of many massively parallel computers (SGI, Parsytec, IBM).

We have implemented four different classes of processes:

1. *Screen*
2. *Control*
3. *VBox process*
4. *RTCore*

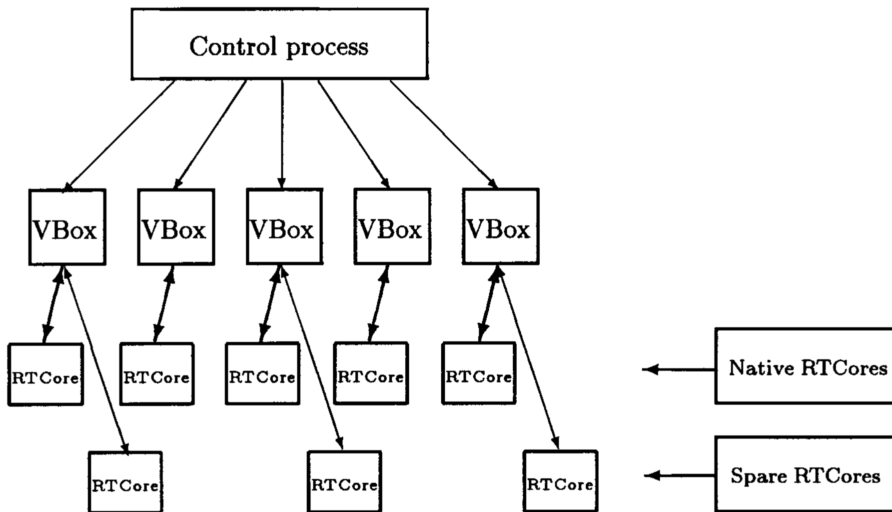

Figure 5: The process structure

From the point of view of load balancing these processes create three level hierarchy shown on (Fig. 5).

The **Screen** is a output process of the system and is independent from all load balancing activities. The **Control** process is the *brain* of the load balancing. It collects information about the current state of the system and control the load balancing.

The VBox processes are placed in the middle between the Control process and the RTCores. Each VBox has to provide the current value of the *LLF* every time the Control process requests this information and also prepare packages of rays for evaluation by the RTCores. The packages are made preferable form the incoming rays. The new rays are generated only if there are no rays in the queue. This helps to keep the entropy of the system low.

The *LLF* is valid only if the VBox is able to fill all packages compltely for all of its RTCores. If there is a RTCore which did not receive a full packge, the VBox sends a message to the Control process. Control process finds a VBox with the highes *LLF* and sends its identification to the spare RTCore. The RTCore then reads the loca scene desciption of the VBox and from the time the requests for rays are directed to the new VBox.

Using this strategy we have experienced a large trashing effect at the end of the computation. In all VBoxes there were no more initial rays to be generated so the VBoxes could not fill the packages. Toward the end of computation more and more RTCores wanted to be reassigned to a different VBox. The computation usualy ended with all RTCores reading the scene of the last working VBox.

To avoid this situation we have implemeted whatch dog based on the variable T_r. If the queue of the incoming rays will be computed in the time twice smaller then T_r, $T_r < 2 \times LLF$, there is no need to assign another RTCore to the VBox. The RTCore would not probably evaluate any rays, but delay the termination of the computation based on the Dijkstra algorithm.

6 Conclusion

We have introuced a load balancing strategy for ray–tracing parallelization based on the spacial subdivison. It preserve the system from rebuilding the datastructures necessary for algorithmical speedup of ray–tracing after every load balancing activity. We have eliminated the trashing effect at the end of the computation which is a side–effect of the load balancing strategy .

At the time of writing the contribution we did not have any measurements done since the load balancing mechanism was not fully implemented yet. The results from messuring the efficiency of

the load balancing strategy will be presented at the conference.

This work is a result of ongoing research on Ray-Tracing Techniques on Virtual Walls (RT-TonVW) which is a common project of the Computer Graphics Research Group at the Czech Technical University in Prague, Paderborn Technical University and Paderborn Centre for Parallel Computing. The future work will be mainly focuse on the scene partitioning during the preprocesing phase and improvemnt of the load balancing strategy. In the current state only the spare RTCores can take a place in the load balancing activities and Control process is necessary in the system. The research is direted toward designing a self balancing architecture exploiting all RTCores for load balancing suitable for the parallel implmentation of the ray-tracing algorithm.

7 References

[1]Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F.: **Computer Graphic Principles and Practice,** Addison-Wesley Pub., New York, 1987.

[2] Žára J. a kolektiv: **Počítačová Grafika – principy a algoritmy**, Grada, Praha, 1992.

[3]Rubin, S. M., Whitted, T.: **A Three-Dimensional Representation for Fast Rendering of Complex Scenes,** Computer Graphics 19(3), July 1980, pp. 127-142.

[4] Přikryl J.: **Dipoma Theses,** CTU, Praha 1994.

[5] Holeček A.: **Diploma Theses,** CTU, Praha 1994.

[6] Žára J., Holeček A., Přikryl J.: **When the parallel ray-tracer starts to be efficient?,** Proceedings of Spring School on Computer Graphics 1994, pp. 108-116.

[7] Sung, K.: **A DDA Octree Traversal Algorithm For Raytracing,** Eurographics 1991, pp. 73-85.

[8] Glassner, A., S.: **Space Subdivision for Fast Ray Tracing,** IEEE Computer Graphics and Applications 4(10), October, 1994, 16-29.

[9] Menzel, K., Shmidt O., Stangenberg F., Hornung Chr., Lange B., Žára J., Holeček A., Přikryl J.: **Distributed Rendering Techniques using Virtual Walls, First European PVM User Group Meeting,** Roma 1994.

[10] Menzel K., Ohlemeyer M.: **Walk-Through Animation in 3D-Scene on Massively Parallel Systems,** The Visual Computer, Iternational Journal of Computer Graphics, Aug. 93, Vol. 9, No. 8, 1993, pp 417-425.