parimod ©

The parallel computer graphics and
interactive solid modeling system

C. Landwehr, F. Thiesing, O. Vornberger, K. Zeppenfeld
Department of Mathematics and Computer Science
University of Osnabrück, Germany
E-mail: klaus@informatik.Uni-Osnabrueck.DE

Abstract. The *parimod* system is a Transputer based graphics system with an additional interactive solid modeling tool for fast rendering of arbitrary 3-dimensional scenes. It consists of an input tool, a calculation and an output unit which are independent of each other so that each of them is replaceable if changes in hard or software come through.
The input unit is a X based solid modeling tool allowing the user to define scenes like an architect on his drawing board. With the massively parallel rendering tool, the user sees the defined scenes in various qualities on-line on the true color output device. The fast output is achieved by the implementation of different parallel strategies of well-known shading and rendering algorithms.
The quick response time between changing and showing the different views of a scene makes *parimod* very popular in the application fields of architectural drawings or the visualization of molecules.

1. Introduction

The research area of computer graphics has grown dramatically through the last decade. It is amazing to see how computers produce photorealistic images that cannot be distinguished from real pictures only by doing arithmetic calculations [Fol90], [Wat89].

For calculating such images in a reasonable amount of time several hard or software approaches have been made during the last years. Specific hardware has been built for solving the rendering equation rapidly, but for practical use these hardware solutions are too expensive. On the software side the developers have produced many ideas to increase the quality of the images, but the basic ideas like Gouraud or Phong shading [Gou71], [Pho75], Ray Tracing [Gla89] and Radiosity [Gor84] haven't changed much.

With our approach we've tried to show another way using common available hardware for solving well-known shading and rendering algorithms in a massively parallel way only in

published in: S. Atkins, A.S. Wagner: *Transputer Research and Applications 6, NATUG-6*, pp. 39-52, IOS Press 1993

software. By using parallel algorithms we are able to calculate photorealistic images as fast as specific hardware solutions but staying in a good price/performance ratio. Therefore, our parallel computer graphics and interactive solid modeling system, abbreviated as *parimod*, is a Transputer based graphics system with an additional interactive solid modeling input tool for fast rendering of arbitrary 3-dimensional scenes. The input tool, the calculation and the output units are modular and communicate via special protocols so that each of them is replaceable. On the other side it is possible to adapt some parts of *parimod* to existing software systems, to speed up their runtime and to make the creation of scenes more straight forward.

With the help of the interactive solid modeling system, which is written in C and runs under X [ORe90a/b], the user can create or modify arbitrary 3-d scenes interactively on the screen only by using the mouse. The scenes can be constructed with basic solids, for example boxes, spheres, cylinders, etc. To define a 3-d scene on a 2-d screen the user sees, like an architect, the ground plan and the front view of the scene. Additionally the complete illumination of the scene (ambient, specular, diffuse, spot light sources), the different view angles, all surface properties (color, reflection, refraction, material), the choice of the rendering quality (wireframe, Flat, Gouraud, Phong shading, Ray Tracing or Radiosity) and all other properties which are necessary for the definition of a 3-d scene can be defined with this tool. After the description is finished, the parameters of the scene are sent to the Transputers. During the calculation of the 3-d image the user can prepare a new scene or modify the current one, for example by changing the view. This is possible because of the device independency of the solid modeling tool. It can be used from different terminals whereas the calculation and the output of the scenes are made on other machines.

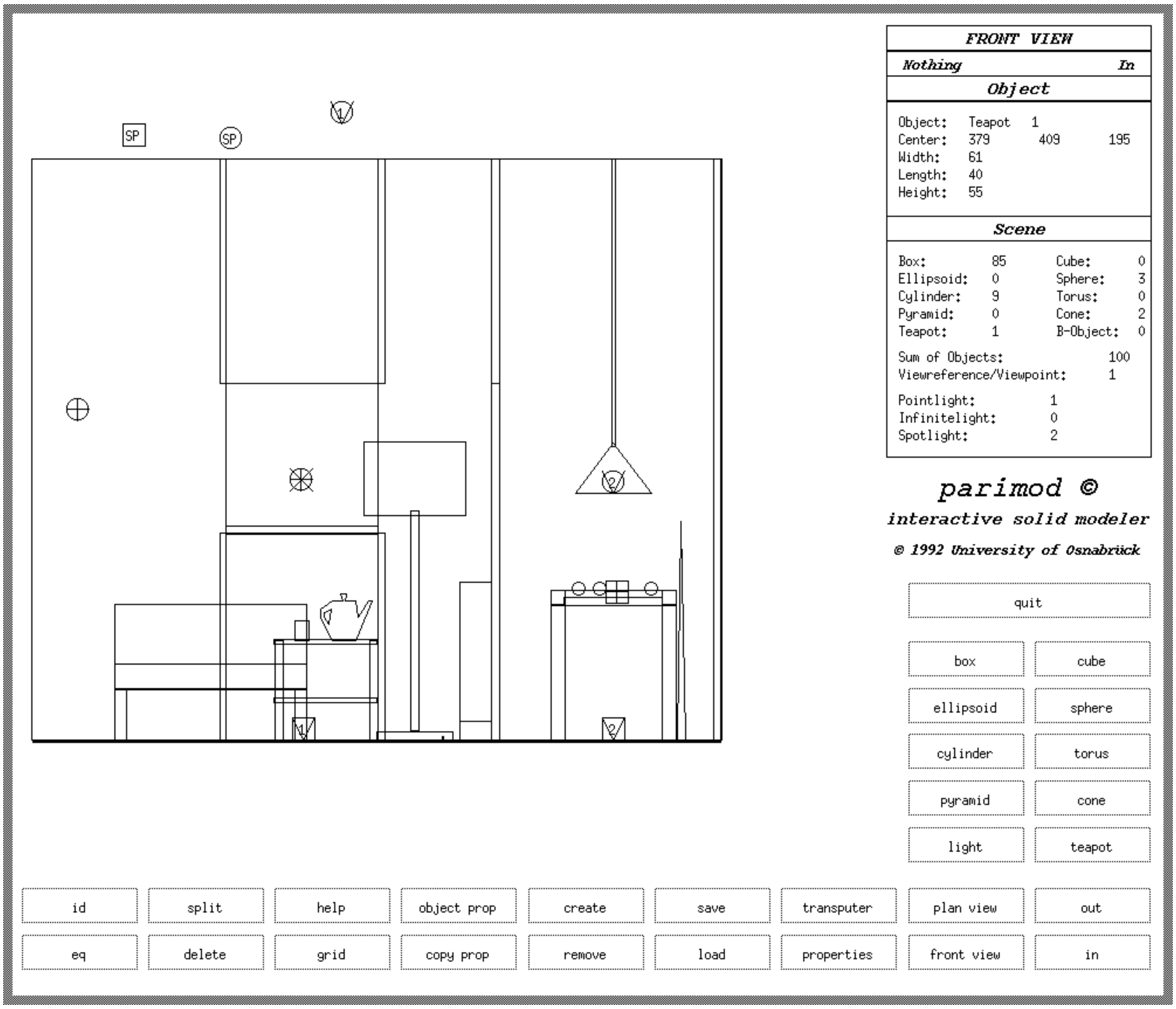
The parallel processing unit, which is the other main component of *parimod*, consists of a T800 multiprocessor system with up to 64 Transputers [Inm89] for which we have developed parallel algorithms, all written in occam2 [Bur88], [Inm88], for the fast calculation of the scenes in the above mentioned modes. The scenes are displayed on a true color, Transputer based Graphical Display System (GDS) [Par90]. To achieve a high performance for calculating 3-d images in software we had to develop good load balancing and routing strategies for the Transputer system. We've made several approaches with different topologies, algorithms and strategies [Gre91], [The89].

The rest of this paper is organized as follows. In Section 2 a detailed description of the facilities of the solid modeler is given. An overview about the parallel rendering algorithms, the used topologies, load balancing and routing strategies is given in Section 3. To see how the different components work together Section 4 shows the complete system, some examples and a performance overview. Finally, concluding remarks in conjunction with our further research plans are presented in the last Section.

2. The interactive solid modeler (*ism*)

For the construction of arbitrary 3-dimensional scenes *parimod* has a front-end, named *ism* (interactive solid modeler). It is a C program which runs under X. This has the advantage of device independency and therefore *ism* can run on any X-terminal independently of the Transputer back-end.

The main problem of constructing 3-dimensional scenes on a 2-dimensional display is to show the user an impression of how his construction will look. This problem can be solved by defining the ground plan and the front view of the scene. Therefore *ism* can be seen as an interactive architectural drawing board in which the user can easily define his drawings only by using the mouse [Boy82], [Män82]. Screendump 2.1 gives an overview about the layout of *ism* as the user sees it on the screen.

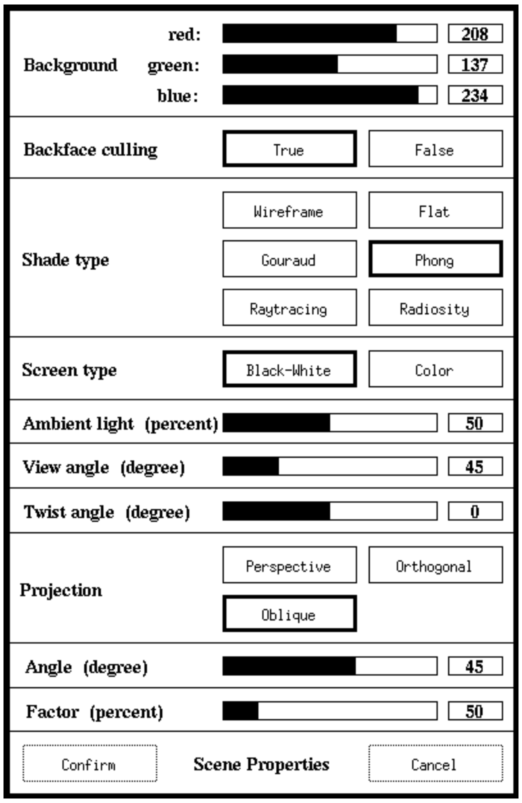


Screendump 2.1: screendump of the application *ism*

On the left side of this screendump the front view of a living room example can be seen. In the upper right corner *ism* shows a complete technical overview about the example. Here the user can see of how many objects his scene consists. Individual object descriptions are presented in one of the windows by clicking the object on the left drawing screen. Informations about the size and the orientation of this object within the scene are shown.

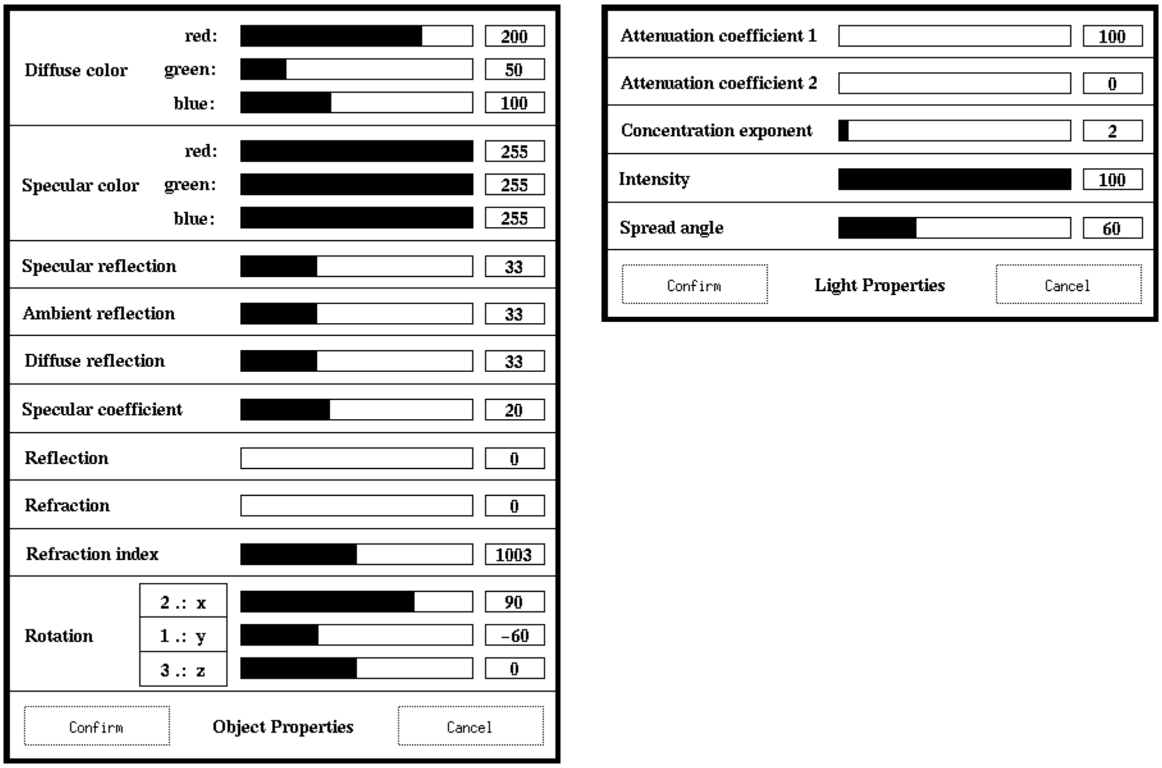
But for the construction of arbitrary scenes the user needs a building box consisting of several basic solids [Req80]. These basic solids are hidden behind the buttons on the bottom and right side of the application. The solid building box consists of boxes, cubes, ellipsoids, spheres, cylinders, toruses, pyramids, cones and the well-known Utah teapot [Cro87]. Every object is created by clicking the object button, moving the mouse on the drawing screen where the object should be placed and after another click by resizing the object on the drawing screen. After this creation the user can modify, move or resize the object whenever he wants. All these operations are implemented with the well-known rubberbanding and handle techniques [Boy82], [Män88], [Mor85]. With these quick object changing techniques the user is encouraged to produce different views or object sizes by playing with the mouse.

To see these results, a constructed scene needs some more definitions like eye and viewpoint and also light sources have to be defined. The eye and viewpoint are created by system default, but can be moved over the screen with the mouse when the user wishes. This facility can be used as a first step of a walk through animation. All objects can only be seen if the user has placed some light sources into the scene. *ism* gives the user three different kinds of light sources: point lights, infinite light and spot lights. The general ambient light can be defined for the whole scene within the scene properties. For the definition and the behavior of light *ism* uses the PHIGS-PLUS standard [How91]. In Screendump 2.1 for example two spot lights are defined. One can see the numbered symbols of source and destination of these lights.



Screendump 2.2: eligible scene properties

After the definition of the geometrical properties the user has to define the object and scene properties. Screendumps 2.2 and 2.3 show that nearly all attributes can be defined with the sliders. We have chosen the basic properties, known from computer graphics, to have a toolkit for the main rendering algorithms. The user can, for example, define diffuse and specular object color, specular, ambient and diffuse object reflectance, object rotations around the three axes, general refraction and reflection, light source intensity with spread angle, etc. For defining the necessary object properties the PHIGS-PLUS standard helps us here too. All these properties are initialized with values which are well defined so that everything works fine even if the user has forgotten to define a certain object property or doesn't want to define all properties by himself.



Screendump 2.3: eligible light and object properties

The possible choice of scene properties are shown in Screendump 2.2. Here the rendering quality of the scene can be defined. For a fast preview wireframe or flat shading are sufficient. Better results concerning the photorealistic outlook of the scene can be achieved by using Gouraud or Phong shading. For producing the final image the Ray Tracing or Radiosity method can be chosen. Some other properties which can be defined, are the selection of the background color, the kind of projection, the view angle, etc.

It is beyond of the scope of this paper to describe all properties which can be chosen for objects and scenes here in detail. We refer interested readers to have a close look to the screendumps and to [Fol90], [Wat89].

However, two other facilities of *ism* are very important and should be mentioned here. Arbitrary objects, different from the basic objects, can be produced by using the *ism* splitting plane. With this splitting plane objects can be divided into different pieces or volumes respectively. The user only has to define the orientation of the splitting plane normal and the object which should be divided. After pushing the split button two new objects are created. The algorithm for splitting objects can be found in [Män88]. The splitting plane is the first step for the implementation of boolean operators like union or intersection of arbitrary objects. These boolean operations are not implemented yet and a goal for further research.

Another important button is the Transputer button which only can be used if *ism* is combined with the parallel computer graphics back end. If the user wants to see the rendered results of his scene constructed with *ism* he only has to push the Transputer button. Then the scene description embedded in a special protocol is sent to the Transputers and rendered in parallel. The parallel rendering and the used hardware configuration are described in the following sections. During the parallel rendering calculations the scene can be modified by the user so that the user can interactively walk through the scene, change object colors or sizes or something else. Every scene description can be saved in a special file description which is an extension to the *neutral file format* (*nff*) by Eric Haines [Hai87]. This file format contains all information about the scene and can be loaded again into *ism* or used by other graphics back ends.

3. Parallel computer graphics

3.1. Sequential viewing pipeline

For the description of the parallel rendering algorithms it is necessary to give a brief overview about the classic viewing pipeline which the objects have to pass through for converting a numerical scene description to a 3-dimensional picture.

For every object type a prototype is stored in its own model coordinate system. Starting from this prototype description, each appearance of this object type in the scene can be constructed with one 4×4-transformation matrix in which operations like scaling, rotation, etc. are hidden. These transformation informations are stored in the *ism* description file. For algorithms like Ray Tracing or Radiosity, where the objects are not transformed through the viewing pipeline, it is sufficient to store the coordinates of the objects in world coordinates. In the *ism* description format this information is also stored.

The second step in the viewing pipeline is the division of the objects into small triangles for which all further operations are performed. After the coordinate transformation of the modeling coordinate system into the normal projection coordinate system a hidden surface removal and a view volume clipping are performed. Only triangles which lie in the view volume and therefore are visible, are sent further through the pipeline. For rendering the triangles, their coordinates are transformed into the world coordinate system and afterwards the lighting is added. For the final step the coordinates are transformed from 3-dimensional world coordinates into 2-dimensional device coordinates. Here the time-consuming shading takes place. For this operation the well-known z-buffer technique is used [Fol90]. After rendering the visible triangles into the z-buffer the image can be shown on the output device.

From this brief description of the viewing pipeline it can be seen that the different steps are independent of each other, the small triangles can be rendered independently into the z-buffer, and that the z-buffer is a large collection of independent screen pixels. If so many things are independent there are a lot of possibilities for parallel rendering strategies.

3.2. Parallel viewing pipeline

To map the different viewing pipeline steps to different Transputers is the first idea that comes to mind, but it is not a good choice because much of the time for the image production is used for the shading of the triangles and a pipeline can only be as fast as its slowest processor. So some alternative ideas must be found.

3.3. Parallel screen

To avoid the pipeline bottleneck it is necessary to distribute the large z-buffer among the network processors. This can easily be done by dividing the view plane and the z-buffer respectively into a large number of small rectangles, the so called subproblems. These subproblems, specified only by their x-y-coordinates and x-y-sizes, are distributed to the processors and every processor has to render the view with regard to its subproblem. The information about the unsolved subproblems runs through the network and whenever a processor becomes idle it can choose one subproblem from the list of unsolved problems. In Figure 3.1 a scheme of this idea is shown. For the fast distribution of the unsolved subproblems it is useful that a Hamiltonian circuit can be embedded into the topology. Therefore we connected our Transputers like a de Bruijn network. These networks have many advantages, especially for Transputers, but are not well known in the Transputer community and therefore a little description of de Bruijn networks is given later in this section.

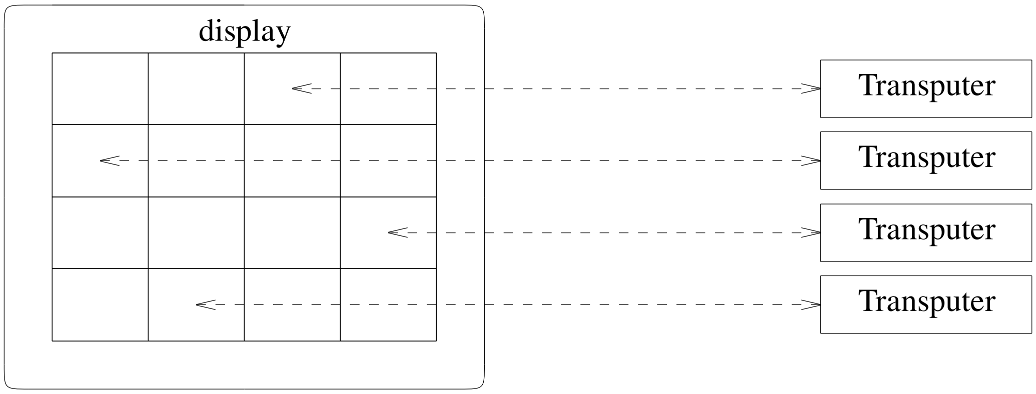


Figure 3.1: distribution of subproblems between Transputers

This parallelisation of the screen into small z-buffers guarantees a very good load balancing in the network. The smaller the subproblems are, the better the load in the network is balanced. Contrarily the smaller the subproblems are, the more work is duplicated because for each subproblem a lot of preprocessing (object generation, hidden surfaces, clipping, etc.) has to be done since the processors have no information about which object lies in their little part of the viewplane. They have to generate the whole scene for every subproblem like in the sequential case, but after clipping only a small part of the scene has to be rendered. This causes a sequential overhead in the network. For making this parallelisation as fast as possible a good ratio between subproblem size and network load has to be found. In Section 4 some experimental results of this parallelisation strategy are shown for several pictures.

3.4. Parallel objects

This strategy should avoid the duplication of work for the processors. Therefore every processor is again responsible for a fixed part of the z-buffer, however, the construction of the scene is done globally by a special processor, called the master processor. This master processor splits every object of the scene into its small triangles and afterwards these small rendering atoms are distributed to the slave processors which are responsible for rendering the atoms into their z-buffer. Figure 3.2 gives an overview of this idea. It can be seen from this figure that it is possible that one triangle is sent to more than one slave processor. If the master processor detects such triangles they are sent to all of the processors which are responsible for a piece of the z-buffer one triangle belongs to. With this idea nothing of the scene construction and rendering work is done twice, but other problems arise. How to distribute the z-buffer among the processors so that the workload is even or how to code the triangle messages to avoid communication overhead?

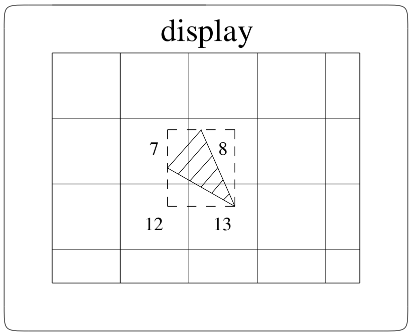


Figure 3.2: one triangle and its rendering Transputers

The termination of the algorithm and the display of the rendered image are additional problems. Whereas in the above parallelisation all processors can send their finished subpictures to the screen at once and the user can see how the final image is built together by the subpictures like a puzzle, in this parallelisation every processor has to wait until the last triangle is distributed. This causes the effect that only the final image can be sent to the output device and the user gets the impression that the calculation of this image takes more time than in the previous parallelisation.

To achieve a good load balancing every slave processor is responsible for different parts of the z-buffer, because it depends on the scene whether or not a processor has to render only some triangles or many triangles. The probability of a well balanced workload increases if one processor gets more than one portion of the z-buffer. In Figure 3.3 an example of a possible assignment of z-buffer pieces to one processor is given. An assignment heuristic which leads to good results is to choose the processors and their related z-buffers at random.

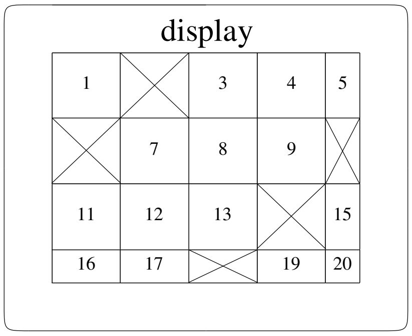


Figure 3.3: subpictures for rendering Transputer 2 (marked)

Another problem of this parallelisation is the number of messages which have to be routed through the network. The parallelisation scheme is shown in Figure 3.4 and it can be seen that at the beginning all atoms have to be routed downwards to the slaves and at the end of the algorithm back to the host. It is clear that this bottleneck must drop down the speedup, especially if those slaves located near the host have to spend more time for routing than for rendering. But all these problems can be solved by selecting a topology suitable for these strategy and we achieved good results with this parallelisation idea concerning the rendering time and the speedup.

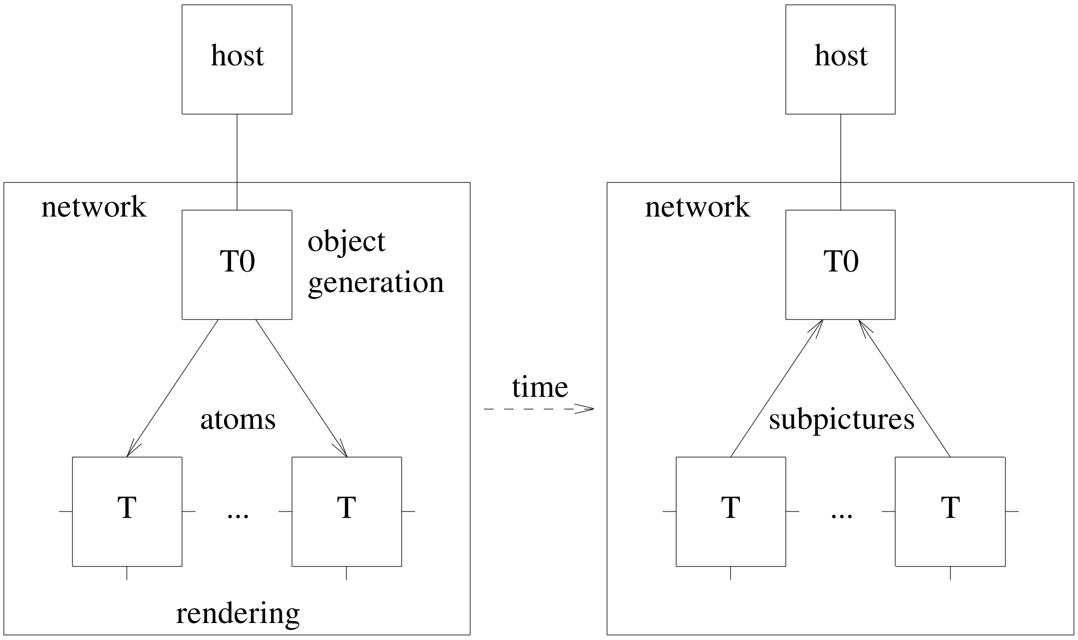


Figure 3.4: the process of rendering (parallel objects)

3.5. De Bruijn networks

De Bruijn networks are related from r-dimensional de Bruijn graphs which consists of 2^r nodes and 2^{r+1} directed edges. Each node corresponds to a r-bit string and there exists a directed edge from each node $u_1\ u_2\ ...\ u_{\log n}$ to nodes $u_2\ ...\ u_{\log n}\ 0$ and $u_2\ ...\ u_{\log n}\ 1$. Every node has outdegree 2 and indegree 2 and if we take no consideration about the direction of the edges every node has a degree of 4 like the number of Transputer links. Obviously the two bit strings 0...0 and 1...1 are connected by themselves and the two bit strings 101...101 and 010...010 in which the bits alternate between 1 and 0 are double connected (see Figure 3.5(a)). But these link ports are very useful for the connection with the Graphical Display System (GDS) where a Transputer is embedded to display the rendered images. So this Transputer can be connected with the 0...0, 1...1, 101...101 and 010...010 Transputers and one additional link of Transputer 0...0 can be connected with the host (see Figure 3.5(b)).

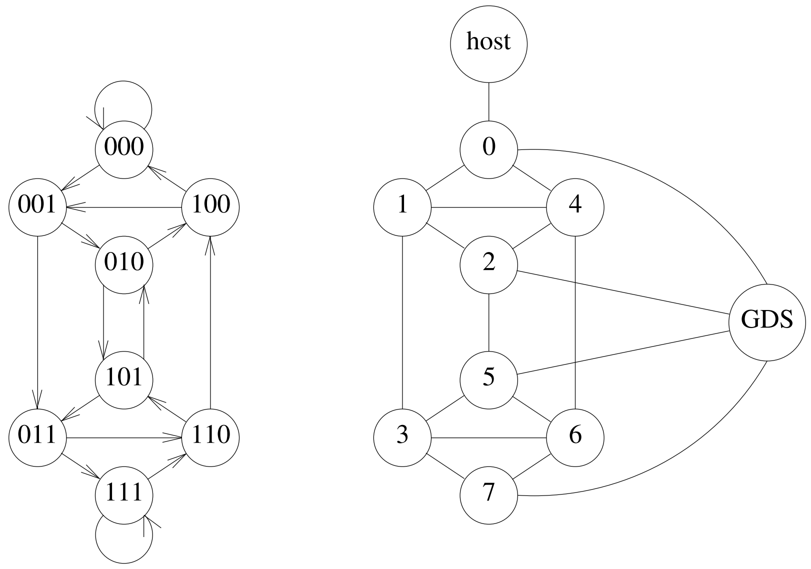


Figure 3.5: (a) de Bruijn(3) (b) Transputer topology with GDS

For every degree the de Bruijn networks can be used as a complete rendering network in which the graphical output system is integrated. The de Bruijn networks have some other likely advantages with which some of the above mentioned routing problems are solved very easily. In every network a Hamiltonian circuit can be embedded and the diameter of the network is only $\log n$.

3.6. Ray Tracing

The implementation of the parallel Ray Tracing algorithm was quite easy because we used the well-known strategy of distributing the rays and the pixels of the view plane respectively. This can be found in [Gre91]. Every processor of the de Bruijn network is responsible for one pixel line of the view plane. After calculating the results, this line is sent on the shortest way to the GDS Transputer. Although this parallel calculation is faster than with conventional Ray Tracers and very realistic images can be produced with a linear speedup, the time for generating one picture takes several minutes or hours, especially if many teapots are within the scene. So this kind of rendering quality should only be used to produce final images.

3.7. Radiosity

Our parallelisation of the Radiosity method is done by using the progressive refinement Radiosity approach [Coh88]. As long as no changes in the geometrical layout of the scene are made, several processors produce the delta form factors whereas the other Transputers are responsible for the fast image rendering. By parallelising the progressive refinement approach we achieve the first image after several seconds. The quality of the image increases according to the numbers of iterations of the delta form factors. But with the speedup of this parallel version shadows and light distribution can be seen after one minute. We also use the progressive refinement approach for the animation of the scenes to add a walk through facility to *parimod*, but this work is still in hand.

4. Hardware and Examples

After introducing the two main components in this section a complete overview of the *parimod* system is given, showing how the components work together.

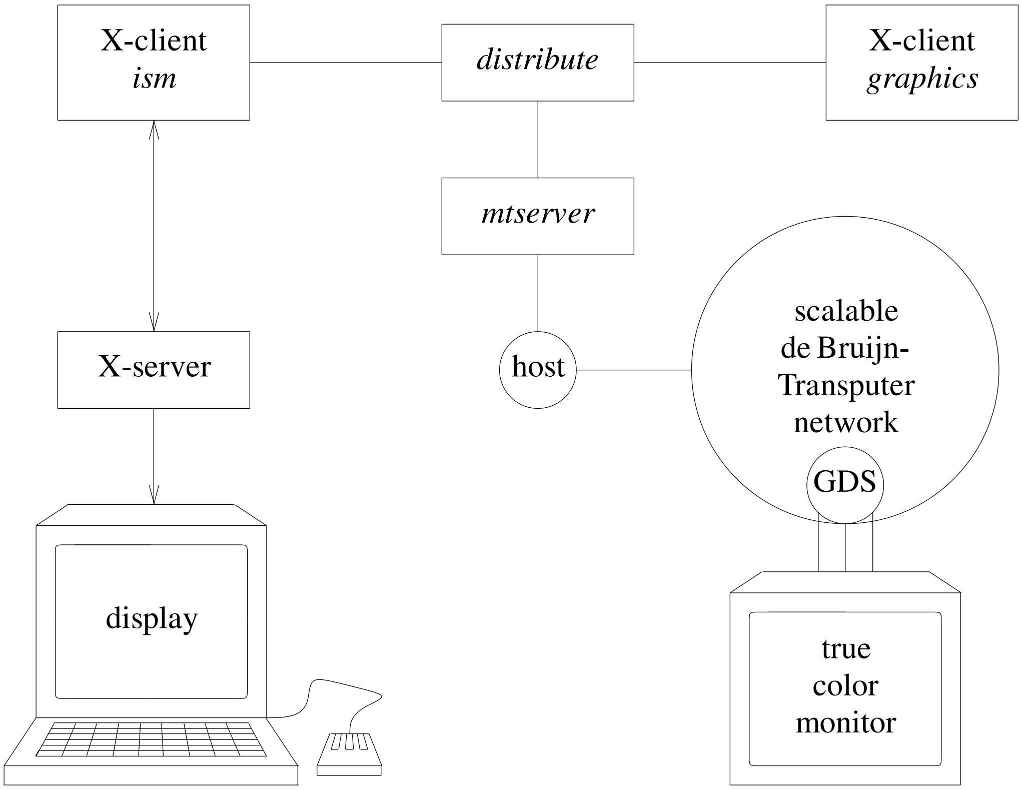
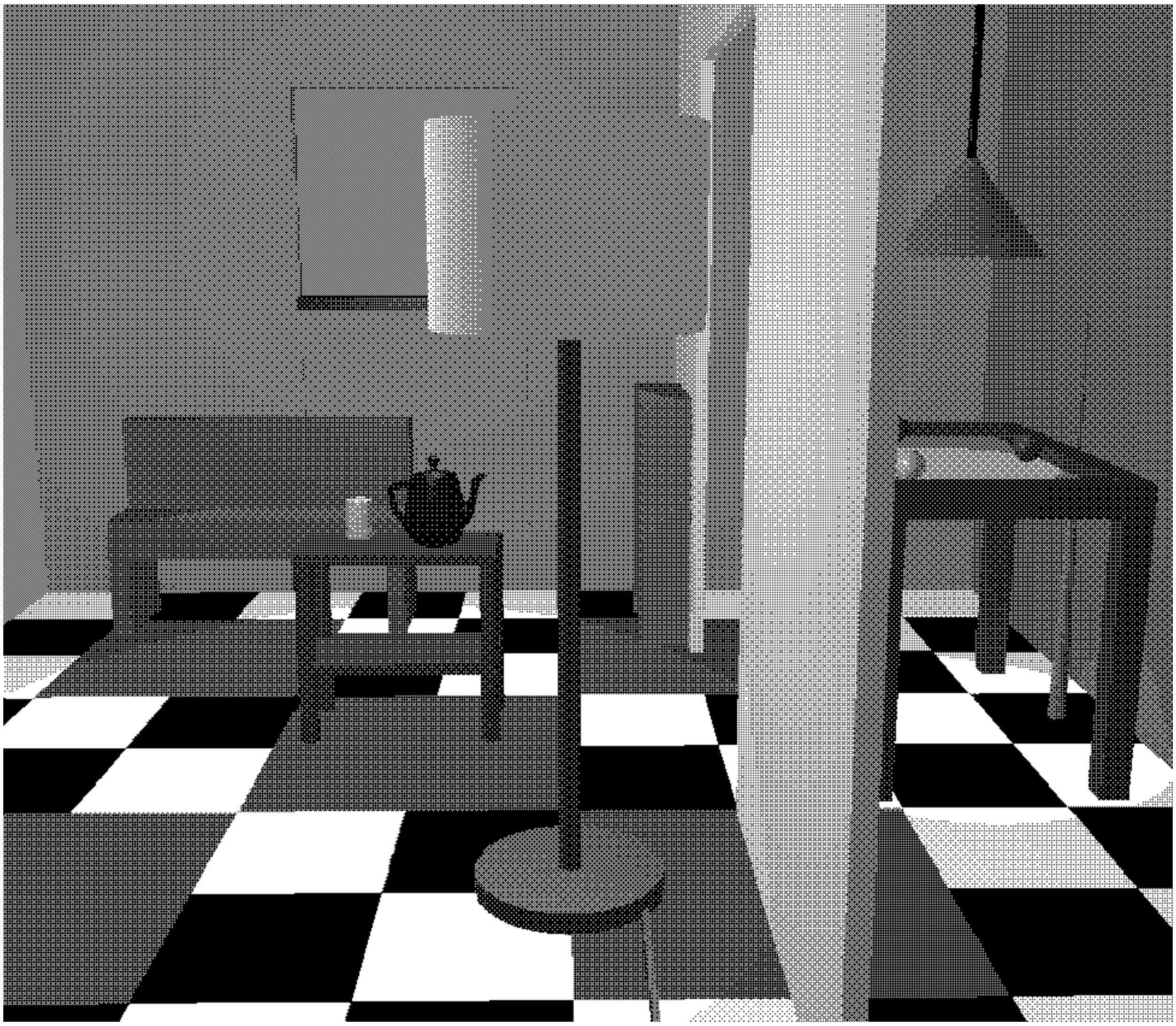


Figure 4.1: the *parimod* system with GDS

As shown in Figure 4.1 *parimod* combines *ism* with another X-application, named *graphics* and the parallel computer graphics Transputer system and vice versa. The scene description which is produced with *ism* is sent via the distributor process and the Transputer operating system (mtserver) to the rendering network [Par89]. This rendering network is a de Bruijn network and therefore scalable. The output is generated with the above described parallel rendering algorithms and can be seen afterwards on the Transputer based Graphical Display System (GDS) [Par90] in true color, or with the help of the X-application *graphics* on a workstation with a 256 color display. Therefore the image has to be routed back through the mtserver and the distributor to *graphics* where it is converted from true color to 256 colors, if necessary. With *graphics* a black/white dithered output of the image on monochrome screens is also possible (see Screendump 4.1).



Screendump 4.1: black/white dithered output

The configuration and installation of *parimod* into an existing workstation network (e.g. Sun4 workstations) can be done only by initializing several UNIX environment variables which indicate where to map the processes and where the paths of the dates and programs are found. This guarantees a high degree of portability and flexibility.

Screendump 4.1 shows the living room scene constructed with *ism* in Screendump 2.1. It is a black/white dithered image and can therefore only give a poor impression of the quality of the related true color picture. Over one hundred of such scenes have been produced with *ism* and then rendered in parallel to get a feeling of *parimod*'s power and also to judge the quality of the different parallelisation techniques. It is clear that for every rendering or parallelisation technique good or bad scenes can be constructed in the sense that for the same picture one strategy out-performs the other strategy and vice versa.

We have rendered the main part of the pictures on a 32 processor network and the calculation time ranges from 1 second (for wireframe images) to 2 minutes (Phong shaded teapots). The time for Ray Tracing pictures ranges from 5 minutes to several hours depending on the complexity of the scene. Whereas in the Ray Tracing version we always achieved a linear speedup, in the Gouraud or Phong shading versions the speedup ranges from 5 to 25 if 32 processors are used. Not all speedups have satisfied our expectations but with the above described parallelisation techniques for some pictures large communication and calculation overhead was detected and an improvement of this overhead should be a topic for further research.

But more important than the speedups are the short rendering times in general and the subjective impression of the user to see the image built up piece for piece on the screen which makes the waiting time as little as possible.

5. Conclusion and further research

With the *parimod* system we have built a portable and flexible parallel computer graphics system. It shows that Transputers can be applied into existing workstation networks to improve the performance of those networks.

Especially in the field of computer graphics where only special hardware solutions achieve a high performance, we have shown that comparable performance is achievable with clever combined conventional hardware and well designed parallel software.

As mentioned earlier to integrate the Radiosity walk through animation into *parimod* is state of the art research. But additionally there are a lot of aspects to consider in the improvement of the system, e.g. boolean set operations for constructing objects, texture mapping and a lot of questions about the parallelisation techniques are open too.

However, *parimod* is a good example to show just how flexible and powerful Transputers can be.

© 1993 by University of Osnabrück, Germany

References

[Boy82] J.W. BOYSE, J.E. GILCHRIST; *GMSolid: Interactive Modeling for Design and Analysis of Solids*; IEEE, Computer Graphics & Applications 2(2), S. 27 - 40, 1982

[Bur88] A. BURNS: *Programming in occam 2*; Addison-Wesley, 1988

[Coh88] M.F. COHEN, S.E. CHEN, J.R. WALLACE, D.P. GREENBERG: *A Progressive Refinement Approach to Fast Radiosity Image Generation*; Computer Graphics, Vol. 22, No. 4, Aug. 1988, pp. 75-84

[Cro87] F. CROW: *The Origins of the Teapot*; IEEE Computer Graphics & Applications Vol. 7, No. 1 (Januar 1987), S. 8-19

[Fol90] J. FOLEY, A. VAN DAM, S. FEINER, J. HUGHES: *Computer Graphics; Principles and Practice; Second Edition*; Addison-Wesley, 1990

[Gla89] A.S. GLASSNER: *An Introduction to Ray Tracing*; Academic Press, 1989

[Gor84] C.M. GORAL, K.E. TORRANCE, D.P. GREENBERG, B. BATTAILE: *Modeling the Interaction of Light Between Diffuse Surfaces*; Computer Graphics, Vol. 18 No. 3 July 1984, pp. 213-222

[Gre91] S. GREEN: *Parallel Processing for Computer Graphics*; The MIT Press, Cambridge MA, 1991

[Gou71] H. GOURAUD: *Continuous Shading of Curved Surfaces*; IEEE Transactions on Computers Vol. c-20, No. 6 (Juni 1971), S. 623-629

[Hai87] E. HAINES: *A Proposal for Standard Graphics Environments*; IEEE Computer Graphics and Applications 7 (11) Nov. 87, pp. 3-5

[How91] T.L.J. HOWARD, W.T. HEWITT, R.J. HUBBOLD, K.M. WYRWAS: *A Practical Introduction to PHIGS and PHIGS PLUS*; Addison-Wesley, 1991

[Inm88] INMOS LIMITED: *occam 2 Reference Manual*; Prentice Hall, 1988

[Inm89] INMOS LIMITED: *Transputer Technical Notes*; Prentice Hall, 1989

[Män82] M. MÄNTYLÄ, R. SULONEN; *GWB: A Solid Modeler with Euler Operators*; IEEE, Computer Graphics & Applications 2(7): S. 17 - 31, 1982

[Män88] M. MÄNTYLÄ; *An Introduction to Solid Modeling*; Computer Science Press, 1988

[Mor85] M.E. MORTENSEN; *Geometric Modeling*; John Wiley & Sons, 1985

[ORe90a] T. O'REILLY: *X Window System, Volume 1: Xlib Programming Manual; Second Edition*; O'Reilly & Associates, Inc., 1990

[ORe90b] T. O'REILLY: *X Window System, Volume 2: Xlib Reference Manual; Second Edition*; O'Reilly & Associates, Inc., 1990

[Par89] PARSYTEC GMBH: *Multi Tool 5.0 Technical Documentation — Transputer Programming Environment*; 1989

[Par90] PARSYTEC GMBH: *GDS-2 — Graphic Display Subsystem*; 1990

[Pho75] B.T. PHONG: *Illumination for Computer Generated Pictures*; Communications of the ACM Vol. 18, No. 6 (Juni 1975), S. 311-317

[Req80] A.A.G. REQUICHA; *Representations for Rigid Solids: Theory, Methods and Systems*; ACM, Computing Surveys 12(4): S. 437 - 464, 1980

[The89] T. THEOHARIS: *Algorithms for Parallel Polygon Rendering*; Springer, 1989

[Wat89] A. WATT: *Fundamentals of Three-Dimensional Computer Graphics*; Addison-Wesley, 1989

---

talk presented at: Sixth Conference of the North American Transputer Users Group NATUG-6, May 10-11, 1993, Vancouver, Canada