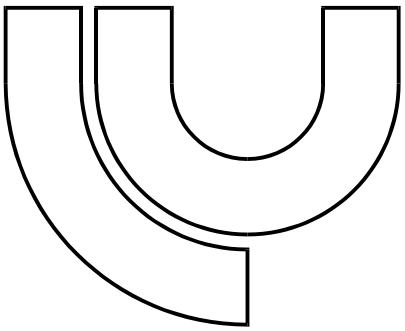

Limburgs Universitair Centrum

Faculteit Wetenschappen

# A STUDY OF GLOBAL ILLUMINATION MODELS AND THEIR USE IN A TRANSPUTER BASED PARALLEL SYSTEM

# EEN STUDIE VAN GLOBALE ILLUMINATIEMODELLEN, TOEGEPAST IN EEN TRANPUTER-GEBASEERD PARALLEL SYSTEEM

Proefschrift voorgelegd tot het behalen van de graad van

**Doctor in de Wetenschappen, richting Informatica**

aan het Limburgs Universitair Centrum te verdedigen door

**WIM LAMOTTE**

Promotor : Prof. Dr. E. Flerackers

Copromotor : Prof. Dr. T. D'Hondt

**1994**

# Dankwoord

Een werk als dit is nooit de realisatie van één persoon alleen. Het komt tot stand door voortdurend contact met mensen uit de onmiddellijke omgeving, zowel direct, door de discussies over de materie, door hulp bij de effectieve realisatie, door het samen evalueren van de voorgestelde algoritmes in een groter kader, enzovoort, als indirect door steun op persoonlijk vlak. Daarom zou ik graag enkele mensen willen danken, die elk op hun manier hun steentje hebben bijgedragen.

Vooreerst dank ik mijn promotor, Prof. Dr. Eddy Flerackers, die me de kans heeft gegeven om bij het wegvallen van de groep Theoretische Informatica, onderzoek te verrichten in het kader van het Laboratorium voor Toegepaste Informatica binnen het fascinerende domein van parallellisme en computer graphics. Mijn oprechte dank voor deze vijf jaren van goede samenwerking. Ook dank ik Prof. Dr. Frank Van Reeth, voor zijn begeleiding, hulp en verfrissende ideeën, zowel over de tekst als over de implementatie; Koen Elens, voor de meeste ray tracing implementaties, voor de lange discussies en het plezier dat we daaraan beleefd hebben; Luc Vandeurzen, voor zijn werk op het gebied van de radiosity; Erwin Vanhees en Philip Schaeken, voor het beroep dat ik mocht doen op hun programmeerervaring; Liesbeth Beckers en Paul Akkermans, voor hun hulp bij de testscènes en de illustraties; Roger Claes, voor al het administratieve werk; alle andere collega's en de diverse thesisstudenten, voor de stimulerende werkomgeving en de vriendschappelijke sfeer op het labo. Chantal bedank ik voor het maken van de vele kleurenfoto's en -dia's. Ook zou ik de universitaire instanties willen danken voor het bieden van de faciliteiten om het hier beschreven onderzoek te kunnen uitvoeren.

Op het niet-professionele vlak bedank ik mijn ouders, omdat ze me de kans hebben gegeven om te studeren en voor hun jarenlange steun. En *last but not least* bedank ik mijn verloofde Sylvia. Zonder haar geduld en steun was dit werk er nooit geweest.

# Samenvatting

# Preface

We live in a period in which the image culture is growing day by day. The visual information imposed upon us is sometimes overwhelming. Special effects appear in almost every movie, advertising becomes visually more attractive, video editing is being used very widely and children learn much of what they know from television (which is not always what their parents would like).

Many amongst us are not aware of the fact that computers are being utilised for the generation of complex and sometimes very realistic images and animations. This is being done in the field of two-dimensional (2D) graphics and animation (like "classical" Disney-style animations) as well as in the more advanced three-dimensional (3D) imaging. The main differences between these two streams is the flatness of 2D images and animation, as opposed to the volumetric appearance of 3D graphics, which gives it a more realistic look. Just think about the well-known effects in recent motion pictures like "*Terminator 2*" and "*Jurassic Park*", where the computer animators create a world of impossibilities with an overwhelming realism.

The image culture is not only growing in the entertainment, but also in business applications. In Computer Aided Design (CAD) computer graphics are used to design mechanical devices like machine parts, car bodies or aeroplanes. Industrial designers and architects can offer their clients a realistic three-dimensional view or walk-through of projects that only exist in the mind, on paper and in the memory of computers. By adding more and more realism, customers obtain a better insight in their projects, faster than with any other prototyping method. (For applications of our research in the areas of architecture and animation, see [Van Reeth 91], [Lamotte 93a], [Lamotte 93b] and [Van Reeth 93]; for a view of our transputer-based computer animation system, see Colour Plate 1).

For the generation of this realism, powerful software is written to generate images of high quality, starting from nothing but some sketches of the things the director wants to show in his film. The main goal of these software programs is to create a scene of objects in three dimensions (width, height and depth), illuminating it with a set of lights and then viewing this scene

through an imaginary camera in order to generate an image (or a sequence of images for animation).

One of the key factors within this context is the way in which the lights should illuminate the scene and how this information is then put on the film of our virtual camera. As we will show in Chapter 1, there are many ways of doing this, but we chose to investigate two main illumination models: ray tracing and radiosity, two very computationally expensive algorithms. The reason is that we were interested in the usefulness of a parallel computer to speed up image generation: if one wants to speed up something, it must be worth the while.

# Table of Contents

# Introduction

When one starts research in a certain field, one has some goals and directions in mind: it is always the intention of a researcher to add some personal "discoveries" to the world's knowledge in a field that interests him. At the origin of this thesis lies our interest in two important streams in computer science: computer-generated imagery on the one hand and the introduction of parallelism in computation-intensive algorithms on the other hand. In this short introduction we state some goals that we had in mind and give an outline of the rest of the text.

## Goals

The general goal was to generate realistic pictures on the transputer-based parallel machine that we had at our disposal (see Colour Plate 1), using two popular but computation-intensive global illumination models: ray tracing and radiosity.

This includes several sub-goals (which will be explained in detail in the rest of this text):

- use database coherence in ray tracing in order to render larger databases (Chapters 4 and 5),
- investigate techniques for reducing the number of ray-object intersections, combined with the large databases (Chapters 4 and 5),
- find techniques for rendering patches in our parallel ray tracer (Chapter 6):
  - fine-tune Toth's algorithm for ray tracing patches,
  - "fill the gaps" left open in the conclusions of [Lischinski 90]: the authors left "smarter control of the depth of surface trees" and "a more sophisticated strategy for releasing cached surface trees" to further research,
- introduce parallelism in progressive radiosity using a novel approach (Chapter 7),
- and, maybe most important, actually implement the algorithms on a transputer-based parallel machine (Chapters 4 to 7).

We will show in the remains of this thesis that these goals were achieved in a uniform and elegant way.

## Outline

The first two chapters give an introduction to the two main streams underlying this work: three-dimensional computer graphics is the subject of Chapter 1, while Chapter 2 describes briefly some aspects of parallelism using transputers.

Chapter 3 describes in general the basic ideas behind parallel ray tracing. The processes and network layout are briefly introduced.

Then Chapters 4 to 7 each outline one of the main blocks in our research. These chapters are all structured the same way: we state some goal(s), give background on the subject, work towards our solution or contribution, give results of the approach described, draw our conclusions and point at possible directions for future research in this sub-domain of our work.

In Chapter 4, we look at the first implementation of the parallel ray tracer, based on a regular voxel grid. This version copes with large databases in a dynamic way.

An alternative approach is elucidated in Chapter 5: we now use a data structure based on a tree of bounding volumes. We outline the main pros and cons of both methods at the end of this chapter.

Chapter 6 outlines the tree caching algorithm for ray tracing patches within our parallel system.

Chapter 7 switches to radiosity. In this chapter we show how the principles underlying the parallel ray tracer can be adopted to calculate form factors for radiosity in much the same way.

In the final chapter, some conclusions are summarised. We evaluate the achievements in relation to the goals. We also give some possible directions of further research that could be done in this field.

After the references, a colour section is appended, showing a colour version of most black-and-white picture in the text, together with some additional pictures to which is referred in the text.

# 1. 3D Computer Graphics

One of the main purposes of 3D computer graphics is the generation of images, starting from some representation of real-life or imaginary "things" that we will call "objects". As an example, let's assume that a car manufacturer wants to see the looks of a new prototype. One can build a model in wood, plastic or polyester. Or, one can port the design to a CAD-program on a computer and generate a realistic image of the car. For such a simulation, not only a good representation is necessary for the surface of the coach-work to look realistic, but also the influence of light on this surface has to be imitated as good as possible.

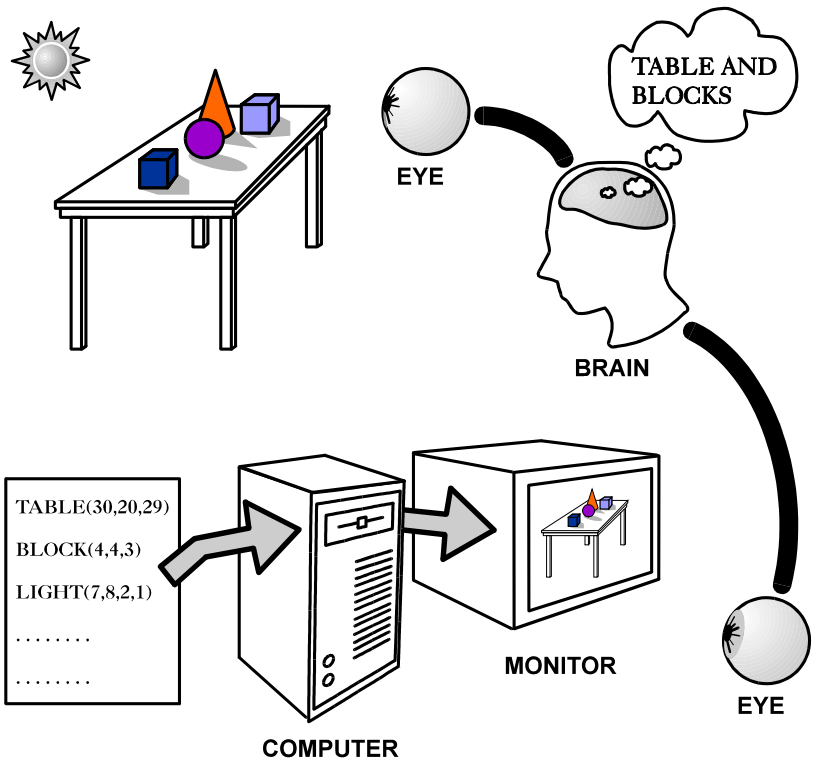


*Figure 1-**Error! Unknown switch argument.**: Principle of visualisation*

In this quest for realism, the ultimate goal is to give the eyes of the one who sees the computer-generated images the impression that they are looking at a real scene. This principle is depicted in Figure 1-1, after [Cohen 93]. The top half of this drawing shows how we are able to see with our eyes: light is illuminating the things that we look at and bounces off into our

eyes. These stimuli are then interpreted by the brain that forms an abstraction of the scene, in this example a table with some blocks. The bottom half of the image depicts in what way a computer system can give almost the same impression to the brain: some kind of model exists in the memory of the computer, which generates an image from this description and shows it on a monitor. Our eyes see this image and the brain tries to interpret it as if it was a real scene we are looking at. If the image is realistic enough, our brain is "fooled" in such a way that it builds the same abstraction of a table with some blocks on top of it.

This means that some prerequisites have to be met: the computer system has to be able to (1) represent or model the three-dimensional scene in its memory, (2) position these objects with respect to the viewer's position and (3) realistically mimic the influence of lights on these 3D objects in the scene. We will have a closer look at each of these steps.

## 1.1. 3D Object Representation

There are many ways of representing an object in three dimensions. The diagram in Figure 1-2 (after [Wyvill 90]) shows a taxonomy of different modelling techniques.

In solid modelling, an object is described by the volume it occupies in space, mostly based on standard geometric objects like boxes, spheres, cylinders etc. Constructive solid geometry allows to apply Boolean operations (union, subtraction, difference) on such solid primitives to create more complex shapes.

Surface models do not describe an object by its volume, but by (an approximation of) its surface. The most common are polygon models and surface patches. Polygon models approximate the surface using flat polygons. Their main advantage is the speed at which a polygon model can be processed and rendered, but they have the disadvantage of jagged contour lines when used to approximate curved surfaces and a lack of control over the curvature of the surface. These disadvantages can be overcome by using surface patches. These are a parametric definition of a curved surface. A patch is represented by a set of control points (16 points in the case of a Bézier patch) arranged in a grid. By moving a control point, the surface's shape will closely follow the displacement. This is a very powerful means for modelling complex curved surfaces using only a very small amount of actual data (e.g. the famous Utah teapot (defined in [Crow 87] and shown in Colour Plate 2) can be represented by

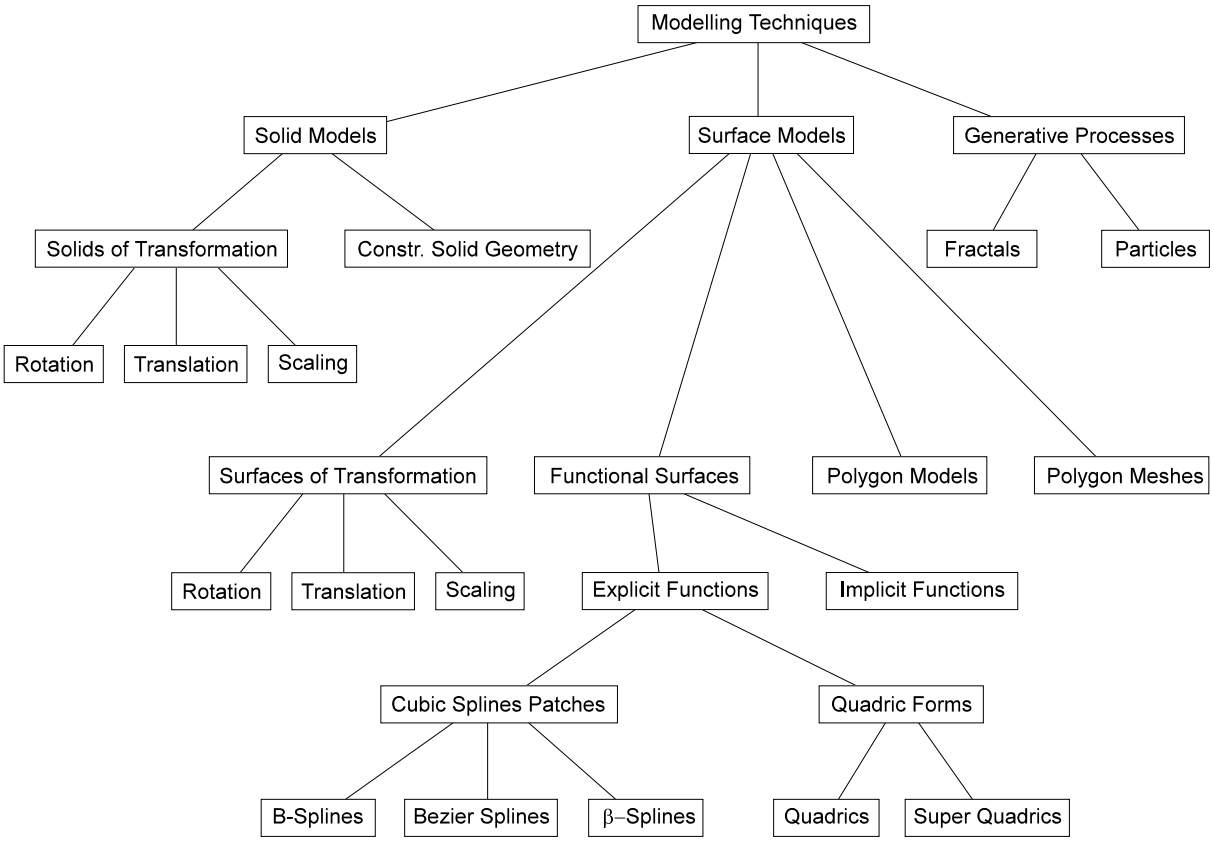


*Figure 1-**Error! Unknown switch argument.**: Taxonomy of Modelling Techniques*

32 Bézier patches, i.e. 512 three-dimensional points). The major drawback of this representation is the amount of computation necessary to render patches, compared to polygon rendering (for which we propose a parallel solution in [Lamotte 91]).

Generative processes create objects based on the two previous categories, using an algorithmic process, e.g. particle systems (see [Reeves 83]).

Our implementations and algorithms are based on the two surface models mentioned: polygon models and Bézier surface patches.

## 1.2. Transformations: Putting The Models In Scene

Once we have built the basic objects that will be part of the scene we want to visualise, they have to be positioned with respect to each other. This is what is called “object transformation”. Three different transformations are possible in order to obtain the right proportions and relative positions: (1) translations put objects in another position without altering their

size or orientation, (2) rotations can be use to rotate an object about one or more of its axes, and (3) scaling is available for resizing.

After all necessary transformations have been carried out on the objects, the scene is ready to be visualised on the screen.

## 1.3. Rendering

The third step after modelling and positioning is displaying the scene ("rendering"). There are many ways of calculating a rendered image of 3D objects, but I will give a brief overview of the most widely used methods, in order to position our work in a larger context.

Rendering a scene involves two major steps:

(1) visible surface determination: one has to use a method to determine which parts of the objects are visible on which point in the resulting image

(2) illumination and shading: when it is determined which object's surface is visible in a certain pixel, the influence of light on the surface has to be calculated, including effects like shadows, transparency, reflections, etc.

### 1.3.1. Visible Surface Determination

When one wants to render a 3D scene, one of the key factors to be determined is which surface is visible through which pixel of the resulting image. This seems a rather simple thing to do, but visible surface determination involves a lot of calculations. Various techniques have been developed throughout the evolution of computer graphics. They can be grouped into two main streams (which are sometimes combined in one algorithm):

(1) *image-precision algorithms*: for each pixel they determine which object is closest to the viewer. In pseudo-code:

```
for each pixel in the image do
      begin
      determine which object is closest to the viewpoint along the
            line between the viewpoint and the pixel
      determine the colour at that position
      end
```

In a brute-force method, for each pixel all objects are compared with each other to determine which is closest to the viewer, but of course this method is much too expensive.

(2) *object-precision algorithms*: all objects are compared with each other to determine which parts are visible. In pseudo-code:

```
for each object in the scene do
      begin
      determine which parts of the object are not obstructed by other
            objects (or a part of the object itself)
      determine the colour on those parts
      end
```

Here also, an expensive, straightforward implementation is possible: every object is compared to all others (and itself) and all invisible portions are discarded. But since this basic operation of finding visible portions is difficult and time-consuming, this method also is not desirable.

Please note that image-precision algorithms are normally performed at the resolution of the desired output image, whereas object-precision algorithms have to be followed by a step in which these objects are actually displayed at a certain resolution.

We discuss the most widespread algorithms in somewhat more detail.

#### *1.3.1.1. Z-Buffer*

This algorithm is also called the *depth-buffer* algorithm and is introduced by Catmull [Catmull 74]. The algorithm not only stores the colour values of the pixels in a frame buffer, but also maintains a buffer with the depth (z-value) of the nearest polygon found so far in each pixel. If during the scan of the database an object is encountered with a depth closer than the one stored in the z-buffer, the new point's colour and depth are taken to replace the old values.

This very simple algorithm can be extended by using depth coherence. If one inserts the projection of each polygon into the buffers one scan line at a time, the z-values of each point on the scan line can be derived from the z-value of the previous point, just because of the fact that a polygon is planar. Because of its simplicity, the z-buffer method is often implemented in hardware. Its only real requirement is to have enough memory to store the depth buffer (which has become less of a problem, since the memory got "cheap").

#### *1.3.1.2. List-priority Algorithms*

These combine object-precision and image-precision. The principle is that the objects are ordered on visibility, such that a correct result is obtained if the objects are rendered in that order. There are several possibilities:

- There are no overlaps between objects in the depth (z) direction: in this case the objects can be sorted on their z value and rendered back to front.

- Some objects overlap, but a correct order can be found: this gives the same result as the previous case.

- There is a cyclic overlap in z (between two or more objects): in this case no unique order can be found. Therefore, the objects have to be split to order the pieces linearly.

The ordering and object splitting are performed at object precision, whereas the drawing of one object over another is done at image precision. But of course, the ordered list of objects can be redisplayed at any resolution without having to recalculate the order.

Some of the most well-known methods are the depth-sort algorithm (or, in simplified version, the painter's algorithm) and binary space-partitioning trees. For a short discussion of these algorithms, see e.g. [Foley 90].

#### *1.3.1.3. Scan-line Algorithms*

These work at image precision to generate an image one scan-line at a time. It is an extension of the general polygon scan-conversion algorithm that is used in two-dimensional graphics. The general idea is as follows. First an edge table (ET) is created for all non-horizontal edges of all polygons projected on the view plane. These edges are sorted on their smallest y-value (height co-ordinate in the projection). The image is scanned from bottom to top. At all times, an active edge table (AET) is stored, containing all the edges that are crossing the current scan-line, sorted on their x-value. By maintaining Boolean flags that tell if a scan is inside or outside of a polygon, the algorithm is always able to determine which polygon is closer than all others on each point on the scan-line.

A popular derived method is based on the scan-line principle for finding the candidate polygons, but uses a one-line z-buffer for the depth-comparisons. In this case, there is no need for a large, full-resolution depth-buffer, which enables very high-resolution images to be calculated.

#### *1.3.1.4. Visible-surface Ray Tracing*

This is the reduced version of the full ray tracing algorithm, which is also called ray casting. Its principle is first introduced in [Appel 68].

Ray casting is basically what we described in the general discussion of image-precision algorithms. A projection centre is placed at the viewer's eye position and an arbitrary window on a view plane is selected. This window can be seen as a grid of points which correspond to the pixels of the resulting image. For each such pixel, an eye ray is fired from the viewpoint through the pixel, into the three-dimensional scene. This ray (topologically a line) is intersected with all objects in the scene and the closest intersection point is retained. The colour at this point is calculated and stored as pixel value in the resulting image.

One uses the term "ray tracing" for the recursive version of ray casting. The same algorithm is used recursively, together with the shading at each intersection point. We will explain this in detail in section 1.3.4.1.

### 1.3.2. Illumination Models

In the explanation of visible surface determination, we elucidated that the algorithms mostly end with filling the screen pixel with the colour of the nearest object. Illumination and shading take care of this part of the rendering. These two terms are sometimes used to indicate the same thing. In this text, we will use "illumination" to indicate the way a colour can be calculated on a given surface at a given point. "Shading" is used with a broader meaning: it indicates which illumination model should be used at which point. Some shading models calculate the illumination for each pixel in the image, while others determine the illumination at certain points only and perform interpolation techniques for all other pixels.

We have to stress the fact that a physically exact illumination calculation is a very complex task. Therefore, most illumination and shading models are based on rough approximations that are mostly just empirically based.

In this section, we describe basic illumination models that will be integrated into the shading models in the next section. The description will first be based on monochromatic lights (having nothing but an intensity) for reasons of simplicity, but will be extended to coloured lights.

#### *1.3.2.1. Self-illumination*

This is the most simple illumination model: each object is associated with some intrinsic intensity and is therefore displayed using only this intensity.

We can then write the illumination equation as follows:

$$I = k_i \qquad \textit{(Eq. 1-1)}$$

where $I$ is the resulting intensity and $k_i$ is the object's intrinsic intensity.

#### *1.3.2.2. Ambient Light*

Assume that the objects are not self-illuminating, but that there is a uniform, diffuse light, without direction, that equally illuminates all objects in the scene. This light is termed "ambient light" and is thought of as the accumulated light reflected off the surfaces of all objects in the scene.

Then we obtain a new illumination equation:

$$I = I_a k_a \qquad \textit{(Eq. 1-2)}$$

where $I_a$ is the ambient light's intensity and $k_a$ (with a value between zero and one) is the ambient-reflection coefficient that indicates the proportion of the ambient light that is reflected from the surface of the object at hand.

#### *1.3.2.3. Diffuse Reflection*

With ambient light, the intensity of the objects are influenced by an external factor, but they still have a uniform colour over their entire surface, which is not at all realistic. Let's assume we have a point light that shines equally in all directions. An object that receives light from such a light source will not be illuminated uniformly, depending on the angle between the surface and the light source's position (e.g. those parts facing away from the light will receive no illumination at all and will remain dark, whereas the side which is perpendicular to the light's rays arriving there will be lit most). This is what we call "diffuse reflection" or "Lambertian reflection".

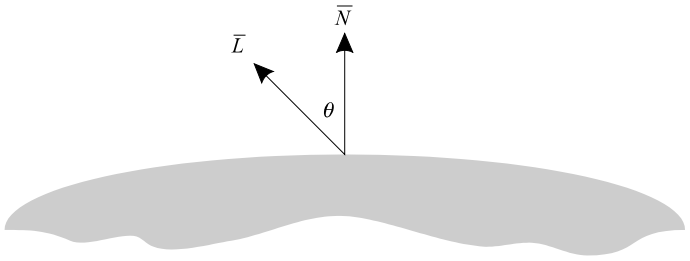


*Figure 1-**Error! Unknown switch argument.**: Diffuse reflection*

For a given surface, the brightness depends only on the angle $\theta$ between the direction to the light source ($\overline{L}$) and the surface normal ($\overline{N}$) (see Figure 1-3).

The following illumination equation covers this diffuse reflection:

$$I = I_p \, k_d \cos \theta \qquad \textit{(Eq. 1-3)}$$

where $I_p$ is the point light's intensity and $k_d$ is the diffuse-reflection coefficient, varying from 0 to 1 and depending on the surface characteristics. $\theta$ must have a value between 0º and 90º, otherwise the surface is facing away from the light source and the object occludes itself.

If the vectors $\overline{L}$ and $\overline{N}$ are normalised (which is common practice in computer graphics), the cosine can be calculated as the dot product of the vectors $\overline{L}$ and $\overline{N}$:

$$I = I_p \, k_d \, (\overline{N} . \overline{L}) \qquad \textit{(Eq. 1-4)}$$

Objects that are rendered using equation 4 seem to be lit by a flashlight in a dark room: there is no light coming from elsewhere than the light source. Therefore, most renderers combine ambient and diffuse reflection into one, more realistic illumination equation:

$$I = I_a \, k_a + I_p \, k_d \, (\overline{N} . \overline{L}) \qquad \textit{(Eq. 1-5)}$$

These computations are still performed under the assumption of monochrome lights and objects. Since computer-generated images are mostly shown on computer monitors that work with the three-component colour representation using the primary colours red, green and blue, light and surface colours are often expressed using these three "RGB" components. The object will have a diffuse colour which is expressed as ($O_{dR}$, $O_{dG}$, $O_{dB}$). Likewise, the point light's colour can be represented by the triple ($I_{pR}$, $I_{pG}$, $I_{pB}$). When we use this RGB approach, we obtain three equations, each describing the illumination for one component, e.g. for the red component this will yield

$$I = I_a\, k_a\, O_{dR} + I_{pR}\, k_d\, O_{dR}\, (\overline{N}.\overline{L}) \qquad \textit{(Eq. 1-6)}$$

This three-component model is very often used and can easily be implemented, but actually it is wrong (the explanation for this would carry us too far, but the interested reader is referred to section 16.9 of [Foley 90]). In order to be more correct, it should be written as a function of the light's wavelength ($\lambda$):

$$I = I_a\, k_a\, O_{d\lambda} + I_{p\lambda}\, k_d\, O_{d\lambda}\, (\overline{N}.\overline{L}) \qquad \textit{(Eq. 1-7)}$$

#### *1.3.2.4. Specular Reflection*

This kind of reflection is observed on shiny surfaces. If one looks at a shiny object, at some angles the light source seems to reflect on the surface, while at other angles no such reflection can be seen. Specular reflection is, unlike diffuse reflection, not uniform in all directions: on a perfect mirror, only in the exact light reflection direction ($\overline{R}$), a specular reflection can be observed (see Figure 1-4).

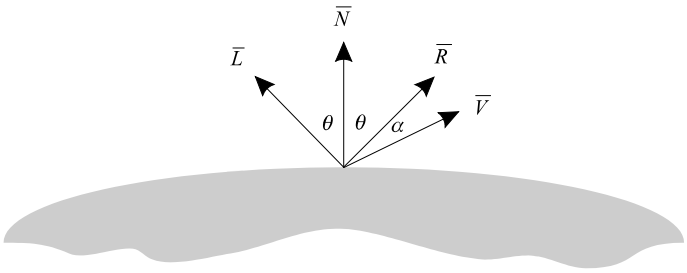


*Figure 1-**Error! Unknown switch argument.**: Specular reflection*

So normally, only if the viewer ($\overline{V}$) is looking at the surface in the reflection direction (i.e. if $\overline{V} = \overline{R}$, or the angle $\alpha$ is zero), he will be able to see the reflection.

Perfect mirrors however are not existing in real life. Most shiny surfaces reflect most light in the actual reflection direction, but still some light is reflected at angles that differ slightly from this direction. The most popular method for approximating this effect is known as Phong-shading. In this approach, it is assumed that maximal reflection is found at the specular reflection direction (i.e. where $\alpha$ is zero) and fall of very quickly when $\alpha$ increases. This is approximated by a factor $cos^n\ \alpha$ , where $n$ is called the "specular-reflection exponent" and depends on the material's properties. It assumes values between 1 and infinity: a value of 1

gives a very broad and soft highlight, whereas higher values give sharper highlights. (Note that for a perfect mirror, an infinite value would be used.)

In the Phong model, under the assumption that all vectors are normalised (yielding that $cos^n \alpha = (\overline{V} . \overline{R})^n$ ), we obtain a new extension to the illumination equation:

$$I = I_a k_a O_{d\lambda} + I_{p\lambda} [k_d O_{d\lambda} (\overline{N} . \overline{L}) + k_s (\overline{V} . \overline{R})^n] \qquad \textit{(Eq. 1-8)}$$

where $k_s$ is called the specular-reflection coefficient, ranging from 0 to 1, and assuming that all vectors are normalised. Observe that in the specular reflection no colour component from the object is taken into account: remember that specular reflection is just the effect of reflecting the light colour in the specular reflection direction. For some materials however, some object colour (not necessarily identical to the diffuse colour) is mixed with the light colour. If we want to take this into account, we have to introduce a specular colour for the object: $O_{s\lambda}$, which yields:

$$I = I_a k_a O_{d\lambda} + I_{p\lambda} [k_d O_{d\lambda} (\overline{N} . \overline{L}) + k_s O_{s\lambda} (\overline{V} . \overline{R})^n] \qquad \textit{(Eq. 1-9)}$$

All illumination equations we have introduced so far only took one point light into account. If more sources are put in the scene, their contributions have to be summed together:

$$I = I_a k_a O_{d\lambda} + \sum_{i=1}^{m} I_{p\lambda i} [k_d O_{d\lambda} (\overline{N} . \overline{L}_i) + k_s O_{s\lambda} (\overline{V} . \overline{R}_i)^n] \qquad \textit{(Eq. 1-10)}$$

### 1.3.3. Shading Models for Polygons

The brute-force method for shading an object is to apply the illumination equation (Eq. 10 to obtain the best effect) on every visible point on the object's surface, but this is too expensive in terms of computation. More efficient approaches are available.

#### *1.3.3.1. Flat Shading*

This method is also known as constant shading. For each polygon in an object, just one colour value is calculated (e.g. at the centre position or in the first vertex) and used as a constant colour value over the rest of the polygon. This approach produces the right result if the light is positioned at infinity, the viewer is at infinity and the object actually needs to have a faceted surface.

If we want to obtain smooth changes and apparently curved surfaces, some interpolation method is needed to use information available on surrounding polygons and points. Two such methods are very popular in computer graphics: Gouraud shading and Phong shading.

### *1.3.3.2. Gouraud Shading*

Gouraud [Gouraud 71] introduced a method to use interpolation of colours for obtaining a smooth appearance on curved surfaces. The principle is quite simple: one calculates the colour values at the polygons' vertices, depending on the surrounding polygons' orientation. This assumes that we have the surface normal at each vertex, which can be obtained in several ways: if a higher-level, analytical description of the surface is available, the normal can be calculated directly from this description; otherwise, it can be approximated by averaging the normal vectors of all surrounding polygons (see Figure 1-5).

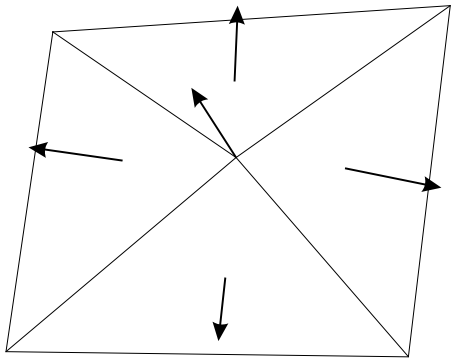

*Figure 1-**Error! Unknown switch argument.**: Interpolated vertex normal*

Using this interpolated normal vector, any illumination equation can be applied to compute the vertex colour. Once this process is performed for each vertex, interior points are shaded by interpolating between the values in the vertices, by first calculating increments along the edges. This interpolation can easily be integrated in a scan-line algorithm, because depth interpolation is already performed.

### *1.3.3.3. Phong Shading*

Phong shading [Bui-Tuong 75] does not interpolate intensities, but normal vectors. The algorithm starts with the same normal interpolation at the polygons' vertices as the one that is performed in Gouraud shading and then performs the normal interpolation in much the same way as the colour interpolation in Gouraud shading.

Using the interpolated normals, again any illumination model can be applied. This method yields better results than the previous one, because the illumination is recalculated at each visible point using a more accurate normal. Of course, this better visual performance is associated with more complex calculations.

We have to add that these interpolation methods (Gouraud and Phong shading) have some drawbacks (of which most can be avoided by using triangles instead of general polygons):

(i) because of the calculations in screen space, one gets a perspective distortion for polygons of which some points lie further away in the depth direction,

(ii) the calculations depend on the orientation of the polygon: values are interpolated between vertex values along horizontal scan-lines, which means that an interior point can receive another value after a rotation, e.g. in Figure 1-6(a), the point P is interpolated between the values in A, B and D, whereas in Figure 1-6(b), the values of A, B and C are used, even though it is exactly the same point P and the polygon has only undergone a rotation.

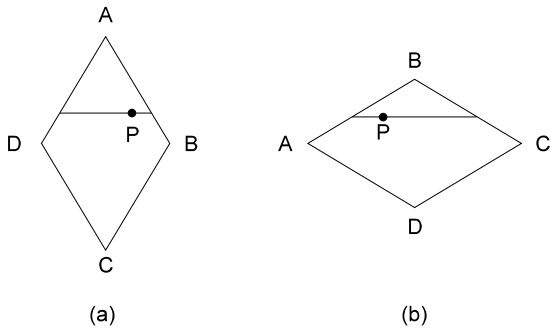


*Figure 1-**Error! Unknown switch argument.**: Orientation dependence*

(iii) the interpolated vertex normals should be representative for the whole surface; if this is not the case, strange shading can occur.

### 1.3.4. Global Illumination Models

The models we described above are appropriate for local shading: this is illumination that can be calculated only depending on the surface characteristics, the light parameters and the viewer's position. However, if one wants to take into account indirect lighting, shadows, reflection, refraction and other effects that depend on the rest of the scene, the traditional shading models do not suffice anymore (note: some extensions have been made to incorporate a simulation of these effects too, e.g. reflection mapping ([Blinn 76]) and two-pass z-buffer shadow algorithms ([Williams 78]), but these are not as exact as the global illumination models).

Therefore, there is a need for illumination models that intrinsically include these necessary extensions. Global illumination models try to mimic the indirect influences on the colour at a given point. Thus far, this indirect term has been represented in the illumination equations by the ambient part.

Ultimately, the goal of these illumination models is to approximate as closely as possible what Kajiya called the "Rendering Equation" ([Kajiya 86]), which expresses the light that travels from point B to point A, in terms of

1) the intensity that is emitted directly from B to A and
2) the light that is emitted from all other points to point B and from there to point A (which is in its turn recursively defined by the same rendering equation).

Kajiya formulated the rendering equation as follows:

$$I(A,B) = g(A,B)\left[\varepsilon(A,B) + \int_S \rho(A,B,C)I(B,C)dC\right] \qquad \textit{(Eq. 1-11)}$$

where

- *I(A,B)* is the energy arriving at A, coming from B;
- *g(A,B)* is a geometry term that is zero when A and B are invisible to each other, otherwise it equals $1/r^2$ with $r$ the distance between the two points;
- $\varepsilon(A,B)$ is the intensity that is emitted from B to A directly;
- the integral is over all points C on all surfaces S;
- $\rho(A,B,C)$ is the proportion of the light reflected (both diffuse and specular reflection) from C to A via the surface at B;
- *I(B,C)* is the energy arriving at B, coming from C (which is thus defined recursively).

Two global illumination models that are approximations of the rendering equation are recursive ray tracing and radiosity. Recursive ray tracing interleaves the visible-surface ray tracing with shading, to incorporate shadows, reflection and refraction. Radiosity on the other hand, completely separates visible-surface determination from shading. Here the (view-independent) influence of all objects (including light sources) on each other is first fully calculated, whereupon several different views can be generated using Gouraud shading.

This distinction between the view-dependence of ray tracing and the view-independent calculations in radiosity, is important: in a view-dependent algorithm, a discretisation is performed on the viewport to find the points where we have to apply the illumination equation, whereas in a view-independent algorithm the scene is discretised and processed in order to provide the information that is needed to evaluate the illumination equation at any point from any given view direction.

### *1.3.4.1. Recursive Ray Tracing*

As we stated before, this is the recursive version of the ray casting algorithm, interleaved with shading. In order to shade the nearest intersection point adequately with global illumination, it is necessary to determine which objects are visible on a reflective surface or through a transparent object. This principle is depicted in Figure 1-7. Once the nearest object (object A in the figure) is found, the illumination equation is applied to it, yielding the "own colour" of object A under the influence of direct lighting. Until this point, we only have applied the visible-surface ray tracing technique. If A's surface is reflective, a reflection ray ($R_1$) is spawned, performing the same algorithm recursively to determine the colour of an object B that is reflected on the surface of object A (again by calculating the colour at the intersection point on the surface of object B and tracing a reflection ray $R_2$). Likewise, if object A is transparent, a refraction ray ($T_1$) is created according to Snell' Law and is also traced further recursively. (An example of reflection and refraction is shown in Colour Plate 3, rendered without shadows: on a checkerboard, 9 refractive spheres are placed with refraction indices increasing from top left to bottom right).

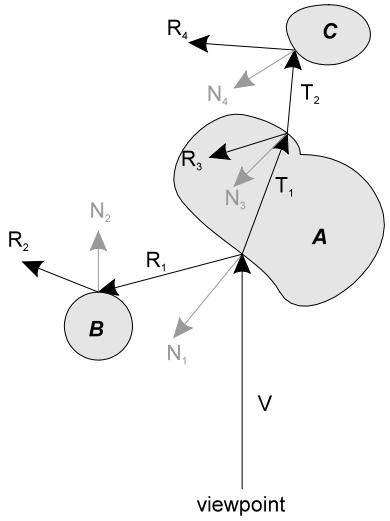


*Figure 1-**Error! Unknown switch argument.**: Recursive ray tracing*

According to surface characteristics, these three values (own colour, reflection colour and refraction colour) are weighed to obtain the final colour that will appear in the pixel where object A was visible. The reflection and refraction rays are called "secondary rays".

In order to be exact, this recursion should proceed for as long as there is any contribution left to be filled in by secondary rays. Mostly, the user can specify the maximum recursion depth. Colour Plates 4, 5 and 6 show three consecutive levels of recursion: the scene consists of a striped plane with five spheres stacked on its surface. Depth 1 (Colour Plate 4) gives the same result as a standard Phong shading algorithm (with shadow casting): just shiny surfaces, but no reflections. Depth 2 (Colour Plate 5) shows that the spheres are reflective and that two perpendicular mirrors are positioned vertically on the ground plane. Of course, the reflections of the spheres in those mirrors do not appear reflective, because one more level of recursion would be needed. This is what is shown in Colour Plate 6, where we traced one step further, such that the reflections become reflective, but also a new virtual image appears at the cross-

ing of the two perpendicular mirrors. This way, one can go on to any arbitrary depth, revealing more and more reflections.

Of course, one can put a threshold on the need for further recursion (see [Hall 83]): it is reasonable to state that, if a ray contributes less than, let's say, 1 % of the final colour of a pixel, it will no longer be visible to the observer if this contribution is added or not. Indeed, every surface will absorb some portion of the incoming light, while composing the final colour from a certain percentage of its own colour and some percentage from secondary rays. An example: assume that all the objects in the scene are made of a material with the following properties:

- 20 % absorption
- 40 % own colour
- 20 % reflection's colour
- 20 % refraction's colour

In this case, if each recursive step yields an actual intersection point (still on a surface with the same properties), the 1 % threshold will be reached after three steps (0.2 * 0.2 * 0.2 = 0.8 % < 1 %). This way, one can cut off many unneeded calculations that would not yield a visible difference. Of course, the choice of the threshold depends on the characteristics of the scene: if one wants to show many levels of reflections, a low threshold is needed.

In order to gain even more realism, using the same paradigm of tracing light rays yields the calculation of shadows: by tracing a ray in the direction of each light source, it can be determined if the surface lies in the shadow with respect to each light source, just by checking if the shadow ray intersects an object. Of course, this shadow test has to be performed at every level in the recursion: the surface colour has to be calculated under the influence of the light sources.

Observe that the key operation of the ray tracing algorithm is the calculation of the nearest intersection point. Naively, one could take all the objects in the scene, intersect them with a ray and then compare all these values to obtain the nearest one. However, if one would like to generate an image at a reasonable resolution, let's say 800 x 600 pixels, this brute force approach would yield 1,440,000 rays, assuming that 50% of all rays yield two secondary rays and two shadow rays (i.e. there are two light sources). Assuming that there are 10,000 primi-

tives in the scene (which is still a rather modest scene), 14.4 billion ray-primitive intersections have to be calculated ! When sub-pixel sampling will be used for anti-aliasing purposes, this number will increase even more. Obviously, research has to be done on speedup techniques and ways to reduce this number of intersection calculations. In Chapter 3, we will give an overview of these speedup techniques for ray tracing.

#### *1.3.4.2. Radiosity*

Remember that ray tracing takes into account specular reflection and refraction. This approach yields sharp, mirror-like reflections. But in real life, a whole lot of diffuse, indirect reflections contribute a great deal in the illumination: if we put a white light in a white room, all objects in the room will have their natural colour, but if the walls are painted red, everything will appear reddish, due to the fact that most surfaces reflect some of the incoming light in a diffuse way.

Radiosity is a rendering technique that takes this diffuse reflection into account. If we recall the final illumination equation (Eq. 10), we see that the colour of an object only depends on the influence of ambient light and directed light sources on the surface of the object (with the extension in ray tracing that the colour of other objects may be reflected, too). Radiosity takes one step further, by taking into account light that bounces off each surface diffusely. Therefore, the following equation is solved for each surface in the scene:

$$B_i A_i = E_i A_i + \rho_i \sum_j B_j F_{ji} A_j \qquad \textit{(Eq. 1-12)}$$

where

- $B_i$ $(B_j)$ = radiosity (energy per unit area) of the surface $i$ ($j$)
- $A_i$ $(A_j)$ = area of the surface $i$ ($j$)
- $E_i$ = emitted energy per unit area
- $r_i$ = reflectivity of surface $i$
- $F_{ji}$ = form factor from surface $j$ to surface $i$ (fraction of the energy that leaves surface $j$ and arrives at surface $i$)

This equation can be applied to every surface $i$ in the scene, yielding a system of simultaneous equations containing as many entries as there are surfaces. The solution of this system of equations determines the radiosity value of each surface. The final step in the radiosity

method is to use these surface radiosities to determine the radiosity at the vertices, which can then be used in an interpolative shading technique like Gouraud shading.

Before one can solve the radiosity equations, the form factors between each pair of surfaces have to be calculate, which is an $O(n^2)$ operation, because each surface has to be compared with all other surfaces. In chapter 7, we will show how a progressive radiosity method, based on ray tracing for the form factor determination, can be used to overcome this problem of complexity.

# 2. Parallelism and Transputers

The central idea behind parallelism is the use of multiple interconnected processors that work together on a certain common task. One of the first and most widely used processors, specifically designed for this purpose, is the transputer. This is a processor based on the RISC principle. Its physical layout is shown in Figure 2-1.

One can see that there is an on-chip floating point unit and 4 K on-chip RAM, but most important is the availability of the four bi-directional communication links. These provide the hardware necessary to communicate with other processors in the transputer network. Using these links, one processor can send data to another transputer, provided that the other one is waiting for input from the sender. If this is not the case, the sender will wait for the receiver to become ready (or vice versa). In other words, there is one-to-one synchronous communication. Hardware for scheduling and descheduling, together with the use of different priorities, enable parallel tasks to be executed while another process is waiting for synchronisation.

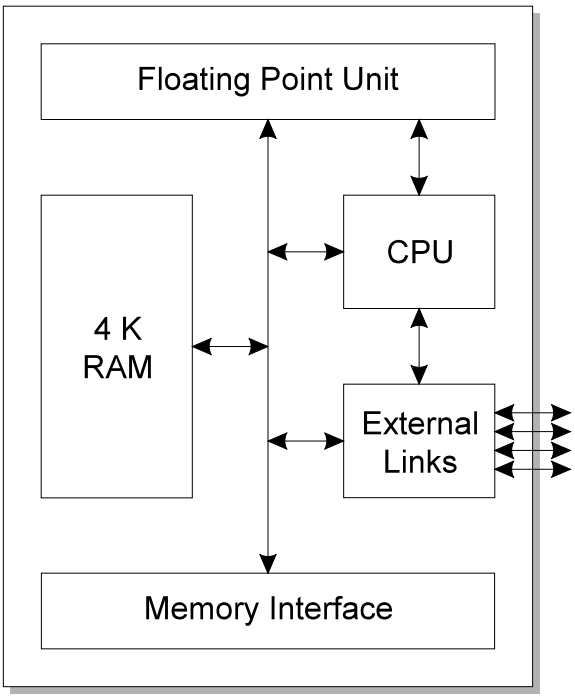


*Figure 2-**Error! Unknown switch argument.**: Transputer hardware layout*

In parallel program design, some key aspects have to be taken care of:

(i) program distribution

(ii) load balance

(iii) reduction of communication overhead

(iv) data sharing and distribution

## 2.1. Program Distribution

Writing a program for a parallel system is (in general) not just running the same program on more than one processor. Of course, if one wants to run a very short program a huge number of times, this SIMD approach (Single Instruction Multiple Data) will suffice, because in this particular case, the program just performs a minimal amount of calculations, but they have to be carried out many times. Most of the time however, one will have a complex program with a lot of different calculation steps which take a considerable amount of time to run only once. In this more general case, it will be more interesting to divide the different steps between multiple processors. Of course, often the results of one step in the calculation will be the input for the next step. If these two subsequent steps have to be executed one after another, the second process will have to wait for the results of its predecessor. Keeping this in mind, it is important to divide the work in such a way that this waiting time can be reduced as much as possible. This means that it will not be a good practice to let one processor do a very simple calculation, while the next has to perform a very complex one: the second would make the first wait almost constantly. Finding these dividing lines between tasks that can be put on different processors, is one of the first things to be investigated when implementing a parallel system. In this context, it is important to find parts of the program that are more or less independent from each other's results and to try and distribute these over different processors.

## 2.2. Load Balance

This issue is strongly connected to the previous one. Even if one can find sub-tasks in a program that are totally independent of each other's results, and if one puts these tasks on different processors, it is still possible that one processor has to perform calculations that are far more complex than those performed by all others. To give a very simple example: suppose we have 9 processors that just perform an addition, while a tenth has to compute a square root, the latter will be occupied much longer than all the others. If the output of the program depends on the combined result of the 10 processors, 9 will be idle almost immediately, while one still has to wait for the tenth to complete.

Therefore we can conclude that we have to find a way of giving the processors an almost equal workload, in order to keep the overall load in balance.

## 2.3. Reduction of Communication Overhead

As mentioned in the introduction on transputers, communication is synchronous: the processor that is first ready to communicate, has to wait for the other one to become ready. As a consequence, the first one cannot continue its computations unless the other processor has participated in the communication. Moreover, if the amount of data that has to be transferred from one processor to another is very large, this will sometimes generate an unacceptable overhead. This is another important issue to keep in mind. Of course, if the process that is waiting for communication is running concurrently with other processes, the scheduling mechanism makes sure that those concurrent processes can be scheduled for execution during this otherwise idle CPU time.

## 2.4. Data Sharing and Distribution

As we are using a system in which each processor is equipped with its own local memory (typically 2 or 4 Mbytes, but sometimes 16 or even 32 Mbytes), that cannot be accessed by any other processor, algorithms have to be devised, bearing this limitation in mind; for example, there cannot be a central database, accessible by all processors. Moreover, if we want all the processors to “know” some global data, this can be done by copying it into all processors’ local RAM, but in this case, the size of the global data block is limited to the size of the smallest RAM available in the network. Therefore, whatever algorithm is implemented on such a transputer system, this limitation has to be borne in mind.

# 3. Ray Tracing Speedup and Parallelism

Ray tracing has always been renown for its simplicity and for the stunning images that could be created. But it could never get rid of its "fame" of being incredibly time-consuming either ! It was common knowledge that a ray traced image took hours or days to compute. Throughout the research that has been carried out in the field of ray tracing, several ways of speeding up the calculations have been introduced. They can be grouped roughly into the following categories (which can very well be combined to produce even more efficient algorithms):

- faster intersections
- generate fewer rays
- generalised rays
- calculate less intersections
- introduce parallelism

We took recourse to a combination of parallelism, techniques to generate fewer rays and methods to calculate less intersections. In the next few sections, a brief overview is given of the four categories mentioned, followed by a general discussion of the parallelisation introduced in our system.

## 3.1. Faster Intersection Calculations

This is the most straightforward way of speeding up the bulk of ray tracing: just try to make the ray-object intersection calculations faster. Some examples of this category are

- the use of object bounding volumes: by enclosing an object with a bounding volume that is easier to intersect, it is only necessary to intersect the object itself if the bounding volume was intersected;
- find efficient methods for certain ray-object intersection calculations, just on a numerical basis.

## 3.2. Fewer Rays

Some rather obvious possibilities are available for reducing the number of rays that are actually traced. A simple examples is the use of adaptive tree depth control: this is what we explained in the general discussion of recursive ray tracing in section 1.3.4.1 on page 22: by defining a threshold value on the percentage that each ray should contribute to the final pixel colour, one can adaptively limit the depth of the ray tree;

## 3.3. Generalised Rays

Other methods use an extension to the concept of a ray to a more general notion of beams, cones or pencils. The idea behind this generalisation is that many rays are traced simultaneously. These techniques no longer work with intersections between thin rays and objects but with coverage of objects by whole bundles of rays.

## 3.4. Reducing the Number of Intersection Calculations

This is perhaps the most effective category. Since testing the whole scene for intersection with each ray is much too time-consuming, some ways have to be found for efficiently searching those objects that are most likely to be hit by a certain ray. The methods in this category can be divided into three subgroups: space subdivision, bounding volume hierarchies and directional techniques.

Space subdivision divides the bounding box of the scene into a regular grid of boxes ("voxels") (see [Fujimoto 86], [Amanatides 87]) or into a structure with non-uniform cell sizes, e.g. octrees ([Glassner 84], [Wyvill 86]) or binary space partitioning (BSP) trees ([Kaplan 87]). In either case, references to objects are stored in the cells of the voxel or octree cells. In the rendering stage, these cells are traversed in ray order. This means that just a few cells will be traversed and once an intersection is found, all cells behind this point can be discarded for further consideration. This technique reduces the number of ray-object intersections very drastically. The key factors for the choice between the voxel structure and the octree is a trade-off between memory usage and traversal speed: a voxel grid can be traversed very efficiently using a kind of 3D DDA algorithm, but empty space sometimes consists of

many empty cells; on the other hand, this empty space can be represented with just a few octree cells, but here the traversal is more complicated, due to the non-uniform box size.

The hierarchical approach builds a tree-based hierarchy on top of the primitive objects, mostly based on the relative positions of the primitives' bounding volumes (see, e.g. [Weghorst 84], [Kay 86]): one takes several objects together and a common bounding volume is positioned around the group. On the next level, some of these volumes are grouped within a higher-level volume and so on, until one root volume is reached. A ray-scene intersection then involves traversing the tree until the nearest intersection is found and/or the rest of the tree can be discarded, based on the position of the bounding volumes. The key factors in this approach are the fact that the bounding volumes have to be easier to intersect than the underlying primitives and that through an appropriate algorithm large branches can be pruned.

## 3.5. Introducing Parallelism

The next step after all the speedup techniques we described, is the introduction of parallelism. Much research has been dedicated to the use of parallelism in ray tracing, which can be categorised in four groups:

1. General-purpose vector processors
2. General-purpose multi-computers
3. Special-purpose hardware
4. General-purpose multiprocessors

We will highlight some of the applications that have been developed within these categories.

### 3.5.1. General-purpose Vector Processors

One of the first vectorised algorithms for ray tracing was described in [Max 81]. It was implemented on a CRAY-1 supercomputer, to render ocean surfaces, without taking into account shadows. The program traced the primary rays as a vector, traced only 2 reflections deep and all objects were intersected with each ray. The algorithms used did not lend themselves for further optimisation techniques.

Another example was an implementation by Plunkett and Bailey ([Plunkett 85]) on a CDC Cyber 205. Here also an exhaustive intersection was performed on the whole scene, but rays

were treated in a more uniform way, allowing shadows, reflections and refractions. Due to the exhaustive ray tracing, only limited databases could be rendered.

### 3.5.2. General-purpose Multi-computers

In those cases where a sequence of ray traced images are needed, e.g. for rendering an animation, an obvious way of speeding this process up is by using a network of connected general-purpose computers. In [Peterson 87], some statistics were published on the work load of computer systems. It turned out that most machines were not heavily used, even not during office hours, just because much time is spent in simple tasks such as reading mail or editing; on average, they were in idle state 90 to 95 % of their time. Peterson gives an example of an animation that was rendered on 30 machines in 24 hours, while the total CPU time was 24 days and 10 hours.

### 3.5.3. Special-purpose Hardware

Because of the complexity of the ray-surface intersections, many researchers have tried to build this basic operation in special-purpose hardware. Nishimura et al. [Nishimura 93] constructed the LINKS-1 system, consisting of a parallel-pipelined multi-microcomputer. Each of the pipelines performed the three steps of object sorting, ray tracing and shading, but a prerequisite was that each pipeline was equipped with enough memory to store the whole database. Adding too much pipeline nodes made the connection to the root computer become a bottleneck.

A proposal for a special-purpose VLSI design for intersecting bi-cubic patches can be found in [Pulleyblank 87]. Another VLSI component, described in [Gaudet 88], is the Pipelined Engine for Ray Tracing (PERT). It consists of three processors connected in a cyclic way: a *shade* processor that shoots rays and performs shading calculations, a *shell* processor to perform the intersection tests for rays and bounding volumes (called “shells”) and a *prim* processor that performs ray-primitive intersection calculations. Multiple such constellations could be connected through buses. The system was never actually implemented, but simulations indicated that only 66% of the processing time was really used, due to the imbalance in the three pipeline stages.

Other approaches divided the object space between several processors (e.g. [Ullner 83], [Cleary 83], [Dippé 84]). The main problem in these approaches was obtaining a good load balance between the processors, which was addressed in [Nemoto 86] and [Kobayashi 87].

Most of these proposals, however, were never actually constructed. Moreover, general-purpose multiprocessors became more widespread, so the use of special-purpose hardware was no longer feasible.

### 3.5.4. General-purpose Multiprocessors

Salmon and Goldsmith ([Salmon 88]) made two implementations of ray tracing on an NCU-BE hypercube multiprocessor. In the first implementation, all primary rays are treated independently and are distributed between the processors in a static, scattered manner, which is sufficient for small databases that can be copied on each processor. For larger databases, a second approach is employed, in which a bounding volume hierarchy is constructed. The higher levels are copied on all processors, while the deeper levels are distributed between processors. This way, any processor can start handling a primary ray, but if the necessary lower levels do not reside on this processor, the computation will be passed to the appropriate processor for completion. Because of the complexity of the intersection calculations, the overhead created by these communications did not prevent from obtaining near-optimum speedups by increasing the number of processors. A similar scheme was used in [Scherson 88].

The first application of transputers in ray tracing is reported in [Packer 87]. Here a processor farm (explained in the next section) was used, on which the object database was distributed. Only a very limited number of primitives were included and no speedup techniques were incorporated, since the application was intended as an exmple to show in which way transputer-based parallel systems could be implemented.

Another well-known example of general-purpose multiprocessors is the Pixel Machine, developed by AT&T and described in [Potmesil 89], in which the processors acted as pixel processors, that could access a large frame buffer in parallel. The object database should be copied on each processor, but paging was used in case of memory overflow.

## 3.6. Parallelisation on the Transputer System

Let's now take a look at the general ideas behind the parallel ray tracing implementations on transputers. The main issue in parallelism is always the question how one has to divide the work between the processors one has at its disposal: should every processor be responsible for one of the steps in the algorithms, such that the data has to be pumped through this pipeline in a systolic way, or do all the processors have to perform the same calculations on a different data set ?

When looking at the ray tracing algorithm, it can be easily observed that the visible surface determination can be calculated in each pixel entirely without any knowledge about other pixels (as opposed to the algorithms that need interpolation across scan-lines, for example): indeed, it is sufficient to trace a ray through the pixel and to perform the ray tracing algorithm recursively, totally independent of what happened in a previous pixel.

With this locality observation in mind, the most straightforward way of parallelising the ray tracing algorithm, is to let each processor calculate pixels on a ray-by-ray basis: for every pixel, a ray is fired and the resulting colour can be calculated by one processor. Therefore, theoretically our parallel system looks like the following: one central processor is responsible for the work distribution. This controller knows which pixels are already calculated, and which ones aren't. When linked to all other processors in a star-shaped network (as in Figure 3-1), the controller is able to send each new ray to an idle processor and after some time, it receives back the results, using a buffer.

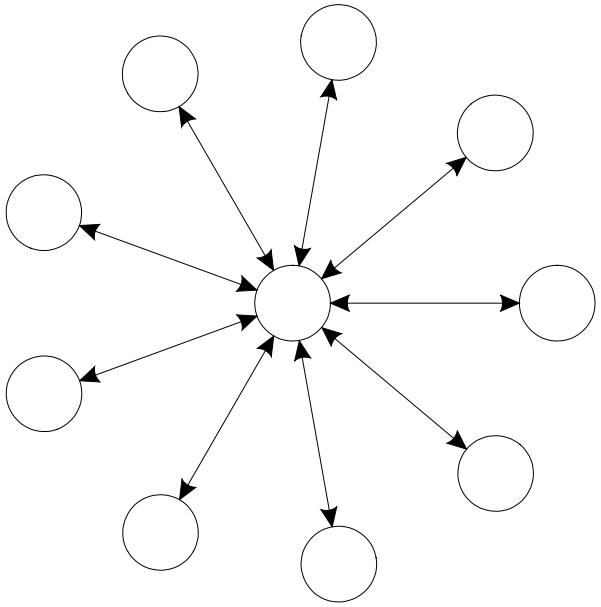

*Figure 3-**Error! Unknown switch argument.**: Star shaped network*

Unfortunately, this kind of networks cannot be created yet using transputers, since only four links are available per processor.

In the system that we could use for our research, the topology was linear list of processors. Of course, this imposed a severe limit on the network topology. In practice, we only had the

possibility to use either a farm or a ring topology (given the fact that the algorithms at issue had to be integrated with other system components and that dynamic linking of processors is not available).

In the farm topology (Figure 3-2), the processors are linked in one chain and communication is performed forward and backward. This means that if the first processor has to send data to the last, this data package has to travel through the whole network. The main advantage of such a farm topology is the expandability: if more processors have to be added, they can be appended at the end of the chain without changing any of the existing links.

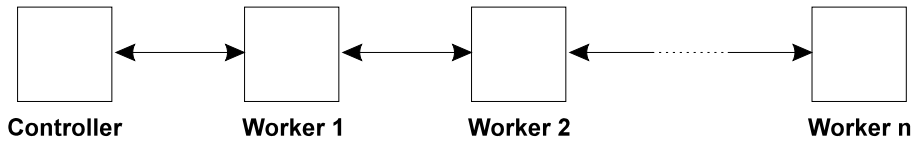


*Figure 3-**Error! Unknown switch argument.**: Farm topology*

In other cases, we could use a ring topology (Figure 3-3): in essence, this is the same as the farm, except for a back-link from the last processor to the first. The obvious advantage is that it is no longer necessary (though still possible because of the bi-directional links) to send data backward: there is only one stream necessary, which simplifies the communication protocol.

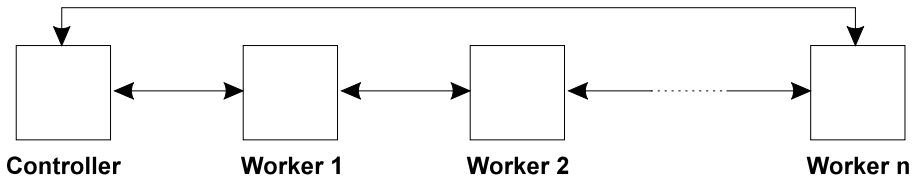


*Figure 3-**Error! Unknown switch argument.**: Ring topology*

If we look at the farm topology in closer detail, we see the following: the first processor serves as a controller and is connected to the host machine (the PC in this case). The controller is responsible for the user interface, loading data from disk and sending output to the screen or disk. It is also responsible for the work distribution over the rest of the network. Only the controller "knows" which parts of the screen still have to be calculated and sometimes even which processor is busy and which is not.

On the workers the following processes are running (Figure 3-4):

- *forward*: communication process that takes incoming data and checks if it is meant for this processor; if so, the data is sent to the *render* process, otherwise, it is forwarded to the next processor.

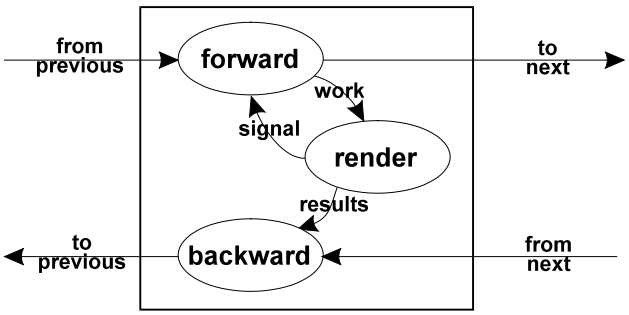


*Figure 3-**Error! Unknown switch argument.**: Worker processes*

- *render*: this is the actual renderer that calculates the value of pixels in the image. Its results are sent to the *backward* process and a signal is given in order to let the *forward* process know that the renderer is ready again.
- *backward*: takes in results both from the *renderer* and the next processor, and sends these back to its predecessor.

We have to mention here that on a transputer, two process priority levels are available. When using the above process structure, tests have shown that the communication processes need a ghigh priority, while the calculations have to be performed at low priority. This is necessary not only to avoid deadlocks, but also for efficiency reasons: if the data has to be sent before other processes can start their tasks, it is best to send it as soon as it is available.

Given this process layout, the most straightforward way of dividing the ray tracing program over a farm of processors is as follows: the controller processor loads the scene data from disk (or interactively from the user interface) and sends it to the first worker processor, which stores this data locally and propagates it to the next. This way, a copy of the whole data base is created on every processor.

Then the image is divided into sub-regions that have to be rendered each by another processor independently. For example, if we have 10 worker processors in the network, each worker could be responsible for one tenth of the resulting image. This means that every processor can start firing rays into the scene and intersecting them with all the objects, thereby calculating its own part of the image. Once finished, it can send its results through the farm back to the controller, which puts all the results together again, generating one coherent image.

The obvious advantage of this straightforward implementation is its simplicity: it is, essentially, a normal sequential ray tracing program that runs on different processors and that operates

on different pixels in the resulting image. The communication processes are the only real extras that have to be added in order to make the program parallel. This means that an existing sequential program that doesn't use too much RAM can easily be parallelised in this way.

There are two major drawbacks to this (naive) approach:

1. The whole scene database has to be small enough to fit within all worker processors' memories. This is of course a very severe limitation: the database size is limited by the processor with the smallest amount of RAM. Taking some overhead for data structures both for database and for dynamic data when ray tracing recursively, only a few thousands of polygons will fit into a "normal" processor equipped with 2 Mbytes of RAM.
2. All processors have to do the intersection tests between their respective rays and every object in the whole scene, as indicated in the introduction. Since ray-object intersections are the most time-consuming part of ray tracing, a huge amount of rendering time is wasted on unnecessary intersections. Moreover, if one processor has to perform more ray-object intersection calculations that the other processors, the overall speed will be determined by this "slow" processor (we will come back to this problem in the next Chapter). Because of this inefficiency, we have to get rid of this brute force technique of intersecting each ray with each object, by combining our parallelisation with one of the techniques for reducing the number of intersection calculations, introduced in Section 3.4.

As will become clear in the next chapters, we took two different strategies: one is voxel-based (see also [Van Reeth 92]) and the other is based on hierarchical bounding volumes (see also [Lamotte 93b]). They both cope with above-mentioned problems in their specific way.

# 4. Voxel-based Parallel Ray Tracing

In this chapter, we propose a novel approach to exploit database coherence in order to ray trace scenes that do not fit in the rendering processors' memories. We will show how Amanatides' voxel structure can be used not only for speeding up the intersection process (reducing the number of actual intersection calculations by exploiting space coherence), but also for database distribution over the processor network (by exploiting database coherence). The system is able to ray trace large scenes, i.e., scenes of which the element database cannot be copied in the memory of each rendering processor. By using appropriate buffering techniques on different levels within a dynamic load balanced scheme, an almost linear speed-up is encountered in the ray tracing process.

## 4.1. Load Balancing

In section 2.2 (on page 27), we indicated that load balancing is an important topic in the design of parallel programs. We also mentioned the naive approach of dividing the screen between the available processors (see page 36) as an example of bad load balancing. Indeed, suppose we have a scene as the one in Figure 4-1 and that there are four rendering processors in our network, then the division scheme that is formed by assigning each quadrant of the screen to one processor is not optimal at all: processor 1 will have to do all the work, while the other three do not encounter any objects. Therefore, we have to look for a better work distribution scheme.

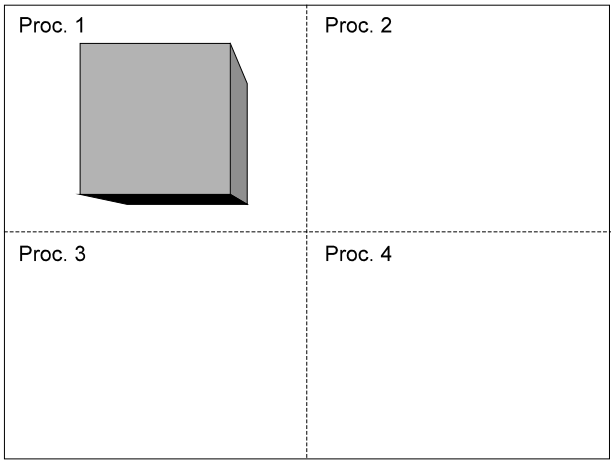


*Figure 4-**Error! Unknown switch argument.**: Bad load balance*

As a first step in overcoming this problem of keeping all the processors usefully busy, we take smaller screen parts as units for calculation: by subdividing the target image into small square

regions (called “screen patches”) of 4x4, 8x8 or 16x16 pixels, and assigning these screen patches to the different processors in an interleaved way, it is more likely that “hard to calculate” parts of the resulting image will be computed by several processors in parallel. E.g. in Figure 4-2, the smaller screen patches are divided amongst the four processors as indicated by the processor numbers in the squares. This way, all processors have to calculate some part of the actual ray-object intersections where the cube is visible, while they also have a comparable share in the empty space that is seen in the rest of the image. This way, a more equal load is obtained, even though the processors will have to compute the same amount of pixels as in the situation of Figure 4-1 (i.e. 1/4th of the whole image).

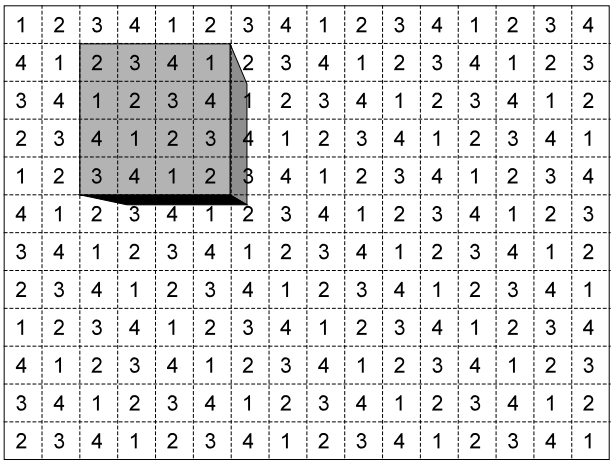


*Figure 4-**Error! Unknown switch argument.**: Interleaved work distribution*

The next step in obtaining a better load balance is to use a dynamic load balancing scheme. In this scheme, the screen regions are not distributed statically between the available processors, as shown in Figure 4-2, but in a dynamic way: the master transputer first gives each processor some work to do and then keeps track of the regions that still have to be processed, while the worker processors ask for new screen patches to be rendered after finishing a certain patch.

In order to avoid elapsed waiting time on the workers, a simple buffering technique is used: one extra region to be rendered is asked for at the start. This way, there will be new work waiting in the input buffer while the master is being informed of the fact that this particular processor will be in the need for a next screen patch.

## 4.2. Introducing the Voxel Structure and Large Databases

In our very first implementations of the parallel ray tracer, we took the approach of the copied data base on each processor. In this system, the first extension was the incorporation of a voxel grid to avoid too much ray-object intersections (see Section 3.4 on reducing the number of intersections). Naturally, this voxel grid was copied along with the database on each

worker processor in the transputer network. But as the possibilities of the ray tracer grew, the scenes to be rendered tended to grow also. Therefore, there had to be a way of storing such a database that did not fit into the memory (typically at least 2 MB of RAM) of the worker transputers.

We took recourse to the following method: the first worker processor is "sacrificed" to become a database processor and has to store the whole database (by allocating more memory to this processor, of course). The workers will obtain a smaller part of the data structures and will sometimes have to "ask" the database processor for some more data. How this is achieved, is explained in the next section.

## 4.3. Database Structure

The database stored in the database processor consists mainly of the element database (defining the scene to be ray traced) and the voxel structure. Figure 4-3 illustrates this database schematically: the *elements* data structure stores information concerning the scene to be rendered, e.g., object descriptions in terms of triangles; the *voxel refs* data structure is part of the voxel structure and stores indices to all the elements which are member of particular voxels (note that elements can be referenced from different voxels, as they can be member of different voxels); the *voxelgrid* data structure (resolution typically 20 x 20 x 20) stores index pointers to the *voxel refs* structure, so the voxel contents can be derived through these *voxel refs* (a special pointer value is used to indicate an empty voxel). The whole of the *voxelgrid* and *voxel refs* is called the *voxel structure*.

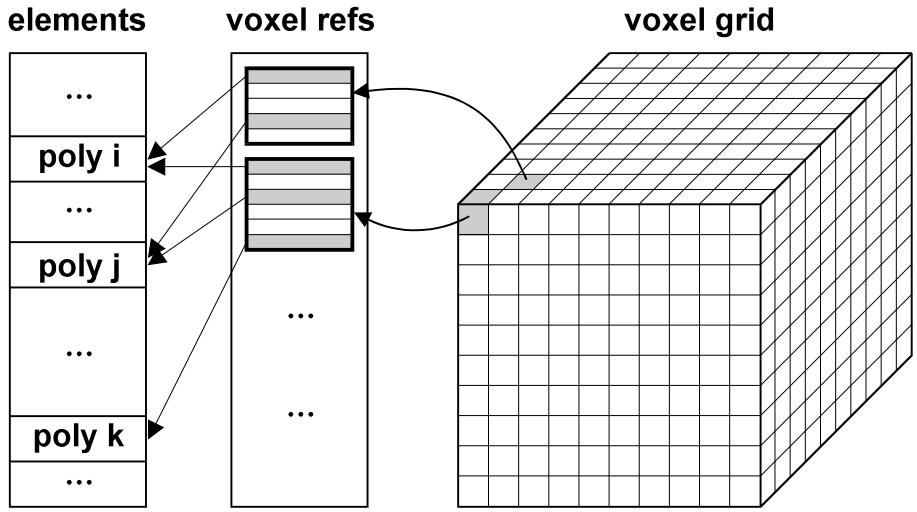


*Figure 4-**Error! Unknown switch argument.**: Schematic overview of the database structure*

One can take recourse to two approaches: (i) sending over the database on regular timeslices, giving the processors the possibility to filter out the data needed at that particular time, or (ii) transmission of data upon request by the slave processors. The former method has several disadvantages:

- potentially large amounts of data are transmitted over and over again without potentially even being used, e.g. an object that is not seen (directly, or indirectly in reflections) within the viewing angle (a situation which occurs very often in animation) will nevertheless be sent throughout the pipeline without being "picked up" by a worker;
- the type of communication implies regular synchronisation points in time, which causes much elapsed waiting time due to the inherently non-uniform work load of ray tracing.

These problems will be even more pronounced when the number of processors increases from just a few to several hundreds.

Keeping these considerations in mind, we took recourse to a client-server approach: the database processor acts as a server for requests coming from the clients (worker processors). Of course, one should beware of sending over too large chunks of data at a time, because of the communication overhead. There has to be a way of sending just those parts of the database that the worker processor's rays will most probably encounter. To achieve this, we have to use coherence in the data. The *voxel structure* is utilised to realise this "database coherence".

## 4.4. Database coherence

The data in the *elements* structure relating to a specific voxel can be treated as an elementary chunk of data to be sent over whenever needed. In order to obtain a kind of "dynamic database" on the workers, the data structure is somewhat different from the full-blown database of Figure 4-3 that is stored on the database processor. Instead, the set of elements that is contained within a voxel cell is treated as a whole and the voxel grid just contains an index into

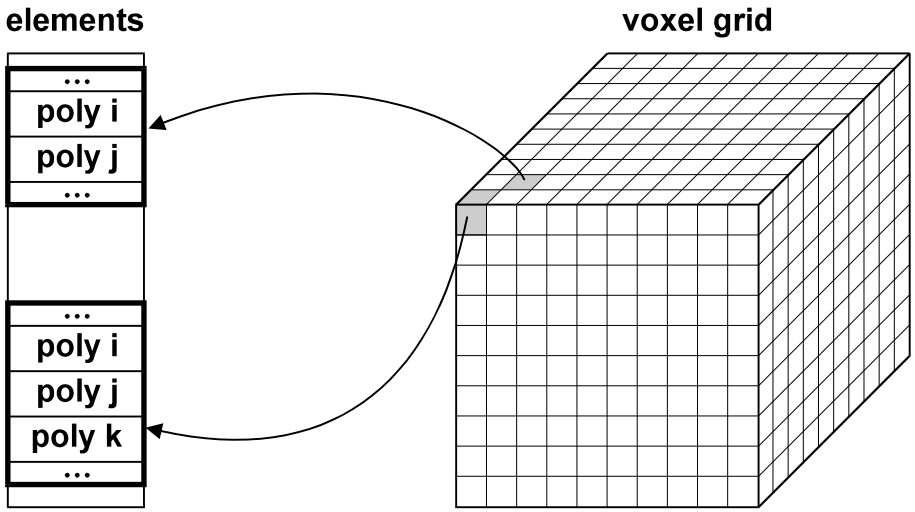


*Figure 4-**Error! Unknown switch argument.**: Voxel-based data structure on the workers*

this pool of element sets (cf. Figure 4-4). This way, we obtain the following work- and data-flow scheme in our parallel ray tracing system:

(i) the master processor sends to the slave processors new screen regions to be ray traced;

(ii) the slave processors perform the ray tracing process; i.e., the voxel grid is traversed and for each non-empty voxel that is entered, and for which no element set is stored in the *elements* buffer, the element data chunk is requested upon.

Please note that the step of *voxel refs* is omitted on the workers, just to be able to treat an element set that is associated with a voxel, as one piece of data. An implication of this approach is of course that the data of primitive elements being member of different voxels (traversed by the rays emanating from the screen region at hand) will be stored multiple times in the slaves' *element* structure. As only large-sized primitive elements appear in many different voxels, this is not such a big problem: large elements tend to be intersected by many rays, implying that no further intersection calculations (and thus potential communications)

are necessary on that level. Another justification for not having to worry too much about the copying of larger elements is the fact that only voxels traversed by rays are requested upon, so only a limited number of copies will be stored within the slaves' element structures.

In the current implementation, the *elements* buffer is initialised for each new screen region to be ray traced as the rays emanating from screen regions assigned to the slaves will most likely traverse different voxels. The *voxelgrid* data structure is still copied onto each of the workers, as it occupies only a few Kbytes of memory.

## 4.5. Evaluation

For the evaluation of the approach we explained, it is important to know how often a new set of elements has to be sent over from the database to a worker processor. An important metric we can utilise for this is the number of non-empty voxels traversed by the rays emanating from one screen patch, since this indicates how many element sets have to be requested for. Note that we utilise the number of *non-empty* voxels, as an empty voxel doesn't imply a supplementary communication. As we will show further, it turns out that, on average, the number of non-empty voxels traversed by rays emanating from a particular screen patch, is small in comparison with the total number of non-empty voxels.

To get an impression of this metric, we counted in the testing scenes in Figures 4-6 up to 4-9 for every screen patch the number of non-empty voxels traversed by its emanating rays and consequently grouped them in ranges of 10. The results are given schematically in Figure 4-5 and the exact numbers appear in Table 4-1.

The numbers indicate that for rendering the major part of the screen regions, only few non-empty voxels have to be traversed. Although these results are of course scene dependent, very similar metrics can be found for other scenes. Three reasons account for this: (i) only non-empty voxels have to be considered, (ii) the more full a voxel is, the more likely it will be that a ray visiting that voxel actually intersects with an element contained in that voxel (implying no further tracing has to be performed in the ray direction on that particular level), and (iii) in general an image of a scene tends to hold object coherence within a region (scene 1, a highly incoherent scene is an exception to this "rule", which is also reflected in the much larger numbers of non-empty voxels traversed per screen region).

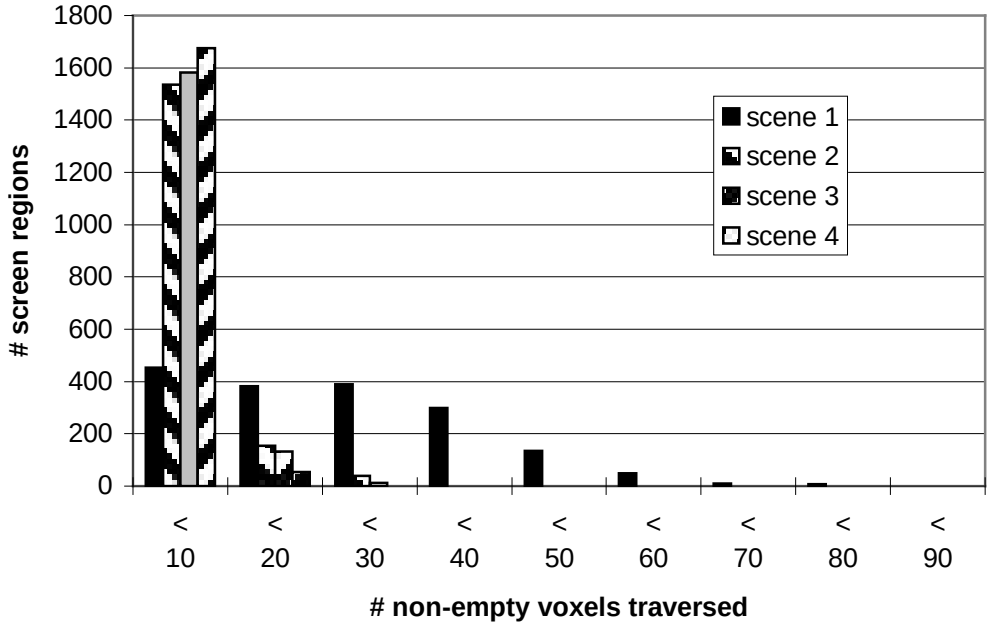


*Figure 4-**Error! Unknown switch argument.** : Number of non-empty voxels traversed per screen region in the four test scenes*

| filled voxels traversed | scene 1 | scene 2 | scene 3 | scene 4 |
|---|---|---|---|---|
| < 10 | 454 | 1535 | 1583 | 1674 |
| < 20 | 381 | 155 | 133 | 54 |
| < 30 | 390 | 38 | 12 | |
| < 40 | 300 | | | |
| < 50 | 135 | | | |
| < 60 | 50 | | | |
| < 70 | 9 | | | |
| < 80 | 8 | | | |
| < 90 | 1 | | | |

*Table 4-**Error! Unknown switch argument.**: Numerical data for Figure 4-5*

*Figure 4-**Error! Unknown switch argument.**: Scene 1: "Autumn Triangles" (cf. Colour Plate 7)*

*Figure 4-**Error! Unknown switch argument.**: Scene 2: Terrain (cf. Colour Plate 8)*

*Figure 4-**Error! Unknown switch argument.**: Scene 3: Car (cf. Colour Plate 9)*

## 4.6.Results

Scenes fitting into the 2 Mbytes of DRAM of the slave processors (up to a thousand of primitives) can be ray traced with an almost linear speed-up. For scenes not containing “difficult” primitives (e.g. bi-cubic patches), rendering timings of around one minute are typical on a 12 processor network (images in true colour, at standard PAL 768 x 576 resolution, with reflection and shadowing, 3 lights on average, tree depth typically 3).

*Figure 4-**Error! Unknown switch argument.**: Scene 4: Truck (cf. Colour Plate 10)*

Scenes not fitting in the memory of the workers take more time to render, depending upon the amount of voxel data that has to be sent over. For the testing scenes in Figures 4-6 to 4-9, the rendering timings (and other statistics) on different numbers of rendering processors are given in Table 4-2. The timings are given for scenes without shadows. These timings show, and this is important, that also in this case of a distributed database, an almost linear speed-up is still maintained.

| | scene 1 | scene 2 | scene 3 | scene 4 |
|---|---|---|---|---|
| # objects | 8000 | 3200 | 4074 | 8981 |
| # non-empty voxels | 7978 | 1211 | 2529 | 1527 |
| objects / non-empty voxel | 9.8 | 9.6 | 6.6 | 2.8 |
| rendering time 9 procs | 204 / 1088 | 158 / 448 | 203 / 467 | 332 / 879 |
| rendering time 18 procs | 107 / 546 | 77 / 225 | 100 / 221 | 159 / 412 |
| rendering time 29 procs | 78 / 341 | 52 / 154 | 78 / 341 | 101 / 284 |
| linearity 9-> 18 | 0.95 / 1.00 | 1.03 / 0.99 | 1.02 / 1.06 | 1.04 / 1.07 |
| linearity 18 -> 29 | 0.86 / 1.00 | 0.91 / 0.91 | 0.94 / 1.00 | 0.98 / 0.90 |
| linearity 9 -> 29 | 0.81 / 0.99 | 0.94 / 0.90 | 0.95 / 1.05 | 1.02 / 0.96 |

*Table 4-**Error! Unknown switch argument.**: Statistics on the different test scenes*

*Note: In this table, the numbers separated by a slash are for the image without and with anti- aliasing, respectively. The linearity factors have the following meaning:*

*linearity = 1 : perfectly linear*
*linearity < 1 : sub-linear*
*linearity > 1 : super-linear*

## 4.7. Conclusions - Further Research

In this chapter, we showed a way to solve the problem of ray tracing large scenes on our transputer-based parallel network. Parallelism is introduced by dividing the screen in m x n "screen patches" that are ray traced using a dynamic load balancing scheme. Amanatides' uniform object space subdivision technique is utilised to exploit space coherence (to cut down the number of ray-object intersection calculations), as well as a form of "database coherence". The latter form of coherence is used for reducing communication overhead while distributing the element database. Measurements show that, for many screen patches, typically few non-empty voxels are traversed by rays emanating from the regions at issue, so unnecessary communication overhead can be kept low. Timings indicate that the linear speedup of the copied database method is still maintained when using the reduced database on the workers.

The given realisation moreover allows the following optimisations to be implemented:

- pre-fetching of voxel data: this means that the contents of the voxel that the ray will enter next (if no intersection should be found in the current voxel) is requested for beforehand if it isn't already in the element buffer; this way, we can reduce the elapsed time when waiting for element sets;
- not going to the master transputer in order to fetch voxel data; indeed, it is very well possible that a worker w' (linked between the master and a particular slave processor at issue) is storing the voxel data required by a particular worker w. Hence, worker w' having stored "coincidentally" the expected data can send over the requested data itself, thus cutting short communication for all the processors between itself and the master processor;
- shadow ray optimisation: shadow rays don't have to access voxels in the order in which they appear along the ray, as only a simple "light visible / light not visible" answer is needed. Hence, data of voxels along the shadow ray path already in the memory of the processor at issue can be traced first, so potentially no further data requests are needed regarding this shadow ray;
- the ray-ID technique (mentioned in [Amanatides 87]), of storing a ray ID with every primitive element intersected with it (and thus omitting multiple intersections between a given ray and a given element), which we implemented for scenes fitting in the slaves' memories, can also be realised for scenes not fitting in the slaves' memories, at the expense

of consuming somewhat more memory; obviously, this requires some more book-keeping in the distributed-database version.

# 5.Tree-based Parallel Ray Tracing

Although the voxel-based approach from Chapter 4 was quite simple and appealing, two deficiencies attracted our attention: (1) element data could be stored multiple times in the workers' memories and (2) it is not at all known beforehand how much memory the data structures will occupy on the workers, since the memory usage depends heavily on the voxels that are traversed. After this observation, we aimed at an approach in which it was possible to overcome these problems. First of all, we needed an alternative for the voxel structure. A hierarchical tree of bounding volumes (also mentioned in Section 3.4, on the reduction of the number of ray-object intersections) turned out to be a good candidate, since the primitives are stored only once and the total size of the tree is very small and can be calculated from the number of objects in the scene.

The method we use is quite similar to the algorithm described in [Kay 86]. We will call our own approach the "hierarchical bounding volume" (HBV) method. In this chapter, we first take a look at the basic HBV algorithm (we will indicate on which points our method differs from the one from [Kay 86]), secondly explain the parallel implementations and finally elucidate how large databases are handled.

## 5.1. The HBV Sort Algorithm

In our HBV technique, the bounding box (or a set of bounding slabs) of every primitive is calculated and organised in a binary tree. The nodes in this tree have the following layout:

```
struct node
{
      void *primitive;          // pointer to the primitive
      struct node son[2];       // pointers to the left and right son
      REAL primitive_box[3][2], // bounding box of the primitive
           tree_box[3][2];      // bounding box of the subtree headed by
                                // this node
}
```

(Here is already a first main difference with Kay and Kajiya's approach: in their tree, the primitives were stored in the leaves. We store them directly in the interior nodes, for reasons that will be explained further in this section.)

The tree is built as follows: for each primitive, the bounding box is stored in a node. The nodes are then sorted using an algorithm based on the median cut principle: at every level a bounding box surrounding the set of remaining primitives at that level is created. Then a co-ordinate axis (usually the one where the surrounding box has its largest span) is selected. The set of objects is searched for a median along this axis and subsequently subdivided around this median. The surrounding bounding box is stored in the median node and subtrees are created recursively, one for the set of objects left of the median, one for those at the right. This results ultimately in a balanced binary tree in which each node contains the bounding volume of the primitive it is referencing as well as the bounding volume of the subtrees it is heading.

When a ray is presented for intersection testing with the scene, it is first tested against the tree bounding box which is stored in the root node. If there is no intersection found at this point, the ray at hand can already be discarded without any problem. In the other case a ray/box intersection test is performed on the primitive bounding box which also is stored in this node. This can eventually result in a ray-object intersection test. In case of an intersection, the intersection point is temporarily stored and will be used for comparison with future ray-object intersections.

Subsequently, the subtrees of the root node are tested for intersection. One of the intersecting trees is processed next, while the other is pushed on a stack (in [Kay 86], a heap is used). This process is repeated recursively until no more subtrees are found to be suitable and/or can be popped from the stack.

During the process (and this is not included in [Kay 86]), each time an actual ray-object intersection is found, this intersection point's distance (ray-parameter) can be used as a cut-off distance for further intersection tests: indeed, once an actual object intersection is found, this will be the largest distance at which we will have to look for a possibly closer intersection. This cut-off distance will play a role in every suitability test of bounding boxes (even for boxes that are popped off the stack). And here lies the reason for storing the primitives directly in the interior nodes: if an intersection is found at a higher level in the tree, it is more likely that subtrees will be cut also at a higher level. In the case of primitives being stored in the leaves, the search algorithm has to go all the way down the tree in order to find such a cut-off distance.

The main advantages of this HBV algorithm over space subdivision techniques alike voxels and octrees are the following:

(i) little storage overhead: in voxel and octree approaches, objects can reside in more than one cell of the grid. This means that for large objects, many references are stored in the data structure. In the HBV algorithm, only one reference is stored for every object.

(ii) no overhead from empty space: in voxel grids and octrees, empty space still have to be represented explicitly, which can generate quite some overhead on scattered databases.

(iii) the size of the tree can be calculated exactly from the number of elements in the database, because each element is referenced only once.

## 5.2. The "Ideal" Parallel System

Ideally, we would like to implement this algorithm on a parallel system where the object database and the box tree can be accessed directly from every processor (this observation was already made in general, in Section 3.6). This way, all processors would be able to do all necessary computations for one pixel, independently from all other processors: all data concerning this pixel (position, intersection point, ray tree, ...) would be stored in local memory of the processor, while all global data (object database, textures, box tree, ...) would be accessed in global, shared memory (ideally not bothered with access-collision). This approach would yield an almost perfectly linear speedup over a single-processor implementation, due to the minimal communication overhead. Unfortunately, this ideal shared memory concept still cannot be implemented for larger numbers of processors. Therefore, we have to take existing parallel hardware and look for ways to reduce the need for global storage.

## 5.3. First Implementation

As in the voxel-base approach, the most straightforward implementation again just copies the whole object database, together with the box tree on every processor. This means all workers can calculate a part of the image totally independent of each other. In other words, they perform the basic algorithm each on their set of pixels.

The host processor keeps track of which parts of the screen have not yet been calculated and sends groups of rays to processors which are ready to take some work. Actually, a buffer is

implemented to keep the processors busy as much as possible: when a packet of rays arrives at a processor, it first checks if it is idle or if its buffer is empty; in either of these cases, the ray packet is pulled into the processor at issue, otherwise it is transmitted to the next worker in the pipeline. As the host processor knows how many processors are present in the farm, the number of ray packets circulating through the pipeline can be kept optimal (i.e., the buffers can be kept full: each time a ray packet is finalised, a new packet will be sent).

The advantage is that the only communication overhead is caused by sending the ray packets and transmitting the resulting pixels back to the host processor; both are very small amounts of data and can be transmitted very quickly. This advantage comes directly from the fact that every processor has stored all necessary data locally and does not need to ask for more data elsewhere.

The main disadvantage is, again, the memory limitation: if the database and box tree don't fit into the workers' memory anymore, the system will not be able to render the scene. This way, the scene size is limited by the processor with the least amount of memory. Because this limitation is much too severe, we have to look for another approach.

## 5.4. Large Databases

### 5.4.1. In General

Before describing the modified approach for large databases, we have to stress the fact that we are working mainly with polygonal databases, since the ray tracer has to be incorporated into an existing 3D animation system (see [Van Reeth 93], [Lamotte 93a]. In this system polygons are grouped in surfaces, which in their turn are grouped in objects. In the discussion that follows, the term 'object' will refer to a collection of surfaces, while 'primitives' will stand for the lowest-level primitive objects, in this case polygons.

The main issue to be handled now is the ability to render databases that are too large to fit into each processor's memory. Some conditions have to be imposed upon the approach at hand:

(i) because of the simplicity and efficiency of the implementation with the database copied into local memory of each processor, the large database version should converge as much as possible to the previous algorithm (with the database copied on every processor);

(ii) not every processor necessarily has to be equipped with an equal amount of memory, allowing the one with the largest RAM to act as a database server.

### 5.4.2. The Solution: a Two-level Bounding Box Tree

Keeping the above conditions in mind, we took the following approach. When loading the scene into the database processor's memory, a bounding box tree is constructed at two levels: for each surface, the primitives contained within it are arranged in a box tree (a PBT, primitive box tree), using the median cut algorithm. At the same time, the bounding boxes of surfaces are put in another box tree (an SBT, surface box tree), built in exactly the same way as the lower-level (primitive) box tree. This high-level SBT is then copied on each processor, while the lower-level PBTs remain stored in the database processor's memory.

The work distribution over the processors is achieved in a dynamic way, alike our voxel implementation described in the previous chapter: the image to be rendered is subdivided into screen patches (in the implementation typically 16 x 16 or 8 x 8 pixels). Such a screen patch is sent by the host processor to a particular worker processor, which will be responsible for the full calculation of the pixels in the screen patch at hand (including secondary rays). For each pixel, a ray is fired into the scene and the algorithm described in the previous section is applied on the high-level SBT. This means that the ray traverses the SBT (with the median cut principle as explained before) until one of the surface boxes is intersected. The data concerning the surface at hand, i.e. this surface's PBT together with the primitives, is then requested from the database processor, which sends the information through the network to the requester. Here it is stored in local memory, in a priority queue of surfaces, allowing the processor to reuse the surface data for subsequent rays: each time a locally stored surface is accessed, its reference count is increased. In case of a shortage of memory for storing a new surface that does not reside in the queue yet, the database sends the needed size ahead of the actual surface data. The worker process then sorts its priority queue and releases as much local surfaces from the tail of the priority queue as needed.

The feasibility of this client-server approach to access the database is proven by the fact that for one screen patch, only a limited number of surfaces will be visited: for the image in Figure 5-1, the average number of (different) surfaces visited (including primary as well as secondary rays) was 14.22 per patch, given that the whole scene consists of 9685 faces, arranged in 90 surfaces.

The size of the surfaces at issue determines how many bytes have to be sent over on a request. Hence relatively large surfaces have to be subdivided into smaller ones in order to have a balanced communication/calculation ratio. Determining a good average size for a given communication and calculation performance is subject for future research. In this implementation, the database needs more memory than the worker processors, because it is obliged to be able to store the whole scene in local memory. The transputer network that is used for these tests contained one processor equipped with 16 Mbytes of RAM, but it would be possible to use a processor with 32 Mbytes or 64 Mbytes of RAM, or to allow this processor to access a disk, in order to store an even larger database.

## 5.5. Examples and results

For testing the performance of the proposed method, we measured the renderings of the Figures 5-1 up to 5-4. At the time of the tests, the scene of Figure 5-4 (a yacht interior - model appearing on the European Projects CD-ROM from Autodesk) did not yet fit into the memory

*Figure 5-**Error! Unknown switch argument.**: Dangerous Swimming Pool (cf. Colour Plate 11)*

*Figure 5-**Error! Unknown switch argument.**: BB (cf. Colour Plate 12)*

of the transputer network that we had at our disposal; therefore, we did the measurements with a sequential version of the algorithm on a stand-alone PC with a large swap-file, just to get an idea of the number of surfaces that had to be accessed.

The scenes were rendered using 16x16 screen patches, with one level of reflection and shadow rays.

Table 5-1 shows the scenery information, together with the average number of different surfaces used per screen patch. This last measurement shows that for this kind of scenes, no large overhead is produced by requesting the necessary surfaces from the database processor.

| Scenery Information | # faces | # surfaces | avg. # surfaces / patch |
|---|---|---|---|
| BB (Fig. 5-2) | 2048 | 11 | 3.4 |
| Geometrical objects (Fig. 5-3) | 3698 | 8 | 2.7 |
| Dangerous swimming pool (Fig. 5-1) | 9695 | 90 | 14.2 |
| Yacht interior (Fig. 5-4) | 47266 | 93 | 12.3 |

*Table 5-**Error! Unknown switch argument.**: Scenery information for the four test scenes*

For these scenes, we observed the rendering times (in seconds), depicted in Table 5-2, on a network of 24, 18 and 12 processors respectively. From these numbers, one can see a linear processor / render time ratio is maintained.

*Figure 5-**Error! Unknown switch argument.**: Some geometrical objects (cf. Colour Plate 13)*

| Scene | Rendering time (sec.) | | |
|---|---|---|---|
| | 24 procs. | 18 procs. | 12 procs. |
| BB | 454 | 596 | 921 |
| Geometrical objects | 469 | 651 | 952 |
| Dangerous swimming pool | 710 | 941 | 1458 |

*Table 5-**Error! Unknown switch argument.**: Rendering times for three test scenes*

Note: To give an idea of the transputer performance: the sequential implementation of the algorithm runs about seven times faster on a PC (486 at 66 MHz) than on a single transputer (T805 at 30 MHz).

## 5.6. Conclusions and Future Research

The tree-based approach proved to be a good alternative for the voxel method, especially in cases when memory requirements are crucial or in very non-uniform scenes where the voxel grid would be too sparse.

Regarding parallel processing, the algorithm described in this chapter can be optimised on several levels:

- Rather than referencing the central database for requesting needed surface/tree data, a processor could fetch this information from other workers in the rendering farm. Indeed,

*Figure 5-**Error! Unknown switch argument.**: Yacht interior (cf. Colour Plate 14)*

as with the voxel-based solution, it is very well possible that a worker w' (linked between the database processor and the requesting processor at issue) is storing the required data. Hence, this worker w' can send over the requested data itself, thus cutting short communication for all the processors between w' and the database processor.

- In the current implementation, each worker begins with an empty surface database. There are ways in which we could find an initial local surface database. For example, if we let processors each work in their own coherent area of the screen, instead of on smaller randomly distributed screen patches (i.e. start out with the normally "bad" load balancing scheme of Figure 4-1, we get means for exploiting database coherence: it becomes possible to find these first surfaces by checking the volume of bounding boxes of surfaces which lie in the screen region to be traced by the processor at hand. In a pre-processing stage, a processor could build a list in which the surfaces are sorted according to their degree of importance to that processor.

  Using this approach, it will become more interesting if a neighbouring processor holds a neighbouring region of the screen, and thus also a neighbouring part of the database. This way, when a worker needs additional surface/data information, there will be a larger chance of finding this information on a neighbouring processor, cutting down communication overhead.

  Obviously, in this approach of letting each processor start with 1/nth of the image, there has to be incorporated a way of letting workers that finished their own screen part, take over from others that are still working. This topic requires some more research.

# 6. Ray Tracing Patches

In the previous chapters, we have been concentrating on ray tracing polygonal databases. This kind of database is sufficient for most applications: e.g. in the fun industry, rendering speed is one of the most important factors, which can be achieved using hardware rendering of polygonal models with the straightforward shading techniques. Nevertheless, if an exact representation of a curved surface is needed, e.g. to be able to perform calculations for strength evaluations or “wind tunnel” tests, on models of cars, planes and the like, it is necessary to be as accurate in the representation as possible. Also, a higher degree of continuity is required for this kind of applications. These requirements can be met using patches.

There are some well-known problems in ray tracing parametrically defined surfaces. On the one hand, one has to deal with “hard to control” round-off errors, while on the other hand there is the performance issue. With respect to both difficulties, we will present some appropriate solutions by utilising a surface tree caching methodology implemented on our parallel network of transputers.

## 6.1. Rendering of Curved Objects

In the ray tracing algorithm, the basic rendering operation is the intersection between a ray and an object in the scene, which in fact boils down to solving the system of the ray's equation and the object's “equation(s)”. For our polygonal databases, this was a straightforward mathematical operation which could be calculated exactly (if we do not take into account that round-off errors are inherent in computer calculations).

The problem with patches, however, as with any third order function, is to calculate the intersection, since the system of equations cannot easily be solved analytically. Therefore one needs to have recourse to an appropriate approximation. But inherent to the higher degree of realism that can be gained here, there are some requirements to the accuracy of the intersection calculation. If this requirement is not met, the result will not be satisfactory. Therefore, it is absolutely necessary to calculate the intersections as closely as possible to the actual

intersection points. There are three main streams in the intersection techniques: (1) fixed subdivision, (2) adaptive subdivision and (3) numerical approximation.

### 6.1.1. Fixed Subdivision

In the most straightforward way, one can subdivide patches into a fixed number of polygons, which will then be rendered with the traditional polygonal methods. This approach involves a nasty decision: what is the criterion for subdivision depth: shall we subdivide once, twice, 10 times ? There is not one answer to this question that is suitable for all objects we would like to render, since it depends on the scene and how precise we want the approximation to be. E.g., if we zoom in on an object which is subdivided a fixed number of times, the inaccuracies of the approximation will become more and more visible. Moreover, relatively flat parts are subdivided exactly as much as very curved parts are. Another disadvantage of this polygonisation is the fact that we lose the advantages of patches (exactness, small number of control points, ease of manipulation, ...) while introducing the disadvantages of polygonal rendering (need for interpolation techniques, averaged normals, ...). This also means that for reflection- and refraction-rays, an interpolated normal has to be calculated, instead of the exact normal on the surface, that could normally be derived easily from an exact intersection point. Otherwise, we obtain results as in Colour Plate 15, where the face normal is used for each point on a face. It is obvious that there have to be better methods.

### 6.1.2. Adaptive Subdivision

Adaptive subdivision into polygons (see, e.g., [Lane 80]) is a better method, since this overcomes the depth problem of uniform subdivision: in this approach, the number of subdivisions depends on the “flatness” of a patch. This way, more subdivisions can be done for very curved patches, while rather flat parts will not be subdivided that deep. In Colour Plates 16 to 20, five successive steps in adaptive subdivision are shown. In this technique, however, another artefact is introduced: if two adjacent (sub)patches are not subdivided with an equal number of subdivisions (depending on the flatness criterion), holes can appear (see, e.g., [Clark 79]). An example of such a hole can be seen in

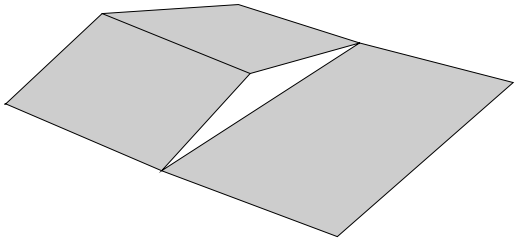

*Figure 6-**Error! Unknown switch argument.**: Problem with adaptive subdivision*

Figure 6-1: the right patch is considered to be “flat enough” while the left one still had to be subdivided once, as shown. This yields a “crack” in the surface on the edge between the two adjacent patches. Of course, the other disadvantages of polygonisation remain as in the case of fixed subdivision. Therefore, we have to get rid of these polygonal methods and consider algorithms that search for the exact intersection point on the surface without approximation by an intermediate representation.

### 6.1.3. Numerical Approximation

Another adaptive method is numerically approximating (by Newton iteration) the actual curved surface of an object ([Blinn 78], [Whitted 78], [Toth 85]). There are two main groups of Newton-variants, depending on the convergence characteristics. Quadratic convergence requires a matrix inversion for every step in the iteration. Linear convergence just requires 1 matrix inversion at the beginning of the iteration but in order to be able to use this one matrix for the whole iteration, the starting value has to be chosen close enough to the actual intersection point (see [Toth 85]).

## 6.2. Toth's Algorithm

As Toth’s algorithm is fundamental in our method, we will give a brief overview of its main principles here. For proofs and more information, the reader is referred to [Toth 85].

### 6.2.1. Newton Iteration

Suppose we have a surface defined by $s(u,v) = (x(u,v), y(u,v), z(u,v))$ and a ray defined by $r(t) = S + tD$ (where $S$ and $D$ are the ray’s starting point and direction vector, respectively), then an intersection occurs if the following system of equations has a solution:

$$s(u,v) - r(t) = 0 \qquad \textit{(Eq. 6-1)}$$

If we use the notation $f(u,v,t) = s(u,v) - r(t)$ and $x = (u,v,t)$, we obtain the equivalent vector form:

$$f(x) = 0 \qquad \textit{(Eq. 6-2)}$$

This type of non-linear system can be solved by three-dimensional multivariate Newton iteration. If Y is a 3x3 non-singular matrix, the Newton scheme can be expressed as:

$$x_{k+1} = x_k - Yf(x_k) \qquad \textit{(Eq. 6-3)}$$

The vector $-Yf(x_k)$ is called the "Newton step". If $Y$ is the inverse Jacobian of $f$ at $x_k$, then the result of a Newton step ($x_{k+1}$) is the intersection point (in parameter space) between the ray and the tangent plane to the surface at the point $x_k$.

There are two ways of using this Newton iteration scheme. In ordinary Newton iteration, this matrix $Y$ is updated at each Newton step (by calculating the inverse Jacobian in the newly found approximation $x_{k+1}$), which yields a quadratic convergence to the solution. If $Y$ is calculated only once and from then on kept fixed, Eq. 6-3 yields "simple" Newton iteration, which converges only linearly (while only one matrix inversion is needed for the whole iteration).

Newton iteration only converges if it is started with a good initial "guess" for $x_1$. Toth uses the so-called Krawczyck operator (K-operator for short) for this purpose.

### 6.2.2. The Krawczyck operator

Suppose we have a continuously differentiable vector valued function $f$ on the open subset $D$ of the n-dimensional Euclidean space (in our case, this is the three-dimensional parameter space of $(u,v,t)$-values) and that the interval vector (box) $X$ is a subset of $D$. Then for any real vector y in $X$ and a non-singular matrix $Y$, an operator $K(X,y,Y)$ can be defined as follows:

$$K(X,y,Y) = y - Yf(y) + (I-YF'(X))(X-y) \qquad \textit{(Eq. 6-4)}$$

(Note : details on the K-operator - e.g. the meaning of the $F$ function - can be found in [Toth 85]).

The interesting thing about this K-operator is that it yields a region within which the Newton iteration steps will remain: for each $x$ in $X$, the result of one step $x-Yf(x)$ will be in $K(X,y,Y)$. It can even be shown that all solutions in $X$ to Eq. 6-2 are in $K(X,y,Y)$. As a consequence, if $K(X,y,Y) \cap X = \emptyset$, then there are no solutions in $X$ to the system in Eq. 6-2.

In his theorems 3.2 and 3.5, Toth identifies two cases in which a region $X$ can be called "safe" for Newton iteration: if $X$ satisfies one of these two criteria, simple Newton iteration is guaranteed to converge to a unique solution for any starting point in $X$.

### 6.2.3. Searching Intersections

Using these observations, the general flow of Toth's algorithm is as follows: given a candidate surface, the K-operator of the associated region *X* (which is the bounding box of the surface, in parameter space) is computed and Toth's criteria for "safe" regions are tested. If one of these two criteria are satisfied, simple Newton iteration is used to obtain the solution. Otherwise, *K(X,y,Y)* Ç *X* is calculated and used as new, refined region. A region is excluded from further searching in several cases, amongst which: (i) the ray-extent intersection is empty or (ii) a solution has already been found and the nearest point of the extent is further away from the ray origin than this intersection point.

By applying this scheme on each of the surfaces that are candidates for intersection, the closest intersection point is found.

### 6.2.4. Problems

When implementing and using Toth's algorithm, some difficulties arose. Firstly, a number of tolerances were needed to control the algorithm, e.g., a flatness criterion, a smallness criterion, a tolerance to decide whether a point is inside a region, etc.[1] The problem was that these tolerances actually should be coupled with every object that was to be rendered, because a set of values that was well-defined for one object could yield nasty holes or impurities in other objects. On the other hand, when the tolerances are chosen too strict, the algorithm will be very slow, since too much time is wasted on unnecessary calculations.

Another problem is indicated by Lischinski and Gonczarowski in [Lischinski 90]: Toth continues his recursive search algorithm with the intersection between *X* and its K-operator (which are defined in parameter space, not in object space), subdivided into two subregions. This is not a good practice when one wants to incorporate ray coherence in Toth's algorithm, as will be explained in the next section, where we summarise the modified algorithm by Lischinski and Gonczarowski (L&G for short).

[1]Even Toth mentioned in his Concluding Remarks section that he "found certain tolerance values useful", and adds that the trade-off between these tolerances is a very hard problem: if e.g. a region is allowed to become very small, round-off problems will be introduced.

## 6.3. Tree Caching According to Lischinski and Gonczarowski

L&G observed that the subdivision used in the recursive steps of Toth's algorithm prevented them from using ray coherence: Toth continues with $X \cap K$ subdivided into two subregions, but since K depends on the ray direction, different rays will yield different K-operators. As a consequence, if two rays try to intersect the same surface, the search tree constructed by the recursive steps will be totally different, already from the first recursion level. The essence of ray coherence is to reuse (at least part of) the results of complex computations from one ray to another, which is not possible in this case. Therefore, L&G propose to use regular subdivision into four subpatches which will be tested separately against Toth's safeness criteria. This way, the search trees are regular quadtrees that follow the same subdivision scheme for all rays that try to intersect a certain patch. This approach allows one to store such a subdivision tree (called "surface tree") and use it again for the next ray.

The L&G algorithm distinguishes two different cases. If a patch is searched for an intersection for the first time, a quadtree is constructed. Afterwards, the surface tree is cached. When another ray has to be tested against the same surface, it is first intersected with the cached tree. Only those leaves whose bounding boxes are intersected by the ray need to be considered for further testing. At this point, it is important to stress the fact that L&G construct a tree cache for a particular patch only once and that this cached tree remains unchanged ever after, even if it is used again for another ray.

Of course, one cannot create an unlimited number of trees. It is not realistic that a cache can be used where all such surface trees can be stored, so a method of releasing occupied cache space is necessary. L&G use a LRU (least recently used) algorithm, similar to algorithms used in virtual memory systems, where memory pages have to be released in order to use them for another chunk of data. L&G chose to construct a queue containing the roots of all cached trees, that is rearranged according to the usage of the trees. Every time a tree is used in the searching scheme, it is moved to the end of the queue in order to ensure that the least recently used tree is always the first element in the queue. When the available space is totally full, an entire tree is released from the head of the queue and the freed space can be used for another tree.

Another part of their method deals with better sampling orders. In traditional ray tracing systems, there are no special requirements for the sampling order, since every pixel can be

calculated independently from all other pixels. But when coherence is wanted, it is a better practice to use an appropriate sampling order. If one wants to exploit coherence as much as possible, one has to be sure that the reuse of cached trees is encouraged by the algorithm. L&G investigated two different sampling orders in combination with their tree caching scheme. Firstly, they used an item buffer (introduced in [Weghorst 84]) to indicate which objects are associated with a pixel, so the ray tracer can keep working on the same object (and thus use the cached tree) as long as possible. The item buffer is constructed at a low resolution (e.g., 64 x 64) and one surface is kept in every buffer entry. This way, it is possible to determine which surfaces are visible through a certain part of the screen. On this basis, a better sampling order than scanline order can be deduced.

A second, more effective, order used by L&G is the Peano curve order (see, e.g., [Witten 82]). Following the path of this curve, we see that it divides a square region into four quadrants. Each quadrant is fully visited before the curve goes on to the next quadrant. The result of using this order for sampling the image screen, successively sampled pixels stick together much more than in scanline order.

The last part of the L&G theory concerns a better treatment of secondary rays. By making some simple observations about the algorithm, one can easily speed up the evaluation of reflected, refracted and shadow rays. For more details on this matter, the interested reader is referred to [Lischinski 90].

In the rest of this chapter we will state some of our findings from implementing a tree caching algorithm within our parallel ray tracing system. The parallel nature of the system forced us to use a different approach on some levels of the algorithm, than the one described before. We also wanted to "fill the gaps" left open in the conclusions section of [Lischinski 90]: further research could be done on the depth control and on more elaborate schemes for releasing cached surface trees.

When rendering patches, we wanted to use the same topology and a work distribution method alike the ones described in the previous chapters. Naturally, this has some implications on the tree caching algorithm.

## 6.4. Implications of the Parallel Ray Tracer for the Caching Algorithm

The first implication is related to the distributed memory. In the whole discussion of the surface tree caching scheme, the implicit assumption was made that there is just one processor that has to render the whole screen and that has to manage the cache memory. In our system, however, there are many processors working together on one screen and managing their own tree cache (since each transputer has its own RAM and it is impossible as yet to share memory between processors). Hence, if one processor has to intersect a patch for which it does not have a tree cached yet, while another processor does have a tree cached for this particular patch, a possible solution would be to ask the latter processor to send his tree to the former, so this one does not have to calculate these subdivisions itself. If we adopted this way of sharing trees between processors, the communication needed (1) to ask other processors if they have a tree cached for a certain patch and (2) to send back this tree (if there is one), would have some severe implications. Firstly, the processor with the cached tree has to be interrupted in its own work in order to send the tree to the one that is waiting for it. Secondly, the size of trees implies that the time needed to send such a tree from one processor to another will produce a significant overhead. And finally, there is the problem of reachability: the only way of sharing data between two transputers is sending this data via one of the four communication channels. But one particular processor cannot reach any other processor in the network, since it is only connected to two adjacent processors (recall the farm topology of Figure 3-2). Therefore, if e.g., processor 1 wants to send data to processor 6, all processors on the path between these two processors (i.e. processors 2-5) have to be disturbed, too: on each of these intervening processors, there has to be a process that simply takes in data and sends it through to the next processor. Of course, such a process will use some processor time that could be used for other tasks. Note that this reachability problem is not inherent in the farm topology: up until now, transputers have only four physical communication links, so it will be a problem in any topology to reach an arbitrary processor from anywhere in the network. To summarise, the communication overhead implied by the sharing of trees between processors cannot be efficient enough.

Another consequence of our system is related to the dynamic load balancing scheme, that we discussed in the two previous chapters: within this scheme, each processor will only “see” a relatively small number of objects in the scene. After all, it just works within a small screen patch. So, when starting with a new screen square, a limited number of objects will have to be

intersected by this processor. Hence, the number of trees cached at any moment will also be relatively small, so it would not be a good practice to limit the tree depth to a given maximum. Otherwise, the profit gained from tree caching would be too small. On the other hand, most of the time it is not desirable to delete whole trees from the cache, but just to prune some "old" branches.

Finally, the sampling order cannot be exploited anymore, or at least not to the same extent as in the research from L&G, since we like the work distribution to be non-deterministic in a parallel system, in order to retain a good load balance.

## 6.5. The New Surface Tree Caching Algorithm

The general idea behind our algorithm is to keep in memory as much patch subdivision data as possible. During the rendering, processors will run out of memory, so "old" cached patches will have to be removed, in order to free space. Of course, some method is needed to determine which data can be deleted first.

We did not want to use the drastic method of L&G's LRU algorithm that is being applied to entire trees. The reason for this is very straightforward: because our transputers do not have a large amount of memory available for the tree cache, an overflow will occur frequently. Suppose the previously traced ray found an intersection on a reflective patch A. Then the next ray to be traced will be a reflection ray. Most probably, this newly spawned ray will have a very different direction than the previous one, hence it will most likely hit other patches than patch A. In case of shortage of cache memory, the tree of patch A will be deleted in its entirety. For the next pixel, however, a ray will be shot again in the direction of patch A, for which no tree is cached anymore, so subdivision starts again from scratch. Suppose the nearest intersection point is again located on patch A (which is very likely to happen), then a new reflection ray is fired, which will most likely hit the same patch as the previous reflection ray also, etc. This way, the cache is constantly emptied and filled with always the same two trees. If just branches are cut away from the leaf-side of the trees, at least part of the previous trees can remain in memory.

Another thing that we wanted to do differently was the construction of the cached trees. In the discussion of L&G's method, we stressed the fact that cached trees remain static in their

approach: once a tree is constructed, it is never changed again, unless the tree is deleted in its entirety. In our opinion, this is too restrictive: e.g. let's assume that the first ray that visits a certain patch does not yield an intersection, but that one subdivision was needed to decide there was no intersection. If the tree that is cached for this patch is traversed for ten other rays, each time only one subdivision level can be reused, yielding not much of a speedup. If, on the other hand, the tree cache can be updated and expanded, new branches can be added and old ones can be cut away. This dynamic behaviour is better suited for reusing the cached data.

How do we accomplish this branch-based LRU algorithm within our system ? The basic idea is that pruning can be applied on virtually any tree node which has only leaves as children (i.e. the end point of a branch): at leaves recursion stopped because either an intersection had been found or because they were rejected by Toth's safeness criteria. In the former case there is quite a chance that this branch has been visited recently, while in the latter, it may be possible that this branch will not be reached again.

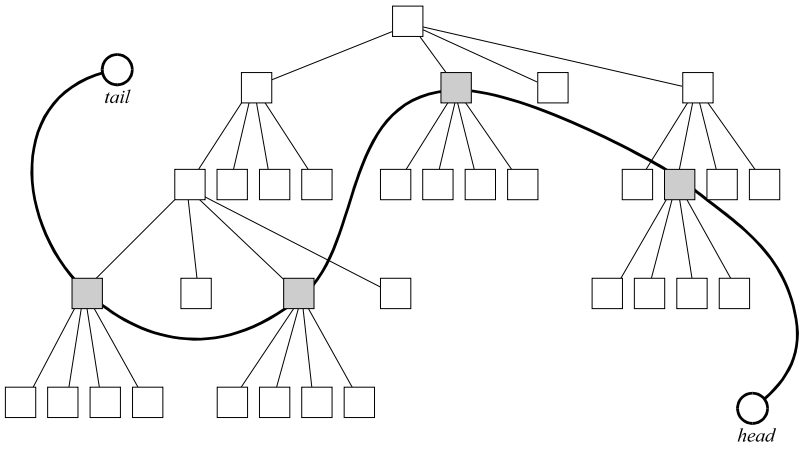


*Figure 6-**Error! Unknown switch argument.**: A cached tree (with candidate list nodes in grey)*

These candidate nodes (nodes which have 4 leaf children) are organised in a sorted list which is maintained throughout the whole tree cache (across different surface trees). E.g., in Figure 6-2, the grey nodes are members of the list. We will show how this list is used to free cache space.

When a patch is presented for intersection testing, the following steps are followed:

```
if <no tree is cached yet for this patch> then
      <create root for new tree> (which will be a leaf)
<take cached root>
```

```
current := FirstValidLeaf (root)
while (current <> NIL) do
      <check Toth's criteria on current node>
      case
            <criteria o.k.>
                  <perform Newton iteration>
            <parameter space too small or patch surface too flat>
                  <perform 1 Newton step>
            otherwise
                  <subdivide leaf's patch into four subpatches>
                  <leaf becomes node with the 4 subpatches as new leaves>
      current := NextValidLeaf (current)
```

Essential to the algorithm are the functions FirstValidLeaf and NextValidLeaf. These functions have several combined tasks. They perform a modified depth-first search to find the first (next) valid leaf. This is a leaf whose bounding box is intersected by the ray and whose value of $t_{min}$ (this is the value of the ray parameter $t$ at the point where the ray enters the bounding box of the patch control points) is smaller than the $t$-value of the closest intersection found so far. Suppose that our cache mechanism already contains a tree as depicted in Figure 6-2. When FirstValidLeaf is called, the algorithm starts the depth-first search with Figure 6-3 as result.

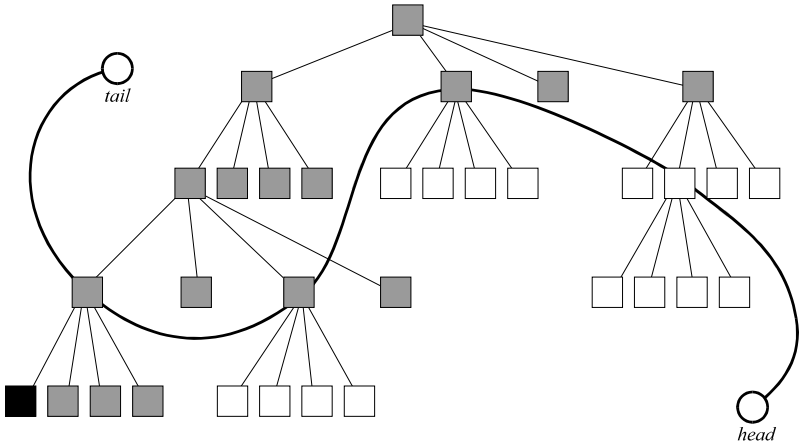


*Figure 6-**Error! Unknown switch argument.**: The result of a call to FirstValidLeaf with the cached tree of Figure 6-2*

At each node, the bounding boxes of the four children are intersected; then the nodes of the children are rearranged so that they are sorted from left to right on their values for $t_{min}$. In the figure, all grey nodes have been sorted. Once a valid leaf has been reached, the function returns a pointer to this leaf (the black node in the tree). If no such leaf can be found after scanning the whole tree, the return value is NIL.

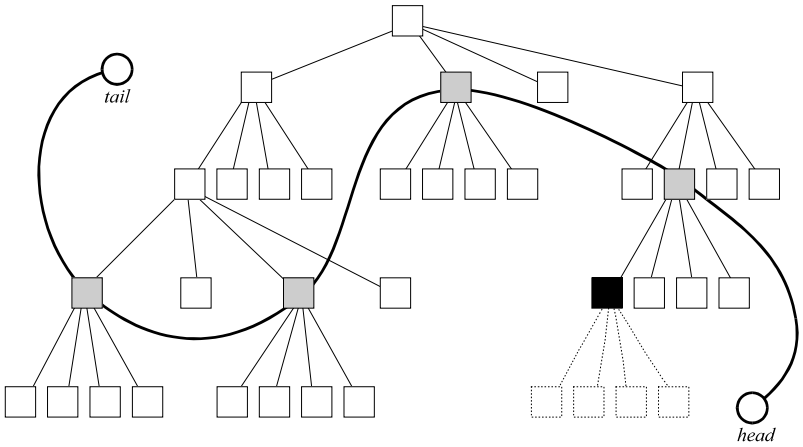


*Figure 6-**Error! Unknown switch argument.**: Children (leaves) have to be added to the current node (black)*

Of course, the global list structure will have to be modified each time new leaves are added to a tree. In Figure 6-4, we have a situation in which new leaves have to be added to the current node (the black one). Since the current node's father is a member of the list, this father has to be removed from the list. Then the black node is inserted at the head of the list, which results in Figure 6-5 (with a new current node in black).

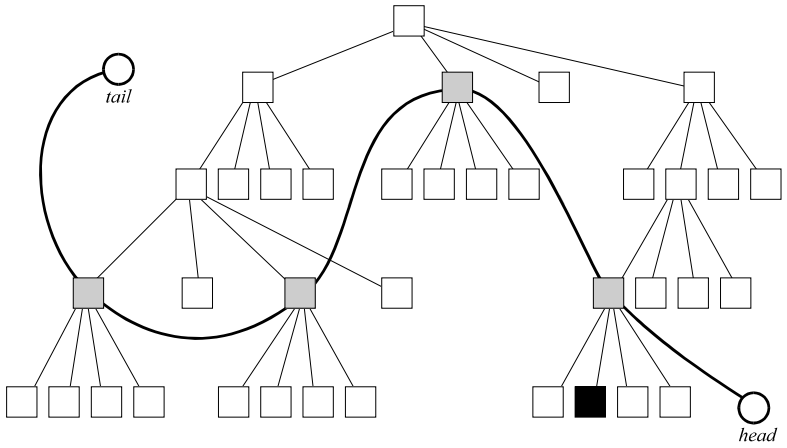


*Figure 6-**Error! Unknown switch argument.**: Children are inserted, the candidate list is updated and a new current node is selected*

Now we are able to see how memory can be freed, using this list. As stated before, we do not use a method as drastic as the one of L&G: instead of deleting entire trees, we prune a set of leaves in a tree in order to free the space occupied by these leaves. We implemented the LRU algorithm such that the nodes that are least recently used will be at the tail of the list. In the situation of Figure 6-6, the node at the tail of the list is the first candidate to release its children for usage elsewhere in the tree cache. When at a certain moment in the tree-building phase some shortage of space occurs, the leaves of the tail node will be pruned away such that

this space will be available again; the tail node becomes a leaf itself. This situation is depicted in Figure 6-7.

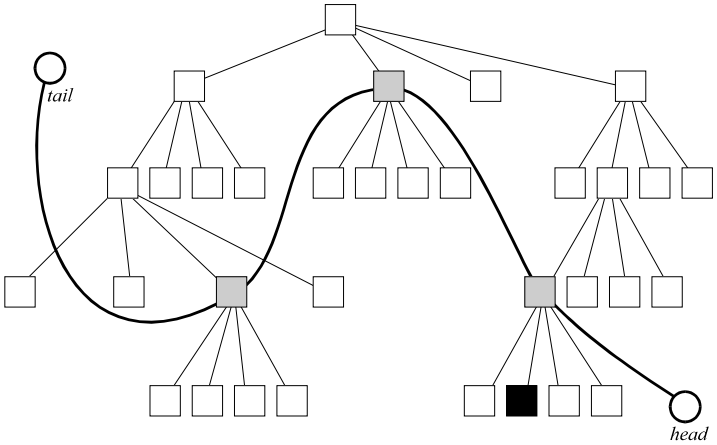


*Figure 6-**Error! Unknown switch argument.**: The node at the tail of the list is the first candidate to release its children*

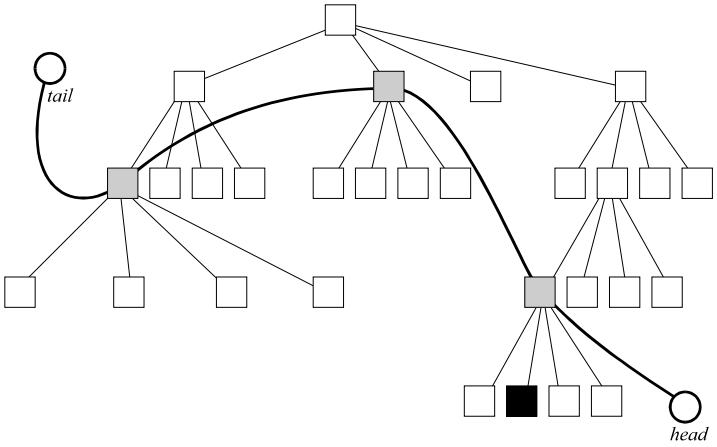


*Figure 6-**Error! Unknown switch argument.**: The result of pruning the children from the tail node*

## 6.6. Performance

In order to get an impression of the relative performance of this new tree caching method, in comparison to previous algorithms, we rendered the classic Utah teapot (cf. Colour Plate 21) using the tree caching algorithm (with the use of Toth's K-operator technique), as well as with Toth's algorithm. In this performance test, the scene contained a reflective teapot (reflection depth 2) and two lights at infinite distance. The video resolution of 768 x 576 pixels was used. The tree cache size was approximately 832 Kbytes (the free memory in the test pro-

gram). In a pre-processing step, all patches are subdivided into four sub-patches. Some tests indicated that this subdivision yielded an increase in speed of roughly 10%, at the cost of using more memory for the object database. This speedup turned out to be a consequence of the fact that the four sub-patches become less curved than the original one. This simple observation was already made after having implemented Toth's algorithm.

Figures 6-8 to 6-10 show the performance of the parallel tree caching algorithm, in comparison with the implementation of Toth's algorithm. At the same time we look at the linearity of both algorithms: we rendered the same teapot scene on a range from 2 up to 23 processors.

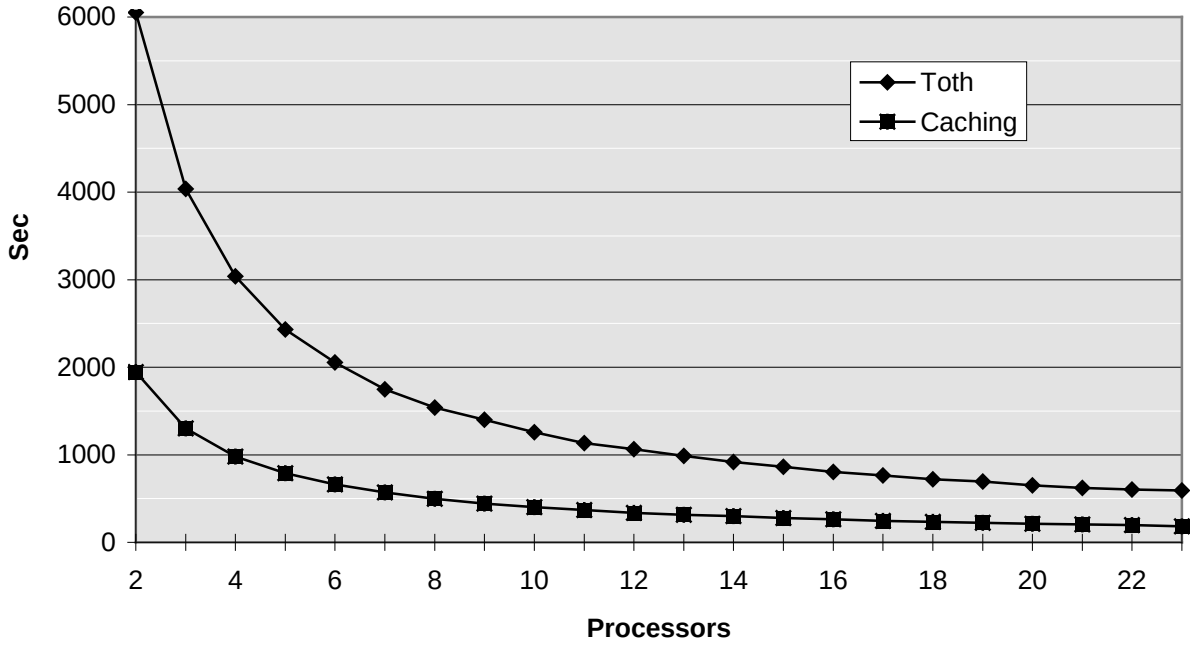


*Figure 6-**Error! Unknown switch argument.**: Rendering times for the teapot scene*

Figure 6-8 shows the time needed to render the teapot scene using each of the two intersection algorithms. It turns out that Toth's method is a lot slower than tree caching: e.g., Toth needs 6 processors to become as fast as tree caching running on just 2 processors (i.e., ± 2000 sec.).

In Figure 6-9, the speed of the two methods is depicted, expressed in terms of pixels / second / processor. It shows that the Toth implementation renders about 35 pixels per processor per second while the tree caching method yields about 108 pixels. This gives a rough impression of the speedup factor gained by this new tree caching method over Toth's algorithm. The exact speedup factor is shown in Figure 6-10: it turns out that tree caching is at least three times faster than Toth.

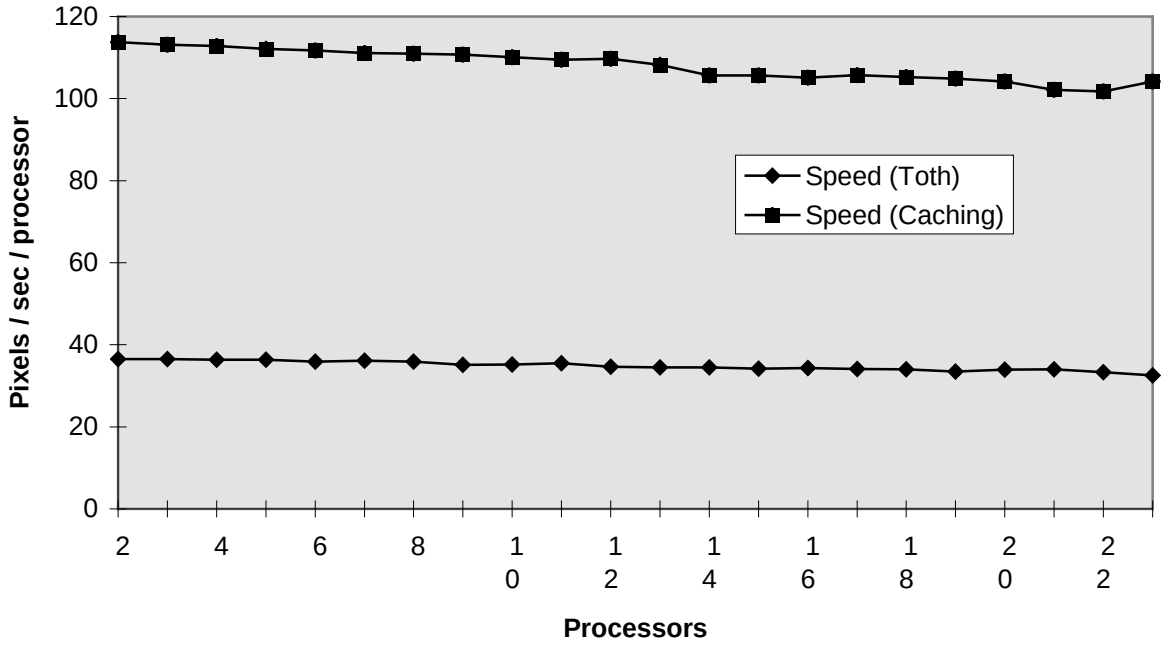


*Figure 6-**Error! Unknown switch argument.**: Speed of both algorithms*

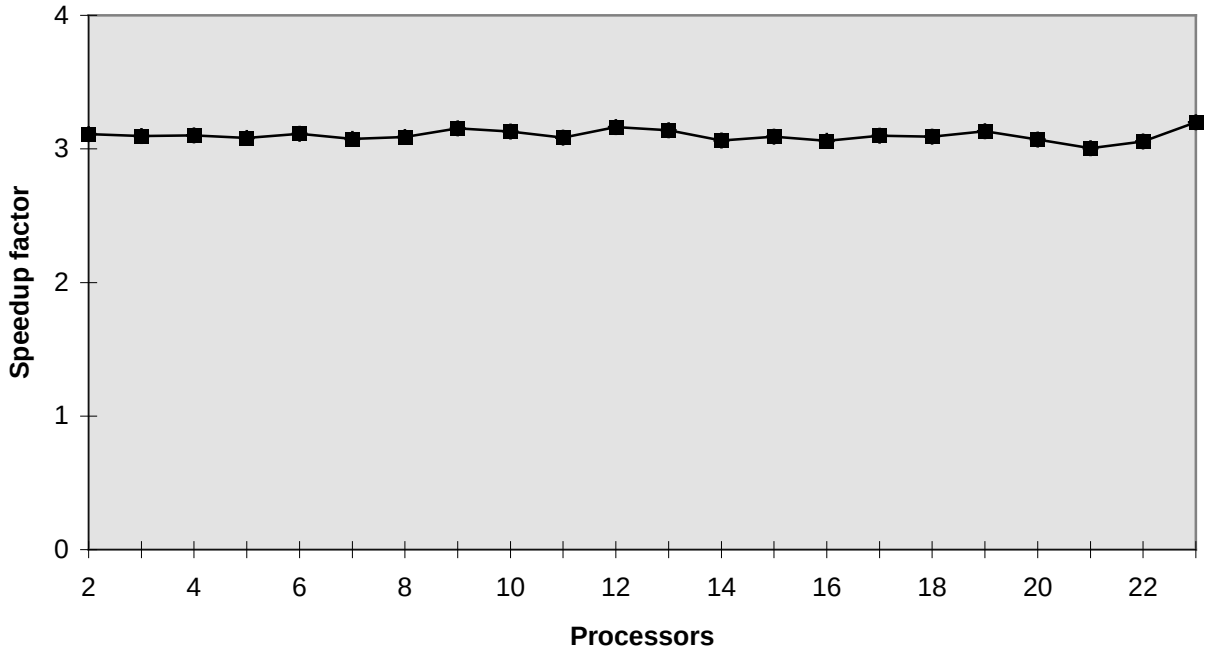


*Figure 6-**Error! Unknown switch argument.**: Speedup factor Caching / Toth*

As a more concrete example of the results, some pictures follow.

Figure 6-11 (Colour Plate 22) shows a scene, consisting of a glass bottle (with 44 patches defining the double-sided surface) lying on top of a grid of reflecting cylinders and a procedural water-textured plane. This scene was rendered with reflection depth 7; it took 2312 seconds on 13 processors (resolution 768 x 576 - not anti-aliased).

For the scene in Figure 6-12 (Colour Plate 23), we were inspired by a picture by Achim Strösser of the University of Karlsruhe in Germany (appearing in [Magnenat-Thalmann 90]). In this picture the trace depth is set to 7. The cocktail glass is also double-sided and consists of 20 patches. This anti-aliased (786 x 576) picture took 23212 seconds (nearly 6.5 hours) on 13 processors.

*Figure 6-**Error! Unknown switch argument.**: Bottle on sea + grid (cf. Colour Plate 22)*

Two other examples (of which we only know the number of processors and the rendering time) are shown in Colour Plates 24 an 25. Plate 24 shows our teapot on top of a sloping surface, also made out of patches, and a curved patch-mirror. This image took 5165 seconds (± 1 hour and 26 minutes) to render on 13 processors. Plate 25 depicts the teapot on the same surface, with two intersecting curved mirrors, totally filling the image space. This one took 4322 seconds (± one hour and 12 minutes) using 23 processors.

*Figure 6-**Error! Unknown switch argument.**: Bottle with cocktail glass (cf. Colour Plate 23)*

## 6.7. Conclusions, current and future research

In this chapter, we investigated the usefulness of previously developed algorithms for rendering curved surfaces in the framework of the parallel ray tracing system. The goal was to exploit coherence between rays, to retain a good load balance between the worker processors and to use the tree cache as efficiently as possible. As a result, a tree caching scheme is developed, similar to that from Lischinski and Gonczarowski, and tuned to the use within the parallel system.

We indicated a fundamental difference with the algorithm of Lischinski and Gonczarowski: in their approach, a cache tree is only constructed for those patches that do not have a cache yet. Once this cached tree is built, it remains unchanged and is just used to be able to start Toth's scheme from a smaller patch. Within their algorithm, Toth's method serves two distinct goals: (1) to assist in the building of a tree cache and (2) to perform patch/ray intersection calculations, starting from a leaf in the cached tree.

In our algorithm however, caching is performed continuously. a patch's tree cache will be updated every time the patch is tested for ray intersection; new nodes may be inserted into the cache tree as soon as the K-operator focuses on previously unvisited sub-patch areas. This way, the tree cache is updated more dynamically than that of Lischinski and Gonczarowski.

At the same time we covered the tree depth control and better methods for releasing cached trees, mentioned as research topics in the conclusions section of [Lischinski 90].

Another topic worth investigating is the influence of the work distribution on the ray coherence, while the load balance has to be retained. E.g., one could provide the work distributor with some more intelligence. In the approach described here, there is no explicit control over which parts will be rendered by which processor; as a consequence, ray coherence cannot be exploited fully. If the work distributor knew in which part of the screen a processor is working, it would be able to let this processor stick to the current screen area as closely as possible.

Another aspect to be considered is the refinement of a father node's $t_{min}$ value, when its children are pruned. Since the father node's bounding box encloses the children's boxes, the father's original $t_{min}$ value is never greater than those of the children. As a consequence, the smallest $t_{min}$ value amongst the children nodes will never be worse than the father's value (most of the time even better). So, it would be a better practice to store this smallest $t_{min}$ in the father node, instead of its original value.

# 7. Parallel Radiosity

In the previous chapters, we have been focusing on the specular part of global illumination. In this chapter, we will concentrate on global Lambertian diffuse reflections. In the graphics literature, quite some research is devoted to this subject. Two broad categories of approaches to solving the illumination problem in diffusely reflecting environments can be recognised. The first one is based on extensions to the traditional ray tracing algorithms ([Cook 86], [Kajiya 86]). The second category incorporates radiosity algorithms ([Goral 84], [Cohen 86], [Cohen 91]). Radiosity is a method based on the theory of heat transfer between surfaces ([Siegel 84]). One of the major advantages of the radiosity approach is that it results in a solution independent from the position of the viewer. This is particularly interesting for fly-through and near real-time camera animation, e.g. in architectural applications.

In this chapter, we first dig a little deeper in the radiosity theory. In the second section, we show what part of radiosity will be parallelised. This parallelisation is further explained in section three and results are presented in section four. We finish this chapter with some conclusions and directions for further research.

## 7.1. Radiosity Theory

### 7.1.1. Background

In radiosity, the surfaces in the environment are assumed to be Lambertian (perfect) diffusers, reflectors or emitters: on these surfaces, light is reflected in all directions with equal intensity. To facilitate the formulation for the system of equations (mentioned in section 1.3.4.2, the introduction to radiosity) that describes the radiosities on the surfaces, the scene is divided into rectangular areas, in the traditional (Cornell) literature called “patches”[2](or “surface patches”). The radiosity over a patch is constant, hence the accuracy of the solution depends on this discretisation of the scene.

[2]This is already the third meaning that is given to the word “patch” !

The radiosity of a patch stands for the total rate of energy that leaves a surface. Therefore, it is equal to the sum of the energy that the surface itself emits and the energy, coming from other surfaces, that is reflected by the surface. The radiosity of a single patch can be calculated, based on a geometric factor, called the "form factor", which indicates the percentage of energy that is transferred between two patches.

The relation of energy leaving each surface element and arriving on each surface element can then be expressed as follows (i.e. the general formulation of Eq. 1-12):

$$B_{dAi}\, dA_i = E_{dAi}\, dA_i + \rho_{dAi} \int_j B_{dAj}\, F_{dAj\text{-}dAi}\, dA_j \qquad \textit{(Eq. 7-1)}$$

(which should be read as: radiosity x area = emitted energy + reflected energy)

where

- $d_{Ai}$ is a differential surface area,
- $B_{dAi}$ is the radiosity (energy per unit area) of differential area $d_{Ai}$,
- $E_{dAi}$ is the energy emission of differential area $d_{Ai}$ (i.e. only non-zero for a "light source"),
- $\rho_{dAi}$ is the reflectivity of differential area $d_{Ai}$ (a value between 0 and 1 indicating which fraction of the arriving light is reflected), and
- $F_{dAj\text{-}dAi}$ is the form factor from $d_{Aj}$ to $d_{Ai}$ (a value between 0 and 1 indicating which fraction of energy leaving $d_{Aj}$ arrives at $d_{Ai}$).

In a closed environment, an equilibrium is reached and a set of linear equations is formed by repeating Eq. 7-1 for each differential area in the scene.

As the number of differential areas is infinite, the number of radiosities to be calculated is also infinite. Hence, a discrete version of the above integral equation is used, by taking the discretised version of the scene, resulting in the following equation (using a simplified notation):

$$B_i\, A_i = E_i\, A_i + \rho_i \sum_j B_j\, F_{ji}\, A_j \qquad \textit{(Eq. 7-2)}$$

with the assumption that $B_i$ and $E_i$ do not vary across the surface of a patch.

In [Siegle 84] it is explained that the energy interchange between patches only depends on their relative geometry. Therefore there is a reciprocity relationship $F_{ij}Ai = F_{ji}A_j$, which yields:

$$B_i = E_i + \rho_i \sum_j B_j F_{ij} \qquad \textit{(Eq. 7-3)}$$

Such an equation exists for every patch. In a closed environment, a set of $n$ simultaneous equations in $n$ unknown values ($B_i$) is obtained:

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_{21} F_{22} & \cdots & -\rho_2 F_{2n} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_n \end{bmatrix} \qquad \textit{(Eq. 7-4)}$$

### 7.1.2. Solving the Radiosity Matrix

Some observations can be made about this equation. Most of the $E_i$ values will be zero, since only a limited number of objects will be defined as being light emitters. Furthermore, for planes or convex shapes $F_{ii}$ will be zero, since none of the emitted energy will directly return to its source. Also, since a form factor $F_{ij}$ is defined as the fraction of the energy leaving patch $i$ that arrives at patch $j$, the sum of the form factors in one row must be unity (in a closed environment); since $\rho_i$ is always less than one, the matrix in Eq. 7-4 is diagonally dominant. Under this condition, the Gauss-Seidel method guarantees to converge to a solution to this type of equation system (see [Cohen 85]).When the equations are solved, we obtain a solution $B_i$ for every patch. These values can be used to calculate the colours at the patch vertices, which can be fed to a standard Gouraud renderer for displaying.

Since the solution is view-independent, several views or even a walk-through can be generated without solving the system of equations again. Note that since the form factors only depend on the geometry of the patches, the colours and reflectivity of patches and the colours and intensities of the light sources can still be altered without having to recalculate the form factors (but of course requiring a new solution for Eq. 7-4). This means that the user can intervene in the whole process at several levels, as depicted in Figure 7-1.

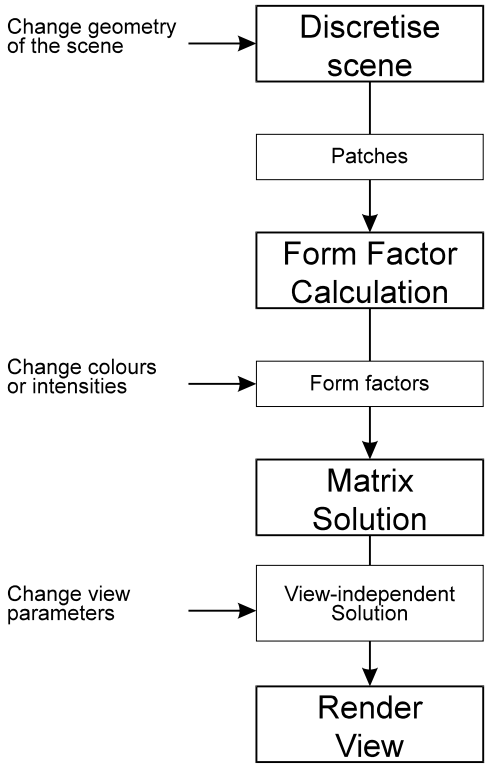


*Figure 7-**Error! Unknown switch argument.**:*
*Radiosity solution steps + user interaction*

In the Gauss-Seidel solution of the radiosity matrix, the solution is calculated one row at a time. Starting with all $B_i$ set to $E_i$ (which means that only light emitters will have a non-zero starting value for $B_i$) as an estimate of the radiosity value. In each iteration step, this estimate is adjusted for one of the patches, by "gathering" the radiosities of all other patches. If one would render the resulting "in-between" solutions, one would first see only the light sources, then gradually more and more patches would be illuminated.

### 7.1.3. Progressive Radiosity

The solution order of Gauss-Seidel is reversed in another category of radiosity algorithms, called "shooting" algorithms, also termed "progressive radiosity" ([Cohen 88]). In these approaches, the radiosity of *all* patches is updated for each iteration step, by choosing per iteration the patch with the most "unshot" energy (i.e. a patch that has not yet fired its energy towards all other patches and which has the largest $B_i$ value amongst the "unshot" patches at that particular moment in the iteration) and letting this patch "fire" its energy towards all others. This restructuring of the solution calculation ensures that with very few iterations, a

very good approximation of the exact solution is gradually generated. At each step *i*, the influence of patch *i* on each of the other patches *j* is calculated:

$$B_j = Bj + B_i \, (\rho_j \, F_{ji}) \qquad \textit{(Eq. 7-5)}$$

where $F_{ji} = F_{ij} \, A_i / A_j$ (because of the reciprocity relationship).

Then the progressive radiosity algorithm becomes:

```
for ( all i )
    {
      Bi = Ei;
      ΔBi = Ei;
    }
while ( not converged )
    {
      pick i, such that ΔBi * Ai is largest;
      for ( every element j )
          {
            Δrad = ΔBi * ρj Fji;
            ΔBj = ΔBj + Δrad;
            Bj = Bj + Δrad;
          }
      ΔBi = 0;
      display the image using Bi as the intensity of element i;
    }
```

This should be interpreted as follows: each element *i* has a value $B_i$, which stands for the radiosity of the element that is calculated so far. It also has a value $\Delta B_i$, which is the energy that still has to be "shot" to the other patches. At the beginning of each step, the element with the largest unshot energy is chosen and its radiosity is shot into the environment. As a result, other patches *j* may receive some new radiosity ($\Delta rad$) which is added to its $B_j$ value, but also to $\Delta B_j$, since this newly received radiosity is still unshot, from element *j*'s point of view. After the shooting operation, element *i* no longer has unshot energy, hence $\Delta B_i$ is set to zero.

Note that while performing one iteration step, the form factors $F_{ji}$ can be calculated "on the fly". This is another advantage over the Gauss-Seidel algorithm where the whole matrix of form factors has to be calculated.

This also means that the control structure of Figure 7-1 is no longer valid in the progressive refinement strategy: now the form factors are calculated one row at a time, immediately

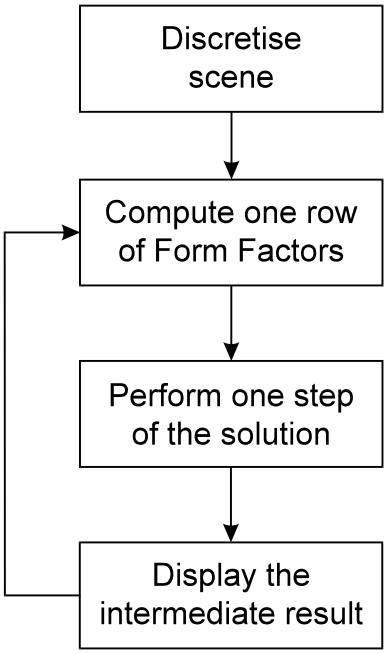


*Figure 7-**Error! Unknown switch argument.**: Progressive refinement control structure*

followed by a solution step that can be displayed for early evaluation. This new control structure is depicted in Figure 7-2.

### 7.1.4. Form Factor Calculation

Until now, the whole discussion of solving the radiosity equations has ignored the actual calculation of the form factors: it was always assumed that these values were known, but we still have to explain how they can be calculated. There are some problems associated with form factors in radiosity:

- The calculation of form factors itself is in general a far from trivial task, unless heavy restrictions are imposed on the shape and relative orientation of patches.
- Form factors have to take into account the fact that some patches cannot be reached from a certain patch, because of intervening patches.
- If one takes the solution method of the Gauss-Seidel type, the whole matrix of form factors has to be fully calculated and stored before the solution algorithm can start solving the system of equations. This imposes some constraints on the complexity of the scenes, since the form factor matrix has dimensions nxn where n is the number of patches.

#### *7.1.4.1. The Hemicube Method*

This is the first efficient method (introduced in [Cohen 85]) for solving the first two problems mentioned: form factor calculation and the intervening patch problem. There are so many aspects and particularities associated with the hemicube method, that we will stick to very brief description of the principles. More information can be found, e.g. in [Cohen 85], [Foley 90], [Watt 92] or [Cohen 93].

The method starts with placing a hemicube at the centre of a patch, with its z-axis aligned with the patch normal. The faces of the hemicube are divided into smaller elements, called "pixels" (typically ranging from 50 x 50 to several hundreds per face). Then all other patches are projected onto the surface of the hemicube (with the patch centre as projection centre), using a viewing pyramid and a standard visible-surface determination algorithm. The coverage of a patch on the hemicube is used as a measure of its radiosity. Part of the form factor values can be precomputed, which makes the method very efficient, particularly when one has fast visible-surface algorithms available.

Of course, since the hemicube method relies on an image-precision visible-surface algorithm, it suffers from the problems associated with these techniques. Typical problems of the hemicube algorithm are:

- aliasing problems, due to the fact that only one patch identity is stored per hemicube pixel and due to the inherent undersampling problems;
- the hemicube is placed around the centre of the patch, which assumes that this point is representative for the patch's visibility to other patches; if this is not the case, it can be broken into subpatches, but this finer structure must be used for all patches that are tested;
- patches must be far away from each other for the hemicube algorithm to produce correct results; this is particularly problematic for adjacent patches, because in this case the form factor will be underestimated;
- a hemicube placed on a source can only determine form factors for receivers with a finite area, not point receivers (differential areas); thus the vertex radiosities cannot be determined directly, but must be calculated by averaging the radiosities of surrounding patches, introducing inaccuracies.

These problems can be avoided by using the method introduced in [Wallace 89], where ray tracing is used for the form factor calculations.

### *7.1.4.2. Form Factors in the Ray Tracing Approach*

In the hemicube approach, the problems that we mentioned originate from the fact that the visibility tests are performed by a set of samples in uniform directions. The method did not take into account that ultimately the radiosity should be known at the element vertices. The ray tracing method eliminates these problems by calculating the vertex radiosity directly.

The ray tracing approach uses the same control flow as the original progressive radiosity method: at each step in the iteration, the patch with the largest unshot energy is taken as light source. Then, instead of placing a hemicube at the centre of this patch, every vertex in the scene is visited and a form factor is computed by firing rays at a set of sample points on the source patch. This way, the illumination will be certainly calculated at each vertex, avoiding the undersampling problem of the hemicube method. Each form factor may be calculated with any desired accuracy and the number of samples may vary from vertex to vertex, allowing area light sources to be approximated as accurately as needed.

For form factor determination using ray tracing, a slightly modified notion of form factors must be used as one now has to compute the form factor from an area $A_2$ to a differential area $dA_1$ at each vertex. The form factor between two differential areas depends on the distance $r$ between the areas, as well as on their mutual orientation $\theta_i$ and $\theta_j$ (Figure 7-3). For the case in which one subdivides the area $A_2$ uniformly in $n$ parts, the following (differential) form factor is to be computed (cf. [Wallace 89] for the derivation):

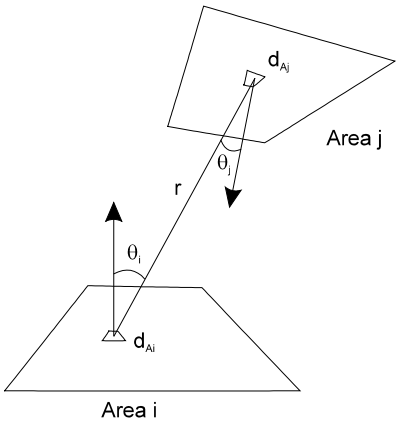


*Figure 7-**Error! Unknown switch argument.**: Geometrical aspects of form factors*

$$dF_{A2\text{-}dA1} = dA_1 \frac{1}{n}\sum_{i} \delta_i \frac{\cos\theta_{1i}\cos\theta_{2i}}{\pi r_i^2 + A_2/n}$$

*(Eq. 7-6)*

where

- $n$ is the number of samples on the emitting patch (source)

- $\delta_i = 1$ if the sample point is visible to the differential area $dA_1$ (i.e. the vertex), 0 otherwise

Equation 7-2 tells us that the radiosity of a surface *i* due to energy received from a single source *j* is given by:

$$B_i A_i = \rho_i B_j A_j F_{ji} \qquad \textit{(Eq. 7-7)}$$

Therefore, by substituting Eq. 7-6 into Eq. 7-7, the radiosity $B_1$ at a vertex 1 received from an area $A_2$ is consequently given by:

$$B_1 = \rho_1 B_2 A_2 \frac{1}{n} \sum_i \delta_i \frac{\cos \theta_{1i} \cos \theta_{2i}}{\pi r_i^2 + A_2/n} \qquad \textit{(Eq. 7-8)}$$

In [Wallace 89] a discussion of sampling the sources can be found. This description would lead us too far from the essence of progressive radiosity using ray tracing, so the interested reader is referred to that paper.

## 7.2. Speeding Up Radiosity

Although the early radiosity methods gave rise to the most realistically rendered images of that period, they suffered from a storage and time cost that were $O(n^2)$ - with n the number of elementary surface patches -, so they were not practical to be used for highly complex scenes. With the introduction of progressive radiosity algorithms, however, the performance has drastically improved ([Cohen 88], [Wallace 89]). Moreover, additional features have been introduced around the radiosity theme that even further enhance the image quality through the introduction of specular reflection ([Chattopadhyay 87], [Sillion 89]) and mapping effects ([Chen 90]), or even more improve performance via hierarchical representations and multi-gridding ([Hanrahan 91]).

A next step upwards with respect to improving performance is to incorporate parallel computations to solve the radiosities. In the radiosity literature, one agrees that the computation of the form factors is the most CPU-time consuming part of the radiosity algorithm: in [Cohen 85], Cohen and Greenberg show that the form factor calculation is approximately an order of magnitude greater than both stages of solving the set of equations and rendering a view. Therefore, the best way to make the radiosity algorithm parallel is introducing the parallelism in the form factor calculations.

In the rest of this chapter, we describe parallelisation of the form factor computations based on the ray tracing method, combined with progressive radiosity. The main arguments we can give for choosing this approach are the following: (1) aliasing and undersampling problems are reduced; (2) traditional advantages (non-physical light sources, shadow testing against exact geometry, ability to turn shadows on and off on a surface by surface basis, utilisation of exact surface normals, shadows of semi-transparent surfaces) as mentioned in [Wallace 89] remain; (3) our transputer based graphics system [Van Reeth 91] doesn't include hidden surface removal hardware to heavily accelerate the hemicube method; (4) transition towards inclusion of specular reflections ([Chattopadhyay 87], [Sillion 89]) will be easier. Moreover, we already mentioned in the radiosity description of the previous section that the progressive radiosity approach yields the possibility to quickly create quite a good approximation of the fully-calculated radiosity.

## 7.3. Parallelising the Form Factor Calculations

### 7.3.1. Previous work

Several references can be found in the literature regarding parallelising the radiosity algorithm. [Prior 89] describes a solution in which the computational power of a transputer network is used for solving directly the form factor matrix in a distributed fashion. Although in some cases even superlinear expandability (based on the fact that the $O(n^2)$ data could be kept entirely in the distributed main memory, rather than on a virtual memory disk) is reported, we didn't want to follow this approach as it doesn't offer the advantages of progressive refinement methods. References to parallelising the progressive radiosity method can be found in [Baum 90], [Puech 90] and [Recker 90][3]. [Puech 90] and [Recker 90] introduce parallelism by distributing the progressive radiosity computations among a number of loosely coupled workstations, whereas [Baum 90] utilises the hardware accelerators and the (fixed set of) multiple processors within a single graphics workstation. These three approaches, however, are based upon a hemicube alike methodology for computing the form factors, with the problems that we indicated in section 7.1.4.1.

[3]After we implemented our algorithm, we received reference to (but didn't as yet manage to get access to) additional work around parallel radiosity in [Chalmers 89] and in Eurographics Workshop reports [Purgathofer 90],[Chalmers 91], [Feda 91], [Guitton 91] and [Jessel 91]).

### 7.3.2. The Hardware Platform

During the definition of the radiosity project within the laboratory, an important condition was the fact that the implementation should be integrated into the existing image synthesis architecture, consisting of a parallel network of transputers (cf. [Van Reeth 1991]; a photograph of the system is shown in Colour Plate 1).

To recapitulate, the hardware platform of the graphics system in the lab consists of the following components:

- a PC, utilised as a host for booting the network and for housing the I/O equipment (mouse, graphical tablet & stylus, scanner, video recorder interface, etc.);
- host transputer boards (with transputers having up to 32 MB of local memory) used for connection with the PC and for storing the database as well as application specific programs (modelling, motion specification, programming environment, etc.);
- a farm of workers (with transputers having typically 2 to 4 MB of local memory) for doing the calculation intensive rendering computations;
- two graphics boards (one delivering a non-interlaced high-resolution signal and one delivering an interlaced true colour PAL video signal) for displaying the user interface and the rendered images.

These processors are arranged in the farm topology of Figure 3-2.

### 7.3.3. Distributing the Work

In the loosely coupled workstation parallelism of [Recker 90], a master-slave approach is used. The master process (processor) selects the patch with the highest energy during the progressive refinement and distributes the work, while the other slave processes (processors) have to compute the form factors of the selected patch in relation with the other patches in the scene. Because the entire scene is available to each of the slaves, a reasonable efficiency can be maintained. In this method, however, the network communication (with a reported bandwidth < 150K / sec) might become a bottleneck, especially when the number of stations increases.

In our ray tracing based progressive radiosity implementation, we also utilise the master-slave approach: a transputer on the host boards (with up to 32 MB) acts as the master, whereas the transputers in the farm act as slaves; the master holds the element database and distributes the work, while a copy of the much coarser surface patch database is kept on each slave. Since each of the four bi-directional links in each of the transputers peak at 20 MBit / sec, communication isn't forming a bottleneck (because the aggregate bandwidth increases linearly with the number of processors in the system).

The form factor calculations are distributed as follows. In each step, the system should trace rays from every element vertex towards the set of sample points on the source patch that s selected as emitter. This is accomplished by first (at the beginning of an iteration) selecting the patch with the largest unshot energy and transmitting this to the slaves. Then the workers trace a subset of the total bunch of rays between all element vertices an the (predefined set of) sample points on the emitter.

In order to keep the slave processors usefully busy - in tracing rays from the vertices to different points on the source patches and in computing equation 7-8 with the results of this ray tracing process - three important issues have to be borne in mind: (i) granularity of the blocks sent to the slaves, (ii) buffering methodologies and process priorities and (iii) acceleration techniques.

#### *7.3.3.1. Granularity*

In the master-slaves methodology we utilise, it is the master who decides how much work is done by each of the slaves. For a given step in the iteration, (1) the surface with the largest amount of energy left to be radiated is selected by the master to be forwarded to each slave, after which (2) the radiosities can be calculated for each of the vertices in the element database (note : vertices shared by more than one element are processed only once). It is the work in step (2) that is performed in parallel. Rather than sending large chunks of work to a slave, as could be beneficial in a loosely coupled workstation approach, the size of the data packets carrying the element vertices to be processed by the slaves is kept much smaller in our closely coupled transputer network. Indeed, the more element vertices to be processed in a burst, the more rays that processor will have to fire and the longer it will take to finish its burst. Hence, a fine granularity is beneficial as it will ensure a low elapsed time of waiting processors at the end of a progressive iteration. On the other hand is it not beneficial to have too many time-

consuming synchronisations (i.e. bursts) in an iteration. These are conflicting requirements, so determining a good number of vertices to be processed in one burst is a far from trivial task: it depends on the average time it takes to trace a certain amount of rays through a scene, the complexity of the scene as well as on the number of processors in the system.

For each progressive iteration, we currently have the rule of thumb of synchronising between 50 and 100 times (this number goes up if the number of processors in the system gets larger). For scenes of a complexity alike the testing scene in the next section, this means we send over between 2 to 8 vertices in a synchronisation: if V equals the number of vertices sent over in a synchronisation and S equals the number of sample points on the source patch, the number of rays to be traced in the entire iteration divided by V x S has to lie between 50 and 100, according to our rule of thumb.

#### *7.3.3.2. Buffering and Priorities*

In order to prevent a processor from going idle (regarding the tracing of rays and the evaluation of equation 7-8) between the moment of having calculated the radiosities of the vertices sent over and the moment of receiving the next package to be calculated, it is necessary to buffer data (this might not be necessary in an approach where the work distribution is fixed beforehand, but we follow the same dynamic work load as in the previous chapters, in which the slaves ask for more work if they are finished). Given the fact that we were physically obliged to use the farm topology, it moreover is important to see that the data packages are send forward and backward across the network as quickly as possible. Hence we utilise on each slave processor the process structure of Figure 3-4. The forward process receives data from a previous processor and checks if the data is sent over for the processor on which it resides. If this is the case, it will buffer the data packet; otherwise it will simply forward the packet to the next processor. The worker process does the actual computational job while the backward process simply ships data packets back down the farm. Hardware logic for context switching on the transputers ensures that scheduling and descheduling of processes on the processor is handled properly with minimal time penalties.

It is important to notice that the forward and the backward process have to be given a higher priority than the work process, in order to prevent a considerable loss in performance. The main motivation for this is that a backlocking communication tends to hold up the other

processors from doing their work properly. The philosophy behind this is that no matter what, the data always has to be transferred anyway, so one can better do it right away.

In alternative approaches, e.g., in which the vertex list is simply distributed equally among all the processors, the problem of finding a good granularity and the difficulty of finding a suitable fine grained communication methodology with appropriate priorities might not be present. However, these approaches would heavily suffer in keeping an optimal load balance, as some processors can coincidentally have been given a set of vertices which need a lot of work in comparison to other processors, introducing possibly much elapsed idle waiting time at the end of each iteration. These problems are comparable to the ones we encountered in the chapters on ray tracing.

#### *7.3.3.3. Additional Speed-up Techniques*

It should be realised that conventional ray tracing (non- screen space based) speed-up techniques can still be applied in order to substantially improve performance even further. We implemented the voxel based algorithm of [Amanatides 87] and found it to be particularly well suited. The fact that "typical" radiosity scenes - often building indoors consisting of coherently grouped surfaces - fit the voxel structure well, can account for this.

The voxel structure we utilise has a size of 20 x 20 x 20. Currently, each worker has its copy of this structure, implying on average only a few thousand surface patches can be handled (note that the original input surfaces are stored in the voxel structure, and not the surface elements, as the ray tracing naturally is done on the non-subdivided input-surfaces). The incorporation of the distributed-database version of Chapter 4 would be a well-suited solution to this problem and is therefore an interesting topic for further research.

## 7.4. Results

For a given scene, the timing results regarding the radiosity calculations heavily depend upon the following factors: (i) the number of original surface patches; (ii) the level of subdivision into surface elements; (iii) the maximum number of samples on the sources; (iv) the number of rays to be shot per vertex; and (v) the number of iterations in the solution.

Hence, it is difficult to find a "general" example for evaluating the solution. We circumvent this problem by giving some basic timings on the testing scene of Figure 7-4 (Colour Plate 26) on a network of 13 processors. This scene contains 238 original input surface patches, which are subdivided into 2029 surface elements. The number of points on which the radiosities have to be calculated is 3351.

*Figure 7-**Error! Unknown switch argument.**: Radiosity test scene (cf. Colour Plate 26)*

Depending on the energy source in a given iteration and the number of samples on a source, the number of rays to be traced varies from 4628 to 29575 (as the element vertices of surface patches behind the energy source don't have to be processed), giving respective calculation timings of 0.7 seconds to 4.1 seconds per iteration.

When we force the number of samples on a source to 16, we get the timings (per iteration, in seconds, for different test scenes and networks), depicted in Table 7-1, given the number of processors and the number of vertices in the iteration:

| | # processors | | | |
|---|---|---|---|---|
| # vertices | 4 | 7 | 10 | 13 |
| 1982 | 15.22 | 8.59 | 6.08 | 4.60 |
| 2743 | 24.04 | 13.49 | 9.52 | 7.25 |
| 3182 | 22.11 | 12.50 | 8.86 | 6.51 |
| 3251 | 25.75 | 14.54 | 10.23 | 7.80 |

*Table 7-**Error! Unknown switch argument.**: Time per iteration for different scenes and networks*

As yet, we haven't had access to a system with more than 13 slaves for testing the radiosity code, but our work that is described in the previous chapters, indicates that we can expect a further linear expandability (tested with up to 50 slaves).

(in between note: As the ray tracing is an important time consumer, it might be interesting if we give an impression of the traditional ray tracing speed: A 13 processor system generates a 768 X 576 resolution ray traced image (tree depth = 1) of the scene in Figure 7-4 (with 4 point light sources) in 38 seconds if no shadowing nor anti-aliasing is used.)

## 7.5. Conclusions and Further Research

In this chapter, we presented a novel approach for parallelising the calculation intensive parts of the ray tracing based progressive refinement radiosity method. It has been shown how a network of closely coupled parallel processors, in this case transputers, can be utilised for speeding up the performance linearly.

Future research is necessary to reveal what happens to the load balance in case of utilising much more processors. Our own work will also involve a study around incorporation of our parallel ray tracing software for large data bases, described in chapters 4 and 5. Moreover, it will be interesting to find out what happens to the expandability of the system if the visibility calculations can be done more quickly; e.g., by incorporating novel "less accurate" ray tracing approaches (cf. conclusion in (Hanrahan 1991)) or by accompanying the transputers with state of the art RISC- or DSP-chips, sustaining a performance more than an order of magnitude larger than that of transputers.

# 8. Conclusions

In this thesis we showed that the problem of global illumination in three-dimensional computer graphics can be tackled in a uniform way on a parallel network consisting of general purpose processors, in this case transputers. We also described how we addressed the goals we had in mind when we started the research in this field. Let's summarise our contributions with respect to these goals:

- In Chapters 4 and 5, we investigated two different techniques for reducing the number of ray-object intersection calculations within our parallel ray tracer, namely uniform subdivision into voxels and a hierarchical tree of bounding boxes. It turns out that the voxel technique is the most straightforward and allows one to use two types of coherence: space coherence when traversing the voxel grid and database coherence for the database distribution. Tests show that, on average, a linear speedup is maintained. However, voxels suffer from some deficiencies: (1) multiple copies of object data are stored and (2) memory usage is not predictable by the size of the database. The tree-based approach on the other hand, gives a solution for these problems: in this technique, the memory overhead needed for storing the hierarchy of trees can directly be deduced from the size of the database. Moreover, only one reference to each object is needed.

- Database coherence is exploited in the large database versions of both approaches: the whole database is stored in the host processor, while the worker processors contain a (coherent) subset thereof. Statistics show that the number of synchronisations needed for additional data requests remain within reasonable limits, allowing larger databases to be ray traced in both methods.

- When comparing both methods, it appeared that it depends heavily on the scene which of the two will be faster: the voxel algorithm is best suited for scenes where the objects are distributed rather uniform and have comparable sizes. E.g. if a huge plane is put underneath a coherent cluster of objects, the voxels will not at all be filled uniformly: the clustered objects may even end up in one row of voxels (or in the worst case even one single voxel !), such that there will be very little benefit in using the voxel structure. In such a

case, the tree-based approach will prove to be a better solution. For a more uniform scene, however, the voxel method tends to be faster, at the cost of the storage overhead we mentioned. So, a space-time trade-off will have to be made, together with a good evaluation of the scene, in order to choose between the two methods.

- We were able to include patch-based objects in our parallel ray tracer. With the basic operations of Toth's method as foundation, a tree caching method is used to cache previously performed subdivisions, in order to use these results again for coherent rays. Statistics show that the surface tree caching method is at least three times faster than the straightforward implementation of Toth's algorithm. We also filled the gaps left open in [Lischinski 90]: we used a tree pruning method instead of L&G's drastic tree cutting; moreover, trees do not tend to grow as deep if one cuts off the least recently used branches. Timings show that complex, patch-filled scenes can be rendered quite fast.

- In the last chapter, we introduced a novel approach to using parallelism in radiosity calculations. Previous work was mostly done in the field of the "gathering" algorithms by solving the radiosity matrix in parallel, or in the "shooting" algorithms, based on a hemicube approach to calculate form factors. In our method, however, we use ray tracing based progressive radiosity. The ray tracing part is done in parallel by distributing the vertices for which the (modified) form factors have to be calculated, among the worker processors, which shoot sample rays in the direction of the light source that is shooting its energy in the current solution step. Timings indicate that in this parallel approach, only a few seconds of calculation time is needed per iteration. If we keep in mind that in the progressive radiosity method a good approximation of the final radiosity solution is reached after a very few steps and that the form factor matrix is solved row after row, one can get a good response time from this system.

- Last but not least, these algorithms were all actually implemented on a network of closely coupled processors, consisting of transputers. In all approaches, buffering techniques are used within a dynamic load balancing scheme, in order to keep the processors usefully busy. Tests show that processors tend to become idle only at the end of the rendering, when all processors are working on their last packet of work, which means that a good granularity has been applied.

# 9.References

[Amanatides 87]

Amanatides J. and Woo A. (1987), *A Fast Voxel Traversal Algorithm for Ray Tracing*. In: Marechal G. (ed.), Eurographics '87 Proceedings, pp. 3-10

[Appel 68]

Appel A. (1968), *Some Techniques for Shading Machine Renderings of Solids*, AFIPS 1968 Spring Joint Computer Conference, 32, pp. 37-45

[Baum 90]

Baum D.R. and Winget J.M. (1990), *Real Time Radiosity Through Parallel Processing and Hardware Acceleration*. Computer Graphics (Proceedings SIGGRAPH ‘90) 24 (2), pp. 67-75

[Blinn 76]

Blinn J.F. and Newell M.E. (1976), *Texture and Reflection in Computer Generated Images*, Communications of the ACM, 19(10), October 1976, pp. 542-547

[Blinn 78]

Blinn J.F. (1978), *Simulation of Wrinkled Surfaces*. Computer Graphics (SIGGRAPH '78 Proceedings) 12 (3), pp. 286-292

[Bui-Tuong 75]

Bui-Tuong, Phong (1975), *Illumination for Computer-Generated Pictures*, Communications of the ACM, 18 (6), June 1975, pp. 311-317

[Chalmers 89]

Chalmers A. and Paddon D. (1989), *Implementing a Radiosity Method Using a Parallel Adaptive System*. Proc. First Int'l Conference on Applications of Transputers

[Chalmers 91]

Chalmers A. and Paddon D. (1991), *Parallel Processing of Progressive Refinement Radiosity Methods.* Proc. Second Eurographics Workshop on Rendering, Barcelona

[Chattopadhyay 87]

Chattopadhyay S. and Fujimoto A. (1987), *Bi-directional Ray Tracing*. Computer Graphics International (Proc. CGI ‘87), Springer-Verlag, Tokyo, pp. 335-343

[Chen 90]

Chen H. and Wu H. (1990), *An Efficient Radiosity Solution for Bump Texture Generation*. Computer Graphics (Proc. SIGGRAPH '90) 24 (2), pp. 125-134

[Clark 79]

Clark J.H. (1979), *A Fast Algorithm for Rendering Parametric Surfaces*. (Distributed only to attendees of SIGGRAPH '79). In: Joy K.I., Grant C.W., Max N.L. and Hatfield L. (eds.) (1988), *Tutorial: Computer Graphics: Image Synthesis*. Computer Society Press, Washington DC

[Cohen 85]

Cohen M.F. and Greenberg D.P. (1985), *An Radiosity Solution for Complex Environments*, Computer Graphics (Proc. SIGGRAPH '85) 19 (3), pp. 31-40

[Cohen 86]

Cohen M.F., Greenberg D.P., Immel D.S. and Brock P.J. (1986), *An Efficient Radiosity Approach for Realistic Image Synthesis*. IEEE Computer Graphics and Applications 6 (2), pp. 26-35

[Cohen 88]

Cohen M.F., Chen S.E., Wallace J.R. and Greenberg D.P. (1988), *A Progressive Refinement Approach to Fast Radiosity Image Generation*. Computer Graphics (Proc. SIGGRAPH '88) 22 (3), pp. 75-84

[Cohen 91]

Cohen M.F. (1991) *Radiosity*. In: Rogers D.F. and Earnshaw R.A.(eds.), *State of the Art in Computer Graphics*. Springer-Verlag, pp. 59-90

[Cohen 93]

Cohen M.F. and Wallace J.R. (1993), *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston - San Diego - New York - London - Sydney - Tokyo - Toronto

[Cook 86]

Cook R.L. (1986), *Stochastic Sampling in Computer Graphics*. ACM Transactions On Graphics 5 (3), pp. 51-72

[Crow 87]

Crow F. (1987), *The Origins of the Teapot*, IEEE Computer Graphics and Applications, 7, pp. 8 - 19

[Farin 88]

Farin G. (1988), *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, London

[Feda 91]

Feda M. and Purgathofer W. (1991), *Progressive Refinement Radiosity on a Transputer Network*, Proc. Second Eurographics Workshop on Rendering, Barcelona

[Foley 90]

Foley J.D., Van Dam A., Feiner S.K. and Hughes J.F. (1990), *Computer Graphics - Principles and Practice.* Addison-Wesley, Reading, Massachusetts

[Fujimoto 86]

Fujimoto A., Tanaka T., and Iwata K. (1986), *ARTS : Accelerated Ray Tracing System*, IEEE Computer Graphics and Applications, 6 (4), April 1986, pp. 16-26

[Glassner 84]

Glassner A., *Space Subdivision for fast Ray Tracing*, IEEE Computer Graphics and Applications, 4(10), July 1984, pp. 15-22

[Glassner 89]

Glassner A. (1989), *An Introduction to Ray Tracing*. Academic Press, London

[Goral 84]

Goral C.M., Torrance K.E., Greenberg D.P. and Bataille B. (1984), *Modelling the Interaction of Light Between Diffuse Surfaces*. Computer Graphics (Proc. SIGGRAPH '84) 18 (3), pp. 213-222

[Guitton 91]

Guitton P., Roman J. and Schick C. (1991), *Two Parallel Approaches for a Progressive Radiosity*, Proc. Second Eurographics Workshop on Rendering, Barcelona

[Gouraud 71]

Gouraud H. (1971), *Continuous Shading of Curved Surfaces*, IEEE Transactions on Computers, C-20(6), June 1971, pp. 623-629

[Hall 83]

Hall R.A. and Greenberg D.P. (1983), *A Testbed for Realistic Image Synthesis*, Computer Graphics and Applications 3(8), November 1983, pp. 10-20

[Hall 89]

Hall R.A. (1989), *Illumination and Color in Computer Generated Imagery.* Springer-Verlag, New York

[Hanrahan 91]

Hanrahan P., Salzman D. and Aupperle L. (1991), *A Rapid Hierarchical Radiosity Algorithm*. Computer Graphics (Proc. SIGGRAPH '91) 25 (4), pp. 197-206

[Jessel 91]

Jessel J.P., Paulin M. and Caubert R. (1991), *An Extended Radiosity using Parallel Ray-Traced Specular Transfers*, Proc. Second Eurographics Workshop on Rendering, Barcelona

[Kajiya 86]

Kajiya J.T. (1986), *The Rendering Equation*. Computer Graphics (Proc. SIGGRAPH '86) 20 (4), pp. 143-150

[Kaplan 87]

Kaplan M.R. (1987), *The Use of Spatial Coherence in Ray Tracing*. In : Rogers D.F. and Earnshaw R.A. (Eds.), *Techniques for Computer Graphics*. Springer-Verlag, pp. 173-193

[Kay 86]

Kay T.L. and Kajiya J.T. (1986), *Ray Tracing Complex Scenes*, Computer Graphics, 20 (4), August 1986, pp. 269 - 278

[Lamotte 91]

Lamotte W., Elens K. and Flerackers E. (1991), *Surface Tree Caching for Rendering Patches in a Parallel Ray Tracing System*. In: Patrikalakis (ed.), *Scientific Visualisation of Physical Phenomena* (Proceedings Computer Graphics International '91), Springer-Verlag, pp. 189-207

[Lamotte 93a]

Lamotte W., Van Reeth F. and Flerackers E. (1993), *Computer Graphics and Animation on a Transputer Platform*, In: Grebe R., Hektor J., Hilton S., Jane M. And Welch P. (Eds.), T*ransputer Applications and Systems '93 - Vol. 1* (Proceedings World Transputer Congress '93), IOS Press, Amsterdam Oxford Washington Tokyo, pp. 233-243

[Lamotte 93b]

Lamotte W., Van Reeth F., Vandeurzen L. and Flerackers E. (1993), *Parallel Processing in Radiosity Calculations*. In: Magnenat-Thalmann N. And Thalmann D. (Eds.), *Communicating with Virtual Worlds* (Proceedings Computer Graphics International '93), Springer-Verlag, pp. 485-496

[Lane 80]

Lane J.M., Carpenter L.C., Whitted T. and Blinn J.F. (1980), *Scan Line Methods for Displaying Parametrically Defined Surfaces*. Communications of the ACM 23 (1), pp. 23-34

[Lischinski 90]

Lischinski D. and Gonczarowski J. (1990), *Improved techniques for ray tracing parametric surfaces.* The Visual Computer 6 (3), pp. 134-152

[Magnenat-Thalmann 90]

Magnenat-Thalmann N. (1990), *Computer Art Forum*. The Visual Computer 6 (4), p. 242

[Max 81]

Max N.L. (1981), *Vectorised Procedural Models for Natural Terrain*, ACM Computer Graphics (Proc. SIGGRAPH ‘81), 15(3), pp. 317-324

[Nishimura 83]

Nishimura H., Ohno H., Kawata T., Shirakawa I. and Omura K. (1983), *LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation*. In: Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 378-394

[Nishita 90]

Nishita T., Kaneda K. and Nakamae E. (1990), *High-Quality Rendering of Parametric Surfaces by Using a Robust Scanline Algorithm*. CGI '90 Proceedings, pp. 493-506

[Packer 87]

Packer J. (1987), *Exploiting Concurrency; A Ray Tracing Example*, Inmos Technical Note 7, Inmos Ltd., Bristol

[Peterson 87]

Peterson J.W. (1987), *Distributed Computation for Computer Animation*, Technical Report UUCS-87-014, Dept. of Computer Science, The University of Utah

[Plunkett 85]

Plunkett D.J. and Bailey M.J. (1985), *The Vectorisation of a Ray Tracing Algorithm for Improved Execution Speed*, IEEE Computer Graphics and Applications, 5(8), pp. 52-60

[Potmesil 89]

Potmesil M. and Hoffert E., *The Pixel Machine: A Parallel Image Computer*, ACM Computer Graphics (Proc. SIGGRAPH ‘89), 23(3), pp. 69-78

[Prior 89]

Prior D. (1989), *An Architecture That Exploits Parallelism In Radiosity Calculations*. Proc. BCS Computer Graphics and Displays Group Seminar "Parallel Processing for Display", 7th April, London

[Puech 90]

Puech C., Sillion F. and Vedel C. (1990), *Improving Interaction with Radiosity-Based Lighting Simulation Programs*. Computer Graphics (Proc. SIGGRAPH ‘90) 24 (2), pp. 51-57

[Pulleyblank 87]

Pulleyblank R. and Kapenga J. (1987), *The Feasibility of a VLSI Chip for Ray Tracing Bicubic Patches*, IEEE Computer Graphics and Applications, 7(3), pp. 33-44

[Purgathofer 90]

Purgathofer W. and Zeiller M. (1990), *Fast Radiosity by Parallelisation*. Proc. Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics

[Recker 90]

Recker R.J., George D.W. and Greenberg D.P. (1990), *Acceleration Techniques for Progressive Refinement Radiosity*. Computer Graphics (Proc. SIGGRAPH '90) 24 (2), pp. 59-66

[Reeves 83]

Reeves W. (1983), *Particle Systems - A Technique for Modelling a Class of Fuzzy Objects*, ACM Transactions on Graphics, Vol. 2, pp. 91-108

[Rubin 80]

Rubin S.M. and Whitted T. (1980), *A 3-dimensional representation for fast rendering of complex scenes.* Computer Graphics (SIGGRAPH '80 Proceedings) 14, pp. 110-116

[Salmon 88]

Salmon J. and Goldsmith J. (1988), *A Hypercube Ray Tracer*. In: Fox G.C. (Ed.), *Proceedings of the Third Conference on Hypercube Computers and Applications*, 1988

[Scherson 88]

Scherson I.D. and Caspary E. (1988), *Multiprocessing for Ray Tracing: A Hierarchical Self-balancing Approach*, The Visual Computer, 4(4), pp. 188-196

[Siegel 84]

Siegel R. and Howeol J.R. (1984), *Thermal Radiation Heat Transfer*, Hemisphere Publishing Corp., Washington DC

[Sillion 89]

Sillion F. and Puech C. (1989), *A General Two-Pass Method Integrating Specular and Diffuse Reflection*. Computer Graphics (Proc. SIGGRAPH '89) 23 (3 ), pp. 335-344

[Toth 85]

Toth D.L. (1985), *On ray tracing parametric surfaces.* Computer Graphics (SIGGRAPH '85 Proceedings) 19, pp. 171-179

[Ullner 83]

Ullner M.K. (1983), *Parallel Machines for Computer Graphics*, PhD thesis, California Institute of Technology. Report Number 5112:TR:83

[Van Reeth 91]

Van Reeth F. and Flerackers E. (1991), *Using Parallel Processing in Computer Animation.* In: Magnenat-Thalmann N. and Thalmann D. (Eds.), *Computer Animation '91*, Springer-Verlag, Tokyo Berlin Heidelberg New York London Paris Hong Kong Barcelona, pp. 227-240

[Van Reeth 92]

Van Reeth F., Lamotte W. and Flerackers E., *Ray Tracing Speed-up Techniques Using a MIMD Architecture*, Proceedings GraphiCon '92, Moscow. Published in Russian: *Methodi Uskorenija Trassirovki Luseji S Ispolzovanijem MIMD-Architekturi*, Programmirovanie, 4, pp. 50-61

[Van Reeth 93]

Van Reeth F., Flerackers E. and Lamotte W. (1993), *Animating Architectural Scenes Using Parallel Processing*. In: Connor J.J., Hernandez S., Murthy T.K.S. and Power H., *Visualisation and Intelligent Design in Engineering and Architecture* (Proceedings First International VIDEA Conference '93), Computational Mechanics Publications, Southampton Boston, pp. 149-164

[Wallace 89]

Wallace J.R., Elmquist K.A. and Haines E.A. (1989), *A Ray Tracing Algorithm for Progressive Radiosity*. Computer Graphics (Proc. SIGGRAPH '89) 23 (3), pp. 315-324

[Watt 92]

Watt A. and Watt M. (1992), *Advanced Animation and Rendering Techniques*, ACM Press, New York

[Weghorst 84]

Weghorst H., Hooper G. and Greenberg D.P. (1984), *Improved Computational Methods for Ray Tracing*. ACM Transactions on Graphics 3, pp. 52-69

[Whitted 78]

Whitted T. (1978), *A Scan Line Algorithm for Computer Display of Curved Surfaces*. Computer Graphics 12 (3), p. 26

[Williams 78]

Williams L. (1978), *Casting Curved Shadows on Curved Surfaces*, Computer Graphics (Proc. SIGGRAPH '78), pp. 270-274

[Witten 82]

Witten I.H. and Neal R.M. (1982), *Using Peano Curves for Bilevel Display of Continuous-Tone Images*. IEEE Computer Graphics and Applications 2, pp. 47-52

[Whitted 80]

Whitted T. (1980), *An improved Illumination Model for Shaded Display*, Communications of the ACM, 23 (6), June 1980, pp. 343 - 349

[Wyvill 86]

Wyvill G., Kunii T.L. and Shirai Y. (1986), *Space Division for Ray Tracing in CSG*, IEEE Computer Graphics and Applications, 6(4), April 1986, pp. 28 - 34.

[Wyvill 90]

Wyvill B. (1990), *The Use of Spatial Coherence in Ray Tracing.* In: Rogers D.F., and Earnshaw R.A. (eds.), *Computer Graphics Techniques - Theory and Practice*, Springer Verlag, pp. 235-242

# 10. Colour Plates