# Simulation of Real Camera for Rendering

**Vlastimil Havran**

# Contents

**1 Introduction** **1**

**2 Fundamentals of Ray–Tracing** **3**
2.1 Models of Real World . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Photons, Color and Light . . . . . . . . . . . . . . . . . . . . . . 3
2.3 Tracing of Ray . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.4 Rays Classification . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.5 Reflection Model . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.6 Refraction and Reflection . . . . . . . . . . . . . . . . . . . . . . 8
2.7 Anti–aliasing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.8 Evaluation of the algorithm . . . . . . . . . . . . . . . . . . . . . 10
2.9 The Acceleration Methods . . . . . . . . . . . . . . . . . . . . . . 11

**3 Image Formation Model and its Orientation in the Scene Space** **13**
3.1 Mathematical Background . . . . . . . . . . . . . . . . . . . . . . 13
3.1.1 Vector Algebra . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.2 Transformation . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 The Pinhole Camera Model . . . . . . . . . . . . . . . . . . . . . 14
3.3 The Real Camera Model . . . . . . . . . . . . . . . . . . . . . . . 15
3.4 The Orientation of Camera in the Scene . . . . . . . . . . . . . . 16

**4 Simulation of the Camera** **20**
4.1 Reasons for Simulation . . . . . . . . . . . . . . . . . . . . . . . 20
4.2 The Basics of Lens Camera . . . . . . . . . . . . . . . . . . . . . 20
4.3 Potmesil Postprocessor . . . . . . . . . . . . . . . . . . . . . . . 25
4.4 Camera Preprocessor . . . . . . . . . . . . . . . . . . . . . . . . 26
4.4.1 Calculation of Rays Inside the Camera . . . . . . . . . . . 26
4.4.2 Generation of Rays inside Camera . . . . . . . . . . . . . . 27
4.4.3 Calculation the Ray Passing Through Ideal Lens . . . . . . 28
4.4.4 Calculation of Ray Passing Through Refractive Surface . 29
4.4.5 Elements of Camera . . . . . . . . . . . . . . . . . . . . . 31
4.4.6 Methods for Focusing and Auxiliary Methods . . . . . . 34
4.4.7 Camera Elements Design . . . . . . . . . . . . . . . . . . . 37

4.4.8 Complexity Evalution of Camera . . . . . . . . . . . . . . . 38
4.4.9 Camera Preprocessor Modul . . . . . . . . . . . . . . . . . 39

**5 Parallel Ray–tracing** **40**
5.1 Approaches to Parallelization of RT . . . . . . . . . . . . . . . . . . 40
5.2 Metrics for Processor Load and its Distribution . . . . . . . . . . . 42
5.3 Load Distribution and Balancing Strategies . . . . . . . . . . . . . . 44
5.3.1 No Load Balancing . . . . . . . . . . . . . . . . . . . . . . . 44
5.3.2 Static Load Balancing . . . . . . . . . . . . . . . . . . . . . 45
5.3.3 Dynamic Load Balancing . . . . . . . . . . . . . . . . . . . . 46
5.4 Types of Coherence in RT . . . . . . . . . . . . . . . . . . . . . . . 46

**6 Implementation of Parallel Ray-tracing on Shared Memory Architecture** **48**
6.1 The Library for Parallelization on Shared Memory . . . . . . . . . 48
6.1.1 Philosophy of the Library . . . . . . . . . . . . . . . . . . . 49
6.1.2 System Functions . . . . . . . . . . . . . . . . . . . . . . . . 50
6.1.3 Process Functions . . . . . . . . . . . . . . . . . . . . . . . . 51
6.1.4 Group Functions . . . . . . . . . . . . . . . . . . . . . . . . 52
6.1.5 Memory Management Functions . . . . . . . . . . . . . . . . 53
6.1.6 Timing and Resources Utilization Functions . . . . . . . . . 53
6.1.7 Identification Functions . . . . . . . . . . . . . . . . . . . . 54
6.1.8 Synchronizing Functions . . . . . . . . . . . . . . . . . . . . 55
6.1.9 Monitoring and Logging Functions . . . . . . . . . . . . . . 56
6.1.10 Communication Functions . . . . . . . . . . . . . . . . . . . 58
6.1.11 Process Mapping . . . . . . . . . . . . . . . . . . . . . . . . 59
6.2 Parallelization of RT . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.2.1 Algorithms for Task Management . . . . . . . . . . . . . . . 62

**7 Ray–tracer testing** **65**

**8 Conclusion** **72**

**Acknowledgements**

I would like to thank to ing. Ji ra, CSc., for his help and support to my work, to my colleague Jan Burinek for his discussion on implementation and the scenes preparation. I would like also thank to all my friends for their remarks concerning my diploma thesis. I want also thank to my parents for giving me the chance to study at Czech Technical University.

# Chapter 1

# Introduction

The evidence of computer graphics is very difficult to ignore, in last few years its application has spread in user programs a lot. It covers different things for different people and it depends very much on computer platform used. The increase of the popularity has been enabled by unbelievable decrease of the price of graphics workstation, among them also the PC's of nowadays can be included. The continual decreasing the price is not the one aspect of progress in computer hardware. The second one is also remarkable increase of performance. It gives to number of users the chance to create its own images and present some pieces of the information in graphics format. Among the versatile methods for generation of the impressive photorealistic images belongs ray–tracing, which has become very popular for its simplicity and practical illustration of its algorithms. As a result of much development of ray–tracing by many researches, it has become established as one of the most powerful and widely used techniques for realistic image synthesis.

Its main drawback, which holds liable to the most of the algorithms used in computer graphics, is time complexity of the ray–tracing. Although the principle of computing the image from the scene description is based on simple calculation, the number of such a calculation is enormous. This artefact disables the usage of ray–tracing in the interactive applications, which covers the virtual reality or the visualization of scientific data of the various sort. Another drawback is the size of the memory required by complex scenes, because they imply not only a geometric description, but also textures mapped onto the surface.

Another disadvantage of classic ray–tracing is the infidelity of the synthetized pictures. It covers the unrealistic representation of colors in generated image and the properties as the depth of field are not taken into account as well. This is important mainly for generation of images in cinematographics for animated sequences, where some special optical effects are required. Among them belongs the adaptation of the techniques as fade in, fade out, uniform defocusing of a scene, depth of field and distortion by the lens. It gives the ability to capture the viewer's attention to a particular segment of the scene, that is, it allows selec-

tive highlighting either through focusing or some optical effects. This additional features of enhanced ray-tracer model can be implemented by some ways, which all have some drawbacks. The common one is the increasing the time complexity of the computation. Thereby the question of the possibility to increase the performance of the classic algorithm become more relevant.

There are some ways, how to solve the time complexity of the algorithms. One way is to optimize the computation inside the algorithm. These methods well-known as the acceleration techniques, can decrease dramatically by two orders. But this is still not enough. The second method is the increase the performance of the computation unit used, but this method will one day reach the limits given by physical law. The last method consists of parallelization of the algorithm and it also is limited to a certain extent caused by communication links. The way which solves the problem consist in combination of all three methods, which brings some reasonable speed of ray–tracing algorithm.

This diploma thesis is divided into three parts. The first one outlines the ray–tracing in general and introduces the physical laws required for comprehension of the algorithms used. The second part treats with the principles used for photorealistic rendering and the rendering techniques, which are not commonly used even by commercial products. The last part covers the area of parallel ray–tracing. As the platform system has been chosen shared memory architecture for its fairly easy accessibility and its probably promissory results, the real multiprocessor architecture is PowerChalenge from SGI with six processors available.

The main focus is put on the explanation of enhanced ray–tracer model for special effects. The performance of parallel implementation has been evaluated with respect of number of processors used. The overhead for processes caused by other tasks on the machine has also been considered, because this situation is usual for current user taking advantage of the multiprocessor system. æ

# Chapter 2

# Fundamentals of Ray–Tracing

In this chapter is shown the basic principles of the algorithm with its natural laws behind beneath. The basic idea of ray–tracing is the simplification of the light behaviour by means of geometric optics to single beams and tracing this ray in reverse order. This principle enables the simulation of real world phenomena as the refraction and the reflection to some extent of the image fidelity.

## 2.1 Models of Real World

If we want to generate some synthetic image, then we need to describe the real world with all its properties. Description of real world is given by collection of constants and parameters that describe each object. The constants specifies the size, location, and the surface properties of the object as well such as glossiness, color and other parameters. These constants are plugged into equations that specify how to render a particular type of object. The geometric description is usually given by three-dimensional vectors, for example the location. Now only depends on the algorithm how it comprehends the given data and how the data are handled to generate the image. There are a lot of formats describing the real world or the scene. For example last few years format Open Inventor has become widely used in graphics applications. The scene description has to cover not only the objects in the scene, but also the sources of the light in the scene, that influences the properties of the synthetized image a lot.

## 2.2 Photons, Color and Light

From physical point of view foton is certain amount of energy. Photons are radiated from the atom in the process of jump the electron from higher orbit to lower orbit. Photons are emitted from the light source to all directions and oscillate in the direction perpendicular to the direction of radiation. The light from this physical point of view is of dual nature, one is called wave and second

one particle. Further it will be handled only with wave essence of light, which will be even simplified for our purposes. The reciprocal value of the frequency multiplied by the velocity of the radiation of the photons is the wavelength, which designates the color perceived by human eye. The color of the photon is changed by the interaction with the energy of similar type. The color of the surface is given by absorption of the range of light wavelength and the the human retina reacts to the rest of received photons.

The stream of huge amount of photons forms the light beam, which is more easy to handle with. This principle is the base of geometrical optics. If the beam is described as the stream of particles, then it is difficult to handle it. That is why this description is simplified by vectorization of the light beam, which is called ray. The ray is also determined by its position, which is originally the light source. It can be as mentioned above ray incident to the reflection or refraction surface. For this simplification is possible to express the ray trajectory using simple geometrical rules.

## 2.3 Tracing of Ray

Having described the objects in the scene, the lights involved in the process and the model for handling with light particles, we can try to designate the model of the computation of the image. Let us define the synthetised raster image as the window, through which the observer watches the scene. This concepts is the simplification of the real situation and it will be described in the next, but is mostly used in the ray–tracing. The computation of the image consists in contribution the rays collided with the window. One way is to copy a nature in entirely scope and simulate the emission of light beams from the light sources to all directions. When the ray encounters the window, then its color is added to the appropriate pixel. This method, which is also used in computer graphics for high quality rendering, is called backwards ray–tracing. The main drawback, of this method is unbelievable amount of rays generated from light sources comparing to the rays really encountering the virtual image plane. Therefore this method is used only in special cases, when the visualization of phenomena as caustics or penumbra is indispensable for synthetized image. The second way to generate the image is to trace the rays in reverse order, it means from the view point through the virtual image plane to the scene. The ray is shot to the scene and then the intersection with the closest object is computed. The color of the pixel is determined by some way described in the next. This method known as ray–tracing gives in most cases sufficiently good images and is widely used nowadays.

The ray is cast from view point through the pixel in the virtual image plane to the scene (see Fig. 2.1). For each pixel in the image is formed the primary ray specified by its position and by the directional vector. The primary ray $P$ comes out from virtual image plane to the scene and then the intersection with all objects

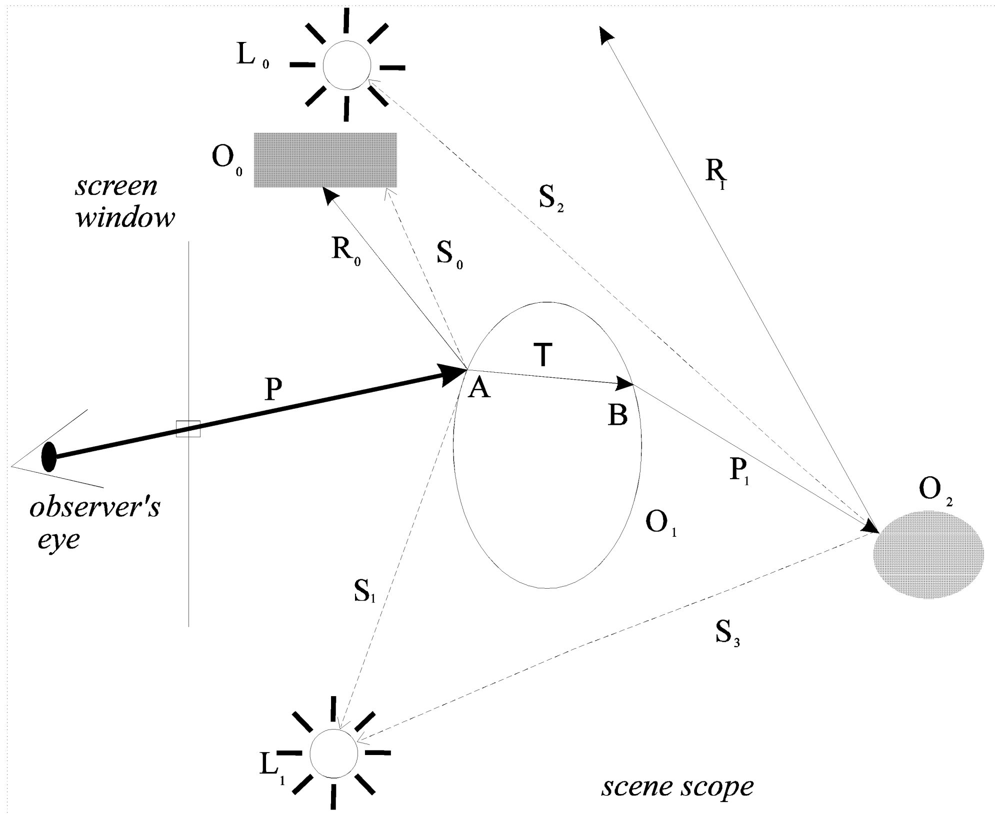


Figure 2.1: Tracing the Ray

in the scene is calculated. The nearest object is $O_1$ and the intersection point on its surface is $A$ . In the scene there are also placed two lights. For each light, if some intersection of this primary ray with scene object is found, is generated the ray from the intersection point to the light, provided the light is point. There are not the point lights in the real world, but this simplification is necessary to hold down the complexity of the algorithm. In our case the shadow ray $S_0$ is generated to the light $L_0$ and it meets the object $O_0$ . The shadow ray $S_1$ does not encounter any object and therefore the color contribution is added to the result color of pixel by certain calculation explained in the next. The ray is reflected on the surface as the ray $R_0$ and for this reflected ray is calculated the intersection with the proximate object. Since the object $O_1$ is transparent there also the ray passes through the object changing the direction in the point of intersection by refraction law and comes out the object in the point $B$. This ray designated $P_1$ meets the object $O_2$ and this process is repeated recursively for the generation of rays. From the point of generation of the refracted and of the reflected ray the process becomes recursive, although the measure of contribution of the color added to the pixel depends on the surface properties defined by reflection model. Mostly the generation of ray is strictly restricted by the depth of recursion or

by color contribution thresholding in the intersection point. The depth of the recursion significantly influences the quality of the synthetized image, but also exponentially the time complexity of the algorithm.

An important feature of the camera model is that it renders objects using perspective projection, which enhances the sense of depth in a two-dimensional image. Perspective projection makes objects appear distorted when they are close to the viewer and parallel lines converge farther away. The amount of distortion is related to the viewing angle. Generally, the larger the viewing angle, the greater the distortion. The next interesting fact is the position of the virtual image plane and the observer int the relationship to the scene. The both mentioned parameters can be positioned out of the scene or inside it, but if the picture has to be realistic, then is also necessary to modell the camera or the observer, because the shadows caused by their position can be visible as well.

## 2.4 Rays Classification

The distribution rays gives the scheme for classification the rays by their use in the model. The rays shot from the view point trough the virtual image plane are called **primary rays** or screen rays. The group of rays casted from the intersection point to the light source determines if the point is illuminated by this light source directly. Therefore they are called **shadow rays**. The group of rays shot from the intersection point using reflection law modells the ability of the surface to reflect the light. That is why this group of rays is called **reflected rays**. The similar situation is for rays refracted on the surface of a translucent object. This group of rays is called the transparency or **refracted rays**.

## 2.5 Reflection Model

The scheme of rays generation gives the recipe for distribution of the rays in the scene, but there are no instructions for the color evaluation of the image pixel. It has to be considered the scene is built up from objects with different behaviour, it means different color, shineness and other optical qualities as the transluency. For this purpose were devised the reflection models derived from natural laws. The reflection models are other simplifications of the reality, which reduce the computation complexity of image synthesis.

The classic reflection model, the most commonly used in computer graphics was designed by *Bui-Tuong Phong*. This empirical model cannot simulate some properties of the real surface. It supposes the surface ideally reflects the lights incident upon a surface. This fact is far away from the reality. Let us to define the convection on the Fig. 2.2, which illustrates the model behaviour.

The $\vec{N}$ stands for the normal vector to the surface in the intersection point.

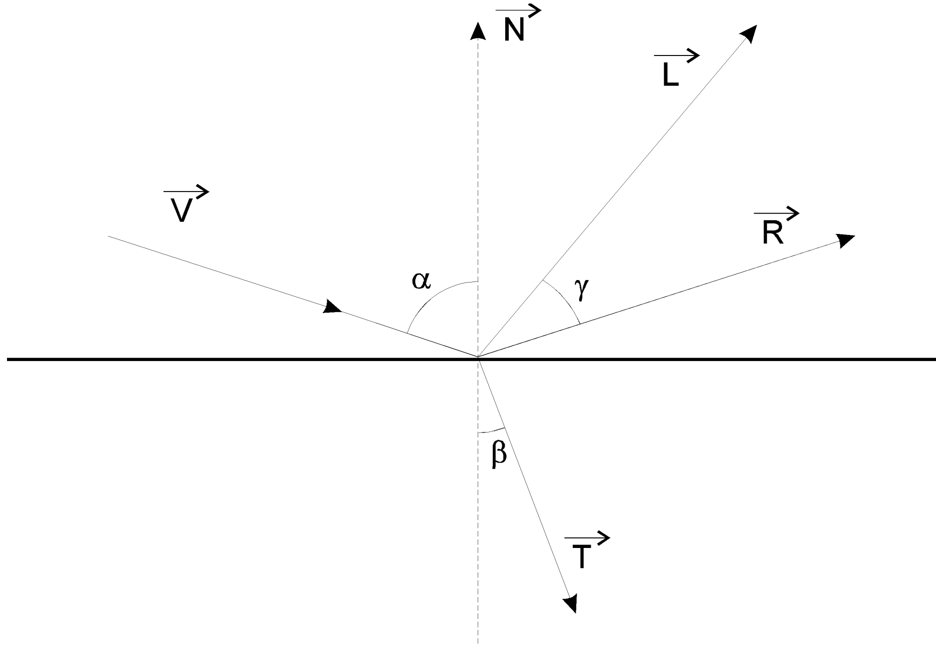

Figure 2.2: Ray incident upon a surface

The V is the primary ray casted from the view point, $\vec{L}$ stands for the shadow ray. For next explanation the term intensity has to be defined, which declares the color intensity given in RGB vector with elements $(r, g, b)$ in the interval $< 0.0, 1.0 >$. The total light intensity for the point without derived secondary rays is the sum of three intensities:

$$I = I_a + I_d + I_s,$$

where:

- $I_a$ is the intensity of ambient light

  $$I_a = C_a * k_a,$$

  where $C_a$ is the ambient intensity for whole scene and $k_a$ is the coefficient of the surface, which present the ability to reflect ambient light.

- $I_d$ is the diffusive part of the light and is declared as

  $$I_d = C_l * k_d.(\vec{L}.\vec{N}),$$

  where $k_d$ is the diffusion coefficient and this intensity declares the measure of illumination of the surface by the source lights independently of the view point, the $C_l$ is intensity of the light source.

- $I_s$ is the specular intensity of the light and is defined as

$$I_s = C_l.k_s.(\vec{R}.\vec{L})^{l_s},$$

where $k_s$ is the specular coefficient indicative the ability of the surface to reflect the light on the surface and $l_s$ is the mirroring quality parameter. The dot product $\vec{R}.\vec{L}$ corresponds the $cos(\gamma)$ of the angle between the vectors.

For more light sources in the scene the specular and diffusive intensity are calculated for each light separately and the intensity contribution is added to the total sum if the dot product in the expression is greater than zero. It is well recognizable in the scene geometry. For diffusive part expresses the oriented surface is turned by the visible face towards the light source. For specular part it expresses the light is in the direction of the reflected ray and the specular contribution has correct physical meaning.

## 2.6 Refraction and Reflection

Besides the color computed from any reflection model for one pixel in the image window there are the contribution of colors acquired from the reflected and transparency ray, if the surface of the object has its properties. If the light ray incides upon a surface, one of its parts is reflected and the second one refracted. The direction of both rays is determined by the surface normal vector and they comply following rules:

- The angle of incidence equals the angle of reflection
- The reflected and refracted ray remains in the half space determined by the normal vector
- The incident angle $\alpha$ and the refractive angle $\beta$ accomplishes *Snell Law*:

$$\frac{sin(\alpha)}{sin(\beta)} = \frac{c_2}{c_1} = \frac{n_2}{n_1},$$

where the $c_1$ is the velocity of the light in the ambience of the incidence and the $c_2$ similarly the velocity of the light inside the object. The coefficients $n_1$ and $n_2$ are absolute refraction indexes, which express the ratio of the light velocity in the vacuum and in the material of the object.

If the ray is propagated from the ambience optically more dense to the ambience optically more sparse, the sinus of the angle can be greater than one. This responds the total reflection of ray. For the value of the sinus one the angle of the incidence is called *Brewster* angle. The direction of the reflected and refracted

rays can be evaluated directly from the incident ray, the normal ray and the absolute refraction index. Direct deducement using vector algebra, computing the auxiliary vectors and goniometric functions for evaluating the arcus sinus of expression takes in the real computer processor a lot of computation and it can be optimized up to factor 3.0 by certain smart technique shown in the next.

The portion of the reflected light is given by Fresnel coefficient:

$$\mathcal{F} = \frac{1}{2}\frac{\sin^2(\alpha-\beta)}{\sin^2(\alpha+\beta)}\left[1+\frac{\cos^2(\alpha+\beta)}{\cos^2(\alpha-\beta)}\right].$$

For some materials the trasparency ray is absorbed after very small distance in the object. For these types of materials is better to express Fresnel coefficient by following formula:

$$\mathcal{F} = \frac{1}{2}\left(\frac{g-c}{g+c}\right)^2\left\{1+\left[\frac{c(g+c)-1}{c(g-c)+1}\right]^2\right\},$$

where $c = cos(\alpha)$ , $g = \sqrt{(n^2+c^2-1)}$ and $n$ is the relative factor pertinent to the refraction index.

The intensity of light ray is decreased by its transition through the refraction ambiancy. This phenomena is expressed by exponential expression parametrized by the distance passed through the material. For the purpose of computer graphics this equation is often simplified by linear expression. The refraction law does not concern only the objects in the scene, but in this thesis the enhanced camera model used for generation of the rays as well.

## 2.7 Anti–aliasing

In the computer graphics the problem of aliasing is quite common. The essence of this unwanted imperfection of the graphical representation is in the finite resolution of the image. The most commonly discussed phenomena in two-dimensional graphics is the jagged edge, which is caused due to the nature of the sampling process. The alias is also problem of the time domain, when the animation sequences of rotating object is performed. The last aliasing problem arises with using a color model, because it gives some boundaries for color storage and the representation on an graphics display.

Strict aliasing problem is caused by inadequate sampling of continuous information, which means undersampling in general. This body of knowledge was originally developed in the field of signal processing and reconstruction, but it can be applied in computer graphics as well. In three–dimensional modelling we define certain objects and in the case of ray–tracing the space is sampled by primary rays and then by secondary rays. The ray gives the color information about the scene precisely in the center of the pixel. However it is only the indelicate

simplification of real situation, because in correct evaluation should be computed through whole pixel area in the virtual image window. This is obviously impossible in practical terms, but the integral of the pixel image can be successfully aproximated by the supersampling within the pixel area and thereby reduces the aliasing artefacts. This method called antialiasing is used commonly in computer graphics and in ray–tracing as the additional feature for high–quality images. The generation of the rays for one pixel can be uniform, adaptive by incremental increase the number of rays in consequence of the color still evaluated or complying stochastical theory of the rays distribution. The disadvantage of the method is the increase of computational complexity of the algorithm.

## 2.8 Evaluation of the algorithm

The recursive ray–tracing in the form described here incorporates the facilities for rendering in a single framework:

- hidden surface removal
- shadows computation
- global specular interaction
- reflection of light
- refraction of light

It is evident the method is computationally complex. Most of the time (about 85 percent) is devoted to the evaluation of the intersections of ray with the objects. The complexity depends particularly on the shape of the scene primitives. The smallest complexity is for sphere, for the objects as quadrics and nurbs is significantly higher.

Let us have the scene for example with 1000 spheres and let us consider the resolution of the image $width \times height = 800 \times 600$. If we do not consider the refracted and reflected rays, we have to evaluate up to $480.10^6 \times (l_m + 1)$ of intersections, where the $l_m$ is the number of lights in the scene. This is the maximal limit, because we need not to evaluate all intersections of the shadow rays with objects in the scene in each case. When the first object in direction to the light is positively evaluated, then no contribution of the light is added to the result color. If we consider the allowed depth of ray-tracing $n$, the number of pixels in the image $width \times height$, then for $p$ objects in the scene the limit number of the intersection calculation is expressed by formula:

$$I_{max} = width \times height \times 2^{(n-1)} \times p \times (l_m + 1)$$

For the resolution and number of primitives mentioned above, two lights in the scene and the depth of recursion 3, which is often used, the whole picture requires to compute the incredible number $10.08 \times 10^9$ of intersections. It is clear not all objects are transparent and reflective, but on the other hand the amount of the primitives in the scene is usually much higher. The computational complexity is also decreased if the reflected ray leaves the scene completely. In real scenes the increase of the depth of recursion leads to the comparatively slight increase of the computations. But the number of the ray intersections is still bewildering and it makes this naive approach totally unusable for real applications.

## 2.9 The Acceleration Methods

The demands on computation resources shown in the previous chapter raise the question how to improve the effectiveness of the algorithm and speed up whole process of the image synthesis. There are some general approaches, which are applicable also for any kind of the algorithm in general:

- **algorithmical decrease of computation demands** - utilizes the division of the scene space to decrease the computational cost for intersection. The space occupation by objects is determined and the objects are assigned to some division of space, which could be any greater object with smaller computation complexity. The bounding objects can be hierarchized in some way or not. These divide and conquer strategies gives the schemes for this type of acceleration. Among these techniques belongs the bounding volume algorithm and its hierarchical modification. Because the speed up given by these algorithm is insufficient, were designed more sophisticated methods as Three Dimensional Digital Diferential Analyser and others, which can be further structured. The first group is uniform space subdivision and the second one adaptive space subdivision. Uniform subdivision consists of the division of the space into lattice or 3D grid. Among the adaptive techniques belongs the famous methods Octree Subdivision and Binary Space Partition. The important attribute of these methods is decrease of computational complexity. The speed up factor can achieve very different values from 2.0 up to 200.0 or more, which highly depends on the scene character. The adaptive techniques mostly bring off better results. The computation overhead is greater and the main disadvantage of this approach is the additional memory requirements used by space division data structures. Nowadays is probably infeasible to improve significantly the best algorithmical technique.

- **increasing the performance of the procesors** - hardware performance is increased by using new technologies. This method alludes to physical

and technical limitations of each technology used.

- **adaptation the algorithm to hardware** - this approach is based on organization the calculation in a such order, which is the most convenient for performance. It concerns the utilization the memory system management by the data dependency to decrease the miss ratio of cache system.

- **parallelization of the computation** - this approach is widely used in computer science in general. The task is distributed among more computers or the processes at the beginning of computation. Then is launched the intrinsic computation, where the computational units can mutually communicate and handle the object data. In the last phase the computed data are gathered and the output image is saved. Unfortunately this method has also bounds given by technical realization. They are caused by the communication between the processors during the computation. Even if the communication is not performed, then the limitation implies that the distribution and both distribution and collection the data is really provided in finitely indivisible time portions. The number of communication links from the initiatior process is also restricted by geometry of three dimensional space.

æ

# Chapter 3

# Image Formation Model and its Orientation in the Scene Space

This chapter describes the camera geometry used for generation of rays. The standard pinhole camera projection geometry and its modification used in ray-tracing is discussed. For more realistic rendering is discussed the camera model, which simulates better the real camera and offers some more additional alternatives.

## 3.1 Mathematical Background

In following text is necessary to understand mathematical calculation using vector and matrix algebra for calculation the transformations in 2D and 3D space. Let us review some basic terms supposing the 2D space is just the decrease the 3D space by one dimension.

### 3.1.1 Vector Algebra

- point $p = [p_x, p_y, p_z]$ in 3D space is the triple of values, which define the coordinates in Euclidian space.
- vector $\vec{v} = [v_x, v_y, v_z]$ declares the direction between two points and is calculated as the substraction of their coordinates. The vector is oriented and its origin can be moved anywhere.
- normalized vector has the length equal to one

$$|\vec{v}| = \sqrt{v_x.v_x + v_y.v_y + v_z.v_z} = 1$$

- dot product of two vectors is defined as

$$\vec{u}.\vec{v} = |\vec{u}|.|\vec{v}|.cos(\angle(\vec{u}, \vec{v})),$$

and the geometrical meaning of dot product on normalized vectors is the cosine of angle between the vector provided that they have equal origin.

- cross product of two vectors ic defined as

$$\vec{w} = \vec{u} \times \vec{v},$$

$|\vec{w}| = |\vec{u}|.|\vec{v}|.sin(\angle(\vec{u}, \vec{v}))$, $\vec{w} \perp \vec{u}$ and $\vec{w} \perp \vec{v}$,

- two vectors are collinear, if one can be expressed by multiplication of the second one. Three vectors are collinear, if one can be expressed by the linear superposition the others.

### 3.1.2 Transformation

The point can be transformed by linear transformation, that changes its coordinates in some way. The transformation can be described by $4 \times 4$ matrix and it supposes the homogeneous coordinates:

$$[x', y', z', w'] = [x, y, z, w]. \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix},$$

where the $[x, y, z, w]$ is the original point and $[x', y', z', w']$ is transformed point. Every point $[x, y, z]$ in 3D space can be expressed in homogeneous coordinates as $[x', y', z', w']$, and the transformation is defined as

$$x = \frac{x'}{w'}, y = \frac{y'}{w'}, z = \frac{z'}{w'}.$$

The $4 \times 4$ square regular matrix and the use of homogeneous coordinates is necessary for some kind of transformation. By means of this mathematical apparatus can be simply expressed not only the rotation around any axis, but also the displacement of the point and the change of the scale. The regularity of the matrix guarantees the feasibility of the inverse transformation.

## 3.2 The Pinhole Camera Model

The standard approach for generation of the rays is simple. It uses the view point as the origin of the rays and the virtual image plane, through which the ray is shot to the scene.

From the image plane in the back of camera box the ray could be cast through the pinhole to the scene, if the simulation of that is required. The image is reverted and from the photographic point of view the camera suffers from a lot

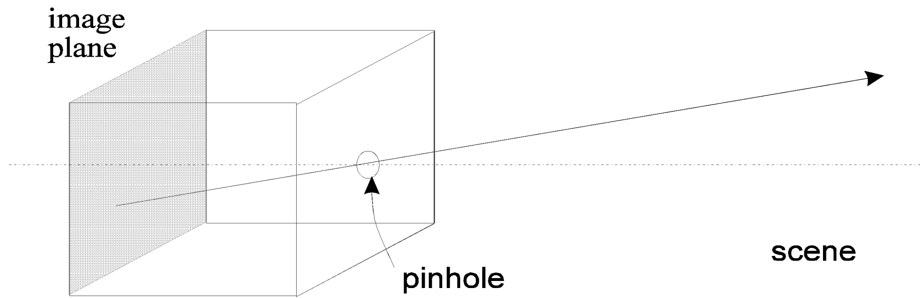


Figure 3.1: Pinhole camera model

of vices. Among them belong long shutter time, because the pinhole has to be enough small to preserve some sharpness of the captured image. This camera model is very contradictory from the practical view, because the decreasing the diameter of the pinhole increases the exposure time, although the sharpness of the captured static scene is higher. It was used in the beginning of photography and before for drawing the reality onto the transparent paper by artists from 16th century. The pinhole camera was originally invented by *Leonardo da Vinci*.

This model is usually used slightly modified by ray–tracer programs. The change consists in the shift of the image plane in front of camera. The rays are then cast from the pinhole point in the scene direction passing through the the shifted image plane as they would be casted from the original image plane. The pinhole point is called the **view reference point** and the image plane the **virtual image plane**. The result image painted on the virtual image plane then is not reverted. This situation never can occur in the real word.

## 3.3 The Real Camera Model

The approach not currently used in graphics renderers is to simulate real camera used in photography. The human eye uses the scheme as well. This situation is shown in the Fig. 3.2.

The geometrical model for capturing pictures is very similar to the pinhole camera model with one exception. The simulation of this concept differs in the fact, the rays are not cast through the pinhole, but through single lens or the system of lenses to the scene. The result image is influenced by the light refraction through the lens and the positioning of the film and the lens position and the orientation. The photographic cameras these days are often constructed with complex system of lenses, which enhance the lens speed of objective and eliminates the distortion given by refraction on the surface.

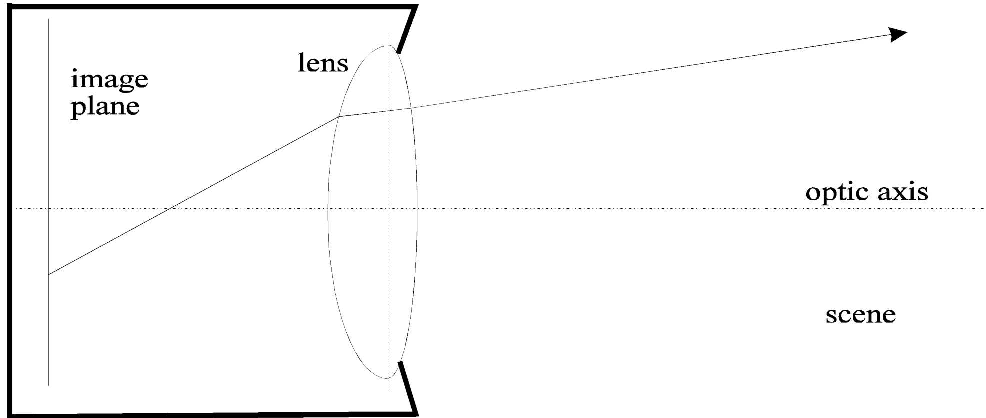


Figure 3.2: Real Camera Model

## 3.4 The Orientation of Camera in the Scene

The important fact of the camera models described above is perspective projection. The other possible projection used in technical science can be also simulated, but they use other model for rendering without the view reference point.

All following ideas suppose the usage of the orthogonal coordinate system. Using the camera raise a question, how to express the generation of rays and the position and orientation the camera with its optical properties in the scene space. The rays are generated within the camera coordinate system with main optical axis. This coordinate system is a space used to establish viewing parameters and a view volume. The main optical axis can be the $z'$ as illustrated in Fig.3.3. It is convenient to lay the image plane or virtual image plane onto the rest of axis , in our case the axes $x'$ and $y'$. Then the result image will be rectangular. These restriction on the camera coordinate system can reach different extent. The reason for alignment the axes with the image plane axes and the main optical axes as well recline in simplification of the ray generation within the camera coordinate system.

The ray leaving the camera is transformed using homogeneous transformation to orientate in some reasonable way to get required portion of the scene captured into the image plane. It is need to cope with the problem of the camera positioning and orientation in the scene space, which can be solved by next cases:

- camera position (view reference point)
- camera orientation
  - the point, that will be rendered in the centre of the image
  - the viewing direction vector - a vector normal to the view plane

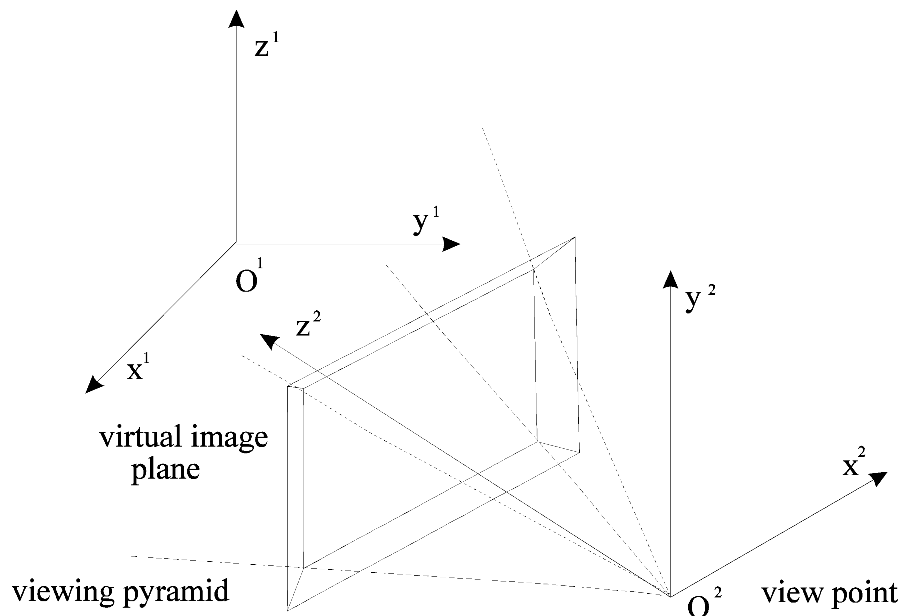


Figure 3.3: Orientation the camera in scene space

  - the azimuth and elevation angle in a spherical coordinate system

- view up vector, that is perpendicular to the viewing direction and orientates the camera about this direction. This vector can be moved that it lies in the image plane and is directed to the top of the synthetized image.

There can be discussed the convenience of view reference point as the camera position with regard to other important points inside the camera model. The selection of this reference point in most cases excluding the macrophotography simulation on real camera model has negligible effect in the result image.

The selection for the viewing direction vector can make difficult the handling the ray–tracer system by the user. Although the three possibilities expressing the same thing, the description of the azimuth and elevation angle gives small notion what will be visible on the image. Commonly in the process of creating the scene the user creates some objects and tries to display all the scene or its part. Mostly the camera position is then displaced many times and the creator of the scene knows the object position. That is why the specification of the angles is inconvenient. The specification by the viewing direction vector directly lies by its convenience inside the other mentioned alternatives. The choice for the best solution to handle the viewing direction depends highly of the renderer's application.

The view up vector specification is more complicated. Mostly is required the view up vector in the sense up in scene coordinate system. But the unit vector in the sense up is not perpendicular to the viewing direction vector. The sensible

strategy is to specify an approximate orientation and then to project this vector onto the image plane and thereby to gain the view up vector. The projection is calculated by $\vec{V} = \vec{V'} - (\vec{V'}.\vec{N}) \times \vec{N}$, where the $\vec{V'}$ is the vector specified by user, $N$ is the viewing direction vector and $\vec{V}$ is view up vector.

The image plane together with the view reference point determines a closed volume called the viewing pyramid or the viewing frustrum, that delineates the volume of space which is to be rendered. The objects outside the viewing pyramid are not visible from the view reference point through the restricted size of the image plane. The reference point together with the size and the position of the image plane determines the size of the viewing angle and thereby shape of the viewing pyramid. This corresponds to the zoom in the classical photographic terminology and the distance between the reference point and the image plane is called **focal length**.

Let us suppose the description given by camera position $C$, the viewing direction vector $N$ and view up vector $V$ perpendicular to the direction vector. The ray in the camera coordinate system is described by the position $P$, by direction vector $R$ identical with axis $z$ and by view up vector collinear with axis $z$. The task is to transform $P$ and $R$ onto the scene coordinate space $P_{sc}$ and $R_{sc}$ , which will be performed using the homogeneous transformation matrix.

Let us suppose right–handed coordinate system for camera and scene space. The viewing direction vector in the camera coordinate system is collinear with axis $z$ and view up vector with $y$ axis. The operation of the transformation matrix composition can be done inversely by mapping $\vec{N}$ and $\vec{V}$ onto the axis $z$ respectively $y$. It is necessary to perform the translation of the view reference point $T(v)$ onto the origin of scene coordinate space. The inverse matrix is not to be really computed using the matrix inversion, because the coordinate systems are orthogonal and it will be shown in the following text.

$$\vec{N} = [n_x, n_y, n_z], \vec{V} = [v_x, v_y, v_z],$$

The rotation of both vectors along the axis $x$ starting from vector $\vec{N}$

$$cos(\alpha) = \frac{\vec{V_x}.\vec{N_x}}{|\vec{V_x}|.|\vec{N_x}|}$$

composes the matrix

$$\mathbf{R}_{yz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) & 0 \\ 0 & -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Vector $\vec{N'} = \mathbf{R}_{yz}.\vec{N}$ has the $z$-part positive, otherwise the $\alpha$ has to be changed to $180-\alpha$. Then the rotation along the axis $y$ is performed for transformed vector $\vec{V'} = \mathbf{R}_{yz}.\vec{V}$ and $\vec{N'}$:

$$cos(\beta) = \frac{\vec{V'}_y.\vec{N'}_y}{|\vec{V'}y|.|\vec{N'}y|}$$

$$\mathbf{R}_{xz} = \begin{bmatrix} cos(\beta) & 0 & sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

In this moment the vector $\vec{N''}$ is collinear with the axis $z$. After the rotation by the last axis $z$ by similar way ($\mathbf{R}_{yz} \leftrightarrow \gamma$) the $\vec{V'''}$ is identical with axis $y$ and $\vec{N'''}$ remains identical to axis $z$.

The transformation $P$ to $P_{sc}$ in scene coordinate system is calculated by the matrix composed from rotation matrixes with the negative angles and also the translation vector.

$$\mathbf{T} = \mathbf{R}_{yz}(-\alpha).\mathbf{R}_{xz}(-\beta) * \mathbf{R}_{yz}(-\gamma) * \mathbf{R}_t(-v),$$

and

$$P_{sc} = \mathbf{T}.P,$$

$$\vec{R}_{sc} = \mathbf{T}.(P + \vec{R}) - P_{sc}.$$

This operation can be simplified thanks to the orthogonality of both coordinate systems. The $3 \times 3$ matrix for vector $\vec{V}$ and $3 \times 4$ matrix for $P$ transformation is used to decrease the computational complexity associated with the conversion to homogeneous, because the last row of transformation matrix remains $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$. For the calculation of intersections with the objects in the scene the directional vector has to be normalized.

The generation of the rays can be also used a method, which cast rays directly through the image view plane which eliminates the necessity of transformation and speeds up the process. This method is convenient only for the pinhole camera model and its modification. The effective speed up of the method in whole process of ray–tracing is totally negligible comparing with the huge amount of intersection calculations. æ

# Chapter 4

# Simulation of the Camera

The objects in the images rendered in normal way are always in sharp focus. This is caused by using the pinhole camera model. In real world cameras and also the human eye have a finite lens apperture and their images on the sensor have a finite depth of field. For human eye this phenomena is not so perceptible, hence the human can focus sharply only on small part of viewing angle projecting onto the fovea. The photographic cameras on the other hand can capture the picture with only a small part of the image at sharp focus. The depth of field can be an unwanted artifact, but it can also be a required by the artist for some desirable effects. Using computers the depth of field can be simulated even in wider range than in the reality.

## 4.1 Reasons for Simulation

The purpose of such synthetic images, which in sense incorporate the constraints of an optic system is twofold:

1. it gives the ability to capture the viewer's attention to a particular segment of the image, it means, it allows selective highlighting either through focusing or other optic effects used in photography.

2. it permits adaptation of many techniques used in cinematography for animated sequences as fade out, fade in, lens distortion, depth of field.

## 4.2 The Basics of Lens Camera

It is necessary for this chapter to comprehend the view function of photographic camera. Each camera consists of converging lens or the system of lenses, which altogether can be considered as one converging lens. Lens has its optic properties geometrically relative its optic axis. One of them is a focal distance. When the

light beam, in our case called ray, passes through the lens in parallel with the optic axis, then after transition to the other halfspace determined by the lens position and the bisects on the optical axis the point called a focus. The are two foci, one for the front of the lens and one for the rear. The distance between the focus and the lens is a focal length. This is true only for the ideal type of lens, which is infinitely thin. In the reality the situation is a little more complicated. The 3focal distance is measured from the focus to the principal point on one lens's side.

In simple double convex lenses two principal points are somewhere inside the lens (in our case $\frac{1}{n}-th$ the way from the surface to the center, where $n$ is the index of refraction), but in a complex lens they can be almost anywhere, including the outside the lens, or with the rear principal point in front of the front principal point. In a lens with elements mutually fixed to each other, the principal points are fixed relatively to the glass.

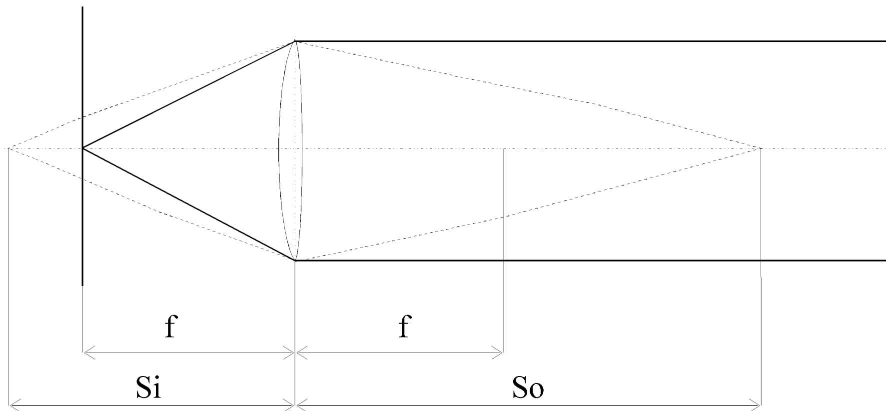


Figure 4.1: Lens Law

In zoom or internal focusing lenses the principal points may move relative to the glass and each other when zooming or focusing. If we define the following symbols:

$S_o$ .. subject (object) to front principal point distance

$S_i$ .. rear principal point to image distance

$f$ .. focal length

$M$ .. magnification, then

Thin lens equation:

$\frac{1}{S_o} + \frac{1}{S_i} = \frac{1}{f} \qquad (GaussianForm)$

$(S_i - f).(S_o - f) = f^2 \qquad (NewtonianForm)$

$M = \frac{S_i}{S_o}$

$M = \frac{f}{S_o - f} = \frac{S_i - f}{f}$

If we interpret $S_i - f$ as the extension of the lens beyond infinity focus, then we see that it is inversely proportional to a similar extension of the subject. For rays close to and nearly parallel to the axis (these are called **paraxial** rays) we can approximately modell most lenses with just two planes perpendicular to the optic axis and located at the principal points. We define more symbols:

$D$ .. diameter of the entrance pupil, i.e. diameter of the aperture as seen from the front of the lens

$N$ .. f-number (or f-stop) $D = \frac{f}{N}$

Light from a subject point spreads out in a cone whose base is the entrance pupil. The entrance pupil is the virtual image of the diaphragm formed by the lens elements in front of the diaphragm. The fraction of the total light coming from the point that reaches the film is proportional to the solid angle subtended by the cone. If the entrance pupil is distance y in front of the front principal point, this is approximately proportional to $\frac{D^2}{(S_o - y)^2}$.

The light from a single subject point passing through the aperture is converged by the lens into a cone with its tip at the film (if the point is perfectly in focus) or slightly in front of or behind the film (if the subject point is somewhat out of focus). This situation is illustrated in Fig. 4.2. In the out of focus case the point is rendered as a circle where the film cuts the converging cone or the diverging cone on the other side of the image point. This circle is called the circle of confusion. The farther the tip of the cone, that is the image point, is away from the film, the larger is the circle of confusion.

Consider the situation of a **main subject** that is perfectly in focus, and an **alternate subject point** this is in front of or behind the subject. We define the following symbols:

$S_{oa}$ .. alternate subject point to front principal point distance

$S_{ia}$ .. rear principal point to alternate image point distance

$H$ .. hyperfocal distance

$C$ .. diameter of circle of confusion

$c$ .. diameter of largest acceptable circle of confusion

$N$ .. f-stop (focal length divided by diameter of entrance pupil)

$D$ .. the aperture (entrance pupil) diameter $D = \frac{f}{N}$

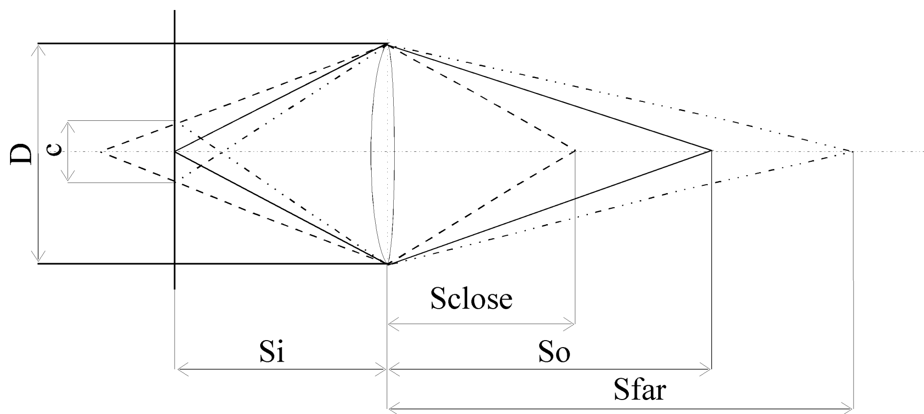


Figure 4.2: Depth of field in general case

The diameter of the circle of confusion can be computed by similar triangles, and then solved in terms of the lens parameters and subject distances. When $S_o$ is finite, then the diameter of the circle of confusion for alternate subject point is:

$$C = D.\frac{(S_{ia} - S_i)}{S_{ia}} = f^2.\frac{(\frac{S_o}{S_{oa}} - 1)}{N * (So - f)}$$

When $S_o \to \infty$, then $C = \frac{f^2}{N.S_{oa}}$

In this formula $C$ is positive when the alternate image point is behind the image plane (i.e. the alternate subject point is in front of the main subject) and negative in the opposite case. In reality, the circle of confusion is always positive and has a diameter equal to $|C|$. The depth of field is asymmetric.

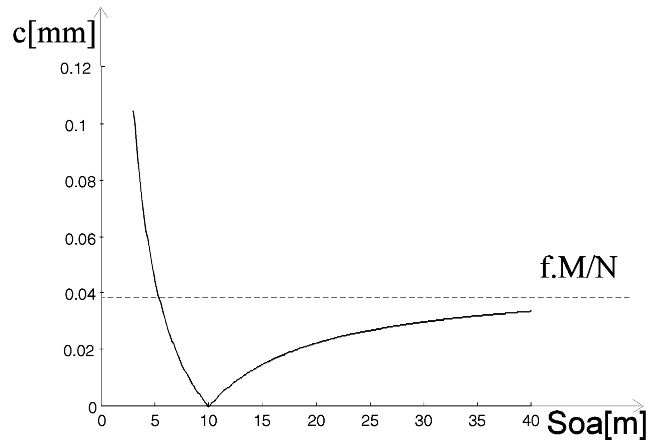


Figure 4.3: Diameter of circle of confusion

If the circle of confusion is small enough, given the magnification in printing or projection, the optic quality throughout the system, etc., the image will appear to be sharp. Although there is no diameter that marks the boundary between fuzzy and clear, 0.03 mm is generally used in $36 \times 24mm$ film frame as the diameter of the acceptable circle of confusion.

If the lens is focused at infinity (so the rear principal point to film distance equals the focal length), the distance to closest point that will be acceptably rendered is called the hyperfocal distance.

$$H = \frac{f^2}{N.c}$$

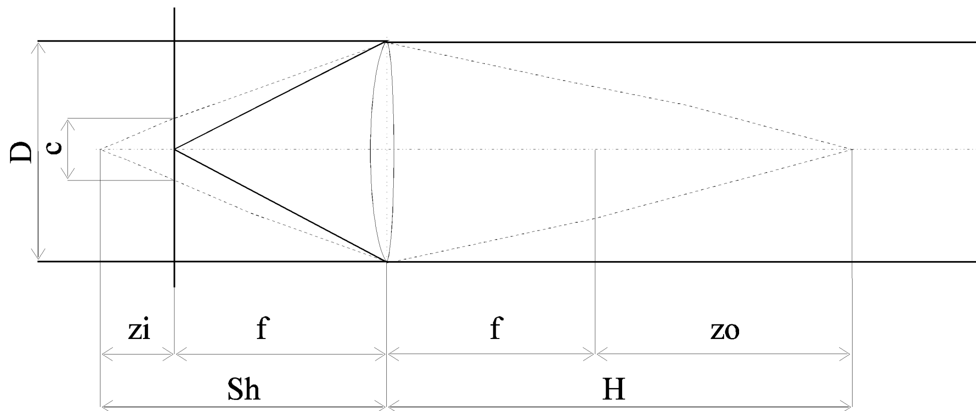


Figure 4.4: Hyperfocal Distance

If the main subject is at a finite distance, the closest alternative point that is acceptably rendered is at distance

$$S_{close} = \frac{h.S_o}{H + (S_o - F)},$$

and the farthest alternative point that is acceptably rendered is at distance:

$$S_{far} = \frac{H.S_o}{H - (So - F)},$$

except that if the denominator is zero or negative, $S_{far} \to \infty$. We call $S_{far} - S_o$ the rear depth of field and $S_o - S_{close}$ the front depth of field.

If a subject is in focus the same way with two different lenses, the shorter focal length lens will have less front depth of field and more rear depth of field at the same effective f-stop (To a first approximation, the depth of field is the same in both cases.).

Another important consideration when choosing a lens focal length is how a distant background point will be rendered. Points at infinity are rendered as circles of size $C = f.\frac{M}{N}$. So at constant subject magnification a distant background point will be blurred in direct proportion to the focal length. As the infinity can be in practical cases called the distance about 500-700 multiple of the focal length.

There are some extensions of the calculation of the depth of field in the photographic theory. They respect the pupil magnification and belows factor, which bear on light intensity. For the computer simulation the depth of field is not necessary to take this extension into account, because the light can be only simulated roughly by the finite number of rays and because the color range is incomparably smaller than in photography.

## 4.3 Potmesil Postprocessor

Depth of field can be simulated by the method proposed by Potmesil. The image is at first rendered in sharp focus and the z-distance from the camera is stored for each pixel. In the postprocessing phase there are computed the diameters of circle of confusion and the intensity of each pixel is distributed by this circle. The postprocessor has one advantage, which is the possibility to compute more sequences from the original image with different apertures and the object point in focus. The disadvantage lies in the infidelity of the blurred images.

For the pixel in the center of the image is evaluated the bad color, because the object O2 is behind object O1 is not visible by casting one ray. The next fault consists in wrong view of objects in the mirror, who's distance from the lens

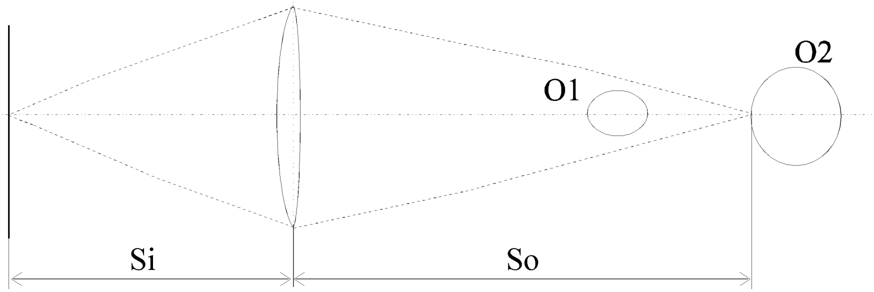


Figure 4.5: Potmesil postprocessor fault

differs from the distance of the mirror. The next drawback is wrong visualization of two overlapping objects, because the objects are sampled in frequency of raster image. It is not possible to estimate the portion of the rear object spreading over the closer one. The method is very approximate, although it can give sufficient result in some cases. For rendering of images in higher fidelity this method is not advisable.

## 4.4 Camera Preprocessor

To increase naturality the depth of field I have designed method, which reduces the imperfection in the image. This method is more time complex, but it is the cost paid for the quality of the image, which integrates the antialiasing as the natural part of the image synthesis. It is based on a classical ray–tracing with the simulation of the real camera. The model for traditional camera in ray–tracing cannot be used. I use the simulation of real lens camera with all aspects included. Simply said the method lies in the generation of more rays for one pixel, which are propagated through optic system and then through the scene. The color is computed as the average from these rays.

### 4.4.1 Calculation of Rays Inside the Camera

For calculation of a ray inside of the camera the camera coordinate system is used. The optic axis of lens is identical with z-axis of the camera coordinate system and the original position of the film frame or the image lies in the $xy$-plane. There are some basic element for composing the camera. Among the refractive ones belong the ideal lens and the refractive surface placed inside of the camera. The ray is cast from the film frame and passes through all elements one by one and then leaves the camera. Then the ray is transformed to the scene coordinate space as was described in previous chapter. The important part is mutual positioning the elements inside the camera in the way to get some acceptable results.

### 4.4.2 Generation of Rays inside Camera

In real world the light beams passes from the scene trough the camera objective to the film. The objective arranges an infinite number of rays. The points in focus are projected on the file as points. The points laying out of focus are projected on the film as a circle of confusion. If the ray is cast from the film frame as is usual in ray-tracing, then the following problem arises. The origin of the ray is clear enough, because it lies on the film frame and it corresponds to the pixel. The much more difficulties are with the ray direction vector. It could be solved by the naive approach, that is, for each pixel is generated the set of rays in any direction and only the rays successfully passing through all optic elements are sent to the next processing. The generation process for one pixel can be stopped after the specified number of rays are successfully cast to the scene. This approach is very time consuming, however it works in all camera systems correctly. It is caused by the generation of the rays in no prerequisite direction.

The improvement of the mentioned primitive approach is based on the definition of the restricted plane area perpendicular to the main optic axis. The ray direction vector is oriented in the way the rays pass from the origin point through this restricted plane area further called a **generation area**. In general case it could be the three–dimensional object, but it is not necessary, because the results acquired by delimitation by two–dimensional area are quite satisfactory.

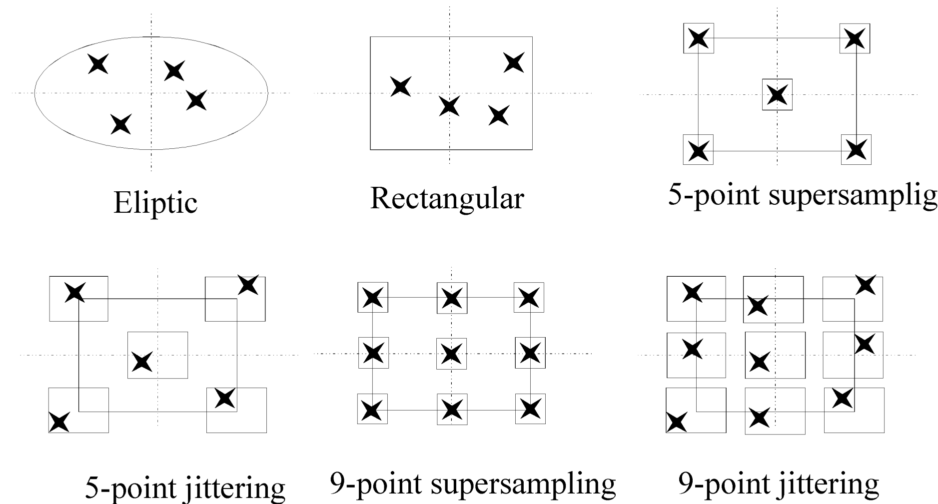


Figure 4.6: Generation area types

The generation area delimits strictly the viewing frustrum for one pixel. There are some methods to assign sharply the points in the area, which are actually used for the vector. These methods are illustrated in Fig 4.6.

The size of the area corresponds the aperture of real camera. The result of it is the depth of field. The important parameter is the position of the center of the

generation area in the camera coordinate system. The suitable position is in the rear principle point of the objective used. The other locations is also possible, but it in consequence the synthetized image could be innaturaly distorted. It can be used for some very special effects, but tuning of parameters requires mostly a lot of time.

### 4.4.3 Calculation the Ray Passing Through Ideal Lens

The camera optics can be simulated by more means. One technique simulates the refraction surfaces of the real lenses. It enables to involve the lens distortion and the other effects caused by real optics imperfections. The effort of objective manufacturers is to produce the system with minimized optic imperfections. This thin lens equation approach can be easily modelled. The ray is determined by its

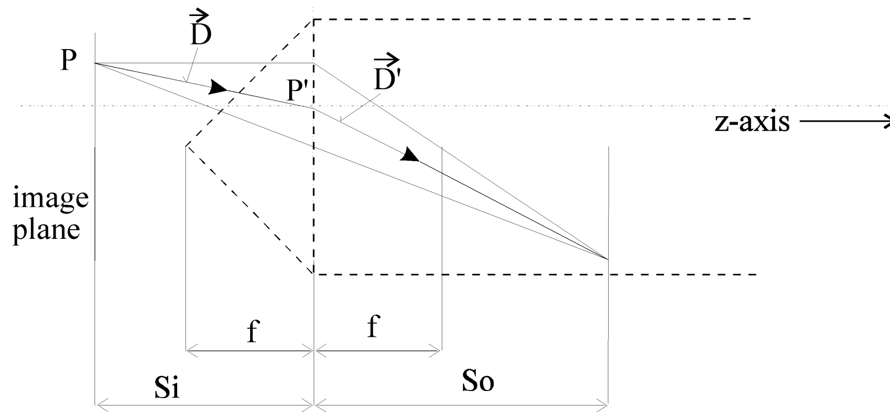


Figure 4.7: Ray through ideal lens

origin $P$ and the normalized directional vector $\vec{D}$. A thin lens is defined by the diameter $d$, focal length $f$, its position $C_z$ provided that its optic axis is parallel with the $z$ axis. The new ray is defined by $P'$ and $\vec{D}'$. The ray intersects the plane of lens at point $P'$. The calculation of this point is simple. By thin lens equation the point in the subject point $B$ and the new vector $P'$ is normalized vector between this two points:

If $(P_z - C_z) < f$ .. the origin before focal plane .. stop

$k = \frac{P_z - C_z}{D_z}$

$P' = C + k.\vec{D}$

If $P_x'^2 + P_y'^2 > d^2$, then the ray intersect the lens out of the bounds - stop

The point $B$ is computed:

$$t = f.\frac{P_z - C_z}{P_z - C_z - f}$$

$$B = t.(P' - \vec{C}) + P'$$

Then vector is $\vec{D'} = B - P'$ and it can be eventually normalized. This formulas enables to simulate both types of ideal lenses, the converging and the diverging.

### 4.4.4 Calculation of Ray Passing Through Refractive Surface

This approach simulates the real situation in the camera. The ray intersect the refraction surface at point $C$, the surface normal $\vec{N}$ is oriented onto the halfspace with point $P$.

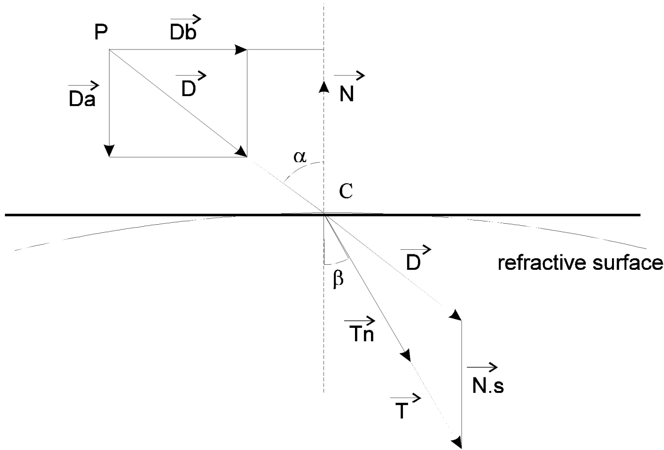


Figure 4.8: Ray passing through refractive surface

The vector $\vec{N}$ and $\vec{D}$ are normalized. The vector $\vec{D}$ can be decomposed into two mutually perpendicular vectors. Vector $\vec{D}_a$ is parallel with $N$ and second vector $\vec{D}_b$ lies in the plane tangent to the surface in point $C$. The refracted vector $\vec{T}$ is expressed as:

$\vec{T} = \vec{D} + \vec{N}.s$, where $s$ is real coefficient. This linear composition of vectors fulfills the requirement given for vector $\vec{T}$, which has to lie in the plane determined by vectors
$\vec{D}, \vec{N}$ and point $C$. The deduction starts by Snell Law:

$$\frac{sin(\alpha)}{sin(\beta)} = \frac{n2}{n1} = n$$

$$|\vec{D}| = |\vec{N}| = 1$$

$$|\vec{D}_a| = cos(\alpha) = |\vec{D}.\vec{N}|$$

$$|\vec{D}_b| = sin(\alpha) = \sqrt{1.0 - |\vec{D}_a|^2}$$

$$sin(\beta) = \frac{|\vec{D}_b|}{\sqrt{(|\vec{D}_a| + |\vec{N}|.s)^2 + |\vec{D}_b|^2}}$$

$$(|\vec{D}_a| + |\vec{N}|.s)^2 + |\vec{D}_b|^2 = (\frac{|\vec{D}_b|}{sin(\beta)})^2$$

Because $sin(\beta) = \frac{sin(\alpha)}{n}$, then

$$(|\vec{D}_a| + |\vec{N}|.s)^2 + |\vec{D}_b|^2 = n^2$$

$$s = \sqrt{n^2 - (1.0 - |\vec{D}.\vec{N}|^2)} - |\vec{D}.\vec{N}|$$

So the form of the result equation is:

$$\vec{T} = \vec{D} + (\sqrt{n^2 - 1.0 + |\vec{D}.\vec{N}|^2} - |\vec{D}.\vec{N}|).\vec{N}$$

The factor $k = n^2 - 1$ can be precalculated. Its positive value signs the refraction vector is aimed towards the normal. It should be normalized for next processing at last. The algorithmic complexity is $(10\pm, 10\times, 3\,, 2\sqrt{})$ if the $k$ is precalculated. This complexity can be compared with the complexity of computation the reflected vector on the same ray upon a surface, which is calculated as $\vec{R} = -\vec{D} + \vec{N}.(2.\vec{D}.\vec{N})$ The algorithmic complexity for reflected vector is $(6\pm, 12\times)$. For computation of $T$ can be used the much more complicated expression, which composes $\vec{T}$ by another way by computing cross product of $\vec{N}$ and $\vec{D}$. Its complexity is $21\pm, 36\times, 5\,, 5\sqrt{}$). The algorithm derived in this paper is about 2.5 computationaly less complex.

### 4.4.5 Elements of Camera

The camera can be constructed from different parts based on theory above. The camera is constructed by putting the elements one by one to the camera body simplified hear to the main optic axis $z$. The order of the elements facilitates the computation of the ray path. Certainly it can be automatized, but it has no practical meaning. The design of the camera is not a stochastic process and it could be very hard to construct the camera with required optic qualities. The designer would have to accurately know the order of the refraction surfaces inside the camera. Let us enumerate these elements with detailed description in C-like language. All parameters are measured in SI units. The camera coordinate system is right handed, the $y$ axis is aimed at the top of the film.

**FILM_POSITION(float xmin, float ymin, float xmax, float ymax)** defines the position of film frame (image plane) in the plane xy. For standard cinefilm 24x36mm corresponding values are -0.018, -0.012, 0.018, 0.012.

**FILM_Z_POSITION(float z)** .. defines the position of the film on the z axis. This value is changed by the functions for focusing stated in the next.

**GEN_AREA(Loc(x, y, z), int Type_Area, int MaxRay, float Xmin, float Ymin, float Xmax, float Ymax, float Delta_X, float Delta_Y)** .. determines the type, size and location of the generation area, which the rays are cast towards. The generation area lies in plane parallel to plane xy. Type_Area defines the method for generation of rays. The allowable types are following:

- **0** .. one ray through the center of generation_area for one pixel and other elements in camera are ignored. It is classic ray-tracing.
- **1** .. one ray trough the center of generation_area for one pixel. This method demonstrates the imperfection of camera lens system.
- **2** .. the set of rays to the ellipse qualified by Xmin, Ymin, Xmax, Ymax. The number of rays is MaxRay and the positions inside the ellipse are determined by pseudo-random generator.
- **3** .. the same as previous, but the area has rectangular shape.
- **4** .. 5-point supersampling onto the center and the corners of the rectangle
- **5** .. 5-point jittering
- **6** .. 9-point supersampling into the center, the corners and the middles of the sides of the rectangle.
- **7** .. 9-point jittering

**FILM_GEN_AREA (Loc(x, y, z), int Type_Area, float Xmin, float Ymin, float Xmax, float Ymax)** .. defines the type, size and location of the generation area on the image plane, which the rays are cast from. This method allows the antialiasing even for the scene objects in focus. The location is relative positioning to the center of the pixel. The allowable types are following:

- **0** .. the origin of the ray is in the center of pixel plus the location of the film generation area.
- **1** .. the origin of ray has to lay in the area circumscribed by ellipse in the plane parallel the $xy$ plane. The size of the area is determined by Xmin, Ymin, Xmax, Ymax.
- **2** .. the area for the film position is the rectangle, the rest of parameters corresponds to the option for ellipse.

It is convenient to choose the size of the film area so the neighbouring areas do not overleap.

**LENS(Loc(x,y,z),int Dir, float Rad, float Delta_z, float N)** .. defines a spherical cap refraction surface by its location, diameter of sphere surface. The parameter Dir determines the ray has to pass from the outside of the imaginary sphere to inside or the other way round, which speeds up the calculation. Parameter Delta_z is the height of spherical cap. N is relative refraction index, which is greater than one if the ray passes through from the environment opticaly sparsier to the opticaly densier one.

**LENS_ELLIPTIC(Loc(x,y,z), int Dir, float kx, float ky, float Rad, float Delta_z, float N)** .. defines an elliptic cap refraction surface. The kx respectively ky is the ratio between the size of x-radius respectively y-radius and z-radius. Other parameters are the same as above. These elements is used to simulate the lenses in panoramatic cameras.

**LENS_PLANE(Loc(x,y,z), Normal(xn,yn,zn), int Dir, float Rad, float Kx, float Ky, float N)** .. defines refractive plane. Some types of lenses (planoconvex and planoconcave) are formed on one side by the plane surface. The valid area for passing ray through is circumscribed by ellipse given by intersection of ellipsoid and the lens plane.

**LENS_IDEAL(Loc(x,y,z,), float Rad, float Kx, float Focal_length)** .. defines the ideal lens with optic axis parallel the optic axis. Positive and negative Focal_length can be defined. The valid area is circumscribed by the ellipse with radius Rad in x-axis radius Kx*Rad in y-axis on the lens plane.

**N_DIAPHRAGM(Loc(x,y,z), int Type, int N, float Xmin, float Ymin, float Xmax, float Ymax)** .. defines the diaphragm perpendicular to main optic axis. The type defines the shape of diaphragm:

- **0** the rectangle related to the diaphragm center
- **1** the ellipse circumscribed in the rectangle
- **2** the polygon circumscribed inside the ellipse with N vertices, this method simulates veritable diaphragms used in photography

**COLOR_ADAPT(int Type)** .. the color is calculated as the average from the colors for each ray. If the generation area is placed improperly, then some rays can be lost inside the camera and they do not pass to the scene. It is caused by total refraction, the size and positioning of the camera elements and also by the restriction given by the diaphragm. The ray behaviour is similar to the vignetation in photography. This method describes how to handle the calculations of result colors.

**0** .. the color is calculated as the average of colors for each ray regardless some rays lost inside the camera.

**1** .. the color is calculated as the average of colors for rays successfully sent to the scene.

**SPECIAL_JITTERING(bool F)** .. this command enables to utilize the generation_area in the place of film_generation_area. The original generation area is not used. This option enables classic jittering method without the possibility of depth of field.

### 4.4.6 Methods for Focusing and Auxiliary Methods

The construction of the camera is very important. If we want to create the picture, then it gives rise to a problem, how to position the film frame to focus at some specified distance or with required depth of field. The theoretical relations were already mentioned, but they are valid only for thin lenses. In this case parameters can be evaluated manually and the diaphragm alternatively the generation area and the film z-position can be set. This procedure is a little tedious. In the case the camera is composed by a set of refractive surfaces, the manual method cannot be used. For these reasons I have designed the iterative method for calculation the focal length of the constructed optic system. Next iterative methods allow setting the required depth of field and point in focus. They are:

**FOCUS_INFINITY(void)** .. sets the film z-position the center pixel is focused on infinity. This method in the same way as the other method supposes the tested optic system is divergent.

**FOCUS_AT(float D)** .. sets the film z-position the center pixel is focused at the distance D measured from the objective's outlet.

**COUNT_FOCAL_LENGTH(void)** .. for given objective is evaluated focal length.

**AUTOFOCUS(void)** .. for concrete scene there are evaluated the z-depth of the objects in the center of image plane. It is focused on the average these z-depths. There are taken into account only points intersecting the objects in the scene.

**SET_SHARPNESS_ID(float D, float T, float Max_c, float Rad)** .. sets the position and the size of the generation area the camera is focused at distance D sharply and at distance D+T the circle of confusion has the radius Max_c. The distance is measured from camera outlet. The parameter Rad defines the maximal radius of the generation area used, because it cannot be bigger than the size of the elements inside the camera.

**SET_SHARPNESS_FROM_TO(float From, float To, float Max_c, float Rad)** .. sets the position and the size of the generation area the camera is focused at distance From and at distance To with radius of the circle of confusion Max_c.

**COUNT_FOCUS_DISTANCE(void)** .. evaluates automaticaly the focal length, the position of the focus and the principal points of the optic system.

These methods are all iterative and some of them uses other. Let us outline how they work.

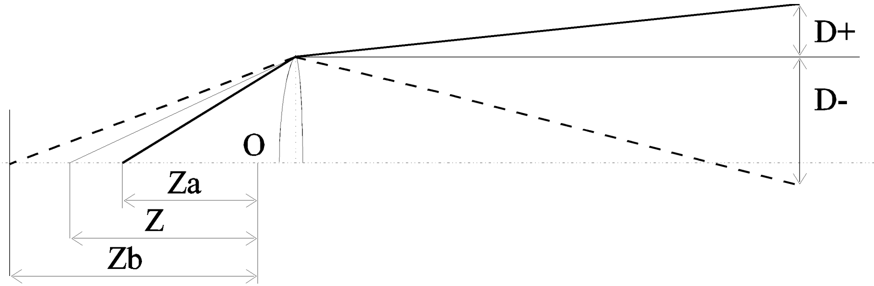


Figure 4.9: Focus at infinity

Inside of the generation area, close to the main optic axis is selected point with zero x-element, on which the ray is aimed to. The origin of the ray for first calculation is on the z-axis at the back side of camera. Then the direction of the ray is determined as the difference of these two points. The ray is cast through the optic system. The important result of output ray is the sign of y-element of the directional vector. If the y-element is positive, then the point on the image plane is in front of focus point and vice versa. The focus point is computed using binary search until required accuracy is achieved. The method "focus at" works similarly.

The method for the focal length evaluation is a little more complicated. In the first step is calculated the position of the ray for infinity $Z_b$ and for point at far distance e.g. at 1000 multiple of expected focal distance $Z$. It is done by previously mentioned methods "focus at" and "focus infinity". The position is determined by z-distance from the camera coordinate system. This values can be put into the system of two equations with two unknown variables, if we suppose the validity of thin lens law:

$\frac{1}{f} = \frac{1}{P_i - Z}$ .. ray focused at infinity

$\frac{1}{f} = \frac{1}{P_i - Z_B} + \frac{1}{D}$ .. ray focused at distance D

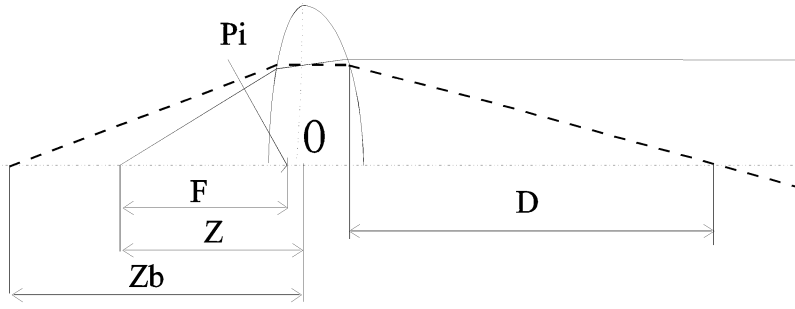


Figure 4.10: Count focal length

$f = D.\frac{P_i - Z_B}{P_i - Z_B + D}$ and $f = P_i - Z$

$P_i^2 + P_i.(-Z_B - Z) + Z.Z_B + D.Z - D.Z_B = 0$

$P_i = \frac{(Z_B + Z \pm \sqrt{(Z_B+Z)^2) - 4.(Z.Z_B + D.Z - D.Z_B)})}{2}$

The smaller value of the quadratic equation determines the z-position of the rear principal point. Then focal length is evaluated simply $f = P_i - Z$.

The method "set_sharpness_from_to" for setting the depth of field works a little different way. The iteration process is nested. In the lower step there is evaluated the position of the origins the rays focused at "from" resp. "to" z_from resp. z_to.

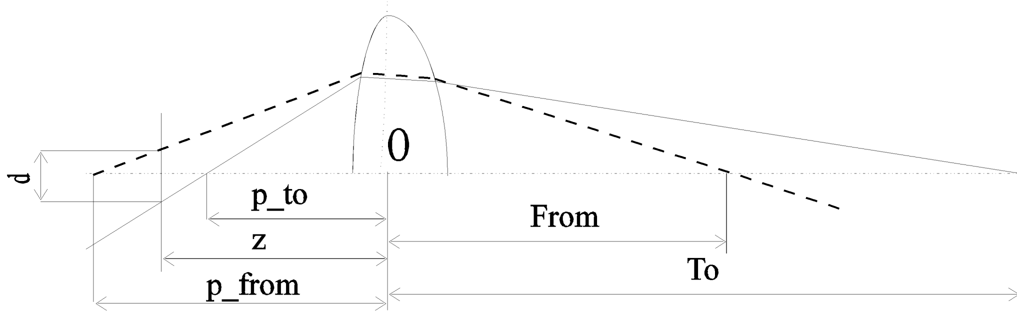


Figure 4.11: Focus from to

The rays direction is determined by the point on the generation area for an initial position on y-axis $py$. The diameter of the circle of confusion is then calculated from the presumption, it is symmetric:

$p_y - p_y.\frac{p_z - z}{p_z - p_{from}} = -p_y + \frac{p_z - z}{p_z - p_{to}}$

The position of the film on the z-axis is

$$z = p_z - \frac{2.0}{\frac{1.0}{p_z - p_{from}} + \frac{1.0}{p_z - p_{to}}}$$

The diameter $d = 2.0.(p_y - p_y.\frac{p_z - z}{p_z - p_{from}})$

The diameter is the input for the second level of the iteration. It is required to be the size specified by Max_c and it is caused by the value of y. If the diameter size is close enough Max_c, then the iteration process is terminated. The method "set_sharpness_id" works similarly.

Zooming process in photography is done by sophisticated change the mutual positioning of the camera elements. It can be simulated by the same procedure, but it is very demanding process for refractive surfaces. In addition to real world the simulation of the camera also allows to change the geometrical properties of the camera. In the case of the ideal lens the viewing angle can be adjusted easily by the change of focal length.

### 4.4.7 Camera Elements Design

The design of the camera is a little difficult particularly for objective formed by refraction surfaces. Without computers the design of any usable lens system with more than seven refraction surfaces takes several months. For the purposes of the computer simulation lens geometrics is available in the books concerning the field "Photographics Optics". The second way is to design own lens. I recommend to

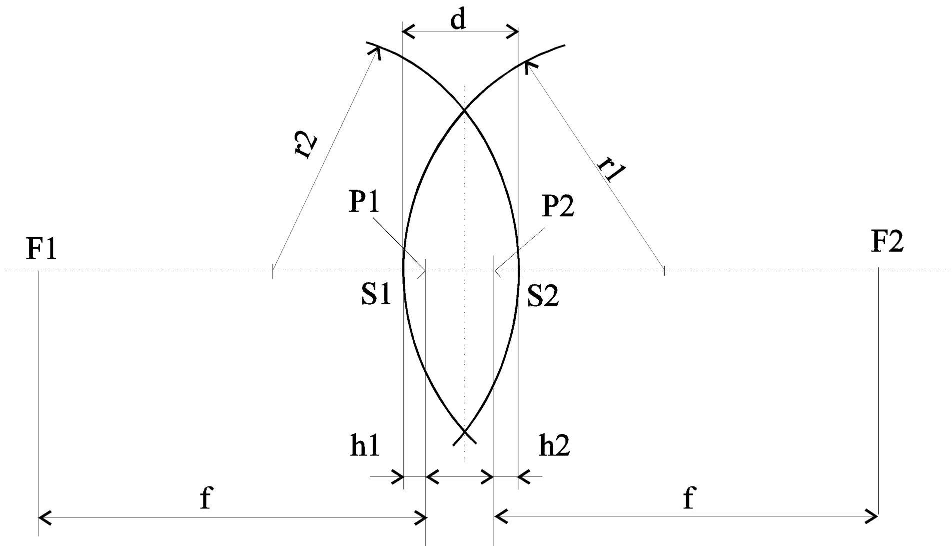


Figure 4.12: Lens Design

design single converging lens only. The successful design of systems with more

than two refractive surfaces can be for the first time a brain-teaser. The single lens can be convex, biconvex, planoconvex or positive meniscus. Optic parameters for given geometry and refraction index $N$ are computed by the following formulae:

$$R = (N-1).[N.(r_2 - r_1) + d.(N-1)],$$

$$f = \frac{n.r_1.r_2}{R}$$

$$h_1 = \frac{r_1.d}{R}, h_2 = \frac{r_2.d}{R}$$

Two lenses of focal length $f_1$ and $f_2$ with the identical optic axis with closer principal points in distance $e$ create one optic system of the focal length

$$f = \frac{f_2 + f_1 - e}{f_1.f_2}$$

.

If both of them are converging lenses, then the result dependently upon $e$ forms diverging or converging lens system.

The refraction index of optic glass is in the praxis from 1.3 to 2.2, but typically 1.6. Because the first estimation of lens geometric attributes is not quite easy for formulae above, the following example is given for simple convex lens:

$$r_1 = 0.1[m], r_2 = 0.051666[m], d = 0.005[m], n = 2.2 \rightarrow f = 0.050[m]$$

### 4.4.8 Complexity Evalution of Camera

The processing of one ray passing through the camera is not usually time consuming. For single converging lens the time complexity for calculating a ray is $20\pm, 20\times, 6\,, 4\sqrt{)}$. The refraction and the calculation of the intersections the sphere cap $22\pm, 20\times, 2\sqrt{)}$ corresponds the complexity of calculation approximately 5 intersections with sphere in the scene. The number of the intersection calculation with objects in the scene is for one ray usually much bigger, even they may be organized in some space structures. Completely different situation is for complexity using the depth of field property in the synthetized image. For one pixel the $n$ rays are sent to the scene and the complexity is $n$-times bigger. Therefore it is convenient to use the adaptive technique for the rays generation. The adaptivity evaluation is based on color distance of two firstly evaluated rays. The rays are aimed at the opposite corners of the generation area for 5 and 9 point methods. If the color distance is acceptably small, then no additional rays are cast to the scene. This adaptive behaviour for casting the rays can perceptibly speed up the computation and the image is mostly of the same quality as counted without this optimatization.

### 4.4.9 Camera Preprocessor Modul

The methods for camera were implemented in ANSI-C language. The modul is implemented as the preprocessor utilizible with common ray–tracers. The camera setup has to be done at the beginning of the rendering. The input for

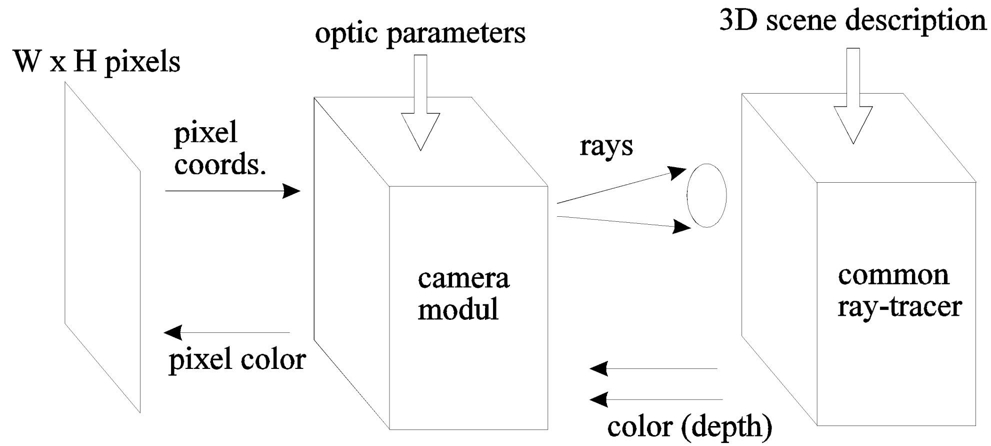


Figure 4.13: Connection of camera modul with common ray-tracer

the preprocessor is the coordinates of the pixel in the image. The output is the ray specified by the position and the directional vector. The connection to the common ray-tracer is done by the function Trace(level,weight,ray,&color,&z). The return value is the color evaluated by common ray–tracer. If autofocusing is required, then the ray–tracer also has to return the distance to the first intersected object. æ

# Chapter 5

# Parallel Ray–tracing

As was stated above, the ray–tracing is one of the most time consuming processes used in computer graphics. If we notice increase of the complexity caused by additional features as the camera preprocessor, then it gives unacceptable time for rendering most of scenes even the quality of such images is very high. After utilization of all possible acceleration techniques on sequential algorithm it remains only one way how to speed up the rendering process. It is the task distribution onto more processors, that is discussed in this chapter.

## 5.1 Approaches to Parallelization of RT

The approaches used to the present days for making sequential ray–tracing parallel can be categorized into groups. The reasons for parallelization are au fond twofold. The first one is the distribution of the computation demands among computation units. Second reason is the distribution of the scene database in the distributed memory. The scene can be very large and the data structures associated with the objects particularly the textures can allocate a huge amount of the memory. Hence the solution and its properties must be evaluated and compared with regard to both problems mentioned.

The first classification is done by the type of the parallel architecture used:

**Vector processors** the machines as Cray Y-MP C-90 and others can be used. They are expensive and are not generally available. Nowadays they are often displaced by other architectures. They are very expensive because of the shortening the instruction cycle to available maximum to the predjudice the number of processors. Although they are designed for vector and matrix operations, it is hard to utilize them for the purposes of RT.

**Special Purpose Hardware** the design and production of the special hardware is very tricky problem with regard to its flexibility and extensibility. The special hardware is mostly used as the additional computational unit inside

normal type of computers to accelerate the calculation of the intersection the rays with objects in the scene. They can be appropriate solution for the interactive rendering workstation, but the development of such hardware is very costly and the solution can become out of date comparatively quickly due to the technology advacement.

**Computers Connected via a Network** this solution seems to be highly effective, because most workstations consume about 95 percent of their time in the idle loop. This is caused by utilisation of the processor unit by relatively easy tasks as text editing. The ray–tracing computation can be block by the communication load between the computers connected via a bus. The bottleneck factor highly depends on a granularity of the data distribution and the method used for load balancing.

**General Purposes Multiprocessors** these architectures provide high degree of flexibility, performance and scalability. They usually support wide range of granularities of parallelism and a variety of different models of programming. With current state of system software the ray-tracer application can be mostly easy ported to the different multiprocessor platform.

The second classification concerns the model for utilization the parallel environment for ray–tracer algorithm:

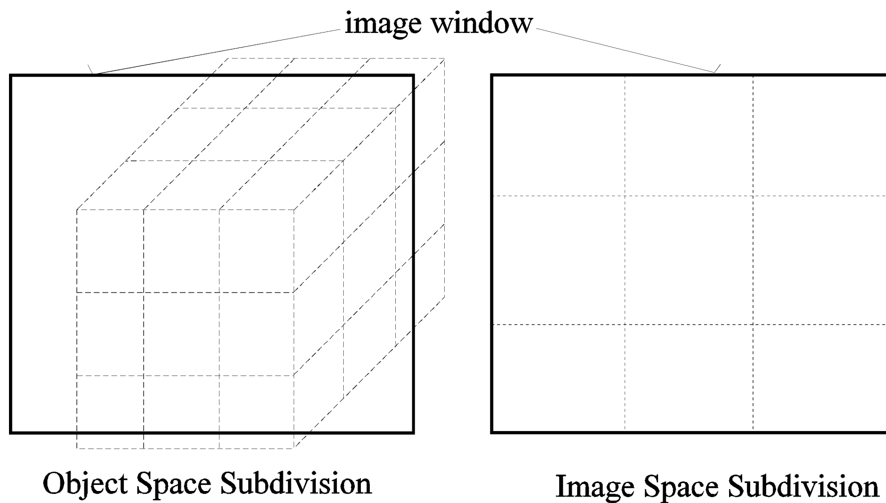


Figure 5.1: Approaches to load distribution in RT

**Object Space Subdivision** the solution is based on the subdivision scene space with objects onto the cells assigned to computation units. The ray is interpreted as the message passing between the neighbouring cells. The objects

database is distributed between the memory of computational elements. It can be convenient for the large amount of objects data. The speed–up of this solution can be depriciated by the growth of communication demand with the increase of granularity. It is the main limitation of this method, especially for a set of workstations communicated through single network bus. Another problem of this solution is load balancing, because the computation of an image is viewpoint dependent and cannot be credibly estimated before the beginning of the computation.

**Image Space Subdivision** the first method consist in distributing the computation by subdivision of image space, which is divided into a number of non-overlapping regions assigned by convenient strategies to the computation units. This approach supposes that the object database is available to all processors, because the rays passing through a pixel in one corner of the image can test the objects rendered in the place of the opposite corner. It is caused by generation secondary and shadows rays. It has to be done by the sharing data between computation units and the solution for sharing depends on the hardware architecture used. For practical solution on some architectures the object database has to be duplicated. These solution is very flexible and general, but for some architectures is hardly scalable from a certain extent caused by hardware limitations.

## 5.2 Metrics for Processor Load and its Distribution

The main effort in management of the distribution the computational demands in multiprocessor environment is aimed to make the load placed on each processor roughly uniform. It is necessary to quantify what is the load and how it can be measured.

The traditional approach in computer science is to evaluate the task complexity. It could be transformed to the terms of CPU cycles or the times for particular type of processor. Unfortunately the ray–tracing is highly data dependent task and the complexity of the algorithm cannot be simply considered proportional to the number of pixels. It gives cause to evaluate the complexity of the algorithm dependently on the data set given. This specification incorporates complete scene definition including light sources, the camera positioning and setting in the scene space, the rendering parameters as the depth of recursion and also the resolution of result image. This is the only one way guarantees the mutual comparability of sequential and concurrent solution of ray–tracing algorithms. The load placed upon each processing unit can be expressed by three ways. The first one concerns the number of rays being processed to fulfill the task. The second one qualifies

the number of references to object primitives within the task. The third criteria is the number of different object primitives within the task, which don't need be the same as the previous one. Let us preassume to divide the tasks over the image space. The rectangular area of the size $w \times h$ and the coordinate $p_x, p_y$ is used for the definition of the metrics. Let the number of rays required for the color determination for one pixel to be $r(x, y)$ and the number of scene data references within $d(x, y)$. The number of the referenced objects $o(x, y)$.

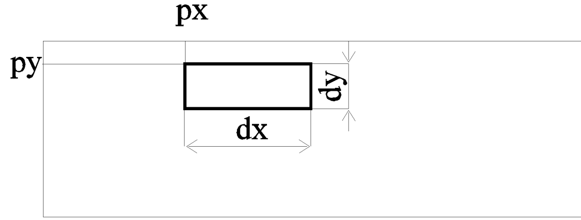


Figure 5.2: Metrics area

Let us define metrics **ray density** over the rectangular area of the image as follows:

$$D_r(p_x, p_y, d_x, d_y) = \sum_{x=0}^{dx} \sum_{y=0}^{dy} r(p_x + x, p_y + y)$$

The **data reference density** is defined similarly:

$$D_d(p_x, p_y, d_x, d_y) = \sum_{x=0}^{dx} \sum_{y=0}^{dy} d(p_x + x, p_y + y)$$

The **object reference density** is the number of unique object primitives referenced:

$$D_o(p_x, p_y, d_x, d_y) = \sum_{x=0}^{dx} \sum_{y=0}^{dy} o(x, y) \text{ exclusively}$$

The objects already involved in the sum by previous pixel are not added to the object reference density again.

The **load** $L$ over a specified area of the image may be considered to be the combination of all the above densities. The contribution to the load has to be considered with regard to each components at this juncture. The load $L$ is defined as:

$$L =< D_r, D_d, D_o >$$

The metrics state above do not involve certain aspects. The first one is the overhead caused by the use of acceleration based on spatial subdivision techniques. The second one is the omission of the difference among the computation requirements for all kinds of object primitives. This negligence is only ostensible. For the different types of objects is supposed the approximately uniform distribution in the image space. The incorporation of acceleration technique can turn out negatively, when the comparison among acceleration techniques is to be performed. The load metrics can be extended for the object subdivision method. For each pixel x,y in the scene is considered also the cell numbering c in the scene space. This approach requires some closer definition of the space subdivision and this load extension looses a lot from the universality.

It is clear for the measuring the effectiveness of the ray-tracing algorithm comparably to the other ones we need some reference samples of scenes. The scene database for such purposes was already designed by *Haines* in 1987, the current database version is from 1994. The databases were designed with the idea of diversity in mind. The variables considered important are the amount of background visible, the number of lights, the distribution of sizes of objects, the amount of reflection and refraction surfaces, and the depth complexity (how many objects a ray from the eye point intersects with). In this paper Haines's Standard Procedural Database (SPD) is used for evaluation the qualities of designed concurrent solution.

## 5.3 Load Distribution and Balancing Strategies

Let us use the image space subdivision scheme for parallelization. This strategy has been chosen for its easy applicability on the shared memory architecture and for expectating good results. The implementation are si discussed in more details. Let us aim the attention to load balancing strategies, which influence the utilization of the multiprocessor system.

The main attribute of image space subdivision is the definition of the rectangular regions. The computation of one region is performed independently off the other regions. This strategy enables the control over the load distribution before or even in the process of rendering. There are some control strategies, which are more or less suitable. Let us enumerate and evaluate their properties.

### 5.3.1 No Load Balancing

The inherent shortcoming with any load balancing strategy is the overhead not present in a conventional serial implementation, required for the load balancing. One approach is based on the assignment contiguous region of the image to each processor in the hope that the complexity will be distributed uniformly enough across the image. This method fails completely in the cases of images with

unequal distribution of complexity because for a large part of the images the uniformness of the complexity cannot be assumed a priory. Let us define load imbalance over the image subdivision as:

$\Delta L =< \Delta R, \Delta D, \Delta O >$, where

$$\Delta R = \frac{max(D_r) - min(D_r)}{max(D_r)}.100$$

$$\Delta D = \frac{max(D_d) - min(D_d)}{max(D_d)}.100$$

$$\Delta O = \frac{max(D_o) - min(D_o)}{max(D_o)}.100$$

The min and max values are defined on the set of rectangular subregions in the image. For convenience $\Delta L$, $\Delta D$, $\Delta O$ are expressed as percentages. They represent the disparity between the largest and smallest density. The operation of substraction for $\Delta O$ is also overloaded with regard to same objects referenced in the rectangular images.

### 5.3.2 Static Load Balancing

This technique performs load balancing by smarter assignment of the rectangular regions to computational units. The mapping is done in way to achieve the assignment of both high and low complexity among the processors. The scheme has one drawback concerning the number of different objects required for referencing during computation. During the computation one contiguous region usually some amount of objects are referenced. For computation the neighbouring pixels the $\Delta O$ is low, it means the set of objects referenced for both of them differs a little. This property is called a coherency. The computation of the pixels in order to preserve coherency decreases computation time. The coherency is related to the drawback of this method, when the computation is changed to new region of the image. The probability, the major of the objects referenced is different, is very high and it invokes the reading the data from the main memory into cache of the processor. For higher granularity the overhead caused by this reading can increase the real time required for computation significantly. The load balancing provides still better performance than the computation with no load balancing. The method also supposes the multiprocessor system is dedicated during the process of rendering, which need not to be true for workstations connected via a network.

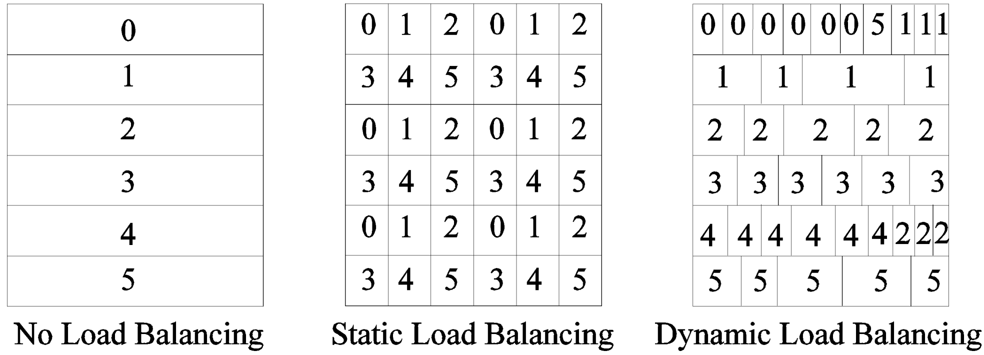


Figure 5.3: Load balancing strategies

### 5.3.3 Dynamic Load Balancing

This method allocates the computational task to each processor at run time. This approach requires some measure of processors activities and performance has to be taken at run time. The mapping tasks to processors is done by a strategy dependently on the performance of the processors. There are schemes designed for distribution of the load and the implementation on a particular architectures can influence the result of balancing. Simple approach can be done using the following strategy. The processes for computing the region of the image demand one single data task from the controller. The controller specifies the position and the size of the region and the processes–workers compute the region. The processes repeat task allocation loop until whole image is computed. These scheme is inconvenient if the response time for the data request is lager. It puts the processor into idle loop. This idle states of the processors can be reduced by buffering the tasks for each processor. Then the process requires next region, although it has still regions of the image to be calculated in its buffer. The qualities of this solution highly depend on the implementation used for communication and on the strategy used. The convenient strategy can be chosen to preserve coherence properties.

## 5.4 Types of Coherence in RT

The ray-tracing is algorithm with high dependency on input data as was already described. Next aspect of the algorithm is dependency on partitioning of the data for the processors. Although the complexity of the algorithm from the point of view of the metrics defined in previous chapter remains the same, the execution time can be influenced by referencing the data for evaluation individual pixels within a region. This phenomena is called coherency. It can be categorized into the forms, which need not to be quite independent. There are more types of coherence in computer graphics. For purposes of RT the most relevant is object

coherence. The objects are confined to lie within local neighbourhood of space. The scene usually are not the objects regularly spread in the scene space and it causes some local nodes of the objects. Distant object are disconnected. In real world the most of the space is also filled by an air. In addition to object coherence, three other principal forms of coherence can exploited in RT.

**Image Coherence** this type of coherence is connected with the object coherence by projecting objects onto the image plane. The local constancy of objects space give rise to the similar coherence in the image space.

**Ray Coherence** the property of coherence has resembling behaviour also for rays. The rays cast for neighbouring pixels has high probability to reference the same data set and approximately the same path of all types of rays.

**Frame Coherence** this property is important for animated sequences of images. It expresses the following frames change only a bit, because the animation has to preserve the continuity of the frames. It can be used by incremental ray-tracing, where are only changed parts of the images re–computed. It supposes the movement of objects in the space, but the camera doesn't change its viewing parameters.

æ

# Chapter 6

# Implementation of Parallel Ray-tracing on Shared Memory Architecture

In this chapter I would like to describe my approach to implementation of the ray–tracer on the shared memory architecture. The image subdivision scheme has been chosen for parallelization. It has been shown by Padon [1] the frequency histogram of data references is quite disproportionate. It is caused by view dependency of data. The most referenced data is fraction of size up to 7 percent of the whole database. Therefore the object space solution is only one convenient way for scenes with huge amount of object primitives, which cannot be stored in memory space of one computer, but it suffers from communication drawbacks. This reason is why I have decided to implement image space subdivision, which is more elegant and promising from the point of view of used architecture. Power Challenge from SGI with six processors was chosen. It is available to academic public at the Supercomputer Centre in Prague. The parallelization is done by distribution of the load to more processes controlled by another process. It enables an easy implementation of different load balancing strategies.

## 6.1 The Library for Parallelization on Shared Memory

In this section I describe the means used for parallelization of ray–tracing on shared memory machine in my implementation. The researchers involved in parallelization use some library for given type of architecture. In the same way I had studied the philosophies of three libraries available from Internet, because I had required the use of AT&T communication package inside of the library. I was disappointed by all libraries design, that is balanced by the other hand by portability to different types of architectures. I have decided to write down my own

library for shared memory machines. It utilizes kernel functions up to maximal extent and it gives chance to paralellize the ray–tracing effectively. The design of the library was conducted in such a way it can be also used for other tasks. Although the design of the library and the resources management is an interesting problem, in this diploma thesis the description of the library is restricted to the functionality important for the practical parallelization.

### 6.1.1 Philosophy of the Library

The library is intended only for shared non-virtual memory architectures. It involves the memory management, interprocess communication and other types of functions required for parallelization. The main ideas of the library are simple. The parallel program assumes to be number of tasks (Unix processes) sharing the same address space. Typically the initial or parent process spawns off the number of child processes. One processor should be used as the controller for additional processors. The cooperating processes are then assigned chunks of work using static or dynamic scheduling for given task distribution. For processes synchronization monitors and rendezvous operations are usually used . Since process creation and destruction are too expensive to be done frequently, processes are spawned once at the beginning of the computation. They do their work and terminate at end of the parallel part of the program.

The library presented here enables nested mastering. It is based on process grouping. The initial process is the main master of all processes. The master is also the first member of current group. More members in current group are created by forking the master process of the group or by forking already created children. The master of the group is not changed during this operations. The created processes conjoin to one process - master process by operation join. Every process, even master process, can create new group and becomes its master process.

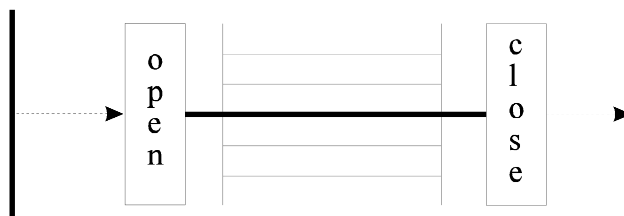


Figure 6.1: Threads in parallelization

The mastering is done for resources management. The are some types of

resources which need to be handled carefully. The most important ones are synchronizations by rendezvouses and monitors. The rendezvouses allocated by the master process inside last created group are valid for all processes belonging to the group. The process, which is alone in the group after the operation join can release allocated resources. It is also the only one, which can cancel the group created and turns its state by operation single into the position before the operation group.

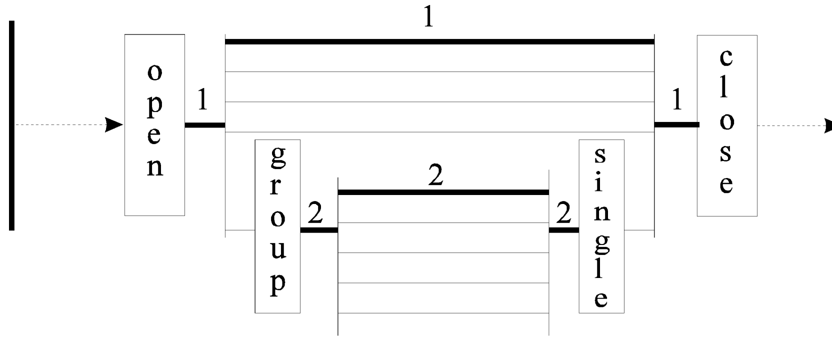


Figure 6.2: Nesting in threads

This scheme establishes some reasonable order for resources assignment. The programming is restricted, but it is more safe and the parallel source codes are easily maintainable. The hierarchy of groups also enables more complex tasks distribution than the simple strategy divide and conquer. It is convenient especially for task required more different group of subtasks running currently.

The processes can use the function for shared memory management concurrently and they can use some other resource assignment functions. They can be divived into the group by their functionality important for parallel application. The groups together with the are describes in more details in following text.

### 6.1.2 System Functions

This group includes the first initialization and termination of shared memory library processes.

- **int shm_open(int argc, char *const *argv)** .. initilizes the shared memory library. It has to be called by one program only once at the beginning of the parallel part of the application. Function creates master group and the current process becomes main master process.

- **int shm_close(void)** .. close the shared memory library and release all resources given by the system. This function should be called also once

during the application flow of execution at the end of parallel computation by main master process.

- **int shm_test(void)** .. provides the test of integrity of the shared memory library including the intrinsic variables and memory management system. It could be used for debugging the application if its behaviour is very strange. The right function guarantees the shared memory system was not corrupted by the application wrong pointering.

- **int shm_detach(void)** .. detaches the process from the shared memory library and the process runs further. It cannot be called by any master process within the shared library system.

- **void shm_logfStatistics(void)** .. logs the statistics about the shared memory library usage as specified by logging functions

### 6.1.3 Process Functions

This group of functions includes the process management, the spawning off, forking and canceling processes.

- **int shm_execv(const char* path, char *const argv[])** .. overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file specified by path. This function does not detach the process from the the shared library. The spawned process should attach to the shared library by shm_open at the beginning of the program. The process can pass the parameters to new process.

- **int shm_procHandle(int value)** .. disables or enables the mapping of the processes within the shared memory system to processors by mapping file. The default mapping file name is "shtmap.txt" and can be respecified by the parameter MAPFILE{name} if the application is created. The mapping file is introduced in the next.

- **shm_forkN(int N)** .. creates $N$ new children in the current group and attaches them tho the shared memory library. The new processes inherits all attributes of the current process as the group and the rendezvous. Message queue and new process id inside the shared library system is allocated for new process. Upon successful completion, fork returns a zero value to a child process. It returns the unix process ID of the child process to the parent process. On a failure returns -1.

- **int shm_forkN_Max(int N, int maxGrp, int maxSystem, int* procc)** .. creates $N$ new children, but the forking is restricted by $maxGrp$ limit for current group and by $maxSystem$ for the number of all processes inside the

shared memory library. The implementation for SGI enables to create up to 100 processes for one user. The return value is the same as for shm_forkN.

- **int shm_join(void)** .. the processes created by shm_fork has to be terminated using this function. The master process of current group is suspended until all other processes reach this function.

- **void shm_setExit(void)** .. set the termination flag for all other processes. The process can ask by shm_getExit function if to terminate. It should be used for simple solving the error situation situation. The processes can save the data computed and complete some other tasks.

- **int shm_getExit(void)** .. returns if the application process should exit. More details are above.

- **int shm_abort(void)** .. aborts this and other processes running inside the shared memory library immediately. The function deallocates the IPC resources used.

- **int shm_getProcsCountByID(int id)** .. returns the number of running processes inside the group, where process specified by id belongs to. This operation is not evaluated recursively.

- **int shm_getProcsCount(void)** .. returns the number of running processes in current group.

- **int shm_getTotalProcsCount(void)** .. returns total number of processes registered in shared memory library.

### 6.1.4 Group Functions

The functions defined in this section handles the group creation and termination.

- **int shm_newGroup(void)** .. creates new group. The original process becomes the master of new group created. The process ID is changed to new one, the old is used for the identification of the created group.

- **int shm_deleteGroup(void)** .. the group is removed from the shared memory system. The function has to be called by master process, which has to be alone in current group. It is typically after the operation join. Before calling this function the resources allocated inside the group should be freed, if they are not used any longer.

- **int shm_deleteGroupRM(void)** .. provides the same functionality as shm_deleteGroup, but additionaly frees automatically all resources allocated for the group.

### 6.1.5 Memory Management Functions

Any portion of a memory is allocated similar to a standard malloc function. The library provides the own memory manager, which provides the allocation from the memory pool. It solves the problem as memory fragmentation optionally by techniques best fit, first fit and the better ones. The use of the functions is exclusive inside the library, because it is necessary for correct function of memory manager. For certain application with frequent accesses to the memory functions it can become the main bottleneck of the parallel application.

- **void* shm_malloc(size_t size)** .. allocates the shared memory and returns the pointer to the memory. When no memory is available, then returns NULL.
- **void shm_free(void* p)** .. release the shared memory specified by the pointer. If bad pointer is specified, then the error message can be handled.
- **void shm_memUnlock(void)** .. when only one process inside the shared memory library uses the function for memory allocation, the overhead caused by critical section required for memory allocated can be eliminated by this function. This function has to be used carefully by the application in well-founded cases. It could be a serial part of the algorithm.
- **void shm_memLoc(void)** .. cases the memory manager is used exclusively for memory operations. It is the default.
- **int shm_freeAll(void)** .. frees all data from the shared memory system.
- **void shm_lock(void* p)** .. locks the variable in the way the other process using the shm_lock has to wait until the shm_unlock is performed. The number of the critical sections for locking the variables is restricted. If no semaphores for locking the variable is available, current process is suspended until some lock is freed.
- **void shm_unlock(void* p)** .. unlock the variable locked previously by shm_lock. Then the variable is available to other process.

### 6.1.6 Timing and Resources Utilization Functions

The functions in this group provided the clock and the resource utilization measurement for the parallel application from the start of shared memory library initialization by shm_open.

- **long shm_clock(void)** .. returns the number of milliseconds from the initialization of shared memory library.

- **long shm_uclock(void)** .. returns the number of microseconds from the initialization of shared memory library.

- **int shm_getrusage(who, struct rusage* usage)** .. returns the rusage of all current process. Dependently on input parameter who (SHM_SELF, SHM_SELF_GROUP_BEGIN, SHM_ALL, SHM_GROUP) returns the rusage for itself, for itself from the start of the last created group, for all processes and for the group process in a whole. The struct rusage is described in <sys/resource.h>. It includes the parameters as user and system time consumed by the process.

### 6.1.7 Identification Functions

The processes in shared memory library are identified by three ways. The first naming convection is based on the processes ID, which is of type integer. This identification is used for all executive interface library functions. For more user–friendly identification are used two types of string names. The first one is a system name, which cannot be changed by the parallel application. The system name of main group is SHM. The second one, the user name, can be changed by user but in the way to be unique in the shared memory library at this moment. When the process is converted into group, then the group is identified by original name of the process (group name). Both new names for the process (master of new group) are created by (group_name)_0. The children created by shm_fork are named similarly (group_name)_n, where n denotes the order of the origin the process creation inside the group. The processes inside the group are arranged into link, and therefore it could be considered the family convention for processes relationships. With the group name is handled the same way as for the process name.

- **int shm_setProcName(char* name)** .. set the process user name of current process. On failure, when the name is already used in the shared memory library, returns -1.

- **int shm_getProcName(int id, char* name, int maxlen)** .. copies the name of the process to pointer specified. The max length of the name is specified by maxlen. On failure function returns -1, when the id is not now used in current system.

- **int shm_getProcSysName(int id,char* name,int maxlen)** .. works the same way as shm_getProcName, but handles system names.

- **int shm_getIDbyName(char * name)** .. returns the id corresponding to user name or -1, when the process specified by name does not exist.

The next part concerns the relationship among the integer ID's. The application can use the function for browsing in the process hierarchy.

- **int shm_getPID(void)** .. returns current ID of current process.
- **int shm_getOPID(void)** .. returns original ID for current process formed during the process creation.
- **int shm_getGID(void)** .. returns ID of the group for current process.
- **int shm_getMainMasterPID(void)** .. returns ID of main master process.
- **int shm_getMainGID(void)** .. returns ID of main master group.
- **int shm_getProcFlags(int id)** .. return flags for process specified by id. The process attributes are find out by bitwise and of flag with one of three constants (PROC_EXIST, PROC_IS_GROUP, PROC_IS_MASTER) .
- **int shm_getMasterPID(int id)** .. returns the master ID of process specified by id.
- **int shm_getParent(int id)** .. returns creator ID of current process, which need not to correspond to group ID.
- **int shm_getLeftBrother(int id)** .. returns left brother process ID by specified id or -1 on failure.
- **int shm_getRightBrother(int id)** .. returns the right brother process ID by specified id.

### 6.1.8 Synchronizing Functions

The function provides the synchronization among the processes. One of them is rendezvous and second one is mutual lock among the processes.

#### Rendezvous

The rendezvous means the all processes in current group are suspended until all processes reach the rendezvous function. The auxiliary structure. passed to the functions is used for rendezvous operation. The rendezvous structure can be reused multiple times without any reinitialization.

- **int shm_initRendezvous(shm_rendzvsT* p)** .. initializes the structure for rendezvous before first call of shm_newRendezvous. The structure can be reused more times.

- **int shm_newRendezvous(shm_rendzvsT* rendzvsP)** .. allocates the semaphore for rendezvous operation inside the current group. This operation can be executed only by master of current group. If no semaphore is at the moment of execution the function free for this purposes, the current process is suspended until any semaphore is released.

- **int shm_deleteRendezvous(shm_rendzvsT* rendzvsP)** .. releases the semaphore used by rendezvous operation. This operation can be executed only by alone master process of the group, typically after the join operation.

- **int shm_rendezvous(shm_rendzvsT* rendezvsP)** .. the current process is suspended until all processes in current group reach this point.

**Monitors**

The monitor provides exclusive run of certain part of the code. The part of the code starts with shm_startMon and finishes with shm_endMon. The allocation and deallocation is the same as for the rendezvous.

- **int shm_initMon(shm_monT* monP)** .. initializes the structure before the first call of shm_newMon. The structure can be reused more times.

- **int shm_newMon(shm_monT* monP)** .. allocates the semaphore for monitor structure provided by the application. If no semaphore is available for monitoring, then the processes is suspended until any semaphore is freed. The operation has to be executed only by master group, typically before fork.

- **int shm_deleteMon(shm_monT* monP)** .. releases the monitor from current group. This action must be executed by alone master, typically after join operation.

- **int shm_startMon(shm_monT* monP)** .. start of the critical section. The processes using this function are mutually excluded from now on.

- **int shm_endMon(shm_monT* monP)** .. the end of the critical section. When this function is called, another process waiting in function shm_startMon is released and can enter the critical section.

### 6.1.9 Monitoring and Logging Functions

This group includes the functions for monitoring and logging the result of the parallel application. The functions are the same as normal printf from <stdio.h>, but in addition of the string specified, they provide additional information in the left part of the printed message. The first column of the message is the time

of the execution the init by function shm_open in milliseconds or microseconds. The second column is the identification of the process. The third part is the group, where the process belongs. The identification of the process and group is done optionally by integer ID, by the user name or by the system name. The format mentioned above for time and identification must be in current version of the library specified during the library compilations. The name of the log file can be the chosen in three ways. The first notation is unix pid connected by underspace with the shared library integer ID. The second one is user name for current process. The third notation is the system name for current process. The format for file name must be specified during the library compilation as well.

- **void shm_printf(const char* fmt, ...)** .. prints the arguments to console. The convention for format string is the same as for printf.
- **void shm_logf(const char* fmt, ...)** .. logs the arguments to log file or the console or both. The destination for logging can be specified by shm_logfSet. The log file should be used for logging the results of the application.
- **void shm_monf(const char* fmt, ...)** .. logs specified argument to monitor log file. This function should be used for monitoring the application, but not for logging the results of the application.
- **void shm_lmonf(int ldebug, const char* fmt, ...)** .. logs the arguments in the case th ldebug value specified is smaller than the value last set by shm_lmonSet. This can be used for control the density of the monitoring for better orientation in the monitoring log file.
- **void shm_lmonSet(int ldebug)** .. sets the priority for shm_lmonf function. The value greater or equal than zero should be specified.
- **void shm_logfSet(int type)** .. sets the destination of log information for shm_logf function. The type could be the bitwise or from the following self-explanatory constants (SHM_ALL_EN, SHM_ALL_DI, SHM_CONSOLE_EN, SHM_CONSOLE_DI, SHM_LOG_FILE_EN, SHM_LOG_FILE_DI, SHM_LOG_COMMON_FILE_EN, SHM_LOG_COMMON_FILE_DI). The common log file for all processes is "log.log". If the arguments connected by bitwise or is contradictory, then the disable function is preferred. The default state is SHM_LOG_FILE_EN. The value is common to all processes inside the shared memory library.
- **void shm_monfSet(int type)** .. sets the destination for the monitor log file. The function is the same as the shm_logfSet. The same values could be chosen for type, but in addiiton to the next values (SHM_MON_FILE_EN, SHM_MON_FILE_DI, SHM_MON_COMMON_FILE_EN,

SHM_MON_COMMON_FILE_DI). The default setting is SHM_MON_FILE_EN. The behaviour of the function is common to all processes inside the shared memory library.

### 6.1.10 Communication Functions

This group of functions includes the communication by messages. The sender and the receiver of the message is specified by process integer ID. The length of the queue of the message is limited by the unix system. For the message is also specified the type, which can be used for selection of the messages.

- **int shm_msgSend(int type, int addr_id, char* msgtxt, int length)** .. sends the message specified by type to receiver. The messages is specified by pointer msgtxt of given length. The function returns zero on success, on failure -1 and sets the variable shm_errno. The details for failure return are in the header file sht.h.
- **int shm_msgSendW(int type, int addr_id, char* msgtxt, int length)** .. has the same functionality as shm_msgSend, but if the receiver message queue if full, the sender process is suspended until the queue is released and the message is sent. Then the sender process continue its execution.
- **int shm_msgRecv(int type, int from_id , char* addr, int* maxlen)** .. receives the message from the process specified by ID and the type. For selection of types can be used the value -1, which denotes any process or any type of the message. On success the function returns zero, on a failure it returns -1 and shm_errno is set. The parameter maxlen specifies the maximal length of the received message. If the message is longer, then the rest of the message is lost.
- **int shm_msgRecvW(int type, int from_id , char* addr, int* maxlen)** .. has the same functionality as shm_msgRecv, but if no message is available in the queue, the current process is suspended until receiving the messages. For selection of the message can be used only the type of the message, the sender ID is not considered.
- **int shm_msgGetType(void)** .. returns the type of message last received by shm_msgRecv or shm_msgRecvW. It is convenient for receive the message with selection -1.
- **int shm_msgGetID(void)** .. returns the ID of the receiver of the message last received by shm_msgRecv or shm_msgRecvW.
- **int shm_msgAvailable(int type, int from_id)** .. returns the number of the messages specified by type and receiver ID. The values -1 for selection any type or any sender ID can also be used.

### 6.1.11 Process Mapping

The processes is in the shared memory environment mapped onto the processor and during its life it can be remapped onto another processor. The remapping is done by kernel handler. This scheme of mapping enables the load balancing for unix tasks. The kernel function of multiprocessor system enables also the mapping the process to specific processor. It decreases the time lost required for remapping the process. The shared memory library presented here supports both schemes. The mapping is described in a configuration mapping file, which is read during the shm_open function. It contains three parts of mapping. The first part defines virtual processors mapping to the physical processors. The physical processor is determined by integer number in the interval $< 0, n-1 >$, where n is the number of physically configured processors. The second part of mapping file declares the mapping the process onto the virtual processor. For the identification of the process or the groups are used the system and user names. Let us suppose the mapping by the configuration file is enabled. After each change of the process name the new names are compared with the process names read from the configuration file. When the names correspond each other, then the process is assigned using the translation by virtual processors table to physical processor. If the mapping by process name is not found, then the mapping by the group authorization is provided. This is declared in third part of the mapping file. The group mapping contains the order of the virtual processors, where the processes belonging to the group are consecutively mapped. The groups mapping definition also includes the maximal number of the processes utilizable by mapping. When the number of the mappable processes in current group is exhausted, the default group is used. The default group need not to be declared in the mapping file. It leaves the mapping the process to the kernel. The behaviour of default group can be also redefined in the mapping file. The concept of the virtual processors enables better flexibility and maintainbility of the map file. In addition to mentioned mapping it also includes the setting of nice value of the unix process or the scheduling priority. The example of configuration file is given for better understanding:

```
NODE_BEGIN   localhost  #hal.ruk.cuni.cz or localhost

PROCESSORS_BEGIN
# max 32+1 processors
# proc_id      proc   max_processes  [|nice_pr   |[sched]]
#              (int)       (int)      (int)        (int)
# (8 chars)  <0,n-1>        >=0       <0 ; 40>     <-20 ; 20>
#           or *

id0           *             3         | 0         | 0
id1           1             3         | 0
id2           2             3
id3           3             3         | 0         | 0
default       *            10         | 0         | 0
PROCESSORS_END

PROCESSES_BEGIN
# max 100 processes
# process_name is unique in the system (system or
#
# process_name   proc_id   [| nice_pr |[sched]]
# (max. 20 chs)  (-ref )   <0;40>       <-20;20>
#
APPL              id0       | 10      |  0
APPL_             id0       | 10
APP2              id0
PROCESSES_END

GROUPS_BEGIN
#
# max 20 groups sequence not longer than 100,
# then the rest is omitted
# groups_name   proc_id, .. [x n], .. ,[|nice_pr|[sched]]
# (max 20 chs)

default        id1 id2 id3 x 2  id1 id3 id2 x 100 |  0 |  0
APPL           id0 id1 id2 x 3  id1               |  0
APPL_1         id1 id2 id3 x 4  id1 id3 id2 x 3   |  0 |  0
GROUPS_END

NODE_END
```

The implementation of the library presented here includes a lot of tricky techniques. I present here only one of them. The shared memory is allocated in large blocks from the kernel and is mapped to available memory address. Let us suppose the situation, when new fragment of the shared memory has to be allocated by shm_alloc and no fragment of required size is available. Then new block of the shared memory is got from the kernel and then the other processes are notified to attach the memory using signals. The shared memory is attached also to the rest of the processes. This scheme for block memory management increases the utilization of the shared memory currently allocated. The other libraries, which I have studied, doesn't support this dynamic memory management. They allocate resp. release the memory at the beginning resp. at the end of their usage. The dynamic procedure is performed for deallocation of the shared memory as well. For detaching the shared memory block is required to have two unused blocks. This precaution excludes the repeated block allocation and deallocation, if the memory used by the application covers all the blocks allocated by shared memory library. Repeated sequences shm_malloc and shm_free does not cause attaching and detaching the shared memory block.

## 6.2 Parallelization of RT

Parallel ray–tracing implementation using the designed library is not difficult. For the parallelization has been chosen the ray–tracer written by Mr. Jan Burianek from the Czech Technical University. The ray–tracer ( called Bursoft in the following text) can read most of Open Inventor files supported by firm SGI. There are lot of prepared scenes in this format widely available.

Bursoft starts by reading the environment file describing the non-scene parameters. It includes the definition of the camera, its positioning in the scene space and other attributes for computing the scene. The Open Inventor does not include the lights. Therefore they are also defined in the environment file. In the second step reads the scene file and creates the inner structures used for storing the scene representation. Then is performed the scene correction extent, which minimizes the bounding box of the scene space and the computational complexity. The acceleration technique using space subdivision is BSP tree. The maximal depth of the tree and the maximal number of the primitives in the link list is also specified in the environment file. The initial phases are followed by real ray–tracing, which is the most time consuming part of the computation.
In order to speed up the ray–tracing this computational part must be parallelized. Although the time necessary for computation of BSP Tree is not negligible for certain scenes, it is relatively small compared with the time for evaluation of the color of the pixel in the prepared scene. The building the BSP tree can be also parallelized using its recursivity, but it was not necessary to implement for our

purposes, when the number of processors available is relatively small.

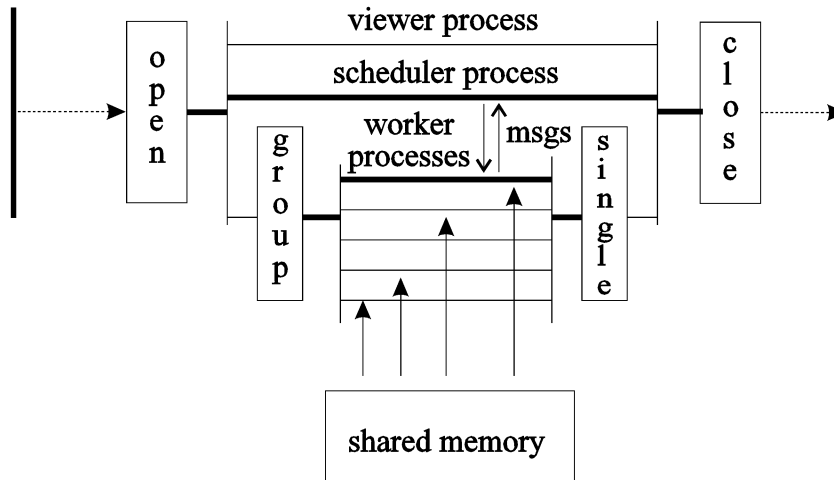


Figure 6.3: Task Management

### 6.2.1 Algorithms for Task Management

The task is divided using image subdivision scheme as was discussed in the details in previous chapter. The image space is divided into rectangular regions by the load balancing strategies. The concept for all load balancing strategies is the same. The situation is illustrated in Fig. 6.2. In terms of the shared memory library are created two or three processes. One process is the scheduler and divides the image to rectangular regions. Second process is the worker process, which is converted into the group. There are created worker–processes inside of the group. The last optional process is the viewer process, which periodically reads the memory used for result image and redraws it on the display using a X window. A region specified by scheduler can be subdivided into indivisible areas assigned to the worker processes by communication messages using shared memory library.

The task management is performed by following steps.

- creation of processes (viewer, scheduler and worker group)
- creation of worker–processes inside the worker group by shm_forkN
- assignment of worker–processes to a scheduler by initial message

  From this point the management depends on the scheduler type used. The common property for all types of management is the assignment of regions to worker processes takes places through the passing of messages. The scheduler sends an area of the image to a worker–process. This type of

the message is called an area message. The worker process computes the required area of the image and immediately after the computation sends the job–done message to scheduler. The message is not used only for indication of the end of the computation, but also stores some additional information about the computational process. It includes the start and finish time for the the area and further the user time consumed really by the worker. This is measured using a getrusage function. Let us suppose the number of worker–processes $N$.

- **No load balancing** .. scheduler uniformly divides the scene into horizontal bands. It prepares the messages and sends them to worker–processes. Then the processes sent the terminate message to worker processes. It is correct, because the worker receives the messages in received order. Then is suspended and waits for all job–done messages sent by workers. Each worker process computes exactly one image area. The number of the messages sent by the scheduler is $N$.
- **Static load balancing** .. scheduler divides the image to rectangular subregions. The division is done by grid divided by $N_y$ horizontal bands and $N_x$ vertical bands. The subregions can differs in their size. Therefore each subregion is divided by special algorithm. The algorithm tries to divide the rectangular area of size $width \times height$ to $N$ rectangles with following properties. The rectangles must have approximately the same area (in $pixels^2$). The next criteria is the squareness of the area calculated as the ratio between width and height of the area. The squareness guarantees certain coherency of the computation in the area. All subregions are one by one distributed by area–messages to worker–processes. The number of computed areas for each worker–process is $N_x \times N_y$. The total number of areas sent by static scheduler is then $N_x \times N_y \times N$.
- **Dynamic load balancing** .. scheduler divides the image to rectangular subregions similarly as the static scheduler. The difference is the number of the subregions $N_x \times N_y$ should be bigger than the number of processes. The subregion is used as the area worker process. In the beginning the scheduler sent the $N_b \times N$ area messages. $N_b$ denotes the size of the message buffer for worker–process. Then the scheduler waits untill it receives the job–done message. The worker, sender of job–done message, is sent next area message. This allocation cycle repeats until scheduler has the all image distributed. Then the scheduler sends to all workers terminate message. The number of the areas computed by workers is not usually the same. The minimal number is $N_b \times N$ and maximal number is $N_x \times N_y - N_b \times (N-1)$. This strategy is designed to distribute computational load uniformly

to worker–processors. The total number of messages sent by scheduler is $N_x \times N_y$. W

- scheduler waits until receives all job–done messages.
- workers terminate by shm_join
- viewer process terminates, if it is running, by shm_join operation
- scheduler becomes the main master process by shm_join operation
- synthetized image is saved to a file

The worker–process is more simple. It receives the area messages, computes the specified part of the image and sends to the scheduler job–done message. The worker repeats this cycle until it receives terminate message from the scheduler. The worker provides the measurement of the computation and passes it to scheduler in job–done messages. æ

# Chapter 7

# Ray–tracer testing

In this chapter I would like to present measurements I have made to test the properties of parallel implementation of ray–tracer. I have used three different scenes. They are the part of SPD and their attributes are known. The first scene is *balls*, which contains 7381 reflecting spheres with three lights in the scene. The second scene *tetra* consists of a fractal tetrahedral pyramid, which is made from 4096 triangle primitives. The last scene shells is made from the balls.

I have tested the speedup on the SGI Power Challenge with six processors. The conditions were not ideal for measuring the load balancing strategies, because the multiprocessor system was also occupied by other time consuming processes. The number of processes were intentionally increased beyond the number of physically configured processor to test the aggressivenees of the ray–tracer in UNIX environment. The strategy for mapping the processes onto the processor was left to the kernel. Following tables contains two or three measured values. The first value $T_r$ is real time required for the application under given conditions in the moment of the measurement. The second value $T_u$ is maximal user time for worker in convention of UNIX operating system. It corresponds the processor time devoted really to the given task. The third parameter $C$ is the measure "change of the load" computed from the user's time devoted to all workers $C = \frac{T_{max} - T_{min}}{T_{max}}$. This measure reflects the imbalance of the load for certain load balancing strategy used. From the time required for the computation of parallel part of the ray–tracer is evaluated a speedup. In the tables is denoted by symbol $D_r$, resp. $D_u$. It gives the ratio between the reference computational time (the time of computation of one processor for whole image) and $T_r$, resp. $T_u$. The speed up is placed at the end of this section.

Table 7.1. shows the performance for no load balancing strategy for all scenes mentioned above.

From the tables is visible following fact. Although the number of physically configured processors is six, the ray–tracer can be speeded up at the expense of other processes running on the multiprocessor system. The linear speedup in real

| | | No load balancing | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | balls.iv | | | tetra.iv | | | shells.iv | | |
| | | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ |
| Number of processes | 1 | 961.8 | 945.3 | 0.0 | 85.7 | 84.4 | 0.0 | 1079.9 | 1005.6 | 0.0 |
| | 2 | 594.1 | 584.1 | 38.2 | 52.5 | 52.0 | 37.4 | 739.6 | 691.4 | 55.3 |
| | 3 | 408.4 | 397.8 | 55.2 | 45.8 | 44.3 | 67.3 | 630.3 | 473.1 | 67.1 |
| | 4 | 334.4 | 302.4 | 65.4 | 48.7 | 45.1 | 84.5 | 696.9 | 572.2 | 83.1 |
| | 5 | 305.5 | 251.5 | 76.3 | 38.6 | 36.4 | 92.3 | 710.3 | 532.5 | 94.3 |
| | 6 | 284.5 | 214.0 | 81.6 | 30.4 | 26.2 | 93.1 | 559.0 | 317.7 | 95.0 |
| | 7 | 274.7 | 183.8 | 84.9 | 34.9 | 29.6 | 94.9 | 615.0 | 391.8 | 98.1 |
| | 8 | 268.4 | 165.5 | 85.5 | 34.6 | 27.5 | 95.3 | 590.3 | 385.9 | 98.7 |
| | 9 | 248.2 | 143.8 | 86.5 | 30.1 | 21.9 | 95.0 | 519.6 | 323.4 | 99.0 |
| | 10 | 236.3 | 136.1 | 87.9 | 27.6 | 22.0 | 95.0 | 484.2 | 288.8 | 99.2 |
| | 11 | 230.3 | 118.6 | 88.5 | 31.6 | 22.0 | 96.0 | 484.4 | 287.5 | 99.4 |
| | 12 | 222.9 | 101.2 | 88.5 | 29.7 | 19.4 | 96.0 | 468.5 | 254.9 | 99.5 |

Table 7.1: Computational times for no load balancing

time measurement was not achieved as I had supposed. æ

| | | Static balancing | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | balls.iv | | | tetra.iv | | | shells.iv | | |
| | | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ |
| Number of processes | 3 | 543.0 | 328.2 | 26.9 | 60.4 | 30.4 | 11.6 | 871.1 | 534.3 | 66.5 |
| | 4 | 421.8 | 257.5 | 27.6 | 46.8 | 30.1 | 47.9 | 789.8 | 428.3 | 62.5 |
| | 5 | 401.8 | 218.0 | 34.9 | 38.9 | 23.9 | 54.7 | 766.6 | 453.7 | 76.1 |
| | 6 | 345.1 | 185.8 | 38.0 | 38.1 | 19.8 | 63.6 | 748.1 | 374.5 | 79.0 |
| | 7 | 403.7 | 155.9 | 38.8 | 32.4 | 17.0 | 53.3 | 678.6 | 357.8 | 81.5 |
| | 8 | 306.8 | 141.2 | 37.0 | 31.8 | 15.8 | 62.6 | 582.1 | 269.1 | 76.5 |
| | 9 | 315.0 | 126.6 | 44.4 | 28.2 | 13.6 | 59.7 | 579.3 | 305.2 | 83.7 |
| | 10 | 279.4 | 116.1 | 46.2 | 31.9 | 13.4 | 61.9 | 607.3 | 328.0 | 86.9 |
| | 11 | 276.3 | 107.1 | 45.8 | 32.1 | 11.8 | 69.4 | 530.6 | 261.2 | 84.3 |

Table 7.2: Computational times for static load balancing

| | | Dynamic load balancing – balls.iv $N_x = N_y$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 x 4 | | | 8 x 8 | | | 12 x 12 | | |
| | | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ |
| Number of procs | 2 | 629.3 | 448.0 | 2.5 | 627.9 | 442.3 | 0.7 | 710.7 | 443.9 | 1.4 |
| | 3 | 495.6 | 313.2 | 9.8 | 478.7 | 299.8 | 3.6 | 531.7 | 309.1 | 11.4 |
| | 4 | 407.4 | 234.8 | 10.1 | 442.2 | 231.6 | 8.4 | 439.3 | 233.2 | 16.2 |
| | 5 | 359.8 | 191.1 | 14.1 | 370.8 | 196.5 | 18.3 | 402.3 | 184.4 | 7.4 |
| | 6 | 322.3 | 160.2 | 17.3 | 338.4 | 153.8 | 6.3 | 364.1 | 163.5 | 16.8 |
| | 7 | 314.2 | 165.7 | 43.5 | 318.0 | 133.3 | 15.2 | 329.0 | 134.2 | 14.5 |
| | 8 | 309.6 | 133.7 | 35.5 | 301.0 | 121.4 | 16.8 | 315.5 | 117.5 | 11.3 |
| | 9 | 294.8 | 124.5 | 33.7 | 282.3 | 106.6 | 12.2 | 298.8 | 104.3 | 11.6 |
| | 10 | 286.8 | 116.2 | 34.3 | 267.5 | 97.5 | 12.8 | 276.9 | 97.9 | 15.2 |
| | 11 | 258.9 | 92.3 | 33.7 | 270.0 | 91.3 | 24.2 | 267.1 | 84.3 | 10.3 |
| | 12 | 253.4 | 87.6 | 40.6 | 254.1 | 79.7 | 13.7 | 253.9 | 80.8 | 15.6 |

Table 7.3: Computational times for dynamic load balancing

| | | Dynamic load balancing – tetra.iv $N_x = N_y$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 x 4 | | | 8 x 8 | | | 12 x 12 | | |
| | | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ |
| Number | 2 | 63.6 | 42.9 | 1.0 | 67.1 | 42.7 | 0.3 | 84.0 | 43.8 | 5.0 |
| of | 3 | 48.6 | 31.5 | 22.8 | 50.7 | 28.8 | 2.5 | 68.0 | 29.4 | 8.5 |
| procs | 4 | 40.2 | 25.5 | 26.9 | 39.7 | 26.1 | 36.0 | 50.8 | 23.1 | 18.9 |
| | 5 | 38.1 | 19.2 | 20.4 | 40.5 | 20.6 | 43.5 | 48.1 | 18.5 | 17.9 |
| | 6 | 37.5 | 18.8 | 49.2 | 35.4 | 15.7 | 22.5 | 39.2 | 18.2 | 39.9 |
| | 7 | 30.8 | 18.4 | 61.1 | 34.2 | 15.2 | 38.7 | 38.1 | 15.1 | 46.8 |
| | 8 | 35.1 | 18.3 | 74.2 | 30.0 | 13.6 | 46.1 | 36.2 | 13.4 | 49.7 |
| | 9 | 33.4 | 18.4 | 87.7 | 30.9 | 12.7 | 47.9 | 30.4 | 13.3 | 43.3 |
| | 10 | 33.7 | 18.3 | 86.7 | 28.0 | 11.3 | 41.7 | 27.9 | 11.0 | 37.3 |
| | 11 | 36.2 | 18.3 | 90.5 | 26.5 | 10.3 | 46.2 | 25.0 | 9.2 | 37.4 |
| | 12 | 38.6 | 18.3 | 94.5 | 27.9 | 10.5 | 55.7 | 24.2 | 9.3 | 33.7 |

Table 7.4: Computational times for dynamic load balancing

| | | Dynamic load balancing – shells.iv $N_x = N_y$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 x 4 | | | 8 x 8 | | | 12 x 12 | | |
| | | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ | $T_r$ | $T_u$ | $C$ |
| Number | 2 | 796.6 | 510.6 | 0.6 | 846.1 | 529.1 | 7.7 | 847.2 | 526.5 | 6.8 |
| of | 3 | 740.0 | 457.7 | 38.9 | 708.0 | 417.0 | 32.0 | 678.8 | 357.7 | 7.7 |
| procs | 4 | 706.1 | 418.9 | 64.3 | 672.4 | 391.0 | 54.8 | 584.7 | 298.3 | 28.1 |
| | 5 | 643.9 | 417.4 | 75.4 | 577.7 | 335.9 | 60.4 | 548.1 | 244.7 | 27.8 |
| | 6 | 671.1 | 381.1 | 79.7 | 591.8 | 332.0 | 66.2 | 496.6 | 235.3 | 46.7 |
| | 7 | 646.7 | 379.7 | 85.3 | 569.9 | 310.3 | 68.8 | 480.3 | 216.6 | 46.2 |
| | 8 | 640.7 | 361.8 | 85.1 | 591.5 | 297.0 | 73.0 | 434.4 | 214.6 | 54.2 |
| | 9 | 616.5 | 358.3 | 90.4 | 531.3 | 297.9 | 77.3 | 416.6 | 194.1 | 58.4 |
| | 10 | 606.1 | 357.3 | 92.6 | 544.8 | 286.4 | 80.3 | 422.8 | 188.0 | 61.0 |
| | 11 | 631.7 | 355.9 | 93.1 | 553.7 | 284.4 | 81.5 | 405.9 | 183.0 | 64.5 |
| | 12 | 646.0 | 357.6 | 98.6 | 556.4 | 279.8 | 84.0 | 407.3 | 183.5 | 70.2 |

Table 7.5: Computational times for dynamic load balancing

| | | Speed up for no load balancing | | | | | |
|---|---|---|---|---|---|---|---|
| | | balls.iv | | tetra.iv | | shells.iv | |
| | | $D_r$ | $D_u$ | $D_r$ | $D_u$ | $D_r$ | $D_u$ |
| Number | 1 | 1.000 | 1.000 | 1.000 | 1.000 | 0.999 | 1.000 |
| of | 2 | 1.619 | 1.618 | 1.631 | 1.624 | 1.459 | 1.454 |
| procs | 3 | 2.355 | 2.376 | 1.871 | 1.907 | 1.712 | 2.125 |
| | 4 | 2.876 | 3.126 | 1.759 | 1.871 | 1.548 | 1.757 |
| | 5 | 3.148 | 3.759 | 2.215 | 2.320 | 1.519 | 1.888 |
| | 6 | 3.381 | 4.416 | 2.819 | 3.220 | 1.930 | 2.747 |
| | 7 | 3.502 | 5.144 | 2.457 | 2.849 | 1.754 | 2.566 |
| | 8 | 3.584 | 5.710 | 2.473 | 3.070 | 1.828 | 2.606 |
| | 9 | 3.875 | 6.573 | 2.848 | 3.851 | 2.077 | 3.110 |
| | 10 | 4.070 | 6.946 | 3.100 | 3.834 | 2.228 | 3.483 |
| | 11 | 4.176 | 7.960 | 2.712 | 3.828 | 2.227 | 3.497 |
| | 12 | 4.314 | 8.058 | 2.882 | 4.360 | 2.303 | 3.945 |

Table 7.6: Speed up for no load balancing

| | | Speed up for static load balancing | | | | | |
|---|---|---|---|---|---|---|---|
| | | balls.iv | | tetra.iv | | shells.iv | |
| | | $D_r$ | $D_u$ | $D_r$ | $D_u$ | $D_r$ | $D_u$ |
| Number | 3 | 1.771 | 2.880 | 1.419 | 2.773 | 1.239 | 1.882 |
| of | 4 | 2.280 | 3.671 | 1.832 | 2.806 | 1.366 | 2.348 |
| procs | 5 | 2.394 | 4.337 | 2.205 | 3.536 | 1.408 | 2.217 |
| | 6 | 2.787 | 5.088 | 2.249 | 4.260 | 1.442 | 2.685 |
| | 7 | 2.383 | 6.062 | 2.642 | 4.960 | 1.590 | 2.810 |
| | 8 | 3.135 | 6.694 | 2.697 | 5.330 | 1.854 | 3.737 |
| | 9 | 3.053 | 7.468 | 3.039 | 6.197 | 1.863 | 3.295 |
| | 10 | 3.442 | 8.140 | 2.685 | 6.290 | 1.777 | 3.066 |
| | 11 | 3.481 | 8.823 | 2.669 | 7.169 | 2.034 | 3.850 |

Table 7.7: Speed up for static loadbalancing

| | | Speed up for dynamic load balancing – balls.iv | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 4 x 4 | | 8 x 8 | | 12 x 12 | | 16 x 16 | |
| | | $D_r$ | $D_u$ | $D_r$ | $D_u$ | $D_r$ | $D_u$ | $D_r$ | $D_u$ |
| Number | 2 | 1.528 | 2.110 | 1.532 | 2.137 | 1.353 | 2.129 | 1.295 | 2.130 |
| of | 3 | 1.941 | 3.018 | 2.009 | 3.153 | 1.809 | 3.058 | 1.765 | 3.072 |
| procs | 4 | 2.361 | 4.025 | 2.175 | 4.082 | 2.189 | 4.054 | 2.053 | 4.031 |
| | 5 | 2.673 | 4.947 | 2.594 | 4.811 | 2.391 | 5.127 | 2.286 | 4.816 |
| | 6 | 2.984 | 5.899 | 2.842 | 6.145 | 2.641 | 5.780 | 2.558 | 5.859 |
| | 7 | 3.061 | 5.706 | 3.024 | 7.090 | 2.923 | 7.044 | 2.738 | 6.864 |
| | 8 | 3.106 | 7.069 | 3.196 | 7.789 | 3.048 | 8.047 | 2.964 | 7.970 |
| | 9 | 3.262 | 7.596 | 3.407 | 8.871 | 3.219 | 9.061 | 3.341 | 9.043 |
| | 10 | 3.354 | 8.136 | 3.595 | 9.690 | 3.474 | 9.661 | 3.502 | 9.705 |
| | 11 | 3.714 | 10.237 | 3.562 | 10.355 | 3.601 | 11.214 | 3.689 | 10.709 |
| | 12 | 3.796 | 10.793 | 3.785 | 11.867 | 3.788 | 11.696 | 3.724 | 11.338 |

Table 7.8: Speed up for dynamic load balancing

# Chapter 8

# Conclusion

In this work I have discussed the important aspects of ray–tracing for higher fidelity of the result image. The main work was done in the simulation of a real camera. The methods designed are usable for other ray–tracers as well. The camera preprocessor modul can be easily incorporated into any common ray–tracer. I have designed the methods, which provides for the animator handy tools for creating image with blur and for controlling the measure of this properties of the image from photographic point of view. The work concerning camera design was from the major part theoretical. The drawback of the method, which is the time complexity was solved by two ways. The first one was the use of the adaptive algorithm for generation the rays inside the camera. In order to speed up the process of rendering was implemented parallel solution of ray–tracing on shared memory multiprocessor system. This work was more practical programming work, even if it had supposed to study the philosophies of parallel libraries widely available. The result of this research is not only parallel implementation of ray–tracing, but the library usable for parallelization of a wide sort of task on non-virtual shared memory machine. I have not succeeded to implement the internet version of the library using for communication TCP/IP. The measurement performed have verified that the behaviour of performance, speedup and effectiveness achieved on shared memory machines are influenced by many factors. It includes the load by other processes on public accessible supercomputer node and the complexity of the scene. In the future I would like to implement the Internet version of ray–tracing, which will be based on the control of shared memory multiprocessor machines for load–balancing purposes and will minimized the exchange of the data through the net. æ

# Bibliography


[1] Paddon, D.J.:*Parallel Processing for Computer Graphics*. Research Monographs in Parallel and Distributed Computing, Pitman, 1993

[2] Holeek, A.:*Metoda sledovn paprsku*. Diploma Thesis, Faculty of Electroengineering, CTU Prague 1994

[3] Pikryl, J.:*Ray–tracing and animation in the parallel computational enviroment*. Diploma Thesis, Faculty of Electroengineering, CTU Prague 1994

[4] Jirek, M.:*Fotografick optika*. Orbis Prague, 1960.

[5] Stehlk, J.:Distribuovan metoda sledovan paprsku. Diploma Thesis, Faculty of Electroengineering, CTU Prague 1993.

[6] Watt, A., Watt, M.:*Advanced Animation and Rendering Techniques*. ACM-PRESS, Addison-Wesley, 1992.

[7] Horiguchi, S., Masayuki, K.:*Parallel processing of incremental ray tracing on a shared memory multiprocessor*, The Visual Computer, Volume 9 pp. 371-380, Springer-Verlag, 1993.

[8] Keates, M., Hubbold, R.:*Interactive Ray Tracing on a Virtual Shared–Memory Parallel Computer*, Computer Graphics Forum, Volume 14, number 4 pp.189-202, 1995.

[9] Potmesil, M., Chakravarty, I.:*A Lens and Aperture Camera Model for Synthetic Image Generation*, Computer Graphics, Volume 15, Number 3, pp. 297-305, August 1981.

[10] Potmesil, M., Chakravarty, I.:*Modelling Motion Blur in Computer–Generated Images*, Computer Graphics, Volume 17, Number 3, pp. 389-399, July 1983.

[11] Glassner, A.S.:*An Introduction to Ray Tracing*. Academic Press, London 1991.

[12] Sung, K., Shirley, P.:*Ray Tracing with the BSP Tree*. Proceedings of Eurographics, 1992.

[13] Havran, V., Zara, J.:*The simulation of the Real Camera for Rendering*. Proceedings of Workskop95, CTU Prague, Czech Republic, January 23-26,1995, pp.183-184

[14] Tvrdk, P.:*Parallel Systems and Algorithms*. Tutorial, Faculty of Electroengineering, CTU Prague 1994.

[15] Janeek, J.:*Distribuovan systmy*. Tutorial, Faculty of Electroengineering, CTU Prague 1993.

æ