On the Complexity of Ray Shooting *

László Szirmay-Kalos, Gábor Márton
Department of Process Control, Technical University of Budapest,
Budapest, Műegyetem rkp. 11, H-1111, HUNGARY
szirmay@fsz.bme.hu

Introduction

Ray tracing [Ge89], which is a promising method to produce photorealistic images, is also famous for its enormous computational requirement. Significant research effort has been devoted to making the solution of the ray-tracing more effective.

The running time t of the ray-tracing method can be expressed by the following formula [SKe95]:

$$t(n, L, P^2, D_{\max}) = O(P^2 \cdot 2^{D_{\max}} \cdot L \cdot Q(n)), \tag{1}$$

where n is the number of objects, L is the number of lightsources, P^2 is the number of pixels, $D_{\max}$ is maximum depth of the recursion of rays and $Q(n)$ is the time required to find the object that is first intersected by a ray. If the required quality of the image (determined by the number of pixels and the depth of recursion) is constant, so is the number of lightsources, then the running time is determined by function $Q(n)$. The task of which the time is $Q(n)$ determines which objects are intersected by the given ray in the 3D space and retains the nearest intersection. This task is called the **ray-shooting problem** in computational geometry. The solution of the ray-shooting problem usually consists of two phases: in the preprocessing phase the objects are transformed into an appropriate data structure; in the query phase the object intersected first by a given ray is searched for.

This paper focuses on the effective solution of the query phase of the ray-shooting problem. From practical point of view, effective solution means inventing tricky algorithms and demonstrating by simulation that these algorithms are really faster for the examined test cases. From theoretical point of view, however, we must formally prove that an algorithm has better complexity characteristics than the others. Complexity measures express the rate of increase of the required time and memory space as the size of the problem grows. The size of the problem is the number of the objects in the scene for the ray-shooting problem. In this paper the number of objects, computation time and required storage are denoted by letter n, $T(n)$ and $S(n)$ respectively. The time and space complexities impose constraints on the functions $T(n)$ and $S(n)$. A function $g(n)$ is said to be in $O(f(n))$ if there exist positive constants c and N so that $g(n) < c \cdot f(n)$ if $n > N$. Thus notation O describes an upper bound. Notation Ω, on the other hand, defines a lower bound: a function $g(n)$ is in $\Omega(f(n))$ if there exist positive constants c and N so that $g(n) > c \cdot f(n)$ if $n > N$. Finally, if $g(n)$ is both in $O(f(n))$ and in $\Omega(f(n))$, then $g(n)$ is said to be in $\Theta(f(n))$.

The main objective of complexity analysis is to find algorithms that have reasonable resource requirements in this asymptotic sense. The required time is expressed by the number of constant time operations needed to complete the algorithm. Before carrying out such an analysis, we have to define what kind of operations can be used to define the algorithm. In the generally accepted

*This work has been supported by the National Scientific Research Fund (OTKA), ref.No.: F 015884

algebraic decision tree model, the following operation type is allowed: an algebraic function is evaluated and according to its result a decision is made to select the next operation. In this framework, the computation can be visualized by a binary tree that is divided into two branches by every single decision (this is why the model is called the algebraic decision tree model).

In the classical approach algorithms are optimized for the worst case, thus complexity measures also express the resource requirements of the worst arrangement of input of a given size.

Thus the worst-case complexity measure is:

$$T(n) = \max_{o_1,\ldots,o_n \in O} t_n(o_1, \ldots, o_n), \tag{2}$$

where o_i represents the parameter set defining object i in the scene. The vector $o_1, \ldots, o_n$ of object parameters represents the input of the algorithm and is also called the **input configuration**.

Worst-case complexity means that the computation time and space must be below the given limits for any input configuration of given size. In computer graphics algorithms optimized for the worst case were usually born in the framework of computational geometry which works with constructs such as hyperplanes, cuttings, etc., thus it usually restricts the type of objects to those that are bounded by planar faces.

The practice of computer graphics, on the other hand, has produced its own techniques, that can be called as heuristic methods. These heuristic methods do not aim at worst-case optimization but rather aim at optimizing for the average case. In the framework of average case complexity evaluation, algorithms are optimized for the majority of the possible inputs not for the worst of them. More precisely, the complexity measure will be the expected value of the running time instead of the maximum running time. If the solution for an input configuration takes very long time, but this configuration has very low or even zero probability, then this does not make too much difference in the average case complexity of the algorithm.

Thus the definition of the average-case complexity measure is:

$$T(n) = E[t(n)] = \int_{o_1 \in O} \cdots \int_{o_n \in O} t_n(o_1, \ldots, o_n)\, f_n(o_1, \ldots, o_n)\, do_1 \cdots do_n, \tag{3}$$

where o_i is the parameter set of object i as before and f_n is the probability density of the possible input configurations.

The naive solution of the ray-shooting problem, when each object is tried to be intersected, and then the nearest intersection is retained, requires obviously linear time both in the worst-case and in the average-case. However, having preprocessed the object space into a sophisticated data structure, this problem can be solved more effectively. Using the concept of worst-case complexity and the tools of computational geometry, relevant research results have already been published, that attacked the sub-linear ray-shooting [dB92]. However, to make the tools of computational geometry feasible, they restrict the analysis to objects bounded by planar faces, although one of the most important features of ray-tracing is that it can handle arbitrary object types for which ray-object intersection test can be implemented.

This paper examines ray-shooting algorithms from the point of views of both worst-case and average-case complexity evaluation. On the side of worst-case analysis, it approaches the problem without requiring the objects to be polyhedra and in the query of ray-shooting the only operation allowed on an object is the ray-object intersection calculation. The paper proposes an algorithm, called the **complementer plane algorithm**, that can solve the ray-shooting problem for arbitrary convex objects in logarithmic time having constructed an appropriate data-structure. Then it is proven that the complementer plane algorithm is optimal in the sense that logarithmic time complexity is inherent in the ray-shooting problem itself, thus no better algorithm can be expected. Having also analyzed the storage complexity of the proposed algorithm, lower bounds of space are tried to be established for any ray-shooting algorithm

that works in logarithmic time. In the second section of this paper, we turn our attention to algorithms optimized for the average case. A very simple method, called the **provocative algorithm**, is presented, and it is demonstrated that the average case time complexity of this method is constant, which is much better than could be provided by the worst-case optimal algorithms. Finally, classical, heuristic ray-tracing acceleration techniques are examined in the context of the provocative algorithm and it is shown that they also have very appealing average-case time complexities.

A Worst-Case Optimal Method: The Complementer Plane Algorithm

For the sake of simplicity, the complementer plane algorithm is introduced first in the 2D flatland, then it is generalized to real 3D space.

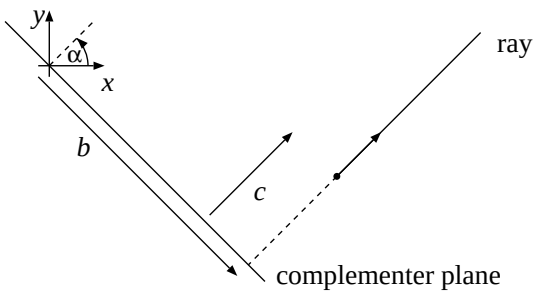

Figure 1: Representation of the ray

First of all, we find an appropriate representation for the ray. An arbitrary ray in the d dimensional space can be defined unambiguously by $2d-1$ independent parameters. Usually, d number of parameters are used to define the coordinates of the origin in a Cartesian coordinate system and $d-1$ parameters describe the direction of the ray. In our representation still $d-1$ parameters are used for the direction, but the origin is described in an unusual way. First of all, a plane is defined which is perpendicular to the direction of the ray and contains the origin of the coordinate system. This plane is called the complementer plane. The origin of the ray is defined by the projection of the ray on this complementer plane and by the signed distance of the starting point from this plane.

On the complementer plane of any ray, the objects of the scene can be represented by generating a subdivision induced by the projection of the objects onto this plane. Since several objects may be projected onto the same point, for each homogenous territory of the subdivision, an ordered list of objects is assigned. The ordering of the list should reflect spatial relation of the objects, that is, for example the distance from the complementer plane.

If this data structure is available, then ray shooting with a ray starts by identifying the territory to which the origin of the ray corresponds. Then the corresponding ordered list is searched for a possible intersection. Finding that territory of a subdivision which contains a known point is called the **point location problem**, which is a classical problem in computational geometry. In this simplified case, point location works on a single line where the territories are intervals. Here, the solution time is a logarithmic function of the involved territories, if, for instance, territories are stored in a balanced binary tree. Since the number of the territories is a linear function of the number of objects, the time of point location is at most a logarithmic function of the number of objects. Similarly, searching in an ordered list can also be done in logarithmic time, with the aid of balanced binary trees. Consequently, the solution of the ray shooting is logarithmic for any ray and object configuration.

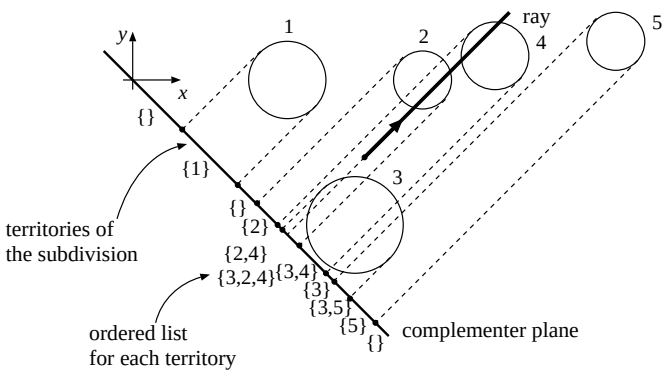

Figure 2: Organization of the data structure

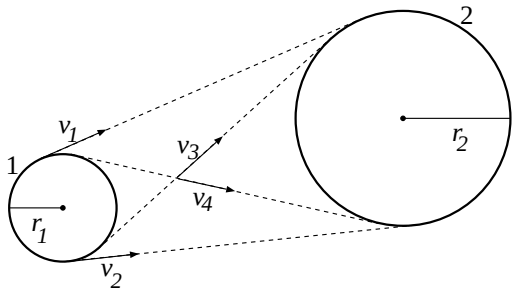

Figure 3: Changes of the structure of projection diagrams

We still have a problem which should be taken care of. The number of possible projection diagrams is infinite, thus they cannot be generated in a preprocessing phase and cannot be stored in a finite data structure.

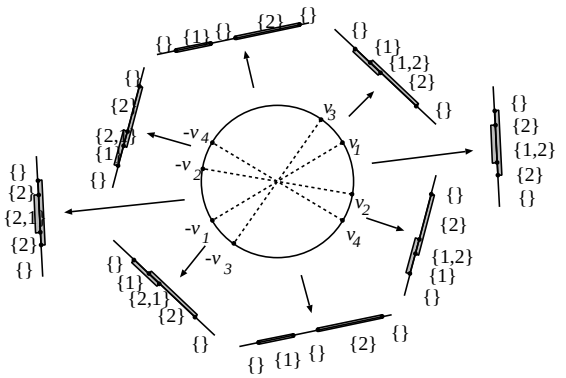

Figure 4: Diagrams of different topology

However, fortunately, the number of topologically different projection diagrams is finite since the structure of projection diagrams changes at discrete steps where there is a common tangent of two objects in the scene (see figure 3).

We can generate metrically unevaluated projection diagrams that will represent a set of metrically different but topologically equivalent real projection diagrams. These metrically unevaluated diagrams should be parameterized on-the-fly with the actual ray parameters. A metrically unevaluated projection diagram will correspond to not only a single ray, but a set of rays and the metrical parameters of its subdivision is determined from the given ray parameters.

Since the run-time parameterization does not modify the complexity, the complementer plane algorithm will find the solution in at most logarithmic time.

Generalization of the Complementer Plane Algorithm to true 3D-space

When it comes to 3D generalization, an appropriate representation of the homogeneous territories on a 2D plane must be found.

We have tried different approaches including

1. the identification of the territories as the intersection of the projections of different objects.
2. the definition of territories as the Cartesian product of the intervals generated by the projections onto two orthogonal lines in the plane [Stü96].

However, these approaches failed. Alternative 1 will generate territories having many (at most n) boundaries, which makes even the check whether a point is in a territory have at least linear complexity. In such a situation, the point location problem cannot be solved in logarithmic time. Alternative 2, on the other hand, sacrifices the property that each homogeneous territory is assigned to a list of objects that are intersected by the ray belonging to the given territory.

The real solution is a combination of the two approaches. Let us define an arbitrary line — called the base line — on the complementer plane and let us orthographically project the endpoints and the intersection points of the projection of the objects onto this line.

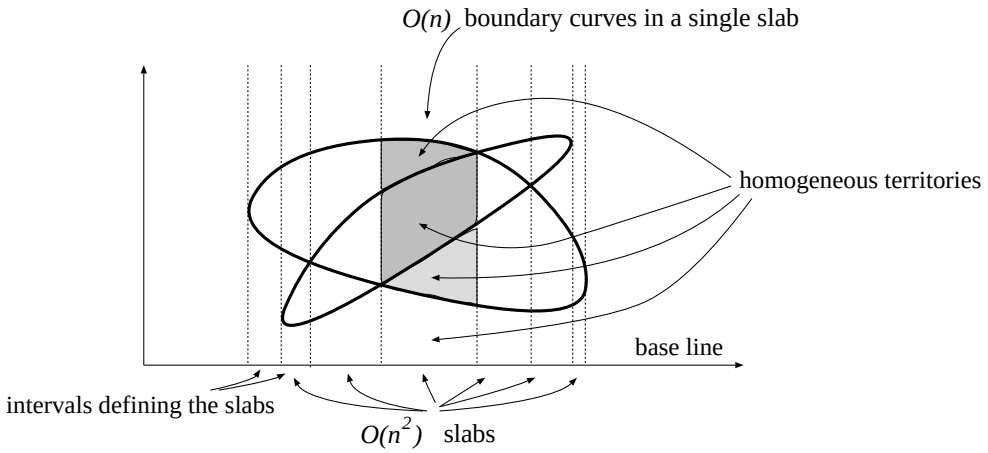

Figure 5: Organization of the 2D complementer plane

The endpoints and the intersection points define slabs perpendicular to the base line. The number of slabs is in $O(n^2)$ since the number of endpoints of n objects is in $O(n)$ and the number of intersections is in $O(n^2)$. The slabs are defined by their orthogonal projection onto the base line, which is a collection of non-overlapping intervals. As in the case of 2D flat-land, the intervals are stored in a balanced binary tree. In each slab the boundary curves of the projections are non-intersecting, since intersection points can only be on the boundaries of the slabs. Between these non-intersecting boundary curves the ordering relation of "above" can be defined if the objects are convex, which allows us to store these boundary curves in a balanced binary tree. The number of boundary curves in a single slab is $O(n)$.

The territories of the subdivision will be those regions of the slabs which are bounded by two subsequent boundary curves.

Using this data structure, the point-location problem on the 2D complementer plane can be solved in the following way:

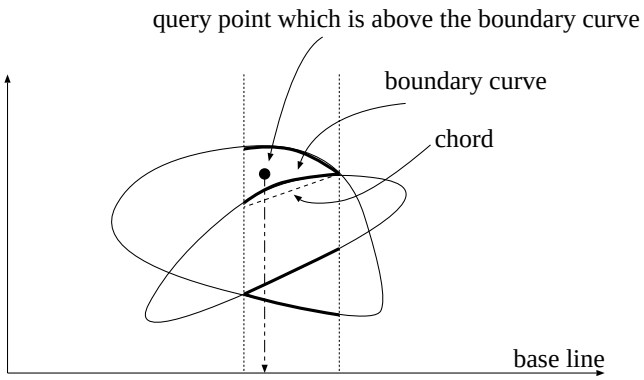

Figure 6: Classification of query point as "above" or "below" a boundary curve

1. First the point of interest is projected onto the base line.

2. Using a binary search on $O(n^2)$ intervals, the slab corresponding to the query-point is identified.

3. Applying another binary search, the homogeneous territory inside the slab is found. During the search, only ray-object tests and simple comparisons are used. To decide whether or not the query-point is above a boundary curve, the ray defined by the complementer plane with the query point is tried to be intersected with the object whose projection is this boundary curve. If the ray intersects the object, then the point is classified as "above" or "below" according to whether the boundary curve is an upper or lower curve of the projection. If no intersection occurs, then the query point is compared to the chord of the curve in between the sides of the slab (figure 6). For convex objects, this comparison tells us whether the point is below or above the projection of the object, thus it also enables us to classify the point as "above" or "below" the boundary curve.

Step 1 requires constant time, step 2 needs $O(\log n^2) = O(\log n)$ time, so does step 3. Consequently, the point location using this data structure can be solved in logarithmic time.

The other part of the algorithm that searches the ordered list of objects associated with the homogeneous territory is the same as in 2D flat-land, and therefore runs in $O(\log n)$ time.

The worst-case storage complexity is determined by the number of topologically different projection diagrams, the number of homogeneous territories of a single projection diagram and the maximum size of an associated list of objects. The size of a list of objects is obviously in $O(n)$. The number of the territories of a single projection diagram equals to the number of slabs ($O(n^2)$) multiplied by the maximum regions inside a slab ($O(n)$), thus it is in $O(n^3)$.

In order to calculate the number of topologically different projection diagrams, we have to realize that, as the direction changes, the topology of projection diagrams changes at discrete steps where the direction is parallel to a common tangent of two objects.

For a pair of objects, the directions, where a change occur, correspond to "circles" on the directional hemisphere (by "circle" we mean a closed curve that is topologically equivalent to a real circle, but not necessarily geometrically as shown in figure 7). Since each pair of objects introduces 4, that is constant number of "circles" on the direction circle, the number of "circles" is $O(n^2)$. M number of "circles" induce a subdivision on the surface of the directional hemisphere, where the number of territories is $O(M^2)$ (this comes from the well-known fact that in a d dimensional space, M number of $d-1$ dimensional objects may generate $O(M^d)$ simplices, that is d dimensional regions, $d-1$ dimensional "faces" , ... and 0-dimensional vertices [Ede87]) Since M is $O(n^2)$ in our case, the number of regions is $O(n^4)$.

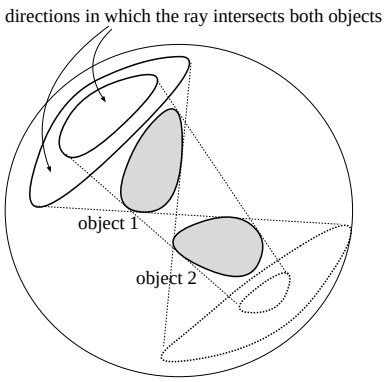

Figure 7: Directions where the topology of projection diagrams changes

Each region of the subdivision corresponds to those directions which have topologically equivalent projection diagrams, thus the number of topologically different projection diagrams is in $O(n^4)$.

Summarizing, the storage complexity of the complementer plane algorithm is $O(n^8)$.

A very high price must be paid for the appealing logarithmic time complexity of the complementer plane algorithm. The calculated upper-bound for its storage complexity seems prohibitively high. We can either try to provide a better upper-bound if possible, or start thinking of how the algorithm can be improved. In doing so, we should like to know how far we are from the optimum. More precisely, the lower-bound of the required time and storage need to be determined. Obviously, the required time and storage are not independent of each other. Therefore, we find first the optimum for the time complexity, then give a lower-bound for the storage requirement of those ray-shooting algorithms that have optimal time complexity.

Lower-bound for the worst case time complexity of ray-shooting

The lower-bounds of the complexity are inherent properties of the problem itself that is to be solved, not of the algorithm. We have to prove that no algorithm can be found for the given problem, that is capable to solve it with better than this lower-bound complexity. This is usually done by tracing back the solution of the problem to another fundamental problem that has a proven lower-bound. Then we have to demonstrate that if the new problem could be solved in lower than the assumed lower-bound time, than the fundamental algorithm can also be solved in less than its proven lower-bound time. This is an obvious contradiction, which proves that the assumed lower-bound is valid for the new algorithm.

For our case, the fundamental problem will be the point location problem along a single line. This line contains n number of intervals and the task of the point location problem is to identify that interval which contains a known point. For the fundamental problem, the following lemma can be established:

The lower-bound for the time complexity of the 1D point location problem is $\Omega(\log n)$ *in the algebraic decision tree model.*

To prove this lemma, a well-known trick can be used. Let us imagine the computations for all the possible input configurations. The computations consist of calculations and decisions that divides the further computation into two branches and they eventually stop. Thus, the computations can be described by a binary tree, which start at the root of the tree and conclude at the different leaves. Since the point location problem may have n different solutions depending on the input point, the computation tree has also at least n number of leaves. A binary tree having n leaves, however, has at least $\log n$ levels, thus $\Omega(\log n)$ decisions are needed to find the solution (the leaf) in the worst case.

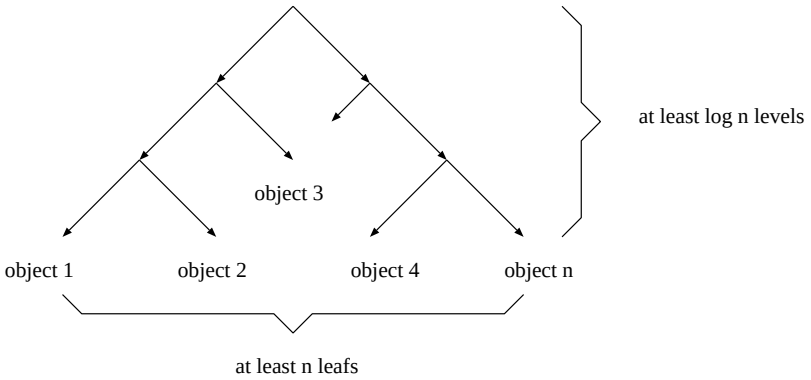

Figure 8: Computation tree

Now, let us turn to the ray-shooting problem. First, a correspondence is established between an arbitrary 1D point location problem and an appropriate ray-shooting problem that can solve the given point location problem.

Let us imagine the line of the point location problem as a ray in the 2D space. Defining an arbitrary direction on the ray, the endpoints of the intervals are classified as starting and ending points. For each ending point with the exception of the last half-open interval, a planar object is placed in such a way that the object intersects the ray in the ending point of the interval.

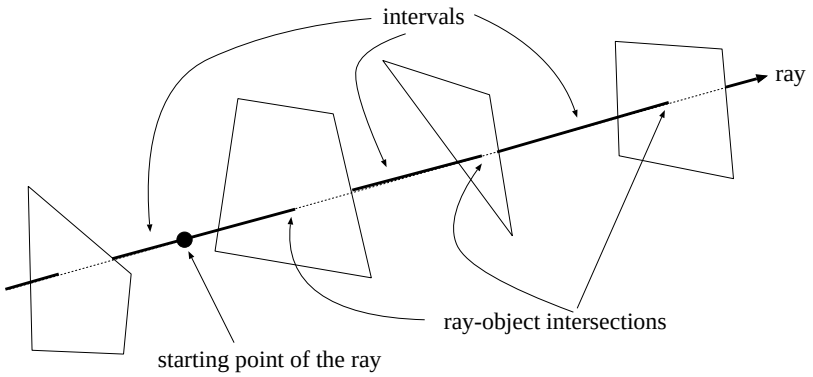

Figure 9: Solving the 1D point location problem by a 3D ray-shooting problem

This is an arrangement of a ray-shooting problem where the direction of the ray is fixed but the starting point is arbitrary along the line of the ray. Since there is a one-to-one correspondence between the objects and the intervals before them, the solution of the ray-shooting problem will also solve the point location problem for the starting point of the ray. Thus, using this construction, the time required by the solution of the point location problem equals to the time of the ray-shooting problem.

If there existed an algorithm that could solve the ray-shooting in less than logarithmic time, then the point location problem could also be solved in less than logarithmic time. As has been demonstrated, it is impossible, thus the assumption that there exists a better than logarithmic ray-shooting algorithm is a contradiction.

Lower bound for the worst case storage complexity of logarithmic time ray-shooting algorithms

In order to develop lower bounds for the required memory of those ray-shooting algorithms which run in logarithmic time, the behavior of logarithmic-time search must be understood. Logarithmic search is possible over a linearly ordered set of objects using a search parameter that induce an ordering on the objects. It means that the underlying data structure must reflect an ordering according to the parameter used in the search, since this makes it possible to eliminate half of the candidates using a single check.

In ray-shooting five independent real parameters define the ray unambiguously. For different five-tuples, the ray may intersect different objects. The collection of those five-tuples, that represent rays intersecting the same object, forms a cell in the 5-dimensional space of ray parameters. The intersection of these cells define those rays that intersect not only a single object but several objects. The results of cell intersection are called sub-cells. These subcells are furher divided into (at least) two sub-sub-cells by a face — called the distance face — which makes distinction between rays according to which object is intersected first from the collection of the rays defined by the subcell. These faces with the faces of cells locate those rays which intersect a given object first. Ray-shooting can also be regarded as a point location in this 5D space where the territories of classification are these sets of rays.

Naive implementation of ray-tracing carries out the search by finding first that sub-cell which represents the object intersected by the ray, then determines the nearest intersection which is equivalent by searching through distance faces. Both the sub-cells and the distance faces are computed on-the-fly.

Logarithmic search, however, requires an ordered set, thus the data structure must reflect the order of bounding faces to which the actual ray is tested (it does not mean that the boundaries themselves must be stored, but it does mean that their order should be built into the data structure).

As can be seen, the total number of bounding faces is not lower than the number of cells and sub-cells.

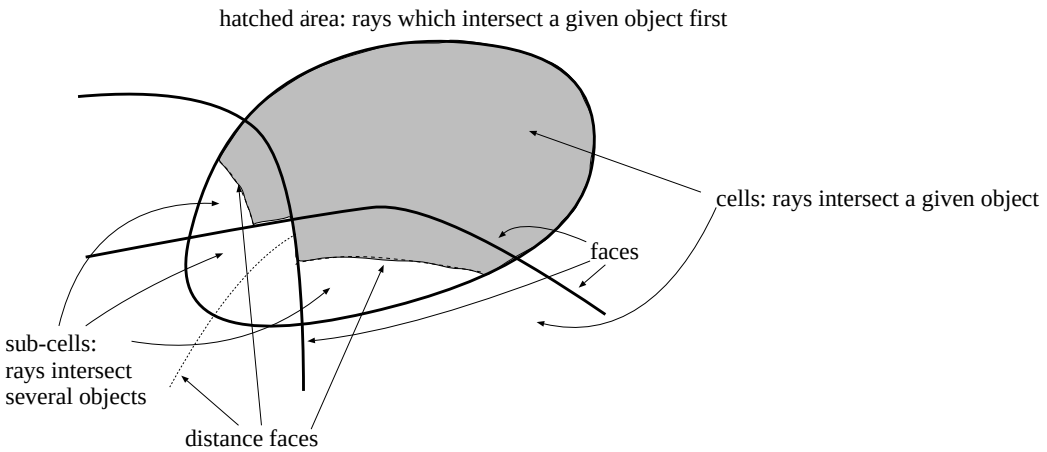

Figure 10: Visualization of the search realized by the algorithm

Since in the d-dimensional space n number of cells may generate $\Theta(n^d)$ number of sub-cells, one might think that the lower-bound of storage complexity is $\Omega(n^5)$. Fortunately, the real situation is a little bit better. The 5 parameters defining a ray is not "totally independent" in the sense that they do not allow the formation of arbitrary cells in the 5D space, which would generate $\Omega(n^5)$ sub-cells. This becomes obvious if we consider the five-tuples representing the class of rays which are incident on the same line but have different starting points. Suppose that

this line intersects objects $o_1, o_2, \ldots o_m$ in the given order. If we approach the objects along this ray following also its direction, then our actual position with the line parameters will first be in a sub-cell of $o_1, o_2, \ldots o_m$, then, as we proceed, in the sub-cell of $o_2, \ldots o_m$, finally in sub-cell of o_m before arriving at cell representing no intersection at all. Clearly, sub-cell $o_1, o_2, \ldots o_m$ is embraced by sub-cell $o_2, \ldots o_m$, which, in turn, is embraced by sub-cell $o_3, \ldots, o_m$, etc. Thus along this degree of freedom no new faces are created. Still, the four remaining parameters seem "independent", thus the number of faces is believed to be in $\Omega(n^4)$, so is the required storage of a logarithmic algorithm.

In order to prove it formally, it is enough to show that the number of possible sub-cells can be at least proportional to n^4 in the worst-case. Recall that a sub-cell represent a collection of objects that are intersected by the rays of the sub-cell. We should only create a ray-shooting problem of n objects for which it can be demonstrated that there exist $\Omega(n^4)$ number of rays which intersect a different collection of objects.

Let the objects be lines. First, it is demonstrated that for every four lines, it is possible to find a ray which intersects each of them. Let us represent the four lines $(\vec{r_1}(t_1), \vec{r_2}(t_2), \vec{r_3}(t_3), \vec{r_4}(t_4))$ and the ray $(r\vec{a}y(t_{ray}))$ by their origin and direction vector:

$$\begin{aligned}
\vec{r_1}(t_1) &= \vec{r_1}(0) + \vec{v_1} \cdot t_1, \\
\vec{r_2}(t_2) &= \vec{r_2}(0) + \vec{v_2} \cdot t_2, \\
\vec{r_3}(t_3) &= \vec{r_3}(0) + \vec{v_3} \cdot t_3, \\
\vec{r_4}(t_4) &= \vec{r_4}(0) + \vec{v_4} \cdot t_4, \\
r\vec{a}y(t^*) &= \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t^*.
\end{aligned} \tag{4}$$

The ray intersects all the four lines if:

$$\begin{aligned}
\vec{r_1}(t_1) &= \vec{r_1}(0) + \vec{v_1} \cdot t_1 = r\vec{a}y(t_1^*) = \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t_1^*, \\
\vec{r_2}(t_2) &= \vec{r_2}(0) + \vec{v_2} \cdot t_2 = r\vec{a}y(t_2^*) = \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t_2^*, \\
\vec{r_3}(t_3) &= \vec{r_3}(0) + \vec{v_3} \cdot t_3 = r\vec{a}y(t_1^*) = \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t_3^*, \\
\vec{r_4}(t_4) &= \vec{r_4}(0) + \vec{v_4} \cdot t_4 = r\vec{a}y(t_1^*) = \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t_4^*.
\end{aligned} \tag{5}$$

This is a system of linear equations where the number of unknown scalar parameters is 14 while the number of scalar equations is 12. Thus it can be solved if the equations are independent, which is true if no line is parallel with the plane generated by other two lines. The solution will correspond to the ray that intersect all the four lines, thus the existence of the intersecting ray has been proven. Note that the fact that the number of unknown parameters is greater than the number of equations does not mean that there are many solutions to this geometric problem since the same line may be expressed algebraically by many different ways. In fact, for a line, only 4 parameters are independent from the 6 (3 for the origin and 3 for the direction), thus concerning the really independent parameters, the number of equations equals to the number of unknowns.

Now let us return to the number of the possible collections of objects that are intersected by the same ray. Since from n number of lines we can select 4 lines in $\binom{n}{4} = \Theta(n^4)$ different ways, the number of different object collections is really in $\Theta(n^4)$. Consequently, the storage complexity of the logarithmic ray-shooting algorithm is in $\Omega(n^4)$.

For a special object type, namely for triangles (or for polyhedra formed by these triangles), logarithmic ray-shooting algorithm that need $O(n^{4+\epsilon})$ (ϵ is an arbitrarily small constant) is really available [dB92].

The minimum storage complexity seems to be the sum of the required storage of all the possible collection of objects that are intersected by the same ray. For restricted object types and arrangements, it can be less than $O(n^4)$. For curtains and for axis-parallel polyhedra, for instance, the logarithmic ray-shooting uses only $O(n^{2+\epsilon})$ storage space [Pel91], [dB92].

Ray-shooting algorithms optimized for the average case

Having got acquainted with this worst-case optimal algorithm, it is no surprise that if we look around in practical implementations, these worst-case optimal algorithms can hardly be found and simple heuristic algorithms seem to be more accepted. Concerning the reasons we should mention that these algorithms are difficult to implement and their memory requirements are prohibitive.

But what is more interesting, not even running time measurements justify the use of such difficult approaches. Heuristic methods, with proven very bad worst case complexities, seem to run faster than these worst-case optimized ones. The resolution of this contradiction is the difference of worst case and average case complexities.

To carry out the average case analysis, the probability distribution of the possible input configurations must be known (equation 3). In practical situations, this probability distribution is not available, therefore it must be estimated, that is, a model of the configuration space must be established. Such a model cannot be very complicated, because that would make the calculation of the expectation of the computation time impossible.

A possible, but also justifiable input configuration model for ray-shooting type analysis is the following:

1. Object space consists of spheres of the same radius r.

2. The sphere centers are uniformly distributed in space.

Since the complexity analysis is interested in asymptotic behavior when the number of objects goes to infinity, uniform distribution in a finite space would not be feasible. Instead, the space should also be expanded as the number of objects grows to sustain constant average object density. This is a classical method in probability theory, and its known result is the Poission point process [Lam72]. This Poission point process will be the basis of our input configuration model. Therefore, the input configuration model is refined further to the following:

1. Object space consists of spheres of the same radius r.

2. The shape centers are the realizations of a Poission point process of parameter ρ.

Having constructed a model of the configuration space, we can start the complexity analysis of the candidate algorithms. Let the analyzed algorithm be a very simple one and let us call it the provocative algorithm.

Provocative Ray-shooting algorithm

For the sake of simplicity, assume the ray origin is fixed to the eye position. The basic idea of this provocative algorithm is that in the preprocessing phase the objects are sorted according to their distance from this eye position. When it comes to ray-shooting, the objects are tested for intersection with the given ray in the order of their distance from the eye position no matter whether or not they are in the direction determined by the ray. This algorithm is simple, it has linear storage complexity and requires modest ($O(n \log n)$) time in the preprocessing phase. Its worst case time complexity is linear, because we can easily construct an object configuration where this algorithm will try all the objects before it finds the nearest intersection. Thus in the worst case it is not better than the naive implementation of ray shooting.

Now let us turn to the average case. The provocative algorithm takes the objects in the given order, carries out a ray-object intersection test and stops if an intersection is found. Thus, in order to find the average time complexity, we have to calculate the expected number of ray-object tests needed before the first intersection is found.

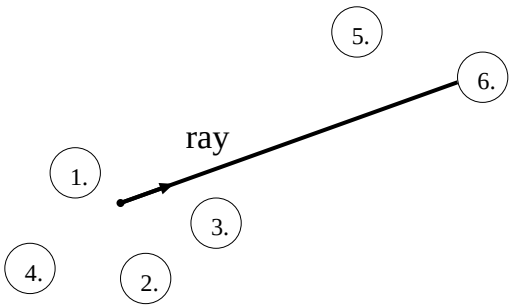

Figure 11: Provocative algorithm

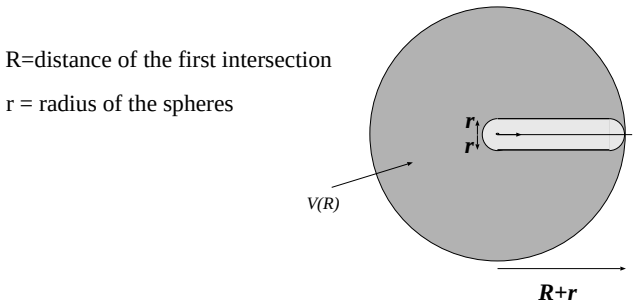

Figure 12: Region of interest

Assume that the distance of the first intersection is R. The probability conditioned by this distance that M objects has been tested before finding the intersection equals to the probability that the sphere of radius R minus the sausage-like cell of figure 12 contains M object centers.

According to the properties of Poission point processes we have:

$$\Pr\{M \text{ tests}|\text{intersection at } R\} = \frac{(\rho V(R))^M}{M!} e^{-\rho V(R)} \tag{6}$$

where $V(R)$ is the volume of the sphere minus the sausage like cell.

The conditioned expected value for Poission distribution is:

$$E[\text{number of tests}|\text{intersection at } R] = \rho V(R) + 1 \tag{7}$$

For this volume, upper and lower bounds can be derived:

$$(R+r)^3 \frac{4\pi}{3} - r^2 \pi R - r^3 \frac{4\pi}{3} \leq V(R) \leq (R+r)^3 \frac{4\pi}{3} - r^2 \pi R \tag{8}$$

Note that the $V(R)$ equals to the lower-bound if $R > 2r$. The upper-bound is only necessary for small τ values which generate strange looking cells (figure 13).

The unconditioned expected value can be calculated by the *total expected value theorem* [Lam72], using the probability density of the distance of the first intersection f_t:

$$E[\text{number of tests}] = \int_0^\infty E[\text{number of tests}|\text{intersection at } R] \cdot f_t(R)\, dR \tag{9}$$

In order to determine the probability density $f_t(\tau)$, first the probability distribution function is calculated:

$$F_t(\tau) = \Pr\{t \leq \tau\} = 1 - \Pr\{t > \tau\}. \tag{10}$$

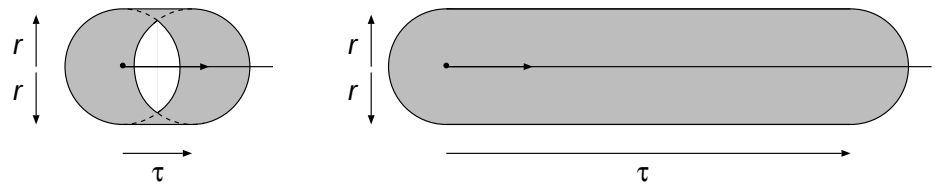

Figure 13: Volume where there are no sphere centers

Complementer probability distribution $\Pr\{t > \tau\}$ expresses the probability that there is no ray-object intersection closer than τ. This event is equivalent to another event that there is no sphere center in the cell of points not farther than r from the ray segment of length τ.

The shapes of this cell are shown in figure 13 for the two distinctive cases corresponding to $\tau \leq 2r$ and $\tau > 2r$. The volume $v(\tau)$ of the cell is:

$$v(\tau) = \begin{cases} \frac{4r^3\pi}{3} + r^2\pi\tau & \text{if } \tau > 2r \\ \\ \frac{11\pi}{6}r^2\tau - \frac{2\pi}{3}\tau^2 r + \frac{\pi}{12}\tau^3 & \text{if } \tau \leq 2r \ . \end{cases} \tag{11}$$

The complementer probability distribution is:

$$\Pr\{t > \tau\} = e^{-\rho v(\tau)}. \tag{12}$$

Consequently, the probability density, as the derivative of the probability distribution is:

$$f_t(\tau) = -\frac{dv(\tau)}{d\tau}e^{-\rho v(\tau)} = \begin{cases} e^{-\rho\pi(\frac{4r^3}{3}+r^2\tau)} \cdot \rho r^2\pi & \text{if } \tau > 2r \\ \\ e^{-\rho\pi(\frac{11}{6}r^2\tau - \frac{2}{3}\tau^2 r + \frac{\tau^3}{12})} \cdot \rho\pi(\frac{11}{6}r^2 - \frac{4}{3}\tau r + \frac{\tau^2}{4}) < 3r^2\rho\pi & \text{if } \tau \leq 2r \end{cases} \tag{13}$$

For the case of $\tau < 2r$, we get rid of the complicated expression by using an upper-bound in equation 13.

Substituting the probability density into equation 9, the following upper-bound is derived for the expected number of objects tested before the intersection.

$$E[\text{number of tests}] = \int_0^{2r} (\rho V(R) + 1) \cdot f_t(R)\ dR + \int_{2r}^{\infty} (\rho V(R) + 1) \cdot f_t(R)\ dR <$$

$$\int_0^{2r} (\rho V(R) + 1) \cdot 3r^2\rho\pi\ dR + \int_0^{\infty} (\rho V(R) + 1) \cdot e^{-\rho\pi(\frac{4r^3}{3}+r^2 R)} \cdot \rho r^2\pi\ dR \tag{14}$$

Similarly a lower-bound can also be found:

$$E[\text{number of tests}] > \int_0^{\infty} (\rho V(R) + 1) \cdot e^{-\rho\pi(\frac{4r^3}{3}+r^2 R)} \cdot \rho r^2\pi\ dR \tag{15}$$

Using also the bounds for the volume $V(R)$ given in equation 8, we end up with the following bounds for the expected number of ray-object intersection tests:

$$\frac{4e^{\rho r^3\pi}}{3(\rho r^3\pi)^2}\Gamma(4, \rho r^3\pi) - \rho r^3\pi < E[\text{number of tests}] < \frac{4e^{\rho r^3\pi}}{3(\rho r^3\pi)^2}\Gamma(4, \rho r^3\pi) + 5\rho r^3\pi + 74\rho^2 r^6\pi^2 \tag{16}$$

It is quite complicated expression, but the important observation is that the bounds are finite and they do not grow forever as the number of objects goes to infinity. Thus the average

time complexity of the provocative algorithm is constant and is independent of the number of objects in the asymptotic case. Formally, the time complexity is $O(1)$.

This is much better than the worst case optimal algorithms which are logarithmic even in the average case.

This provocative algorithm can be regarded as a common idea in almost every heuristic ray-tracing speed-up techniques, including:

- regular space partitioning [AK89],
- octree [AK89],
- ray coherence methods [OM87][HMSK92],
- ray classification [Gla84],
- Voronoi diagram based space partitioning [Már95],

Comparing them to the provocative algorithm, we can note that they can handle arbitrary ray origin and further restrict the search space by decreasing the possible directions in which intersections are tested. This can enhance the speed by a linear factor.

However, the basic idea is still the same, thus these algorithms can also be expected to run in constant time in the average case. A rigorous theoretical analysis has been carried out to prove that formally. It turned out that constant time behavior is really true for the classical speed-up methods, with the exception of the octree method, since the administration of the octree has a slight (logarithmic) overhead.

The following two tables summarize the results for these heuristic ray-tracing acceleration techniques, and in order to allow comparison, they also survey the worst-case complexity of these algorithms.

Average case	$Q(n)$ query time	$P(n)$ preprocessing time	$M(n)$ memory
regular space partitioning	$O(1)$	$O(n)$	$O(n)$
octree	$O(1)$–$O(\log n)$	$O(n \log n)$	$O(n)$
ray coherence	$O(1)$	$O(n^{2+1/3} \log n)$	$O(n^{2+1/3})$
ray classification	$O(1)$	?	?
Voronoi–diagram	$O(1)$	$O(n^{1+1/3})$	$O(n)$

Table 1: Average-case complexity measures

Worst-case	$Q(n)$ query-time	$P(n)$ preprocessing time	$M(n)$ memory
regular space partitioning	$O(n)$	$O(n^2)$	$O(n^2)$
octree	$O(n)$	$O(n^2)$	$O(n^2)$
ray coherence	$O(n)$	$O(n^3 \log n)$	$O(n^3)$
ray classification	$O(n)$	?	?
bounding hierarchy	$O(n)$	?	$O(n)$
Voronoi–diagram	$O(n)$	$O(n^{\lceil 3/2 \rceil} + n^2)$	$O(n^{\lceil 3/2 \rceil} + n^2)$

Table 2: Worst-case complexity measures

Towards general object spaces

The average case analysis has been based on the model of the input configuration space, which involves two basic assumptions: the objects are spheres of the same radius and are uniformly distributed. The credits of the results thus depend highly on whether this model is acceptable for practical situations. In order to improve the realism of the underlying model, we examined the effect of allowing spheres to have different radii from a finite set [MSK95]. Although this makes the average case analysis more complicated numerically, the time complexity of the heuristic ray-tracing algorithm mentioned above is still in $O(1)$.

The second step of generalization is the introduction of arbitrary object types instead of dealing with only spheres. Even if the scene consists of arbitrary objects, spheres can be used as virtual bounding boxes. Using the results valid for spheres, the first bounding box can be found in constant time in the average case. However, finding a bounding box does not necessarily mean finding the object intersected by a ray. Assume that the objects are "well-shaped", which means that the probability that the intersection with the bounding volume implies the intersection with the object itself is greater than a given constant (for instance, lines are not "well-shaped" objects). In the scene of well-shaped objects, the probability distribution of the number of those ray and bounding-volume intersections that do not result in ray-object intersections follow a geometric series, thus its expected value is constant. Thus $O(1)$ average running time is also valid for arbitrary "well-shaped" objects.

The other fundamental assumption of the input configuration model is that the objects are uniformly distributed, which led us to the application of Poisson point processes. Is it valid for practical cases? Unfortunately, no such practical data is available, therefore this question cannot be answered. If it were not be true, then the provocative algorithm, for instance, would not necessarily run in constant time. It is also indicated by the fact that the worst-case complexity of the algorithm is different from the average-case complexity. However, even for the most malicious distribution, the algorithm is not worse than shown by its worst-case complexity which is $O(n)$.

Conclusions

This paper reviewed the worst-case and average-case complexity analysis of the ray-shooting problem. It has been demonstrated that ray-shooting algorithms run at least in logarithmic time in the worst-case. An algorithm, called the complementer plane algorithm has been introduced, that really has the optimal logarithmic worst-case time complexity. Unlike previous methods proposed for logarithmic ray-shooting, the complementer plane algorithm can work with arbitrary convex objects not only with polyhedra or other planar bounded constructs. The storage complexity of complementer plane algorithm is $O(n^8)$ which seems prohibitive. It has been shown, that no logarithmic ray-shooting algorithm capable of handling arbitrary objects can be expected, that uses less than $O(n^4)$ space. This makes the practical use of worst-case optimal algorithms questionable and increases the importance of average-case optimal methods. The paper presented a framework, called input configuration model, for the analysis of average-case complexity of ray-shooting algorithms. It has been shown that a very simple algorithm, called the provocative algorithm, runs in constant time in the average case, so do most of the heuristic ray-tracing methods.

References

[AK89] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Academic Press, London, 1989.

[dB92] Mark de Berg. *Efficient Algorithms for Ray Shooting and Hidden Surface Removal.* PhD thesis, Rijksuniversiteit te Utrecht, The Nederlands, 1992.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[EH94] Gröeller E. and Löffelmann H. Extended camera specification for image synthesis. *Machine Graphics and Vision*, 3:513–530, 1994.

[Ge89] A.S. Glassner (editor). *An Introduction to Ray Tracing.* Academic Press, London, 1989.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[HMSK92] T. Horváth, P. Márton, G. Risztics, and L. Szirmay-Kalos. Ray coherence between sphere and a convex polyhedron. *Computer Graphics Forum*, 2(2):163–172, 1992.

[Lam72] John F. Lamperti. *Stochastic Processes.* Springer-Verlag, 1972.

[Már95] Gábor Márton. Acceleration of ray tracing via voronoi-diagrams. In Alan W. Paeth, editor, *Graphics Gems V*, pages 268–284. Academic Press, Boston, 1995.

[MSK95] Gábor Márton and László Szirmay-Kalos. On average-case complexity of ray tracing algorithms. In *Winter School of Computer Graphics '95*, pages 187–196, Plzen, Czech Republic, 14–18 February 1995. University of West Bohemia.

[OM87] Masataka Ohta and Mamoru Maekawa. Ray coherence theorem and constant time ray tracing algorithm. In T. L. Kunii, editor, *Computer Graphics 1987. Proc. CG International '87*, pages 303–314, 1987.

[Pel91] M. Pellegrini. New results on ray shooting and isotopy classes of lines in 3-dimensional space. In *Proc. Workshop on Algorithms and Data Structures*, pages 20–31, 1991.

[SKe95] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics.* Akadémia Kiadó, Budapest, 1995.

[Stü96] W. Stürzlinger. Verbal communication. 1996.