On the Limitations of Worst-case Optimal Ray Shooting Algorithms *

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

Abstract: This paper examines the lower-bounds of worst-case complexity measures of ray-shooting algorithms. It demonstrates that ray-shooting requires at least logarithmic time and discusses the strategies how to design such optimal algorithms. It also examines the lower-bounds of storage complexity of logarithmic-time algorithms and concludes that logarithmic time has very high price in terms of required storage.

Keywords: Ray-shooting, complexity analysis, data-stuctures

Introduction

Ray shooting is a geometric problem arisen in the framework of the famous ray tracing [Ge89], and can be formulated as follows: *Given a set of objects in the 3D space, a data structure must be built that allows the effective determination of the first object that is intersected by an arbitrary ray.*

The solution of the ray-shooting problem usually consists of two phases: in the pre-processing 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. From theoretical point of view, algorithm efficiency is characterised by complexity measures that express the rate of increase of the required time and memory space as the size of the problem grows. Here the size of the problem is the number of the objects in the scene. 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 it grows at most with the speed of $f(n)$, in $\Omega(f(n))$ if it grows at least with the speed of $f(n)$ and in $\Theta(f(n))$ if it grows exactly with the speed of $f(n)$.

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 in the algorithm. In the generally accepted **algebraic decision tree model** [Rei72], 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 visualised by a binary tree that is divided into two branches by every single decision.

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

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. However, having pre-processed the object space into a sophisticated data structure, this problem can be solved more effectively. Concerning effective ray-shooting, the following questions need to be answered:

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

1. What is the lower-bound of the time complexity of ray-shooting, thus what are the theoretical limits of optimal ray-shooting algorithms?

2. How such an optimal algorithm can be elaborated?

3. What is the minimum price of optimal time complexity in terms of the required memory space and pre-processing time.

4. What is the practical value of worst-case optimal algorithms in ray-shooting?

First, examining the computation tree, this paper shows that the optimal time complexity of the ray-shooting problem is logarithmic. Then the structure of the space involving the objects and the ray is analysed and the results are used to determine the storage complexity. In order to evaluate the lower-bounds of the geometric complexity of this space, we will assume lines to be the objects.

We have to mention that using 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.

Thus, this paper also 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.

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. In the algebraic decision tree model, this is usually done by determining the size of the tree based on the number of possible outcomes of the algorithm.

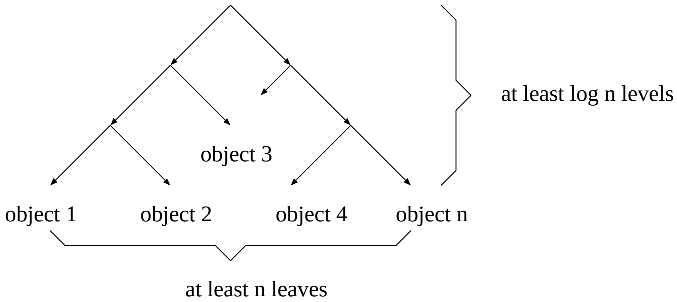

Figure 1: Computation tree

The ray-shooting algorithm can result in at least $n+1$ different outputs, since generally any object can be first intersected by the ray in addition to generating no intersection if the ray parameters are chosen appropriately, thus the computation tree of ray-shooting has at least $n+1$ different leaves (figure 1). Since a binary tree that has at least n leaves should have at least $\log n$ different levels, the algorithm must make at least $\log n$ different decisions in the worst-case to reach the leaf, that is to find the output. It means that no algorithm can be expected that solves the ray-shooting problem in better than logarithmic time.

Analysis of the space containing the objects and the ray

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 — called sub-cells — define those rays that intersect not only a single object but several objects. Since a set of objects may be intersected from two, opposite directions, more than one sub-cells may represent the same collection of objects. For each cell or sub-cell, the object which is first intersected by a ray can be identified.

Ray-shooting can also be regarded as a point location in this 5D space where the territories of classification are the sub-cells.

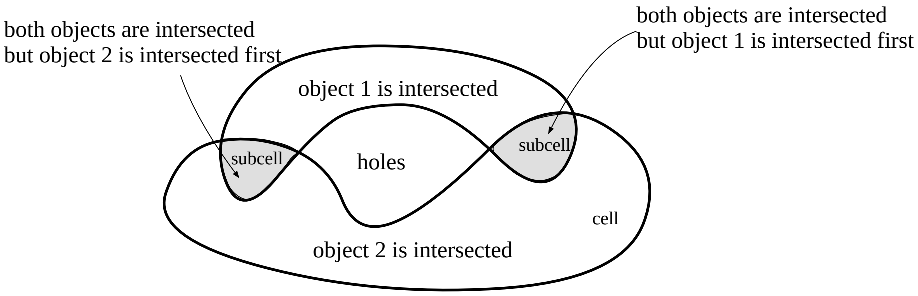

Figure 2: Visualisation of the 5D space of ray parameters

The complexity of the arrangement of cells and subcells will determine the complexity of data structures needed to effectively search in this arrangement. For algorithms discussed below, two measures are crucial:

1. The number of subcells, which means the number of different collections of objects intersected by the same ray.
2. The number of holes between cells, that is the number of equivalence classes of those rays which do not intersect any object. Two rays are said to belong the same equivalence class if the supporting lines of rays can be moved to each other without intersecting any object.

In order to establish lower-bounds for these measures, arbitrary object types can be used. Obviously, a worst-case lower-bound that is valid for a special type will also be valid worst-case lower-bound for arbitrary object types. Thus, for the sake of simplicity, we can restrict the objects to lines.

A lower-bound for the number of collections of objects intersected by a ray

First of all, we show that for every four lines, it is possible to find a ray which intersects each of them if the lines are generally positioned and oriented in the space.

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 $(r\vec{a}y(t^*))$ by their origin and direction vector:

$$\begin{aligned} \vec{r}_i(t_i) &= \vec{r}_i(0) + \vec{v}_i \cdot t_i, \qquad i = 1,2,3,4 \\ r\vec{a}y(t^*) &= \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t^*. \end{aligned} \tag{1}$$

The ray intersects all the four lines if:

$$\vec{r}_i(t_i) = \vec{r}_i(0) + \vec{v}_i \cdot t_i = r\vec{a}y(t_i^*) = \vec{r}_{\mathrm{ray}}(0) + \vec{v}_{\mathrm{ray}} \cdot t_i^*, \qquad i = 1,2,3,4 \tag{2}$$

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 (if any) generated by other two lines. The solution will correspond to the ray that intersects 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.

Since any four lines can be intersected by a ray, the number of object collections is not lower than the number of different combinations of four elements from a set of n number of elements, which is obviously $\binom{n}{4} = \Theta(n^4)$.

A lower-bound for the number of equivalence classes of rays generating no intersection

The restriction of objects to lines allows us to use a result of McKenna and O'Rourke [MO88], which states that the number of such equivalence classes is also in $\Theta(n^4)$.

Using this result let us examine the decision tree of the ray-shooting again. In the decision tree, different leaves either correspond to different objects intersected first or correspond to no intersection at all. For a moment, let us assume that we are only interested in whether or not a given ray intersects any object. This simplifies the ray-shooting problem to a simple decision whether or not the ray corresponds to any of those ray classes which do not generate intersections. Since these ray-classes are disjoint, this is obviously a membership problem, for which the Dopkin-Lipton [DL79] or the Steel-Yao [SY82] theorem on the properties of the decision trees of the associated membership problems can be applied.

According to the Dopkin-Lipton theorem which is valid for linear decision trees, the number of leaves of the decision tree cannot be lower than the number of disjoint connected cells. In linear decision trees, the allowed operations are the evaluation of a polynomial of degree 1 and a branching based on the result. In ray-shooting lines, the operations of the algorithm may involve intersection checks and the comparison of distances for which the linear decision tree model seems to be adequate. Since the number of disjoint connected cells is not lower than $\Theta(n^4)$ (the number of ray classes representing no intersection), the number of the leaves and consequently the number of the nodes of the decision tree are also in $\Omega(n^4)$.

Logarithmic search algorithms generate data structures that store some ordering to be followed during the search. Since the number of paths of computation is $\Omega(n^4)$, even if the information used by different paths differs in only a single bit, the required storage is in $\Omega(n^4)$. This data structure is built up in the pre-processing phase, where the time complexity cannot be less than the size of the data structure, since this determines the output size of the pre-processing phase. Thus the storage complexity and the pre-processing time of logarithmic ray-shooting algorithms are in $\Omega(n^4)$.

If the linear decision tree model may seem restrictive, we can apply the Steel-Yao theorem which allows any polynomials of at most degree d to be the nodes of the decision tree, and states that a single leaf of the decision tree may be associated with at most $d(2d-1)^{m+h-1}$ number of cells, where m is the number of query parameters (5 for ray-shooting) and h is the distance of the leaf from the root of the tree. For logarithmic search algorithms, $h = c \cdot \log n + \Delta(\log n)$ where $\Delta(x)$ represents any function for which $\lim_{x\to\infty} \Delta(x)/x = 0$. Thus $d(2d-1)^{m+h-1} = \Theta(n^{c\log(2d-1)})$, which makes the number of leaves be in $\Omega(n^{4-c\log(2d-1)})$. This means that a tradeoff might be established between the complexity of the computations in a single node and the required storage, but the existence of such a method is not proven yet.

Construction of logarithmic-time ray-shooting algorithms

Logarithmic-time ray-shooting algorithms can be constructed by applying the divide-and-conquer approach. This method attacks a problem of size n by subdividing it into smaller problems of the same type and in a single phase of the search it selects that smaller problem which solves the original problem.

Let us assume that the problems generated by the subdivision are of size at most $n/r(n)$, where $r(n) > 1$. If the solution time of a problem of size n is $Q(n)$ and the time of decomposition is $d(n)$, then the following recurrence expression can be established for $Q(n)$:

$$Q(n) = Q(n/r(n)) + d(n) \tag{3}$$

The algorithm is exactly logarithmic ($\Theta(\log n)$) if $Q(n) = c \cdot \log n + \Delta(\log n)$. Substituting this into equation 3, we get:

$$c \cdot \log n + \Delta(\log n) = c \cdot \log(n/r(n)) + d(n) + \Delta(\log n/r) \tag{4}$$

Thus the following relation must hold asymptotically:

$$\lim_{n\to\infty} \frac{d(n)}{\log n} = \lim_{n\to\infty} \frac{\log r(n)}{\log n} \cdot c. \tag{5}$$

This means that if $\log r(n)/\log n$ converges to zero, then $d(n)$ must be a sub-logarithmic function of n. For example, if $r(n)$ and $d(n)$ are both constant, then this condition holds. On the other hand, if $\log r(n)$ is in $\Theta(\log n)$, then $d(n)$ can also be in $\Theta(\log n)$.

Based on these two extreme cases, two different approaches to logarithmic ray-shooting algorithms can be elaborated:

1. The case when $r(n) = 2$ and $d(n)$ is constant takes us to the simple binary search. The objects should be pre-processed into one (or more) list(s) ordered by conditions on the ray parameter. These conditions determine whether the object intersected first is in the upper or lower sections of the list. Starting at the centre of the list, the corresponding conditions must be examined. If this condition is true, then the upper half of the list is processed in a similar way. For false value, the lower part is examined. The algorithm stops when the examined conditions identify a single object.

2. If both $\log r(n)$ and $d(n)$ are in $\Theta(\log n)$, then the algorithm will be based on recursive subdivision of the object space [dB92]. Here objects should be assigned to some not necessarily disjoint groups of size $O(n/r)$, and a search algorithm must be elaborated which is able to decide which group contains the first intersection in $O(\log n)$ time.

This paper focuses on ray shooting algorithms of the first type.

Binary search type ray-shooting algorithms

Binary search is a well-known technique that can identify an object in an ordered set in logarithmic time.

However, this simple approach does not work in its original form for ray shooting, since it is impossible to generate a single order of objects where half of the objects can be eliminated executing a constant time test on the ray parameters. In order to prove it indirectly, let us assume that the objects are points and it is possible to subdivide them into two subsets for which constant time selection is possible.

At least one subset from the two — let it be subset A — is size of $O(n)$. Let us select a single point in the other subset — called subset B — and consider only those rays that pass through this point and if they pass through a point in A, the point in A is in front of the point

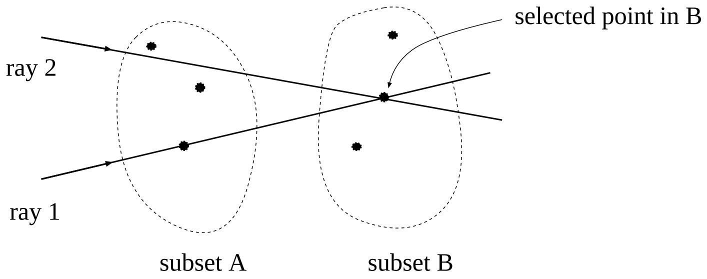

Figure 3: Indirect proof for that no single order exists in ray-shooting

in B. Clearly, if such a ray passes through a point in A, then B must be eliminated, otherwise A must be eliminated. This means that our assumed-to-be algorithm would determine in constant time whether or not the ray passes through any point (object) in A. However, this is impossible due to the pigeon-hole theorem, since the information that can be processed in constant time cannot store the required parameters of the points in A if n goes to infinity. Consequently, no single order on objects exists in the general case, that can be used for applying binary search to identify the first object intersected by a ray.

However, generating different orders to different collection of rays, binary search will be feasible. To avoid the previous problem, collection of rays must be formed in such a way that if a subset of objects are eliminated for some ray, then the number of those eliminated objects, which can be intersected first by a ray in the collection, should be bounded by a constant number. An appropriate collection of rays can be selected by identifying those rays whose supporting lines intersect the same group of objects. For convex and disjoint objects, the sequence of objects intersected by the supporting line can follow an ascending or descending order of a fixed series, thus this series can be used as the basis of ordering.

The logarithmic search on this set is controlled by conditions imposed by planes defined between the subsequent objects. These planes make a distinction between origins of rays that intersect a different object first. The search starts at the middle of the list of these planes. The plane and the origin of the ray is compared to decide whether the origin is "above" or "below" the plane, where the direction of the ray is considered as the "up" direction. If the origin is above, then the intersection cannot be below the plane thus the same step is repeated recursively for those objects that are above the plane. If the origin is below the plane, the first object below the plane is tried to be intersected. If intersection occurs with the first object below, then the object first intersected is in the lower part, thus the search will be continued for the lower part. If no intersection occurs, then the object first intersected is the fist object in the upper list.

In this search a single cycle takes constant time and eliminates approximately half of the objects. Thus the solution is available in logarithmic time.

This approach starts by the identification of that group of objects that are intersected by the line of the given ray, then the first intersection is found by a logarithmic search based on an order defined for this group. Note that this step solves only the second half of the query problem (see figure 4). First, the group of intersection objects must be found, which cannot require more than logarithmic time if the complete algorithm needs to be logarithmic.

Clearly, we need special representations and search methods that assign the given ray to those objects which are intersected by the line of the ray. In the subsequent sections two algorithms are presented that attack this problem in two different ways.

The first method assumes the object to be triangles and uses the methods invented in computational geometry [PS85], [dB92], namely the method of Plücker coordinates. The second method is more general and allows arbitrary convex objects for which ray-object intersections and projections can be calculated.

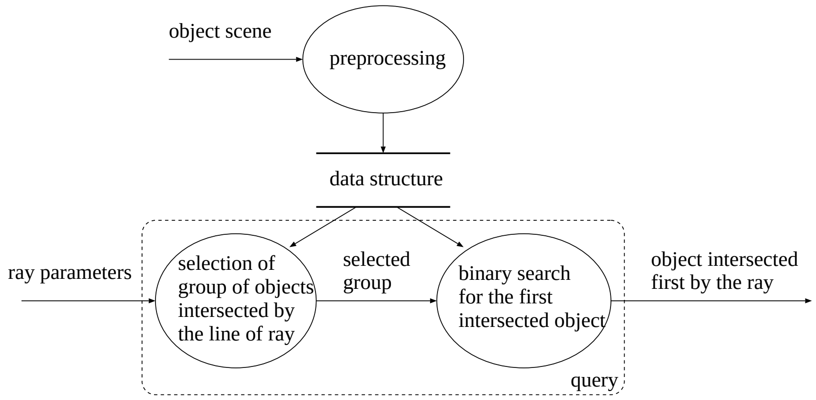

Figure 4: Dataflow of binary search type ray-shooting algorithms

Ray-shooting using Plücker coordinates

One way of tracing back ray-shooting to an equivalent point location problem is the application of Plücker coordinates as proposed in [dB92]. Since Plücker coordinates can be defined for linear varieties, the objects should be assumed to be bounded by lines, thus we restrict the types of objects to triangles.

Let us define a directed line l by two points (a_1, a_2, a_3, a_4) and (b_1, b_2, b_3, b_4) using homogeneous coordinates. This line can be represented by a point π and a plane Π in a 6 dimensional space, where

$$\begin{aligned} \pi(l) &: (\pi_{12}, \pi_{13}, \pi_{14}, \pi_{23}, \pi_{24}, \pi_{34}), \\ \Pi(l) &: \pi_{34}x_1 - \pi_{24}x_2 + \pi_{23}x_3 + \pi_{14}x_4 - \pi_{13}x_5 + \pi_{12}x_6 = 0, \end{aligned} \tag{6}$$

where $\pi_{ij} = a_i b_j - b_j a_i$. Since the relation of a Plücker plane of a line l_1 and a point of another line l_2 expresses the relative orientation of the two represented lines l_1, l_2, it can be shown [dB92] that a line-triangle intersection problem can be traced back to a point location problem in the arrangement generated by the planes of the three similarly oriented lines t_1, t_2 and t_3 defining the triangle.

A list of triangles is assigned to each cell of the arrangement, that stores the triangles that are intersected by the lines of this cell. Point location in arrangements defined by planes is an extensively studied topic of computational geometry, and is known to have logarithmic time solution using, for example, cuttings based on random sampling [Cla87]. Having identified the list of triangles intersected by the supporting line of the ray, the triangle intersected first by the ray can also be found in logarithmic time as suggested by the previous chapter.

The Complementer Plane Algorithm

This algorithm is based on the observation that the identification of the groups of objects intersected by the line of the ray can be traced back to a point location problem if the direction of the ray is fixed. For fixed direction, different groups will correspond to different areas on a plane which is perpendicular to the direction and onto which all objects are projected (figure 5). This plane is called the **complementer plane** in [FMSK96].

On the complementer plane, the objects of the scene can be represented by generating a subdivision induced by the projection of the objects onto this plane. Unfortunately, the number of possible projection diagrams is infinite if arbitrary ray direction is allowed, thus they cannot be generated in a pre-processing phase and cannot be stored in a finite data structure. However, fortunately, the number of topologically different projection diagrams is finite since the structure

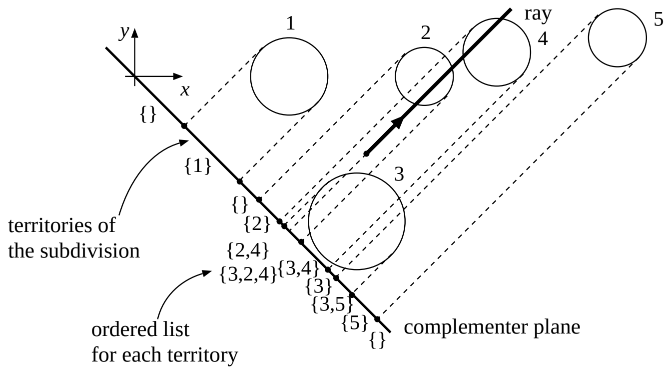

Figure 5: Organisation of the data structure

of projection diagrams changes at discrete steps where there is a common tangent of two objects in the scene (see figure 7). Thus we can generate metrically unevaluated projection diagrams which must be parameterized on-the-fly with the actual ray parameters.

If this data structure is available, then ray shooting with a ray starts by identifying first the associated projection diagram, then continues locating the territory to which the origin of the ray corresponds, finally the corresponding ordered list is searched for a possible intersection.

In order to find an appropriate representation for the territories, 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.

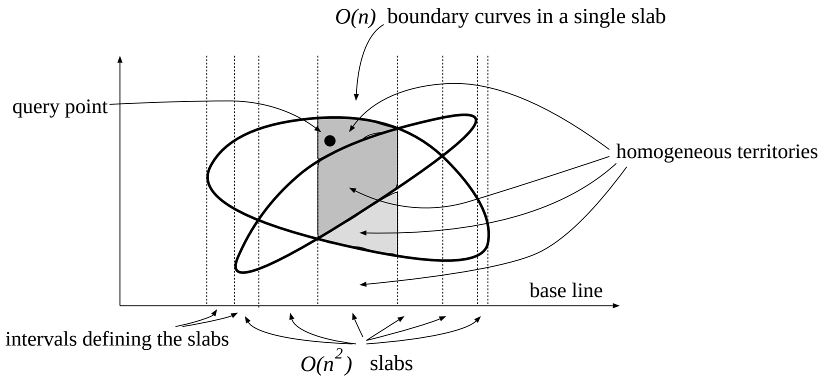

Figure 6: Organisation of the 2D complementer plane

The endpoints and the intersection points define slabs perpendicular to the base line. These slabs 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 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:

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.

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.

In order to find that projection diagram which corresponds to a ray, we have to realise 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 occurs, 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). Each region of the subdivision corresponds to those directions which have topologically equivalent projection diagrams. Thus the projection diagram can be identified by solving another point location problem having transformed the surface of the direction sphere onto an appropriate plane.

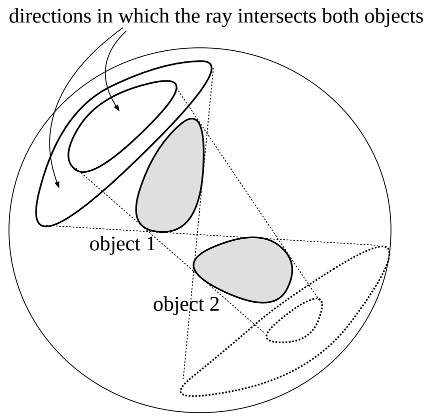

Figure 7: Directions where the topology of projection diagrams changes

The storage complexity of the complementer plane algorithm is determined by the product of the number of projection diagrams and the number of slabs in a projection diagram and the number of curves in a slab and the length of object lists in a homogeneous territory, which is in $O(n^8)$. Thus 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.

Conclusions

This paper reviewed the worst-case analysis of the ray-shooting problem focusing on the theoretical limits. The practical value of this analysis is to point out what can be done and what is not reasonable to do to speed up ray-shooting.

It has been demonstrated that ray-shooting algorithms run at least in logarithmic time in the worst-case and logarithmic algorithms use at least $\Omega(n^4)$ storage and pre-processing time which makes these algorithms not feasible in practical applications. Making the situation even worse, such logarithmic ray-shooting algorithms are rather complicated and difficult to implement. The implementation difficulties are obvious from the discussion of the two presented algorithms. Thus to speed up ray-shooting other approaches should be taken, or the worst-case optimal techniques must be combined with them. A very promising direction is the application of average-case optimal algorithms which are not better than the naive approach in the worst case, but their expected running time is $O(1)$ [MSK95]. However, the study of worst-case optimal methods of the computational geometry is not useless, since they can help us to create efficient, but practically feasible algorithms in addition to making clear where the limits are.

References

[Cla87] K. Clarkson. New application of random sampling in computational geometry. *Discrete and Computational Geometry*, 2:195–222, 1987.

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

[DL79] D. P. Dopkin and R. Lipton. On the complexity of computations under varying set of primitives. *Journal of Computer and Systems Science*, 18:86–91, 1979.

[FMSK96] Tibor Fóris, Gábor Márton, and László Szirmay-Kalos. Ray-shooting in logarithmic time. In *Winter School of Computer Graphics '96*, pages 84–90, Plzen, Czech Republic, 12–16 February 1996. University of West Bohemia.

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

[MO88] M. McKenna and J. O'Rourke. Arrangements of lines in 3-space: A data structure with applications. In *Proc. 4th ACM Symp. on Computational Geometry*, pages 371–380, 1988.

[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.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[Rei72] M. Reingold. On the optimality of some set algorithms. *Journal of the ACM*, 19:649–659, 1972.

[SY82] J. M. Steel and A. C. Yao. Lower bounds for algebraic decision trees. *Journal of Algorithms*, 3:1–8, 1982.