Worst-Case Versus Average Case Complexity of Ray-Shooting

László Szirmay-Kalos, Gábor Márton

Department of Control Engineering and Information Technology,

Technical University of Budapest,

Budapest, Műegyetem rkp. 11, H-1111, HUNGARY

szirmay@fsz.bme.hu

Abstract: This paper examines worst-case and average-case complexity measures of ray-shooting algorithms in order to find the answer to the question why computer graphics practitioners prefer heuristic methods to extensively studied worst-case optimal algorithms. It demonstrates that ray-shooting requires at least logarithmic time in the worst-case and discusses the strategies how to design such worst-case 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. In order to find average-case measures, a probabilistic model of the scene is established. We conclude that algorithms optimized for the average-case are not only much simpler to implement, but have moderate storage requirement and can even run faster for the majority of problems.

1 Introduction

Computer graphics examines the light-object interaction of surfaces in 3D space. In order to determine the reflected light of a surface point, the objects that radiate light onto this point must be known. For a single direction, this requires the emanation of a half-line — called ray — from the surface point and the computation of the object that is first intersected by this ray.

This fundamental geometric problem is called **ray-shooting**. For the complete solution of the rendering problem, a lot of rays should be shot and traced, thus it is worth preprocessing the scene into a data structure that makes ray-shooting more effective.

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 **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:

$$T(n) = \max_{o_1,\ldots,o_n \in O} t_n(o_1, \ldots, o_n), \tag{1}$$

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. 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 **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:

$$T(n) = E[t_n] = \int_{o_1 \in O} \cdots \int_{o_n \in O} t_n(o_1, \ldots, o_n)\, f_n(o_1, \ldots, o_n)\, do_1 \cdots do_n, \qquad (2)$$

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.

2 Previous work

One of the earliest results on the complexity of ray-shooting has been published by Glassner [Ge89], who stated that his octree based acceleration technique runs in logarithmic time, but his statements has been neither precisely

formulated nor proven. For example, he must have meant average-case complexity since the octree algorithm is definitely not a worst-case logarithmic method (it is easy to construct an object space for which the octree-based algorithm would run in linear time; a simple example is when all objects have a common point or are very close to each-other since this makes a cell contain $O(n)$ number of objects), but he did not specify the assumptions of the underlying probabilistic model.

Later on ray-shooting received attention in computational geometry where worst-case sub-linear algorithms have been proposed. However, to make the tools of computational geometry feasible, they consider only 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.

An excellent coverage is presented by Mark de Berg [dB92] who analyzed different special cases such as ray shooting with rays from fixed point or into fixed direction in a space of axis parallel polyhedra, c-oriented polyhedra, arbitrary curtains and general polyhedra. His main idea was to transform the ray-shooting problem into an equivalent 5D point-location problem using Plücker representation of the line of the ray and the lines of edges of the triangles composing the polyhedra. For point-location, then, logarithmic algorithms are available using, for example, cuttings. The resulting algorithm can solve the ray-shooting problem in logarithmic time using $O(n^{4+\epsilon})$ storage and preprocessing time (more precisely, for any $\epsilon > 0$, there exists a data structure of size $O(n^{4+\epsilon})$ that answers intersection queries in logarithmic time).

Others took a different compromise between the worst-case running time and the required storage. Chazelle [CEG$^+$89], for example, proposed an algorithm using Plücker coordinates that runs in $O(\log^2 n)$ time but requires only $O(n^{2+\epsilon})$ storage space.

Pellegrini, on the other hand, investigated global rays and came to the conclusion that tracing m rays among n triangles can be done in roughly $O(m^{0.8}n^{0.8})$ time [Pel93].

The paper of Schmitt, Müller and Leister [SML88] contains space/query-time lower bounds for ray shooting iso-oriented rectangles. One of their propositions states that it is possible to preprocess n iso-oriented rectangles within $P(n) = O(n\text{polylog}n)$ time into a data structure of size $O(n\text{polylog}n)$ so that an arbitrary ray query for a closest rectangle can be answered within

$Q(n) = O(n^{\alpha}\text{polylog}n)$ time, $\alpha = \log \frac{1+\sqrt{5}}{2} \leq 0.695$ (polylog stands for $\log^c$ for some $c \geq 0$). Their other proposition states that it is possible to preprocess n iso-oriented rectangles within $P(n) = O(n^3\text{polylog}n)$ time into a data structure of size $O(n^3\text{polylog}n)$, so that an arbitrary ray query for a closest rectangle can be answered within $Q(n) = O(\log^3 n)$ time.

These worst-case optimal algorithms are not only very difficult to implement but also not feasible in practice due to their prohibitive memory and preprocessing requirements. In today's complex computer graphics scenes n is in the order of $10^4 \ldots 10^7$, which prohibits the use of $O(n^4)$ or even $O(n^2)$ memory to obtain logarithmic or $O(\log^2 n)$ query times respectively.

According to a generally accepted criterion, a "good" ray-shooting algorithm runs in sub-linear time after sub-quadratic preprocessing and uses linear memory space. Thus instead of implementing the algorithms invented in computational geometry, computer graphics practitioners prefer heuristic ray-shooting speed-up techniques, including, for example,

- regular space partitioning [FTK86][AK89],
- octree [Gla84][AK89],
- ray coherence methods [OM87][HMSK92],
- ray classification [Gla84][AK89],
- Voronoi diagram based space partitioning [Már95a],

These algorithms are usually not analyzed formally. From worst-case complexity point of view, these algorithms are even worse than the naive solution. However, for the majority of the cases they seem to be better than that. Inventors of such algorithms usually intuitively justify why these methods are expected to run faster for "normal environments" than the naive implementation and demonstrate this statement by simulation [OM87].

3 Structure of this paper

This paper examines ray-shooting algorithms from the point of view of both worst-case and average-case complexity evaluation. On the side of worst-case

analysis, it tries to find the theoretical limits of such approaches and to summarize the different possibilities for the elaboration of such algorithms. In order to demonstrate the possibilities, the paper proposes an algorithm, called the **complementer plane algorithm**, that can solve the ray-shooting problem in logarithmic time having constructed an appropriate data-structure.

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 that 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.

4 Worst-case optimal ray-shooting algorithms

4.1 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.

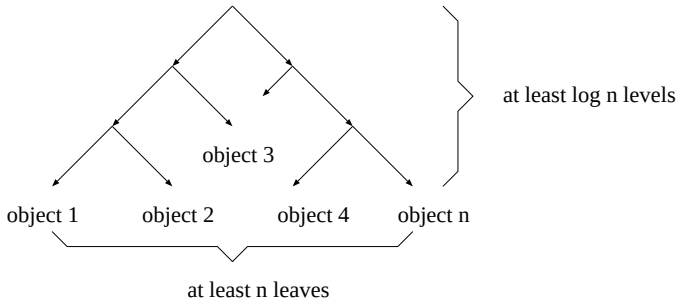

Figure 1: Computation tree

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.

4.2 Analysis of the space containing the objects and the ray

In ray-shooting five independent real parameters define the ray unambiguously. Usually, three scalars specify the origin of the ray in a Cartesian coordinate system while the remaining two scalars define the direction of the ray. 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.

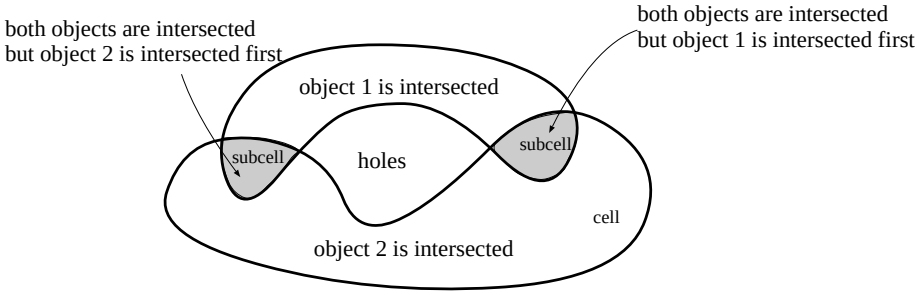

Figure 2: Visualization of the 5D space of ray parameters

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 for a class of object types, arbitrary representative from this class 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 general object types including the selected one. Thus, for the sake of simplicity, we can restrict the objects to lines.

4.3 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:

$$\begin{aligned} \vec{r}_1(t_1) &= \vec{r}_1(0) + \vec{v}_1 \cdot t_1, \\ \vec{r}_2(t_2) &= \vec{r}_2(0) + \vec{v}_2 \cdot t_2, \\ \vec{r}_3(t_3) &= \vec{r}_3(0) + \vec{v}_3 \cdot t_3, \\ \vec{r}_4(t_4) &= \vec{r}_4(0) + \vec{v}_4 \cdot t_4, \\ r\vec{a}y(t^*) &= \vec{r}_{\text{ray}}(0) + \vec{v}_{\text{ray}} \cdot t^*. \end{aligned} \tag{3}$$

The ray intersects all the four lines if:

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

This is a system of bi-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 $\binom{n}{4} = \Theta(n^4)$.

4.4 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. 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 $\Omega(n^4)$.

In order to speed-up ray-shooting, we usually use linear constructs such as bounding planes for which the linear decision tree model seems to be adequate.

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.

4.5 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 $Q(n)$:

$$Q(n) = Q(n/r(n)) + d(n) \tag{5}$$

The algorithm is exactly logarithmic ($\Theta(\log n)$) if $Q(n) = c \cdot \log n + \Delta(\log n)$. Substituting this into equation 5, we get:

$$c \cdot \log n + \Delta(\log n) = c \cdot \log(n/r(n)) + d(n) + \Delta(\log n/r) \tag{6}$$

Thus the following relation must hold asymptotically:

$$\lim_{n\to\infty} \frac{d(n)}{\log n} = \lim_{n\to\infty} \frac{\log r(n)}{\log n} \cdot c. \tag{7}$$

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 center 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 $r(n) = O(\log n)$ (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.

4.6 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.

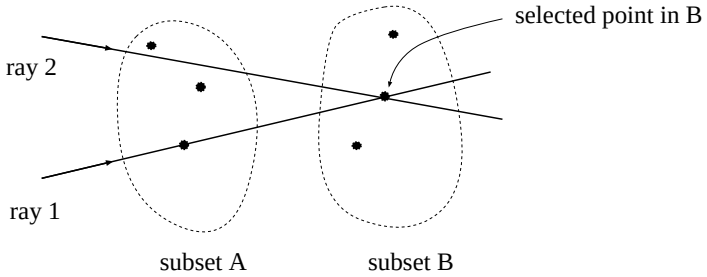

Figure 3: Indirect proof for that no single order exists in ray-shooting

At least one subset from the two — let it be subset A — has $O(n)$ size. 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 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 is 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 can be carried out in the following way. First the object in the middle of the list is tested for a possible intersection. If the ray does not have intersection with the object (just its supporting line has), then this object as well as all objects in the first half of the list are "behind" the starting point of the ray, thus this half of the list can be eliminated from the further search. If the ray does have intersection with this object, then the first intersection obviously cannot happen with an object in the second half of the list, thus this half can be eliminated.

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.

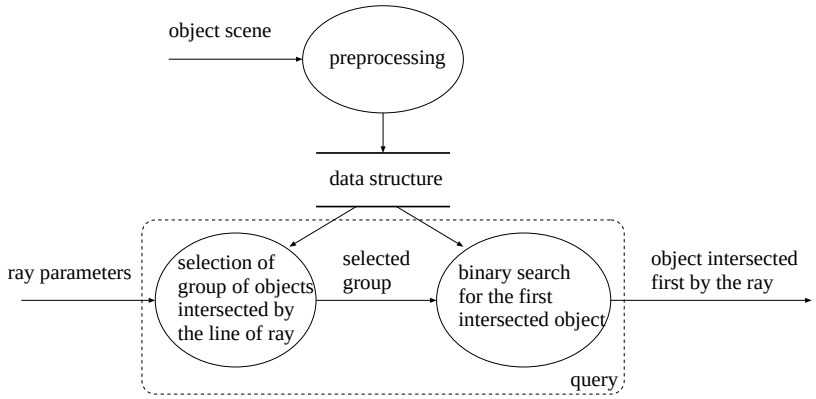

Figure 4: Dataflow of binary search type ray-shooting algorithms

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.

4.7 A Method with Worst-Case Optimal Query Time

In the subsequent sections a worst-case optimal algorithm is presented for demonstration purposes. This algorithm is of binary search type.

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

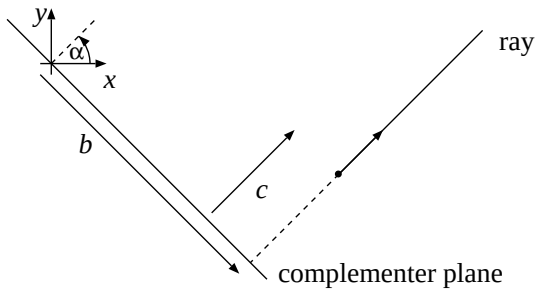

Figure 5: Representation of the ray

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 of the 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.

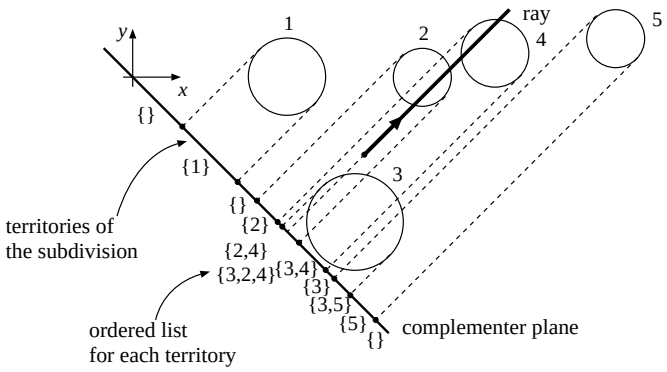

Figure 6: Organization of the data structure

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 (actually, an ordered array would also do; the usage of balanced binary tree enables the incremental construction in an efficient way). 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.

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.

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 7).

We can generate metrically unevaluated projection diagrams that will

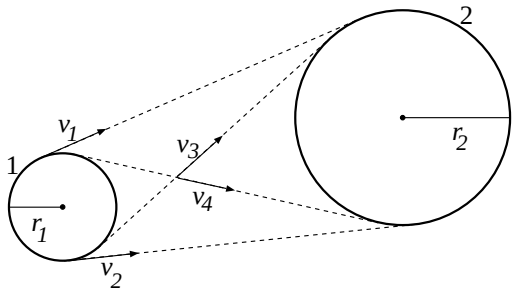

Figure 7: Changes of the structure of projection diagrams

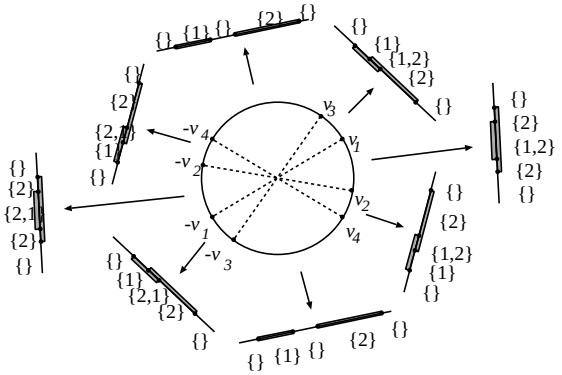

Figure 8: Diagrams of different topology

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 logarithmic time.

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

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.

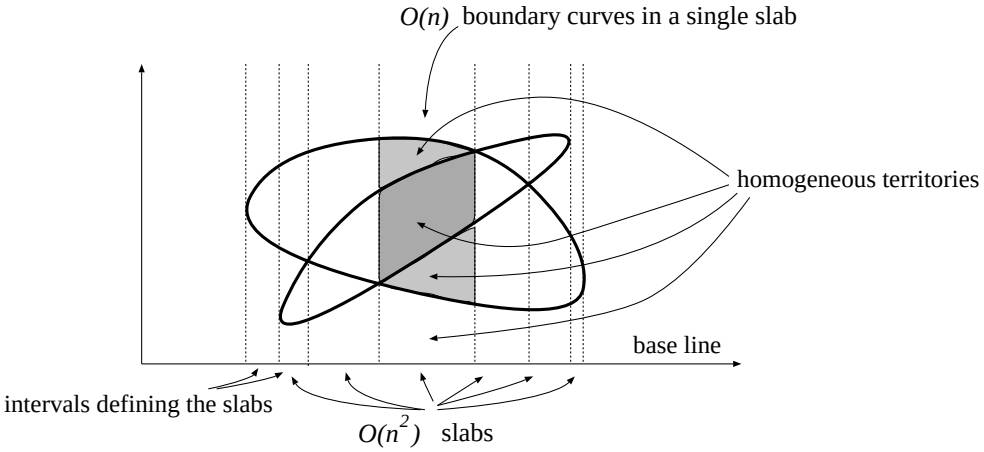

Figure 9: Organization of the 2D complementer plane

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 $O(n^2)$, provided that the objects are convex. 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.

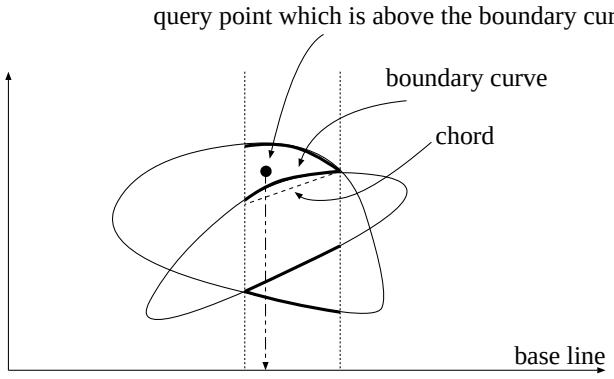

Figure 10: Classification of query point as "above" or "below" a boundary curve

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. 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 10). 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. Note that the chord has been chosen to decide whether the point is outside because it is above the projection of the object, or it is outside because it is below the projection of the object. This above-below information is needed to guide further binary searches and it is not provided by the ray-object intersection calculation.

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 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 11). 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 fact that the number of edges in the subdivision is $O(M^2)$ (each of the M circles can intersect at most $M-1$ other circles), and each edge belongs to exactly 2 territories.

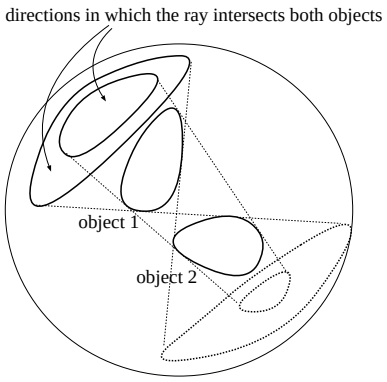

Figure 11: Directions where the topology of projection diagrams changes

Since M is $O(n^2)$ in our case, the number of regions is $O(n^4)$.

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 in terms of storage required. Although, there are possibilities to reduce this storage requirement and to get closer to the $O(n^4)$ theoretical limit, using sophisticated techniques, such as for example list compression [ST86], [Veg93]. List compression is based on the recognition that neighboring slabs are very similar, thus it is not very economical to store the associated binary trees of size $O(n)$ independently for all the $O(n^2)$ slabs. Since the number of regions without slabs is $O(n^2)$ this can reduce the the number of territories of a projection diagram from $O(n^3)$ to $O(n^2)$. However, the resulting storage complexity is still prohibitive.

5 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 2). 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 is the following:

1. The 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 Poisson point process [Lam72] (a Poisson point process $N(A)$ counts the number of points in subsets A of S in a way that a) $N(A)$ follows Poisson distribution of parameter $\rho V(A)$ where ρ is a positive constant called "intensity" and $V(A)$ is the volume of A; b) for disjoint $A_1, A_2, \ldots A_n$ sets random variables $N(A_1), N(A_2), \ldots N(A_n)$ are independent [KT75]).

This Poisson point process will be the basis of our input configuration model. Therefore, the input configuration model is refined further to the following:

1. The object space consists of spheres of the same radius r.

2. The shape centers are the realizations of a Poisson point process of intensity ρ.

Having constructed a model of the configuration space, we can start the complexity analysis of the candidate algorithms. First, a very simple, but fundamental method is analyzed, which is called the provocative algorithm.

5.1 Provocative Ray-shooting algorithm

For the sake of simplicity, assume that 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 and a query ray where this algorithm will try all the objects before it finds the nearest intersection (figure 12, for example, if $n = 6$). Thus in the worst case it is not better than the naive implementation of ray shooting.

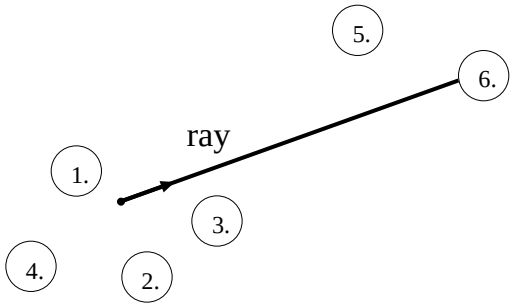

Figure 12: Provocative algorithm

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.

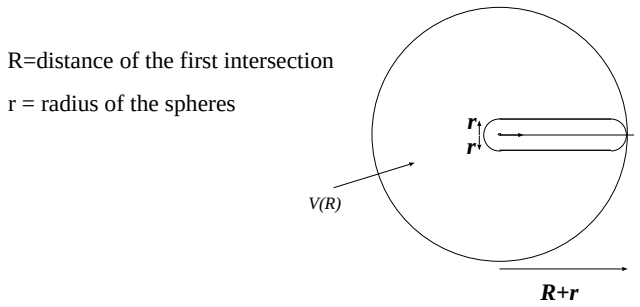

Figure 13: Region of interest

Assume that the distance of the first intersection is R. The probability conditioned by this distance that M objects have been tested before finding the intersection equals to the probability that the sphere of radius R minus the sausage-like cell of figure 13 contains M object centers.

According to the properties of Poisson point processes we have:

$$\Pr\{M \text{ tests}|\text{intersection at } R\} = \frac{(\rho V(R))^M}{M!} e^{-\rho V(R)} \tag{8}$$

where $V(R)$ is the volume of the sphere minus the sausage like cell. The conditional expected value derived from this is:

$$E[\text{number of tests}|\text{intersection at } R] = \rho V(R) + 1 \tag{9}$$

For volume $V(R)$, upper and lower bounds can be derived:

$$(R+r)^3 \frac{4\pi}{3} - r^2 \pi R - r^3 \frac{4\pi}{3} \leq V(R) \leq (R+r)^3 \frac{4\pi}{3} - r^2 \pi R \tag{10}$$

Note that $V(R)$ equals to the lower-bound if $R > 2r$. The upper-bound is only necessary for small R values which generate strange looking cells (left of figure 14).

The unconditional 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:

$$E[\text{number of tests}] = \int_0^\infty E[\text{number of tests}|\text{intersection at } R] \cdot f_t(R)\, dR \tag{11}$$

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

$$F_t(R) = \Pr\{t \leq R\} = 1 - \Pr\{t > R\}. \tag{12}$$

The complementer probability distribution $\Pr\{t > R\}$ expresses the probability that there is no ray-object intersection closer than R. 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 R.

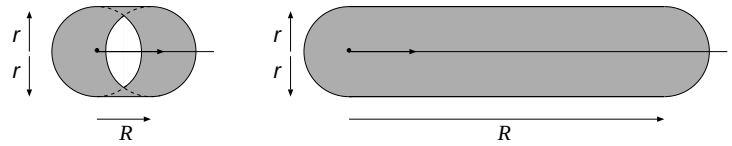

Figure 14: Volume where there are no sphere centers for $R \leq 2r$ and $R > 2r$

The shapes of this cell are shown in figure 14 for the two distinctive cases corresponding to $R \leq 2r$ and $R > 2r$. The volume $v(R)$ of the cell is:

$$v(R) = \begin{cases} \frac{4r^3\pi}{3} + r^2\pi R & \text{if } R > 2r \ , \\ \\ \frac{11\pi}{6}r^2R - \frac{2\pi}{3}R^2r + \frac{\pi}{12}R^3 & \text{if } R \leq 2r \ . \end{cases} \tag{13}$$

According to the Poisson point process assumption, the complementer probability distribution is:

$$\Pr\{t > R\} = e^{-\rho v(R)}. \tag{14}$$

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

$$f_t(R) = -\rho\frac{dv(R)}{dR}e^{-\rho v(R)} = \begin{cases} e^{-\rho\pi(\frac{4r^3}{3}+r^2R)} \cdot \rho r^2\pi & \text{if } R > 2r \\ \\ e^{-\rho\pi(\frac{11}{6}r^2R - \frac{2}{3}R^2r + \frac{R^3}{12})} \cdot \rho\pi(\frac{11}{6}r^2 - \frac{4}{3}Rr + \frac{R^2}{4}) < 3r^2\rho\pi & \text{if } R \leq 2r \end{cases} \tag{15}$$

For the case of $R < 2r$, we get rid of the complicated expression by using an upper-bound in equation 15.

Substituting the probability density into equation 11, the following upper-bound is derived for the expected number of objects tested before the intersection.

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

$$\int_0^{2r} (\rho V(R) + 1) \cdot 3r^2 \rho\pi\ dR + \int_0^{\infty} (\rho V(R) + 1) \cdot e^{-\rho\pi(\frac{4r^3}{3} + r^2 R)} \cdot \rho r^2 \pi\ dR \quad (16)$$

A lower-bound can also be obtained in a similar way:

$$E[\text{number of tests}] > \int_0^{\infty} (\rho V(R) + 1) \cdot e^{-\rho\pi(\frac{4r^3}{3} + r^2 R)} \cdot \rho r^2 \pi\ dR \quad (17)$$

Using the bounds for the volume $V(R)$ given in equation 10, we end up with the following bounds for the expected number of ray-object intersection tests:

$$\frac{4e^{\rho r^3 \pi}}{3(\rho r^3 \pi)^2}\Gamma(4, \rho r^3 \pi) - \rho r^3 \pi < E[\text{number of tests}] < \frac{4e^{\rho r^3 \pi}}{3(\rho r^3 \pi)^2}\Gamma(4, \rho r^3 \pi) + 5\rho r^3 \pi + 74\rho^2 r^6 \pi^2, \quad (18)$$

where $\Gamma(x, z)$ denotes the incomplete Γ–function:

$$\Gamma(x, z) = \int_0^z t^x e^{-t}\ dt. \quad (19)$$

The expression is quite complicated, 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 complexity of the worst case optimal algorithms which are logarithmic even in the average case.

5.2 Classical heuristic ray-shooting algorithms

The provocative algorithm can be regarded as a common idea in almost every heuristic ray-tracing speed-up technique, including, for example, *regular space partitioning* [FTK86][AK89], *octree* [Gla84] [AK89], *ray coherence methods* [OM87][HMSK92], *ray classification* [Gla84][AK89], *Voronoi diagram based space partitioning* [Már95a].

These methods overcome the limitation of the provocative algorithm that assumes fixed eye position, that is, their data structures try to express the distance from all possible positions. Since it would require infinite storage space, these algorithms use some approximation that subdivides the space into finite equivalent regions from where the objects can roughly be sorted according to their distance.

On the other hand, these algorithms 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 in [MSK95] and [Már95b]. 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.

Table 1 summarizes the results for these heuristic ray-tracing acceleration techniques. The question mark in the row of ray classification indicates that it was not possible to analyze preprocessing time and memory requirement separately since the ray classification data structure is built in a "lazy evaluation" fashion during the tracing phase instead of a separate preprocessing phase.

Table 2 surveys the worst-case complexity of these algorithms in order to allow comparison. It illustrates that the heuristic algorithms are much better in the average case than in the worst case.

5.3 Simulation results

Simulations have also been carried out in order to illustrate the theoretical results [Már95b]. Tables 3–6 show some of the simulation results, where n is the number of spheres in the scene, a is the width of spatial subdivision

Average case	$Q(n)$ query time	$P(n)$ preprocessing time	$M(n)$ memory
provocative algorithm	$O(1)$	$O(n \log n)$	$O(n)$
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
provocative algorithm	$O(n)$	$O(n \log n)$	$O(n)$
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	$O(n)$	?	?
Voronoi–diagram	$O(n)$	$O(n^2)$	$O(n^2)$

Table 2: Worst-case complexity measures

cubes, N_{I} stands for the number of intersection tests, N is the side resolution of the directional cube in the case of ray coherence method, $\overline{N_{\mathrm{I}}}$ is the measured average of N_{I}, and $\overline{M}$ is the measured average memory requirement in Megabytes. Each of the values $\overline{N_{\mathrm{I}}}$ was calculated as the average with respect to more than one million random rays shot into a scene consisting of spheres of equal radii and random centers of intensity ρ. The size of the virtual memory was about 100 Mb; this posed the upper limit for the number of spheres. The value c, also shown in the header of the tables, is another measure of object density: it is the percentage of space occupied by the objects, that is the occupancy ratio; in fact $c = 100 \cdot 4\rho r^3\pi/3$.

The theoretical values are shown as intervals in the tables. The results of the octree method are not shown, since it does not result in savings in intersection tests (it does, however, save number of cell-steps).

Two kinds of deviations can be observed in the tables comparing the theoretically predicted and the measured values. The first kind of deviation is that measured values are much better (lower) in the case of small occupancy ratios than the predicted ones. The reason is that the number of spheres is finite in the simulation, and the probability that a ray does not intersect any of the spheres increases as the occupancy ratio decreases; in the Poisson–model, however, any ray intersects at least one sphere (actually an infinite number of spheres) with probability 1. In other words, the situation of small occupancy can hardly be well simulated. The second kind of deviation is that for the spatial subdivision methods (regular grid, Voronoi-diagram), the measured number of intersection tests are worse (higher) in the case of large occupancy ratios than the predicted ones. The reason is that in our implementation the criterion of putting an object onto the list assigned to a given spatial cell is not perfectly that the object intersects the cell but weaker than that: an object is put onto the list if there is no face of the cell the plane of which separates the object from the cell.

There is one more interesting observation in the case of regular space partitioning (Table 3). There is a small but persistent fluctuation in the average number of intersection tests ($\overline{N_{\mathrm{I}}}$), with local minima at $n = 50000$ and $n = 500000$, which is independent of c. The cause of the phenomenon lies in the way the algorithm has been implemented. According to the widely accepted heuristics [FTK86] on the one hand, the number of cubes in the space partitioning is chosen to be proportional to the number of objects. On the other hand, the preprocessing algorithm that builds the space subdivision

REGULAR SPACE PARTITIONING, $r = 1$, $a \in [1,2]\rho^{-1/3}$					
c [%]	ρ	$E[N_{\mathrm{I}}]$	n	$\overline{N_{\mathrm{I}}}$	$\overline{M}$ [Mb]
0.5	0.00119	[483.6, 1154.3]	10000	48.5	0.8
			20000	87.7	1.5
			50000	55.1	4.0
			100000	93.6	7.7
			200000	161.8	15.2
			500000	95.1	39.3
1.0	0.00239	[304.6, 747.7]	10000	47.2	0.8
			20000	82.5	1.6
			50000	51.2	4.1
			100000	82.4	7.8
			200000	136.0	15.3
			500000	77.3	39.9
2.0	0.00477	[191.9, 490.5]	10000	44.9	0.8
			20000	75.4	1.6
			50000	45.4	4.1
			100000	69.6	7.9
			200000	110.0	15.5
			500000	60.4	40.8
5.0	0.01190	[104.2, 290.1]	10000	40.5	0.8
			20000	63.9	1.6
			50000	37.0	4.3
			100000	53.5	8.2
			200000	82.4	15.9
			500000	44.3	42.3
10.0	0.02390	[65.6, 202.9]	10000	36.9	0.9
			20000	56.5	1.7
			50000	31.7	4.5
			100000	45.3	8.5
			200000	69.2	16.3
			500000	37.1	44.1
20.0	0.04770	[41.3, 150.0]	10000	33.8	0.9
			20000	52.0	1.7
			50000	28.7	4.8
			100000	40.8	8.9
			200000	62.8	16.8
			500000	33.8	46.7
50.0	0.11900	[22.4, 114.3]	10000	32.9	1.0
			20000	50.5	1.8
			50000	28.0	5.4
			100000	39.5	9.7
			200000	60.6	18.0
			500000	33.0	52.1

Table 3: Regular space partitioning — simulation results

has been implemented in a recursive (divide-and-conquer) fashion, hence the resolution of the subdivision should be a power of 2 in each of the three spatial directions. These two facts imply that as the number of objects is increasing, the overall resolution of the subdivision drastically increases (by a factor of 8) each time when the number of objects exceeds a power of 8. In the range 50000–500000 there are exactly two powers of 8: the first one is $2^{15} = 32768$ (between 20000 and 50000) $2^{18} = 262144$ (between 200000 and 500000). And the higher the resolution of the subdivision, the less number of intersection tests needed.

RAY COHERENCE, $r = 1$, $N = 100$					
c [%]	ρ	$E[N_\mathrm{I}]$	n	$\overline{N_\mathrm{I}}$	$\overline{M}$ [Mb]
0.5	0.00119	[9.0, 115.9]	10000	3.8	2.1
			20000	2.8	4.1
			50000	2.8	10.8
			100000	2.7	22.3
1.0	0.00239	[6.1, 38.7]	10000	3.8	2.2
			20000	2.8	4.4
			50000	2.7	11.5
			100000	2.5	24.2
2.0	0.00477	[4.9, 16.1]	10000	3.7	2.4
			20000	2.6	4.7
			50000	2.4	12.6
			100000	2.2	26.7
5.0	0.01190	[4.3, 7.8]	10000	3.5	2.7
			20000	2.3	5.3
			50000	2.0	14.7
			100000	1.8	31.9
10.0	0.02390	[4.1, 6.1]	10000	3.2	2.9
			20000	2.0	6.1
			50000	1.7	17.1
			100000	1.6	37.8
20.0	0.04770	[4.0, 5.6]	10000	3.0	3.3
			20000	1.8	7.1
			50000	1.6	20.5
			100000	1.4	46.3
50.0	0.11900	[3.9, 6.4]	10000	2.8	4.1
			20000	1.6	9.1
			50000	1.5	27.5
			100000	1.4	63.8

Table 4: Ray coherence method — simulation results

RAY CLASSIFICATION, $r = 1$, $a \in [1,2]\rho^{-1/3}$					
c [%]	ρ	$E[N_{\mathrm{I}}]$	n	$\overline{N_{\mathrm{I}}}$	$\overline{M}$ [Mb]
0.5	0.00119	[63.4, 1470.2]	10000	78.3	22.5
			20000	142.9	43.7
1.0	0.00239	[32.9, 594.4]	10000	73.1	22.9
			20000	129.2	44.4
2.0	0.00477	[19.2, 275.4]	10000	66.3	23.5
			20000	112.4	45.3
5.0	0.01190	[10.8, 119.8]	10000	55.7	24.4
			20000	89.2	46.8
10.0	0.02390	[7.5, 71.6]	10000	48.0	25.4
			20000	74.8	48.3
20.0	0.04770	[5.5, 46.5]	10000	42.7	26.6
			20000	64.3	48.9
50.0	0.11900	[3.9, 29.9]	10000	39.2	28.8
			20000	59.7	51.7

Table 5: Ray classification — simulation results

5.4 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 or not 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

VORONOI-DIAGRAM, $r = 1$					
c [%]	ρ	$E[N_\mathrm{I}]$	n	$\overline{N_\mathrm{I}}$	$\overline{M}$ [Mb]
0.5	0.00119	[7.5, 5571.7]	10000	19.2	7.5
			20000	23.0	15.0
			50000	27.9	37.3
1.0	0.00239	[4.7, 2218.4]	10000	17.7	7.5
			20000	20.9	15.0
			50000	24.6	37.3
2.0	0.00477	[3.0, 886.2]	10000	16.4	7.5
			20000	18.6	15.0
			50000	21.2	37.4
5.0	0.01190	[1.6, 266.7]	10000	15.6	7.5
			20000	17.4	15.0
			50000	18.6	37.6
10.0	0.02390	[1.1, 110.1]	10000	17.1	7.6
			20000	18.9	15.2
			50000	19.7	37.9
20.0	0.04770	[0.7, 48.1]	10000	24.5	7.8
			20000	26.8	15.6
			50000	28.7	39.0
50.0	0.11900	[0.4, 20.8]	10000	82.3	10.1
			20000	77.6	19.1
			50000	82.8	48.2

Table 6: Voronoi-diagram — simulation results

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 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 and these logarithmic algorithms require very high storage space and preprocessing time. A general framework has also been presented to discuss the construction of such algorithms. An algorithm, called the complementer plane algorithm has been introduced, that really has the optimal logarithmic worst-case time complexity.

The storage requirement 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.

6 Acknowledgements

This work has been supported by the National Scientific Research Fund (OTKA), ref.No.: F 015884 and the Austrian-Hungarian Action Fund, ref.No.: 29p4, 32öu9 and 34öu28.

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.

[CEG+89] B. Chazelle, H. Edelsbrunner, L. Guibas, R. Pollack, and M. Sharir. Lines in space — combinatorics, algorithms and applications. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 382–393, 1989.

[dB92] M. 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.

[FTK86] Akira Fujimoto, Tanaka Takayuki, and Iwata Kansei. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[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.

[KT75] S. Karlin and M. T. Taylor. *A First Course in Stochastic Processes.* Academic Press, New York, 1975.

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

[Már95a] Gábor Márton. Acceleration of ray tracing via voronoi-diagrams. In Alan W. Paeth, editor, *Graphics Gems V*, pages 268–284. Academic Press, Boston, 1995.

[Már95b] Gábor Márton. *Stochastic Analysis of Ray Tracing Algorithms.* PhD thesis, Department of Process Control, Technical University of Budapest, Budapest, Hungary, 1995.

[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.

[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.

[Pel93] M. Pellegrini. Ray shooting on triangles in 3-space. *Algorithmica*, 9:471–494, 1993.

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

[SML88] Alfred Schmitt, Heinrich Müller, and Wolfgang Leister. Ray tracing algorithms — theory and practice. In R. A. Earnshow, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 997–1030. Springer-Verlag, Berlin, Heidelberg, 1988. NATO ASI Series, Vol. F40.

[ST86] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

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

[Veg93] G. Vegter. The visibility complex. In *Proceedings of 9th Annual ACM Symp. on Computational Geometry*, pages 328–337, 1993.