Global Visibility Computations

Jiří Bittner

xbittner@sgi.felk.cvut.cz

Diploma Thesis

Czech Technical University Prague

Faculty of Electrical Engineering, Department of Computers

January 1997

Contents

Preface

This diploma thesis deals with the problem of visibility computations in object space. The description of restrictions in visibility for a bounded region of virtual environment can significantly improve the visualization process and brings many advantages in applications of virtual reality.

The first chapter describes motivation of the research in the field of visibility preprocessing. It defines important features of different approaches of visibility computations and gives an overview about related work. Second chapter discusses the spatial subdivision, which is the first step necessary to perform to gain knowledge about the structure of the virtual environment. The visibility determination is the subject of third chapter. Several ways how to process visibility is shown there, each having different complexity in both theoretical and computational resources. Chapter number four documents one of the most important part of conservative visibility determination - polyhedra enumeration in arbitrary dimension. Chapter five is devoted to implementation issues as an extension to previous chapters. It describes the current implementation, points some troubles encountered in the practical usage of the theoretical concept. In sixth chapter an overview about empirical measurements of features of visibility computations is given. Finally, chapter seven contains summary information and drafts subjects of future work.

Chapter 1

Introduction

1.1 Motivation

Identifying objects, that are visible from a set of viewpoints is an important task in computer graphics. In many complex geometric models of virtual environments, most parts of the model are pairwise invisible, what means there is no direct illumination interaction between them. Despite this, hidden-surface algorithms like *Z-buffer* or *BSP-tree* often spend significant time processing the invisible portions of the *model of virtual environment*. Perhaps it is worthwhile to study restrictions in visibility and to apply some sophisticated methods of visibility preprocessing to speedup queries for visible portion of the model, given certain viewpoint or a set of viewpoints.

Consider a model of a town describing its streets, buildings and their interiors. It is clear, that such model will consist of a very high number of objects. Each such object is usually represented by many geometrical primitives. Table 1.1 illustrates number of the geometrical primitives (polygons) contained in different models of single building. Two of these models are showed on figures 1.1 and 1.2.

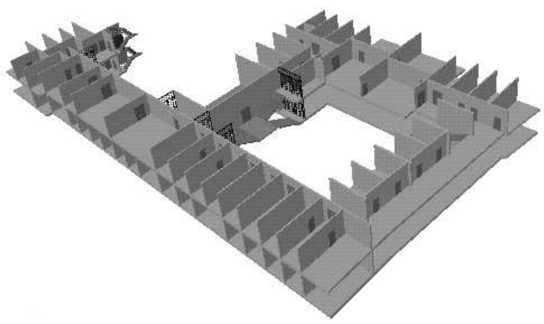

Figure 1.1: The model of the interior of the Department of Computer Science.

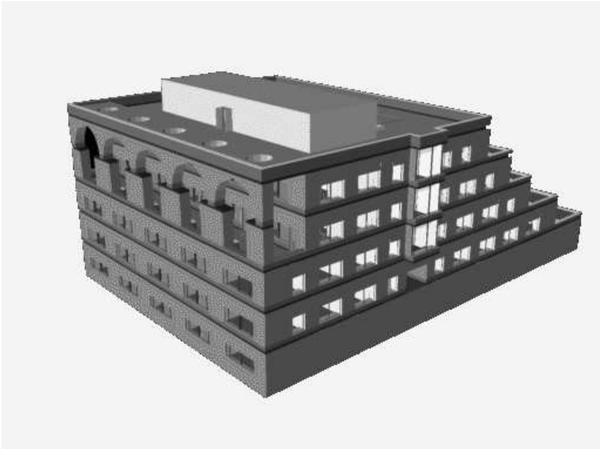

Figure 1.2: The model of the building of the Department of Computer Science at U.C. Berkeley.

Applications, which are said to be interactive, should give a visual response in not more then one tenth of a second. Visualization of the model of virtual environment (MVE) has to be performed in a real-time, while an observer (user) is moving through the model. Hence, the frequency of ten Hz is the minimal *rendering frequency*, when the observer still obtains consistent and visually pleasant experience.

Scene	No. of polygons
Berkeley	9129
Dept. of CS	9478
Furnished Berkeley	2M

Table 1.1: Number of geometrical primitives in several virtual environments.

Although modern graphics workstations equipped with graphics accelerators, such as Rendering Engine or Geometry Engine, are able to render many thousands of (small) polygons per second, this speed may be not sufficient for models describing very large environments, such as those mentioned above. An example of capabilities of the state-of-art hardware for computer graphics is given in table 1.2. This table shows rendering capacity of several graphics accelerators manufactured by Silicon Graphics. The *TMesh* column depicts the throughput of visualization of a triangular mesh in number of rendered triangles per second. The triangles are considered to be 10×10 - 50 pixels each, in true color, unlighted, flat shaded, Z-buffered and arbitrary oriented. The *Polygons* column shows a number of 10×10 pixels large single polygons, which can be rendered per second. These polygons are assumed to be true color, independent of each other, lighted, Gouraud shaded, Z-buffered and arbitrary oriented. Since the *polygon throughput* was not published for some of the rendering engines, the corresponding cells in the table are struck through. Note that a bigger size of rendered polygons can significantly affect the polygon throughput, leading to much worse results than ones shown in the table.

Accelerator	TMesh	Polygons
XL	60k	—
XZ	265k	150k
Extreme	470k	195k
Reality Engine	1.8M	659k
i-station	>5M	—
Infinite Reality	>10M	—

Table 1.2: The rendering speed of some graphics accelerators.

Probably user demands of realism of experience in virtual environment will always lead to models, that are not likely to be rendered at an interactive frame rate, using some naive algorithm, although we process them by the state-of-art hardware. Therefore, it is necessary to gain some more knowledge about the structure of the virtual environment and to use this information in further processing, i.e. during the visualization process.

Apparently, to be able to move interactively in a virtual environment, we have to reduce the amount of geometrical primitives passed to the rendering subsystem. We can conveniently exploit the fact, that at each moment when the observer moves through the environment, probably only a small fraction of it is currently visible. For example, an observer currently visiting a "virtual rector's office", cannot see to the kitchen, which is on different floor in the building.

1.2 Related Work and Theoretical Background

In this section some references to the background and research related to visibility computations are presented. Exhaustive research has been conducted in the area of photo-realistic visualization algorithms such as ray-tracing [1]. However, here and further methods of visibility determination from a single point are not concerned, but we focus just on work related to visibility preprocessing for certain regions (region visibility queries).

Most of the methods of visibility determination use some kind of ordering, such as spatial subdivision. Each partition should represent a class of equivalence in the set of *visual perceptions* of the scene. In other words, the visual perception of the scene has to remain the same or similar for all the view points enclosed by the particular partition.

1.2.1 Exact Visibility

Decent amount of research has been done in the field of *exact visibility calculation*. Determination of exact visibility is adequate to computing an exact description of the set of visible elements of the MVE for particular regions of the environment. Exact visibility computation found its applications mainly in the area of representing three-dimensional object by a set of two-dimensional views and in the field of recognition of 3D objects [2]. The main idea behind exact visibility computation is based on partitioning of the viewpoint space into distinct regions. For all the viewpoints enclosed in each region the structure of the scene drawing must *qualitatively* remain the same. The structure created by such a partitioning is called *aspect graph.* The aspect graph encodes the representation of exact visibility for every qualitatively distinct region. Informally at each vertex of the aspect graph there is a view (aspect), that is a representative of the projection of the scene from a set of viewpoints. For all the viewpoints in the aspect the scene appears to be similar, i.e. it changes just *quantitatively.* Two aspects are adjacent in the graph if the corresponding regions are adjacent. The difference of the scene projections between two adjacent aspects is called *visual event.*

The construction of the aspect graph is computationally very complex [30, 29, 28]. The fastest known algorithms for orthographic projection and polyhedral objects can compute one partition of the aspect graph of a single polyhedral object with the time complexity $O(n^4 \log n + m \log m)$, where n is the number of vertices and m is the size of the partition of the object. Also the number of qualitatively distinct viewpoint regions may be very large ($O(n^9)$). Another drawback of this approach is that the currently visible portion of the scene may have higher complexity[1] than the input scene itself - a scene with n polygons can have a visible portion of size $O(n^2)$. These features of aspect graph make it hardly applicable in the field of virtual reality, where the complexity of visualized environments is mainly too great to allow aspect graph construction.

Exact visibility algorithms have also been developed for viewpoints moving along prespecified paths [34, 35]. Another exact visibility algorithm - the *hierarchical Z-buffer* [36] uses a *temporal* coherence by maintaining a list of polygons that are visible from the current viewpoint. For the next viewpoint, the algorithm draws polygons from this list first. The contents of the hierarchical Z-buffer is used to *cull* those polygons which are invisible. To exploit spatial coherence, this algorithm requires that the Z-buffer supports visibility queries. Such queries are usually not efficiently supported in most graphics hardware, and simulating the Z-buffer in software generally involves significant overhead.

1.2.2 Conservative Visibility

Conservative visibility is another framework, how to deal with the visibility calculations. Conservative visibility is an approximation to the exact visibility approach and we do not insist on determination of the exact regions within which the visibility of the scene does not change qualitatively. This assumption leads to fewer restrictions and gives a lot more freedom to see the problem from a different point of view.

In the case of exact visibility the concatenation of the viewpoints into aspects is driven by information about the visual events in the scene. To compute the conservative visibility a spatial subdivision is assumed to be constructed in advance. During the visibility computation the set or the *superset* of all possibly visible objects in all viewing directions from all possible viewpoints has to be determined for every cell (region) of the spatial partitioning.

Let us assume that we are interested only in determining which other cells are possibly visible from the cell corresponding to location of an observer. Such a query is called *cell-to-*

[1] after subdivision of its elements

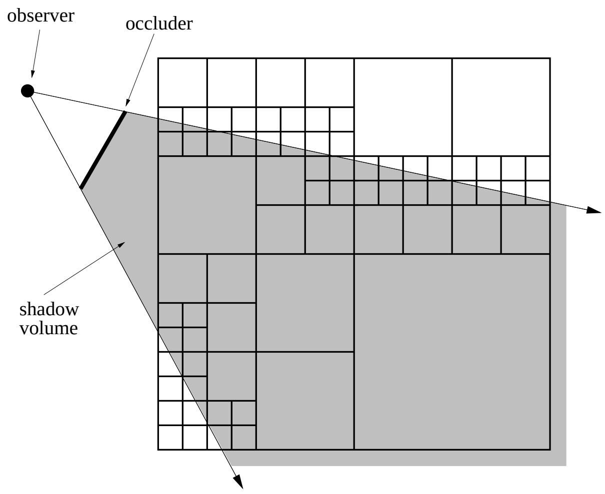

Figure 1.3: Visibility determination in a hierarchical data structure.

cell visibility computation. Apparently, this is an easier task than computing more precise, therefore less conservative, *cell-to-frustrum* visibility, which describes also the portions of the cells potentially visible from the source cell.

Generally the static visibility computation classifies some of the elements of the virtual environment as visible, while in reality they are obscured to an observer with a particular viewpoint and viewing direction. This is basically quite small penalty to pay, in compare to the benefits of the static visibility over the exact visibility approach. In the time of rendering misclassified elements are considered to be paint away by the rendering hardware, or a "lower level" visibility algorithm, such as BSP-tree or Z-buffer. On the other hand, misclassifying an object as invisible may result in presenting a wrong visual cue to the observer. Such errors disturb the user and of course break the realism. Basically, the number of elements of the scene which will be rejected as invisible by the visibility preprocessing grows with density and number of the elements in the scene, although it is strongly dependent on the type of the scene. For models of interior environments of certain type the number of visible objects, or in another words the size of the visible region, remains constant in average. This observation leads to the claim, that knowing the restrictions in visibility, similar number of visible elements must be considered for rendering at each moment, regardless of the size of the environment.

Described outlines of conservative visibility correspond to the work documented in this thesis. This approach is based on the research of Seth J. Teller and his colleagues [31, 38, 32]. Results of their work, were used in the project called RING. From the information we have, it is rather advanced project, focusing mainly on maintaining a constant frame rate during the simulation of environment using distributed database management. It is the research project of the University of California in Berkeley led by Prof. Carlo Séquin.

Other work in the field of conservative visibility can be found in [33], where it has been used to design an optimal algorithm for determining visibility in a scene containing only rectangles with sides parallel to the x and y axes.

A novel method for computing temporally coherent conservative visibility is shown in [39]. This algorithm uses a hierarchical spatial subdivision and description of imminent visual events. The determination of the visible portion of the model is based on culling polygons "hidden" behind major *occluders* (see figure 1.3). This approach seems to be much more independent on scene structure than one based on the opposite of occluders - portals, such as presented in this thesis. However, due to the fact that more computations are performed on-line, the algorithm spends more time determining the visible set of geometric primitives. For interior environments the algorithm based on cells and portals gives probably better results in both - time complexity and the size of approximation of the visible portion of the model.

1.3 Overview of the Implemented Approach

To consider the restrictions in visibility, we attempt to divide the model of virtual environment into regions, for which we can determine the part of the model potentially visible from all viewpoints enclosed by each of the regions. Notice that this is not a problem of determining visibility from a static location (point). We consider a *static visibility* in arbitrary direction from a closed but unlimited set of viewpoints defined by a volume. The viewpoint dependent type of visibility determination is henceforth call *dynamic visibility.*

To determine the static visibility, we exploit the observation, that the model of an environment, namely architectural model, typically consist of large connected elements of non-transparent material such as partitions or walls. If an observer is located in a region surrounded by such *occluders*, significant part of the scene is occluded by them. This implies, there is no need to involve this "invisible" part (*occludees*) of the scene in the dynamic visibility computation and hence in the rendering process.

Based on this observation, the scene is divided into *cells*, using an algorithm of spatial subdivision. We will describe one possible approach of spatial subdivision in chapter 2. An overview about requirements for the spatial subdivision is presented together with the description of postprocessing to meet criteria needed by further visibility computations. An example of postprocessing of the spatial subdivision is the portal enumeration and cell concatenation. During visibility computations, for each cell a set of potentially visible cells is determined. In the case of more precise calculations we enrich this information by a hierarchical data structure – *visibility tree.* Furthermore we describe the potentially visible regions – *visible frustrums.* Determination of these frustrums corresponds to antipenumbra computation for an areal light source. The theoretical background and algorithms necessary to compute the antipenumbra are subjects of chapters 3 and 4. Describing the visible regions helps us, when retrieving visibility information about particular objects or lower level geometrical primitives enclosed in other cells. This description of visibility restrictions, computed for each cell, is stored as a complementary information of the scene [18]. It can be used for on-line dynamic visibility determination as shown in chapter 5, as well as other processing of the MVE. An example of the other field of applicability of visibility computations are lighting calculations in radiosity based algorithms.

During dynamic visibility computations, knowing particular viewpoint and viewing direction, only the set of objects referenced as potentially visible from the *source region* is considered.

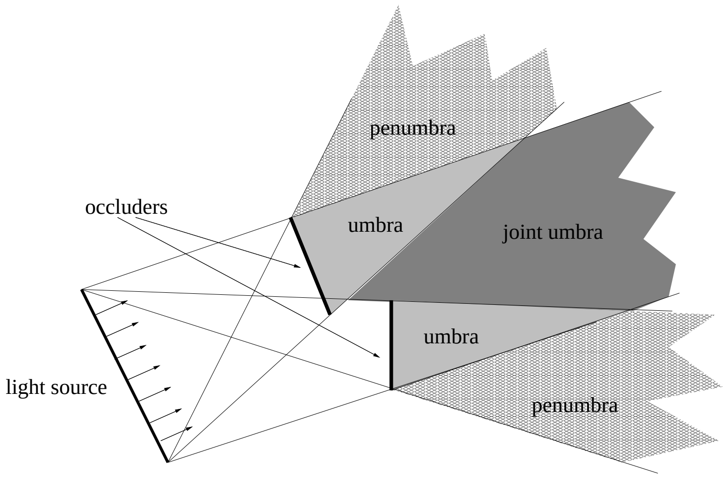

Figure 1.4: Visibility propagation from an areal light source in presence of occluders.

Note, that the source region is the one that contains the current viewpoint. As it has been stated previously, there is high probability that this subset of the database describing the virtual environment is significantly smaller than the entire database. This probability even grows with the number of objects, particularly those classified as occluders, in the environment. Hence it is evident, that the spatial subdivision and the visibility preprocessing can considerably influence the visualization process in positive manner.

As reader could already find out, presented method of object space visibility preprocessing basically consists of two steps. These steps are rather independent, so separate chapters are devoted to each of them. The *spatial subdivision* is used to divide the model into spatially disjunct cells. The *visibility computations* are performed then, to classify the visibility in each of the cells of the spatial subdivision.

In the the next chapter we present an algorithm that subdivides the space of a MVE into cells which are further used to compute the cell-to-cell and cell-to-frustrum conservative visibility.

Chapter 2

Spatial Subdivision

The spatial subdivision of the virtual environment is the first and very important step preceding the conservative visibility determination. In relation to visibility this subject was already examined in [38] as well as in the frame of several semester projects [3, 4].

Definition 1 Let us define a *cell* C^d in d-dimensional space as a d-dimensional region with $(d-1)$-dimensional boundaries. We will denote the set of points of the d-dimensional space creating the boundary of the cell by ∂C^d.

In the three-dimensional space the cell is a general polyhedron with planes as its boundaries. We will show below, that particularly in our case the planes are restricted to planar polygons and moreover convex polygons.

Definition 2 A *spatial subdivision* (SSD) is a set of closed, disjunct and convex cells intersecting only on their boundaries. As an example of three-dimensional SSD see figure 2.1.

Note, that the convexity of the cells of the spatial subdivision implies that also their boundaries are convex.

Definition 3 Two cells C_1^d and C_2^d are said to be *neighbors* in a particular spatial subdivision if and only if

$$\partial C_1^d \bigcap \partial C_2^d \neq \emptyset$$

Typically the neighbor information is maintained during the process of construction of the spatial subdivision. However, this information can be extracted by post-processing of the SSD. When the SSD is created constructively by hand, the neighbor information should be explicitly defined, taking care that it satisfies the definitions 2 and 3.

Definition 4 Define an *occluder* as an opaque, convex, planar polygon which sufficiently occludes some portion of the virtual environment. We say that an occluder in three-dimensional space is *affine* to the plane if the plane completely embeds it. (Figure 2.2)

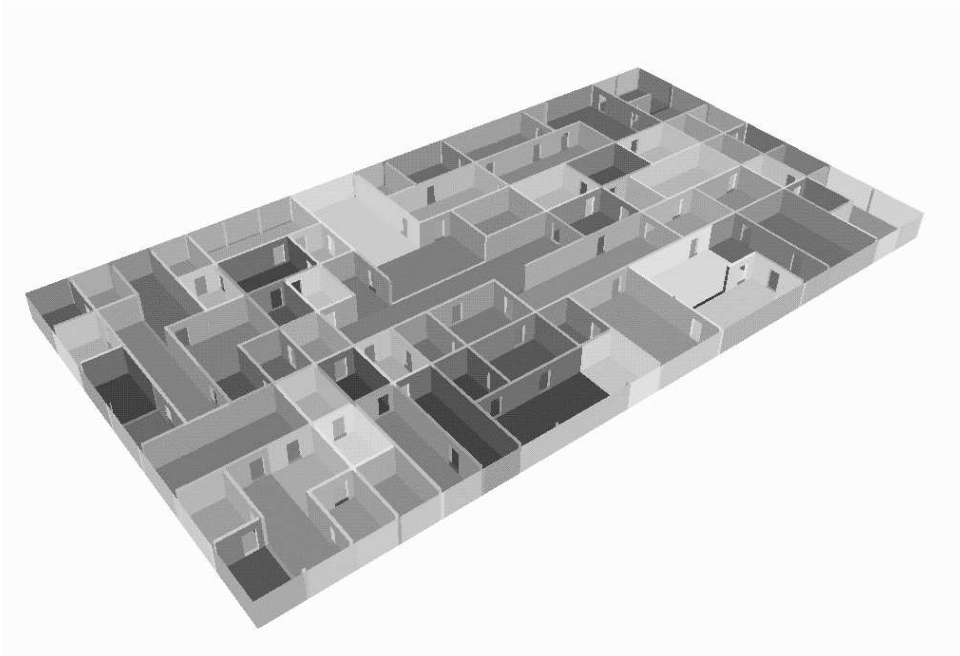

Figure 2.1: Example of the spatial subdivision in three dimensions.

Although, in reality, major occluders and detail objects are interchangeable, the subdivision and visibility algorithms presented here treat them differently. Variety of heuristics (e.g. size, connectivity to other entities) can serve to distinguish if a particular element referenced in the database should be considered an occluder or a detail object. From now on we will consider all the detail objects to be *non-occluding*, hence transparent, although they themselves can be subject to visibility determination.

Definition 5 For a cell C^d, *portals* are defined as a convex decomposition, or a convex hull of a set difference of all boundaries ∂C^d and the union of occluders coaffine to each particular boundary. In other words, portals form the transparent parts of the cell boundary (Figure 2.2).

For purposes of visibility computations and further processing of the MVE the spatial subdivision must support some elementary operations, namely point location, neighbor finding and portal enumeration. A spatial subdivision which supports all these operations is called *conforming*.

Suppose that all the polygons (occluders) which are contained in the input set of the subdivision algorithm are simple, convex and are defined by a counterclockwise ordered sequence of their vertices v_i, where $i \geq 3$. The normal vector $\vec{n}$ of each polygon can be computed as a cross product of two of its edges as follows:

$$\vec{n} = \vec{e}_{3\to1} \times \vec{e}_{1\to2}, \tag{2.1}$$

where $\vec{e}_{i\to j}$ denotes an edge of the polygon defined by ordered tuple of vertices $[v_i, v_j]$. The convexity restriction on the input data is only marginal, because if the data are not provided

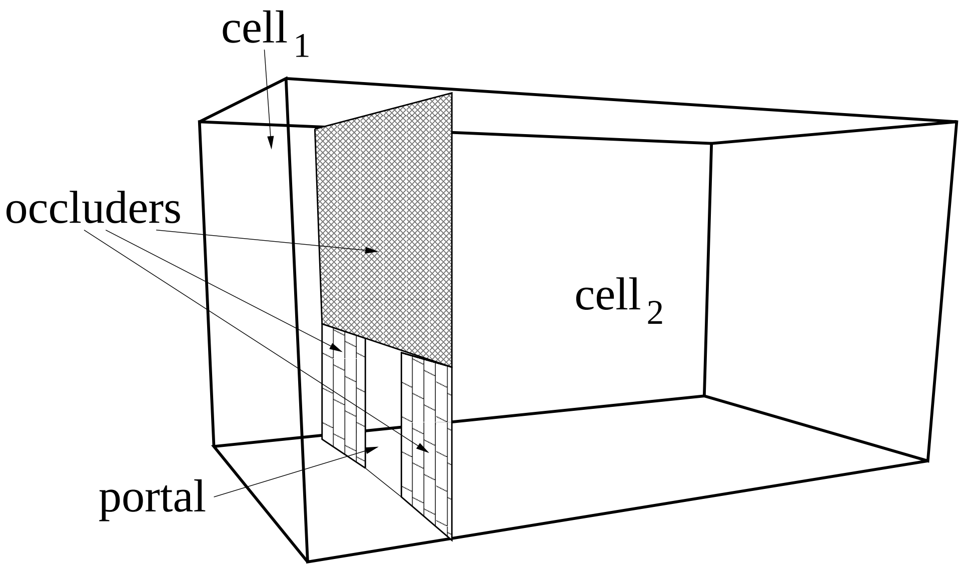

Figure 2.2: Set of three occluders with a portal connecting two cells.

in the requested form, any convex decomposition of each of the polygons can be computed in advance.

In general, an infinite number of spatial subdivisions satisfying the definition 2 exists. We are interested in one, with minimal number of objects visible from each cell in average. Let us define $f(SSD)$ to be a function, which sums the number of visible cells through each of the cells created by the SSD.

$$f(SSD) = \sum_{i=0}^{i=\Omega(SSD)} C(SSD)_i \tag{2.2}$$

where:

- ⋄ SSD is an arbitrary spatial subdivision
- ⋄ Ω is a number of cells created by the subdivision
- ⋄ $C(SSD)_i$ denotes the number of cells visible from the cell C_i

Then the formalization of the requirement on the optimality of the spatial subdivision can be expressed by following equation:

$$SSD_{opt} = \{SSD_{opt} \in SSD^* : f(SSD_{opt}) \leq f(SSD), \forall SSD \in SSD^*\}, \tag{2.3}$$

where:

- ⋄ SSD_{opt} is the set of optimal spatial subdivisions
- ⋄ SSD^* is the closure of all possible spatial subdivisions

To our knowledge, no method for construction of the spatial subdivision provably satisfying this criterion is known. Based on [22] we use heuristics, which, mostly tends to converge to the desired optimal solution. It exploits the binary space partitioning (BSP) and construction of the BSP tree.

2.1 Requirements of Conforming *SSD*

2.1.1 Point Location

The *SSD* supports the operation of *point location*, if given a query point in $\mathcal{R}^d$, the cell containing that point can be determined. Mainly, most of the point location queries are highly coherent, that is, the result is strongly correlated with that of previous queries. This coherence can be exploited to speed up the point location task in practice.

2.1.2 Object Population

Detail objects can be arranged efficiently, using the *population* into the cells of the spatial subdivision. A spatial cell is *populated* with an object, if the object, or possibly some simpler *bounding representation* of it, intersects the cell.

2.1.3 Neighbor Finding

Cell in the spatial subdivision in d-dimensional environment intersects it's neighbors on the $d-1$-dimensional regions, or boundaries. Typically, neighbor information is maintained as a invariant during the construction of the spatial subdivision. Alternatively, neighbor information is recovered in a post-processing stage. Another option is to compute neighbor information *lazily*, as needed.

2.2 Binary Space Partitioning

Building of the BSP tree is a process of recursive subdivision of the space by planar planes. The root node of the BSP tree data structure represents the convex hull of the database containing the geometric description of the subdivided scene. Every other node of the BSP tree is either *leaf*, or an internal node assessed by a *splitting plane* and two *child nodes*. The child nodes correspond to each half-space induced by the splitting plane. Each cell of the BSP tree represents the common intersection of all positive or negative half-spaces encountered while traversing the tree from the root to the particular cell, respectively. Notice, that the cells are always a convex polyhedra.

In practice it is convenient to replace the convex hull as the root of the BSP tree by a rectangular box enclosing the entire scene. This way we omit the construction of the convex hull, which can be time consuming and which is hard to implement robustly in three-dimensional environments with large numbers of objects. The most important factor of the BSP tree construction is the splitting criterion, which has a great influence on the properties of constructed tree.

Common demands posed on the BSP tree in different applications are:

- low depth
- well balanced structure

- memory efficiency.

Constructing the binary space partitioning so that the representation of the final tree is memory efficient is a complex problem. The current best time complexity of the tree construction is $O(n^3)$ for n polygons. The bound for memory requirements of the BSP tree structure is $O(n^2)$ [5].

2.2.1 Splitting Criterion

In compare to other applications in which the BSP trees are used, our goal is not to get well-balanced tree structure with minimal depth, but to reach the best approximation of the criterion expressed by the equation 2.3. The demand of minimizing the number of visible cells indirectly implies, that the cross-sections of the non-occluding parts of a cell boundary have to be minimal. The opposite of such transparent cross-sections are obscure parts of the boundaries, which are represented by occluders, or their parts. This observation implies the primary criterion for creating the spatial subdivision. This criterion is the *maximal obscurity* of the cell boundaries. In other words we want to minimize portal areas on the boundaries of cells created by the spatial subdivision. One good and simple approach chooses the polygon with the *largest cross-section* from the set of polygons enclosed by the cell currently being subdivided. The dividing plane of the cell is a support plane of this polygon. If more polygons meet the criteria of size just with a small deviation, we pick one, which implies a support plane *intersecting the least of other polygons* in the current cell. This is considered to be the secondary criterion of the subdivision.

Note, that the original polygons of the scene may happen to be divided in some of the previous subdivision steps. Therefore, just the "clipped" relevant parts enclosed in the currently subdivided cell, are considered for further computations.

2.2.2 Area of Polygon

The polygon area calculation is one of the dominating tasks of the SSD construction. As it was already stated previously, virtual environments containing a large number of polygons are expected to be subjects to the spatial subdivision, what gives a good reason to include a brief discussion of the polygon area computation.

In vector calculus the area of a polygon is often taught of as a *vector* [6, 7]. Assume bounded polygon positioned in the plane in three-dimensional space with a normal vector $\vec{n}$. The interior of the polygon is denoted S and the boundary is denoted ∂S. We can compute the area-vector using formula:

$$\vec{A} = \frac{1}{2} \oint_{\partial S} \vec{r} \times d\vec{s}, \tag{2.4}$$

where $\vec{r}$ is the position vector, $d\vec{s}$ is the differential arc length vector, and "$\times$" denotes the cross product. The boundary curve ∂S is traversed in counterclockwise order when viewed from the positive half-space,i.e. the half-space into which the normal of the polygon points.

Theorem 1 *Suppose a polygon P with k vertices. Let us denote the vertices of the polygon $v_0, v_1, ..., v_{k-1}$ labeled in counterclockwise sequence defining the boundary of the polygon. The equation 2.4 can be reduced to*

$$A(P) = \frac{1}{2}[(v_{k-1} \times v_0) + \sum_{i=0}^{k-2}(v_i \times v_{i+1})]. \tag{2.5}$$

2.3 Subdivision Algorithm

Based on the subdivision criteria defined in previous sections the outlines of the subdivision algorithm can be described.

1. Compute the root cell as an axial box bounding all polygons defined in the database of the virtual environment. Use the minimum and maximum of coordinate values of all the polygons in each dimension x, y, z to determine extremal vertices of the bounding box. This procedure has time complexity $O(n)$, where n is the number of all polygons in the scene.

2. Generate a list of boundaries for the root cell.

3. Compute the cross-section of all polygons in the current cell. Generate an ordered list of polygons sorted by descending value of the cross-section.

4. Compute the number of intersected polygons by plane coaffine with each of the polygons with the largest cross-section value.

5. Choose the polygon with the largest cross-section and with the smallest number of intersected polygons and divide the cell by the plane coaffine with this polygon. Two new cells are created. According to the half-space defined by the plane we denote; C^+ the cell with a locus in the positive half-space and C^- the cell in the negative half-space.

6. Insert the polygons which are located completely in the positive half-space, negative half-space to the C^+,C^- respectively. Polygons intersected by the division plane have to be split, their fragments are then assigned to the C^+ ,. C^- accordingly and their area is recomputed.

7. Apply the previous step on all the boundaries of the current cell.

8. Based on the edges of the boundaries, which were split in the previous step, create a new boundary and assign it the two new cells.

9. Recursively continue in the cell C^+ until there are no polygons which are classified as occluders in the child cells.

10. Recursively continue in the cell C^- until there are no polygons which are classified as occluders in the child cells.

This algorithm leads in general to the predicted space complexity $O(n^2)$ in number of polygons.

2.3.1 Representation of the *SSD*

In order to preserve the information about the SSD, we define two graph structures describing it.

Definition 6 A *neighbor graph* is a graph structure represented by tuple $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a set of nodes and $\mathcal{E} = \{(\mathcal{N} \times \mathcal{N})\}$ is a set of edges. The set of nodes represents cells of the spatial subdivision. Two nodes are adjacent in the graph (are connected by an edge), if the corresponding cells share a boundary.

Modified version of the neighbor graph is the *adjacency graph*.

Definition 7 An *adjacency graph* is a multi-graph structure represented by triple $(\mathcal{N}, \mathcal{E}, \mathcal{P})$, where $\mathcal{N}$ is a set of nodes, $\mathcal{E} = \{(\mathcal{N} \times \mathcal{N} \times p : p \in \mathcal{P})\}$ is a set of edges and P denotes the set of portals. The nodes $\mathcal{N}$ represent the set of cells created by the spatial subdivision. Two nodes are adjacent in the graph, if the corresponding cells share a boundary with a portal $p \in \mathcal{P}$ (Figure 2.3).

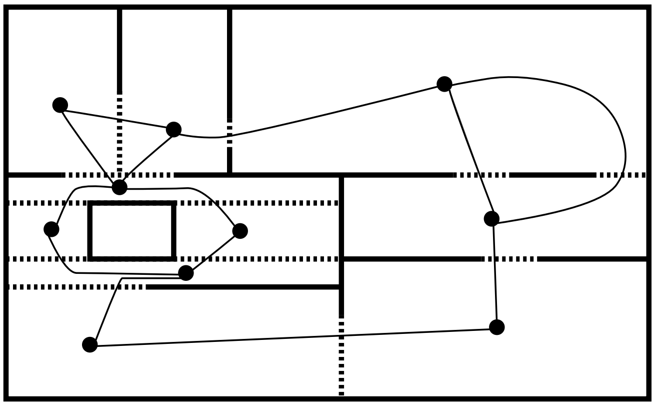

Figure 2.3: An example of an adjacency graph.

Note, that since an edge between two nodes of the graph is present for each portal located on a boundary dividing the corresponding pair of cells, the adjacency graph is generally a multi-graph. In terms of the visibility computation the adjacency graph is a sufficient data structure to represent the relationship of neighborhood between cells. It is sufficient, since we are only interested in the cells which are mutually visible. Notice, that two adjacent cells with a portal on their common boundary are mutually visible and moreover due to the convexity assumption they are “fully” visible with respect to all the possible viewpoints enclosed by these cells.

Exploiting these graph structures the ”logarithmic time” point location feature is lost, since we no longer store the information which was induced in the hierarchy of the BSP tree. However, due to the high coherence of point location queries in practice, the linear time point location query, seldom necessary to perform, does not seem to be a significant drawback.

2.3.2 Post-processing of the SSD

Practical experience with the binary space partitioning and the splitting criteria established in section 2.2.1 shows, that the final the SSD obtained by the BSP algorithm is only suboptimal, as expected anyway. The subdivision of the environment based on the previously described rules often produces cells with some boundaries completely transparent.

An observation was made, that some of these cells can be unified with their neighbors, without violating the restrictions of the conforming spatial subdivision. It is obvious that the unification reduces the number of cells and portals. It will be shown, that it considerably influences the complexity of the visibility computation in positive manner.

Based on this empirical knowledge, we have proposed a randomized algorithm which performs such cell unification of the SSD as a post-processing. This algorithm produces a conforming spatial subdivision with all cells convex and it satisfies all the other requirements to the space partitioning for visibility computation. Moreover, especially for architectural models, the modified partitioning represents more intuitive subdivision of the space, where the cells are more likely to correspond to the rooms of the real environment.

We demonstrate this feature on the following example of floor-plan of a model of building interior. Consider the model with a long straight hallway. It is well seen approximately in the middle of the right image of figure 2.4. This hallway was divided into set of neighboring cells (the left image), during the process of binary space partitioning. At least two boundaries of these cells are completely transparent. Using an iterative unification algorithm described below, we have composed these small cells back into the hallway (right image). In the next chapter it is shown, that such change in the SSD leads to no significant impairment in terms of conservative visibility, although it can significantly reduce the combinatorial explosion of the *visibility tree* size.

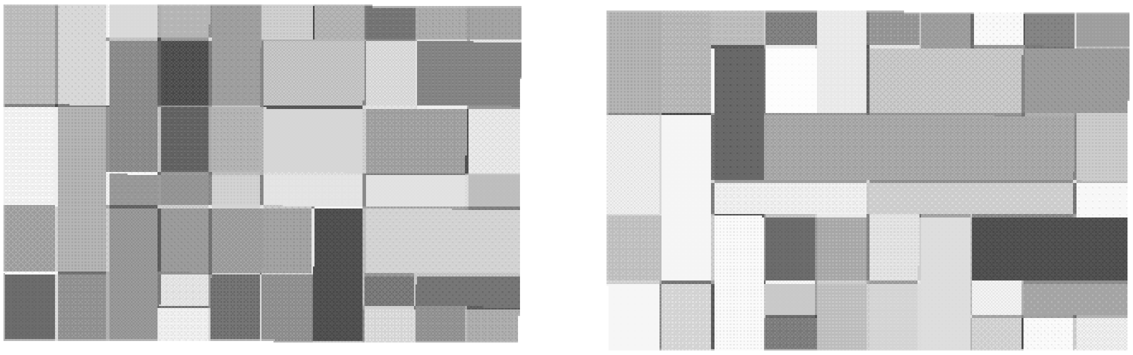

Figure 2.4: The right image depicts a model with a hallway subdivided into several small cells. Left image shows the model, where these cells have been unified back to the hallway.

The unification algorithm is based on the adjacency graph structure (see definition 7). The brief description of the algorithm follows.

1. Pick an arbitrary cell C from the set of all cells of the SSD
2. Denote a completely transparent boundary of C by b. If ∂C does not contain such a boundary continue at 9.
3. From the adjacency graph find a cell Q which is connected to C through this boundary.
4. For each edge e_i of the boundary b (recall that the boundary is a convex polygon), identify coincident boundaries c_i and q_i of the cells C and Q.
5. If the angle β_i between any of the c_i and q_i is bigger than π continue at 9.
6. Unify C and Q, discard the boundary b and if it is possible (they are coplanar) unify boundaries c_i and q_i.
7. Restore and complete all references on the cell C in the adjacency graph.
8. Continue at 2.
9. Choose an arbitrary unprocessed cell C from the set of cells created by the spatial subdivision. If such a cell does not exist end.
10. Continue at 2.

The algorithm as described represents one step of the iteration. We iterate the current solution (SSD) applying this algorithm repeatedly until no unification was performed during the last iteration step. An experience was made, that for real-life scenes this algorithm processes two to five iteration steps in average. Tangibly, in the case of figure 2.4, two steps of the unification were executed.

2.4 Portal Enumeration

A boundary of each cell consists of set of planar convex polygons. It is important to understand, that these polygons do not correspond to the polygons of the scene database, since they describe an abstract spatial structure and do not represent geometrical primitives as models of the real world. We will denote these abstract polygons *boundary polygons*. As we have shown in the previous section, some of the boundaries may even be completely transparent, when no occluders are induced on them. In every step of the spatial subdivision the boundary polygons are divided in the same manner as the occluding polygons referenced in the scene database. Moreover a new boundary polygon is explicitly constructed during each step of SSD construction. This boundary polygon is coaffine to the support plane which was used to divide current cell and it becomes the boundary between the two new cells created at this step.

The portal enumeration is based upon the definition 5. This implies we have to subtract all the occluders coaffine with the particular boundary polygon. Although, there are several relatively simple algorithms to perform such operations on polygons with certain restrictions,

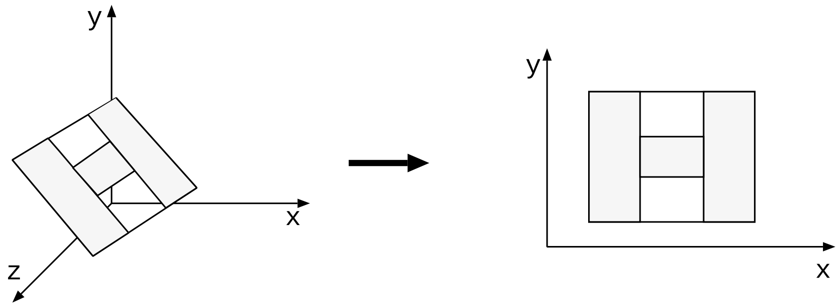

Figure 2.5: Before portal enumeration starts, the boundary polygon and all the embedded occluders are transformed to be parallel with xy-plane and to make one of their edges parallel with the x-axis.

in general these operations are rather complex and require robust algorithms implementing union and difference of a arbitrary set of polygons.

Employing the fact, that all the polygons used for this algorithms are coaffine,the boundary polygon and its occluders can be transformed to the plane, namely xy-plane, with one of their edges parallel to the axis x (Figure 2.5). For the portal enumeration modification of a plane-sweep algorithm, described in [9], was adopted. In the current implementation the occluders are assumed to be rectangles. This assumption significantly simplifies the implementation of necessary polygon operations. The result of the portal enumeration is a list of portals embedded on each boundary polygon comprised solely of convex polygons (rectangles in the current implementation). Obviously, the result meets all the requirements on portals given by the definition 5.

Chapter 3

Visibility Determination

This chapter deals with the visibility computations performed on the conforming spatial subdivision of a model of virtual environment. Given such SSD, several abstract queries can be formulated. Here, the notion of the occluders, detail objects and conforming spatial subdivision are assumed to be defined previously.

3.1 Static Visibility Preprocessing

The process of SSD, creates a *coarse* abstract spatial structure to the virtual environment. Understanding each of the elements (cells) of this structure as one object, we compute a cell-to-cell static visibility information for every cell in the spatial subdivision.

The visibility between cells is based on the observation, that to see one cell from another an unobscured *sightline* (straight line) must exists between them. Due to the convexity of the cells, an observer located in a *source cell*, can see to neighbor cells (defined in the terms of adjacency graph) through portals induced on the source cell boundaries. However, not only cells adjacent in the adjacency graph are mutually visible. Obviously, the visibility is not restricted just to the neighbors of the source cell, but also more distant cells can be potentially visible. Visibility to such further-away-cells is mediated by a *portal sequence*, which admits a sightline.

Definition 8 Define a *portal sequence* as an ordered list of portals, such that each consecutive pair of portals leads into and out of the same cell.

If an observer can see from an interior point of one cell to interior point of some other cell, a sightline through the boundaries of the source and target cells must exists. Since the sightline is not allowed to intersect any occluders in the scene, it has to intersect (*stab*) just portals. Therefore it is sufficient for the cell-to-cell visibility to determine an existence of a *stabbing line* for a particular portal sequence,i.e. existence of a line intersecting all portals in the sequence. Thus, the problem of cell-to-cell visibility reduces to determining an existence of a stabbing line through a portal sequence. We say that the portal sequence admits a sightline if a straight line that stabs every portal (represented by a convex polygon) in the sequence exists. Hence, the set of all visible cells from a source is determined by constrained depth first search in the adjacency graph, i.e. searching through those paths in the graph which admit sightlines.

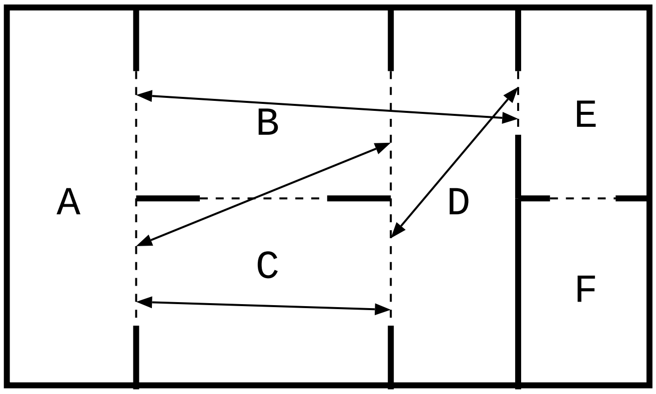

Figure 3.1: An example of simple environment with portal sequences admitting sightline marked.

We demonstrate the principles on a simple example. Let us denote P/Q a portal from cell P to cell Q. Figure 3.1 depicts six cells A, B, C, D, E and F. The portal sequence [A/B, B/D, D/E] admits a sightline. Similarly the portal sequences [A/C, C/B, B/D] and [C/D, D/E] admit sightlines. Thus A, B, C, D and E are mutually visible. On the other hand no portal sequence starting at A and ending at F admits a sightline. That is why cells A and F are not mutually visible.

Evidently the generation of the portal sequences from the adjacency graph structure and queries for an existence of stabbing lines are two crucial operation of the cell-to-cell static visibility computation. Therefore, separate sections are denoted to both of them.

3.1.1 Portal Sequence Generation

To generate portal sequences a depth first search algorithm in the adjacency graph is used. As already mentioned above, this search is constrained by the existence of stabbing line through the generated portal sequence. The algorithm of constrained DFS, to establish cell-to-cell visibility is recursively called procedure, outlined here :

```
Find_Visible_Cells(cell C, portal sequence P, cell_list V)
{
 V = V + C;  // Add the cell C to the set of visible cells
 for (each portal p of C) {
    Q = P + p;     // Add the portal to the sequence
    if (stabbing line through portal sequence Q exists) {
        N = GetNeighbor(p);   // Retrieves the cell portal p leads to
        Find_Visible_Cells(N,Q,V); // Recursive call to this procedure
       }
    }
}
```

3.1.2 Stabbing a Portal Sequence

Recall that the portal is required to be a convex polygon. Thus the query for the existence of a stabbing line through the sequence of portals is equal to determining an existence of stabbing line of the set of convex polygons.

For a given set of polygons, let n be the number of edges of all the polygons comprised in the set. Various stabbing line algorithms for unoriented polygons have been formulated. Avis and Wenger presented an $O(n^4 \lg n)$ time algorithm in [10]. McKenna and O'Rourke improved this to $(O(n^4\alpha(n))$ time [11], where $\alpha(n)$ is the functional inverse of Ackermann's function. If the polygons are triangles with g distinct normals (some of the normals may point in the same direction), an algorithm developed by Pellegrini computes the stabbing line, if one exists in $O(g^2n^2 \lg n)$ time [12]. If every polygon's normal points in different direction (g equals to the number of polygons), the time bound of this approach is the same as the one accomplished by Avis and Wenger. For the case of input polygons consisting of *isothetic* (axial-aligned) rectangles, Hohmeyer and Teller proposed an $O(n \lg n)$ stabbing line algorithm [13]. Amenta improved this with a randomized linear time algorithm [14]. Finally, Teller and Hohmeyer introduced a method, based on the linear programming and five dimensional convex hull algorithm, for solving the problem of stabbing arbitrary, oriented, convex polygons in randomized $O(n^2)$ time [15]. This algorithm expects the input polygons to be convex. Each polygon has a specific *orientation* according to its normal. The normals of the polygons may point in arbitrary direction. The algorithm then searches for a set of *extremal* stabbing lines with direction vectors having a positive inner products with each normal of polygons in the sequence (Figure 3.2).

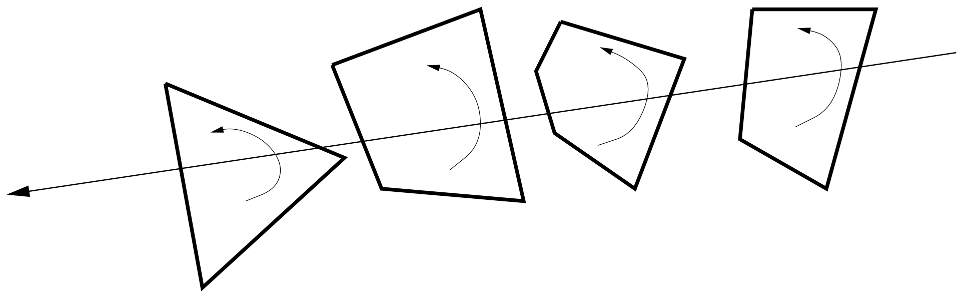

Figure 3.2: A stabbing line for a sequence of oriented convex polygons in $\mathcal{R}^3$.

We have adopted several algorithms to suit the purposes of static and dynamic visibility computations. All the algorithms utilize the analytical geometry of lines in three-dimensional space involving the Plücker coordinate system. By mapping oriented lines in $\mathcal{R}^3$ onto points (hyper-planes) in $\mathcal{R}^5$ the Plücker coordinates provide us with useful and powerful tools for manipulating generally-oriented lines in three dimensional space [12, 38]. In the Plücker coordinates the duality of points and hyperplanes, both images of 3D-lines, can be exploited. Thus, queries of the mutual position of lines reduces to a dot product of their corresponding mappings.

3.1.3 Plücker Coordinates of Lines

An oriented line in three-dimensional space can be expressed by homogeneous coordinates of two its points. Let l be a line in 3-space and let $a = (a_0, a_1, a_2, 1)$ and $b = (b_0, b_1, b_2, 1)$ be two distinct points in homogeneous coordinates lying on l. We can represent the line l oriented from a to b by the following matrix:

$$l = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \end{pmatrix} \tag{3.1}$$

Taking the determinants of the six two-by-two sub-matrices of the above two-by-four matrix, we obtain components of the *Plücker coordinates* of the line:

$$p(l) = (\xi_{01}, \xi_{02}, \xi_{03}, \xi_{12}, \xi_{31}, \xi_{23}), \quad \text{with} \quad \xi_{ij} = det \begin{pmatrix} a_i & a_j \\ b_i & b_j \end{pmatrix}. \tag{3.2}$$

The six numbers ξ_{ij} are interpreted as homogeneous coordinates of a point in five-dimensional space or by duality, as coefficients of a hyperplane in $\mathcal{R}^5$. For a given oriented line l these six numbers are unique and they do not depend on the particular distinct points a and b, that were chosen to describe the line l. We call $p(l)$ the *Plücker point* of l in projective five-dimensional space $\mathcal{R}^5$.

Define the *Plücker hyper-plane* of the line l to be the hyper-plane in $\mathcal{R}^5$ with vector of coefficients $v(l) = (\xi_{23}, \xi_{31}, \xi_{12}, \xi_{03}, \xi_{02}, \xi_{01})$. The Plücker hyper-plane is defined:

$$h(l) = \{p \in \mathcal{R}^5 : v(l) \cdot p = 0\} \tag{3.3}$$

The Plücker hyper-plane induces closed positive and negative half-space given by:

$$\begin{aligned} h^+(l) &= \{p \in P^5 : v(l) \cdot p \geq 0\} \\ h^-(l) &= \{p \in P^5 : v(l) \cdot p \leq 0\} \end{aligned} \tag{3.4}$$

Somewhat asymmetric order in the specification of the Plücker coordinates was adopted in [16] to produce positive signs in some identities about the Plücker coordinates. Let us denote the new permuted coordinates by a six-tuple

$$\Pi(l) = (\pi_{l0}, \pi_{l1}, \pi_{l2}, \pi_{l3}, \pi_{l4}, \pi_{l5}). \tag{3.5}$$

The permutation from the original Plücker coordinates is given by the following rules:

$$\begin{aligned} \xi_{01} &\rightarrow \pi_{l0} \\ \xi_{02} &\rightarrow \pi_{l1} \\ \xi_{03} &\rightarrow \pi_{l2} \\ \xi_{12} &\rightarrow \pi_{l3} \\ \xi_{31} &\rightarrow \pi_{l5} \\ \xi_{23} &\rightarrow \pi_{l4} \end{aligned} \tag{3.6}$$

For an oriented line l defined by a pair of distinct points p, $q \in \mathcal{R}^3$ with homogeneous coordinates $p = (p_x, p_y, p_z, 1)$ and $q = (q_x, q_y, q_z, 1)$, the corresponding projective six-tuple $\Pi(l)$ is enumerated as follows:

$$\begin{aligned} \pi_{l0} &= \xi_{01} = p_x q_y - q_x p_y \\ \pi_{l1} &= \xi_{02} = p_x q_z - q_x p_z \\ \pi_{l2} &= \xi_{03} = p_x - q_x \\ \pi_{l3} &= \xi_{12} = p_y q_z - q_y p_z \\ \pi_{l4} &= \xi_{31} = p_z - q_z \\ \pi_{l5} &= \xi_{23} = p_y - q_y \end{aligned} \quad (3.7)$$

If a and b are two directed lines, and Π_a, Π_b their corresponding Plücker mappings, relation $side(a, b)$ is defined as the permuted inner product $\Pi_a \odot \Pi_b$:

$$\Pi_a \odot \Pi_b = (\pi_{a0}\pi_{b4} + \pi_{a1}\pi_{b5} + \pi_{a2}\pi_{b3} + \pi_{a4}\pi_{b0} + \pi_{a5}\pi_{b1} + \pi_{a3}\pi_{b2}) \quad (3.8)$$

This relation of *sidedness* can be interpreted with the right-hand rule (Figure 3.3). If the thumb of right-hand is directed along line a, then:

- ◇ $side(a, b) > 0$, if line b is oriented in the direction the fingers points,
- ◇ $side(a, b) = 0$, if lines a and b are coplanar (i.e., intersect or are parallel),
- ◇ $side(a, b) < 0$, if line b points against the fingers.

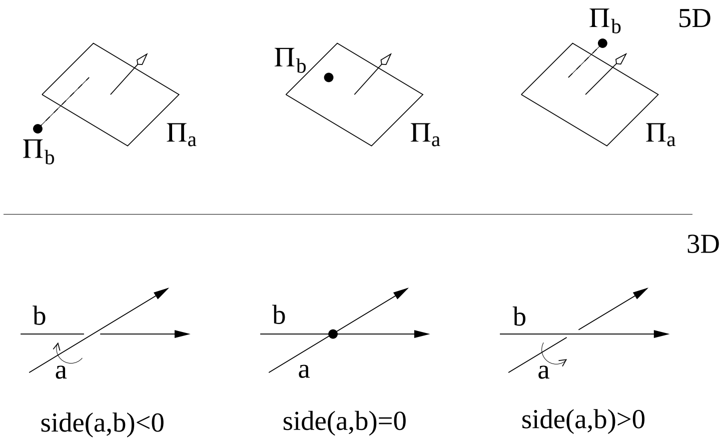

Figure 3.3: The $side(a, b)$, interpreted as the right-hand rule.

The main advantage of transforming lines into Plücker coordinates is that detecting incidence of lines in $\mathcal{R}^3$ is equivalent to computing the inner product of homogeneous point (mapping of one line) with hyper-plane (mapping of the other). This can be expressed by the following lemma.

Lemma 1 *Lines l and l' intersect or are parallel (i.e. meet in infinity) if and only if* $p(l) \in h(l')$

Note that the condition 3.3 states in terms of Plücker coordinates that any line always meets itself.

An important feature of the Plücker coordinates is that although every directed line in $\mathcal{R}^3$ maps to a point in Plücker coordinates, not every tuple of six real numbers corresponds to a line in three-dimensional space. Only the points Π satisfying the condition

$$\Pi \odot \Pi \equiv \xi_{01}\xi_{23} + \xi_{02}\xi_{31} + \xi_{03}\xi_{12} = 0 \tag{3.9}$$

correspond to real lines in $\mathcal{R}^3$. All other points map to lines which are said to be *imaginary*. The set of points in $\mathcal{R}^5$ satisfying the quadratic form 3.9 represents the *Plücker quadric*, also known as Plücker surface,Klain quadric or Grassmanian manifold (Figure 3.4). One can observe, that the six Plücker coordinates of a real line are not independent. First they describe an oriented projective space and second they must satisfy the equation 3.9.

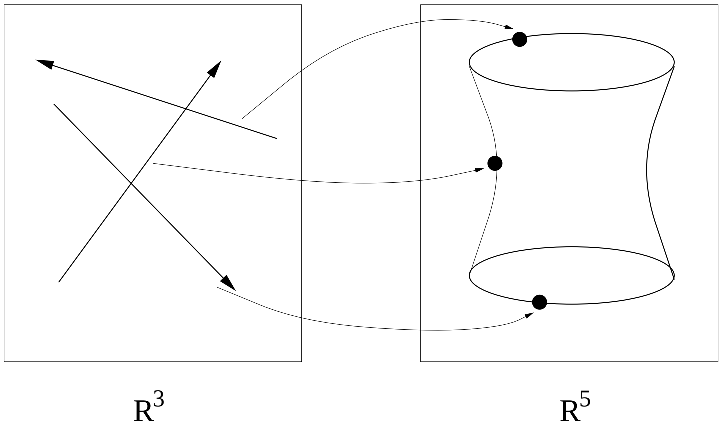

Figure 3.4: Real lines map on points on the Plücker quadric.

Now all the prerequisites to the the problem of computing the stabbing line have been defined, so we can continue with description how to determine the stabbing line existence.

3.1.4 Computing an Existence of a Stabbing Line

The set of oriented polygons used in the query for existence of a stabbing line, with a total of $n > 6$ edges, can be denoted by $E_k,\ 0 \leq k < n$. Each edge E_k is a segment of a directed line e_k. Since the polygons are oriented, the e_k can be directed so that if a stabbing line S through all the polygons exists, it must lie on the same side of all e_k (see figure 3.5). That is S must satisfy

$$side(S, e_k) \geq 0, \quad \text{for} \quad 0 \leq k < n. \tag{3.10}$$

This condition can be rewritten in terms of the Plücker coordinates as $s \odot \pi_k \geq 0$, where $s = \Pi(S)$ and $\pi_k = \Pi(e_k)$. Thus s must lie above all the hyper-planes h_k (see figure 3.6), what means inside or on the boundary of the convex polytope

$$\bigcap_k h_k^+, \quad \text{where} \quad 0 \leq k < n. \tag{3.11}$$

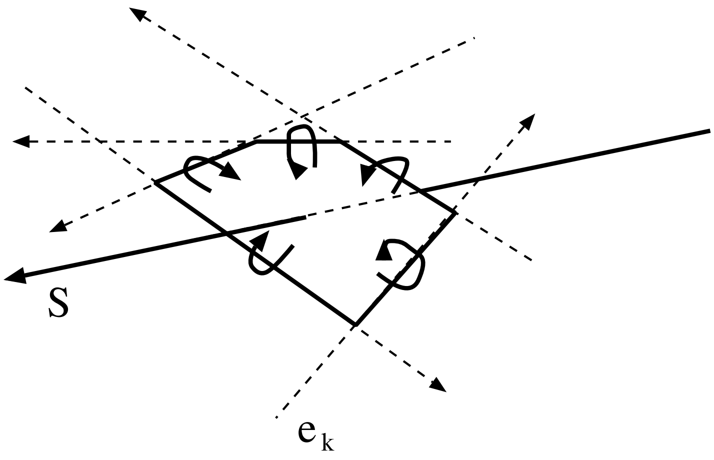

Figure 3.5: The stabbing line has to lie on the same side of all the edges e_k.

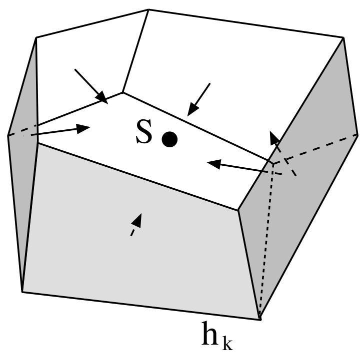

Figure 3.6: If S is a stabbing line, $s = \Pi(S)$ must lie inside h_k^+ for all k.

Points satisfying this condition are called *feasible.*

Recall that not all points of the Plücker coordinates have a real-line dual in three-dimensional space. The points in Plücker space corresponding to a stabbing line through the sequence of polygons must lie inside or on the boundary of the polytope defined by 3.11. Moreover to satisfy the condition 3.9 for mappings of real lines, they must lie on the Plücker surface in the same time. To determine all the real stabbing lines, it is necessary to find intersections of the polytope $\bigcap_k h_k^+$ and the Plücker quadric (Figure 3.7). Note, that when a stabbing exists, there is generally an infinite number of other stabbing lines through the portal sequence, except the degenerate case of just one sightline. To pick up some representative lines the *extremal stabbing lines*, which arise from the intersection of the edges of the polytope $\bigcap_k h_k^+$ and the Plücker surface, can be used.

Since the Plücker space is five-dimensional, the edges of the polytope arise from an intersection of four hyper-planes h_k. The existence of an intersection of any of these edges with the Plücker quadric proves the existence of stabbing line through the sequence of polygons, which

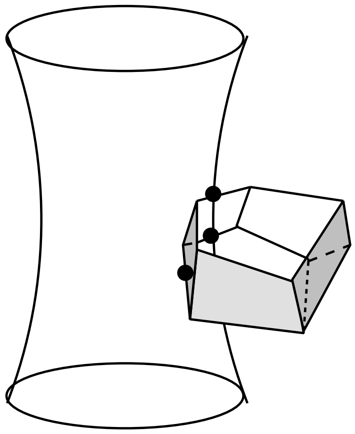

Figure 3.7: The existence of an intersection between an edges of the polytope $\bigcap_k h_k^+$ and the the Plücker quadric implies an existence of a real stabbing line.

have entered the algorithm. As we will see in next sections, these intersections correspond to the extreme stabbing lines, which can be used to describe the *antipenumbra* caused by the portal sequence.

The condition $side(S, e_k) \geq 0$ can be formulated in terms of linear programming. Computing a point inside the polytope $\bigcap_k h_k^+$ can performed by maximizing the function $z(s, c) = c$ with the restrictions $side(s, h_i) - c > 0$. If the set of equations can be solved with $c > 0$, then a point located inside of the polytope exists.

However, this formulation of the six-dimensional linear programming problem is not in the *complete normal form*, where all the possible results are assumed to have positive values. The vector of variables consist of six coefficients of the Plücker coordinates and c denotes the "cost". To solve this problem it is necessary to transform the problem into the complete normal form. It is done by expressing each of the variables as a difference of two positive numbers. This way we obtain the twelve-dimensional linear programming problem in complete normal form [17, 20, 4]. For more information about this transformation and other possible ways to solve the linear programming problem see chapter 5.

3.1.5 Intersection of a Line in 5D and the Plücker Surface.

Before we attempt to compute the intersection of a 5D-line and the Plücker quadric, an expression of the line in the space of Plücker coordinates is defined :

$$\begin{aligned} E &= (e_0, e_1, e_2, e_3, e_4, e_5) = H + Gt, \qquad t \in R, \\ \text{where :} & \\ H &= (h_0, h_1, h_2, h_3, h_4, h_5), \\ G &= (g_0, g_1, g_2, g_3, g_4, g_5). \end{aligned} \tag{3.12}$$

Any point lying on the Plücker surface must satisfy the equation 3.9. From there and the line equation 3.12 we get :

$$E \odot E = H \odot H + 2tH \odot G + t^2 G = 0 \tag{3.13}$$

This simple quadratic equation can be solved easily to get the unknown t. Substituting the result to equation 3.12, we get the desired intersection(s).

3.1.6 Reverse Mapping of Plücker Coordinates.

The mapping of points satisfying the condition 3.9 back to the 3D-space can be performed, what means that parameters of a real line can be computed from the corresponding Plücker mapping. When the result of the reverse mapping to lines in 3D is being described by a point and directional vector, some more information to map the line back from the Plücker six-tuple must be specified.

The directional vector u of the line is straightforward from the π_{l2}, π_{l5} and π_{l4}) components. As the "origin" point p of the line, we use the point, which is an intersection the line and the additionally specified 3D planar plane. Since we transform the first portal in the sequence to point into particular direction (y-axis), we use the $y = d$ plane to compute this intersection. Adding the plane equation to the set of equations used for mapping to Plücker coordinates we get following results:

$$\begin{aligned} p_x &= \tfrac{\pi_{l0} - \pi_{l2} d}{\pi_{l5}} \\ p_y &= d \\ p_z &= -\tfrac{\pi_{l3} + \pi_{l4} d}{\pi_{l5}} \\ u_x &= -\pi_{l2} \\ u_y &= -\pi_{l5} \\ u_z &= -\pi_{l4} \end{aligned} \tag{3.14}$$

In other words the additional plane equation used is the one defining the plane of the first portal in the transformed sequence (see section 3.2.2). It means, that all the "origin" points of the computed stabbing lines lie on the support plane of the first portal.

3.1.7 The Visibility Tree

Previous sections provide us a powerful tool for the cell-to-cell static visibility computation. This section introduces an efficient data structure for storing the computed results.

The recursive nature of the portal sequence generation suggests to store the computed data in a tree-structure – the *visibility tree*. The visibility tree is a general n-tree, where nodes of the tree represent cells visible from the source cell, which is represented by the root of the tree. Edges of the tree correspond to portals stabbed as a part of portal sequences originating on the boundary of the source cell. For each portal sequence there is a different path in the tree. The cell-to-cell static visibility for any SSD of the virtual environment is completely recoverable from the set of visibility trees. One such tree is assigned to each cell created by the spatial subdivision. For better understanding of the structure of the visibility tree see the figure 3.8

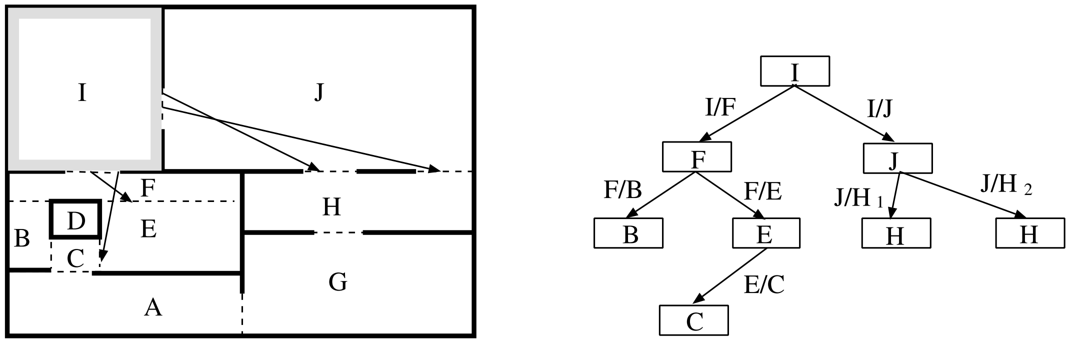

Figure 3.8: An example of a visibility tree in a simple environment.

Since the tree describes portal sequences possible to see through, any of the cells in the tree can appear even several times. Obviously, any of such appearances occurs on a different path in the tree. We have to consider this fact, when on-line searching the tree. For example, when we use the tree for the visibility query during the rendering process it is necessary to create an additional data structure, such as a hashing-table, to ensure that we do not render the same cell more times.

3.2 Description of Visible Regions

The visibility tree as described above, is a data structure considering only the cell-to-cell visibility relationship. The set of possibly visible cells are all the distinct cells appearing as nodes in the visibility tree, which is assigned to the source cell. We can further improve the visibility preprocessing, adding the information about visible frustrums induced by the portal sequences. We call such visibility relation *cell-to-frustrum* static visibility. So far, we used the stabbing line algorithm to answer the query if the portal sequence admits a sightline. When a sightline exists, the portal sequence induces the *visible region* on one side of the sequence and the *region of interest* on the other side (see figure 3.9). We assign the information about the visible region to edges of the visibility tree.

Here we briefly say, that not each edge in the tree has to contain this information. Some heuristics can be applied to decide if including the frustrum information leads to significant improvement, in the terms of number of visible elements or the size of potentially visible region. A simple heuristics can classify the visible region caused by the portal sequence as *assetive*, when it bounds a volume, which is smaller than for example 60% of the volume bounded by the cell induced by the last portal in the sequence.

To determine the visible regions, we use similar approach as to find the existence of a stabbing line. The mapping to the Plücker coordinates is involved again, what simplifies further computations. We search for extreme stabbing lines [15], to get the description of the visible region. Further calculations are performed then, to conservatively approximate the visible region.

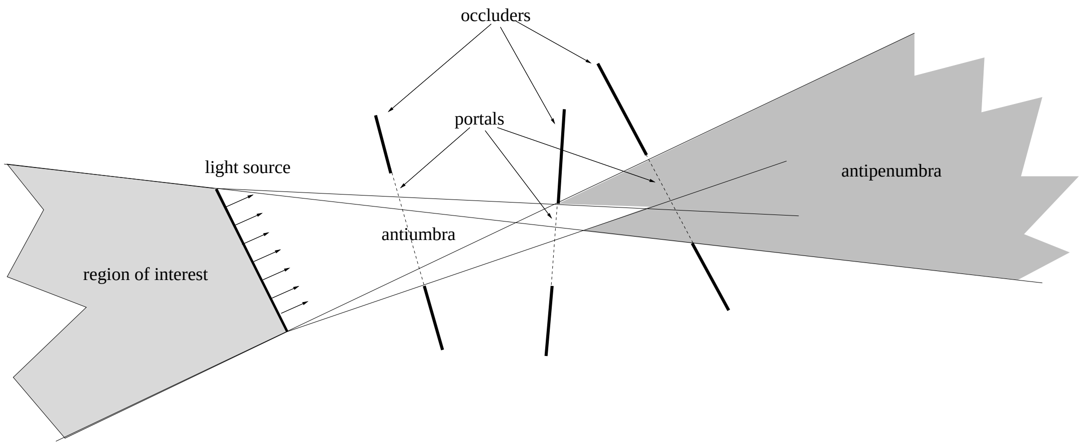

Figure 3.9: Visibility propagation through portals.

3.2.1 Extreme Stabbing Lines

Consider the first portal in the portal sequence an *areal light source*. We wish to describe the set of light rays originating at the light source passing through each of the portals in the sequence. The algorithm exploits the fact that the portals are *oriented* by the sense in which they are traversed during the constrained search in the adjacency graph. The method of generating the portal sequences was already described in section 3.1.1. For a particular portal sequence, each portal can be considered to admit light in only in one direction. When generating the sequence the normals of the portals are transformed so the side which admits light is induced by the positive half-space of the support plane of the transformed portal.

The face structure of the polytope $\bigcap_k h_k^+$, mentioned in previous section, has worst-case complexity quadratic in the number of half-spaces defining it [40], and it can be computed by a randomized algorithm in $O(n^2)$ time [41]. Define the *extremal* stabbing lines as those lines incident on four generator edges [16]. In five dimensional space four lines (i.e. constraints) are necessary to determine a line. Obviously, if some stabbing lines exists, then at least one must be extremal. The structure of the polytope 3.11 yields all extremal stabbing lines [16, 15]. Each such line l is incident, in 3D, upon four of the generator edges e_k. Consequently, the Plücker point Π_l must lie on four of the hyper-planes h_k in 5D, and must therefore lie on a *1D-face*, or edge, of the polytope $\bigcap_k h_k^+$. Thus, we can find all extremal stabbing lines of a given polygon sequence by examining the edges of the polytope for intersections with the Plücker quadric. The extremal stabbing line corresponding to each intersection can be determined from the four relevant generator edges E_k as shown in [42]. If the polytope 3.11 is expressed appropriately it is possible to exploit determination of the stabbing line based just on the edges of the polytope. Figure 3.10 shows the output of the algorithm, which computes all extremal stabbing lines of a sequence of oriented polygons.

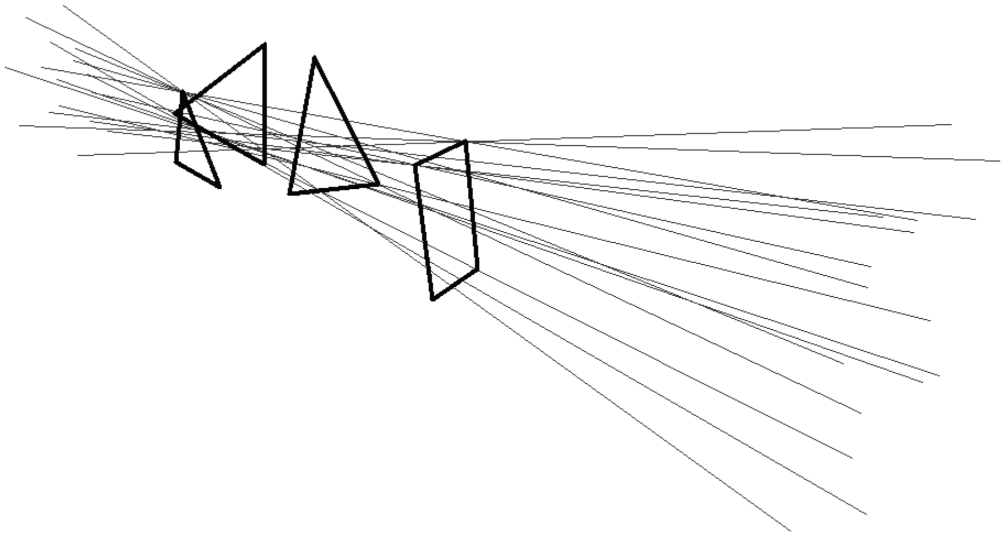

Figure 3.10: Extreme stabbing for a sequence of four polygons.

3.2.2 Transformation of Portals

As mentioned in previous sections we map lines in 3D space to the Plücker points, which are the six-tuples. Since, the Plücker space is five dimensional, we treat these six-tuples as homogeneous points or hyper-planes. Solving the problem in Plücker space, results are constrained to have positive "homogeneous" component, or zero when in infinity.

Therefore we must to know, that any of possible solutions has non-negative homogeneous component. Because we take π_{l5} as the homogeneous, its value must be ensured to be non-negative for any stabbing line. Based on equations 3.7, for any line l mapped to Plücker coordinates the π_{l5} is actually a difference in y-coordinate of 3D space.

Therefore we must transform all the portals in the sequence, so that any possible stabbing line has a non-negative y-value of its directional vector. The first portal in the sequence defines a half-space, which induces directions of stabbing lines. It means, that all the stabbing lines, which stab this portal must have, say positive, dot products with the normals of all the portals in the sequence. So, for the first portal P we compute a transformation, to turn its normal to point along y-axis and this transformation is used for any other portal in the sequences originating at P.

3.2.3 Portal Clipping

In the above mentioned algorithm for computing stabbing lines, problems might arise when the portal sequence is not *spatially monotone*. This means that the support plane of some portal

p_a intersects some other portal p_b. Thus, the portal p_b lies on both of the half-spaces defined by the support plane of p_a. This situation is illustrated on figure 3.11.

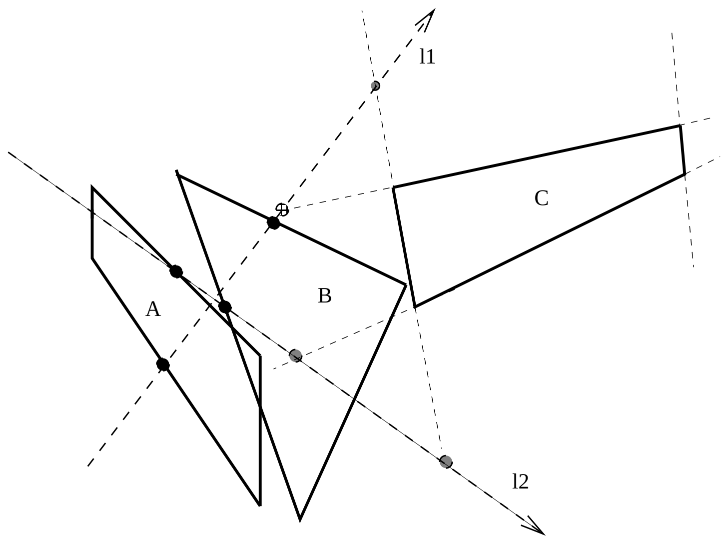

Figure 3.11: Example of a "wrong" portal sequence. Two extremal stabbing lines are shown, which do not intersect each portal in the sequence.

Applying the equation 3.10, the algorithm can output some extremal stabbing lines, which actually don't intersect some of the portals at all. The reason of such misclassification is that the "wrong" stabbing line is embedded in the support plane(s) of some portal(s). When so, it satisfies the side relation by zero, regardless if it really stabs the portal or not.

Our solution to this problem is the *clipping* of the portal sequences, in the time of their generation, to produce spatially monotone ones. Figure 3.12 illustrates the method of portal clipping. We use some additional δ to ensure, that the side relation is no longer satisfied by zero for all the possible extremal stabbing lines laying in the support plane of some of the portals. In fact it is not necessary to perform the portal clipping in 3D space. Exploiting the advantages of Plücker coordinates, we simply add the Plücker mappings of the "clipping edges" to the constrains e_k, which are mappings of generator edges.

3.2.4 Extremal Swaths

In this section some theory about the boundary of antipenumbra caused by a portal sequence is given. This is important to understand for expressing or approximating the antipenumbra, based on extremal stabbing lines.

There are three types of extremal stabbing lines: vertex-vertex (VV) lines, vertex-edge-edge (VEE) lines, or quadruple edge (4E) lines. Imaging "sliding" an extremal stabbing line (of any type) away from its initial position, by relaxing exactly one of the four edge constraints determining the line. The surface, or *swath*, swept out by the sliding line must either be planar (if it remains tight on a vertex) or a regulus, whose three generator lines embed three

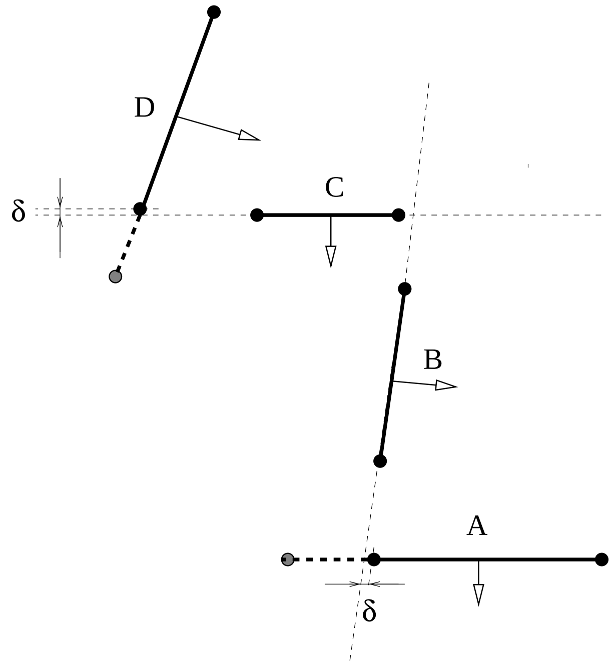

Figure 3.12: 2D projection of portal clipping example.

polygon edges. In the terminology of [29], swaths are VE or EEE *event surfaces* important in the construction of aspect graphs, since they define places at which qualitative changes in occlusion occur.

Figure 3.13-i shows an extremal VV stabbing line tight on four edges A,B,C and D. Relaxing constraint C yields a VE (planar) swath tight on A,B and D. Eventually, the sliding line encounters an obstacle (in this case edge E), and terminates at a VV line tight on A,B,D and E. Figure 3.13-ii depicts an extremal 4E stabbing line tight on the mutually skew edges A,B,C and D. Relaxing constraint A produces a EEE (regulus) swath tight on B,C and D. The sliding line eventually encounters edge E and induces an extremal VEE line, terminating the swath.

Consider the same situations in the 5D space generated by the Plücker mapping (see figure 3.14). Extremal swaths map to particular points in the Plücker space, namely, the intersections of the edges of the polytope $\bigcap_k h_k^+$ with the Plücker quadric. Since swaths are one-parameter line structures, they correspond to curves in Plücker coordinates. These curves are the *traces* or intersections of the 2D-faces of $\bigcap_k h_k^+$ with the Plücker surface and therefore conics (in 5D, a polytope's 2D-faces are planar, and determined by the intersection of some three h_k). The 3D swaths corresponding to these 5D conic traces are called *extremal swaths.* All swaths have three generator lines and consequently three generator edges, arising from the input polygons (portals).

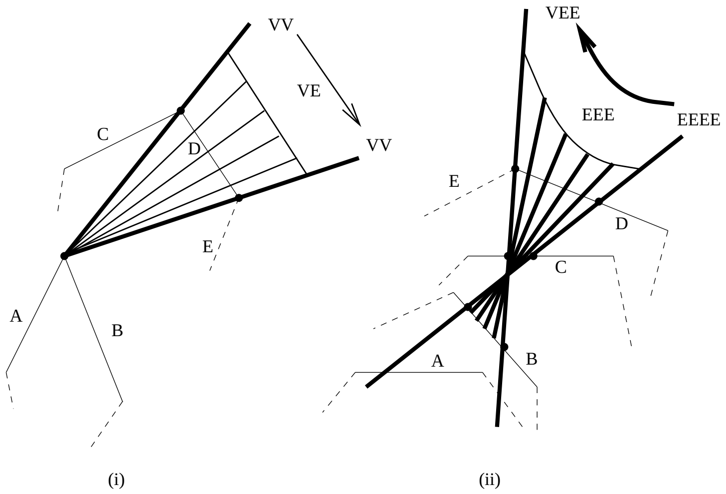

Figure 3.13: Extremal swaths - VE plane (i) and EEE quadratic surface (ii).

3.2.5 Internal and Penumbral Swaths

An object can be extremal (that is, lie on the boundary of the convex hull) in 5D, yet lie wholly inside the antipenumbra of the light source in three-space. Just as there are extremal stabbing lines that lie in the interior of the antipenumbra, so there are extremal swaths in the interior as well. Define a *penumbral* swath as an extremal swath that lies on the boundary of the antipenumbra. All other extremal swaths are *internal*. Examining the 2D-faces of the polytope $\bigcap_k h_k^+$ yields all extremal swaths. However we have to distinguish between penumbral and internal swaths. This distinction can be made locally; that is , by examining the swath's three generator edges and, in turn, their generator polygons. In [21] it is shown how to classify the swath as penumbral or in other case as internal.

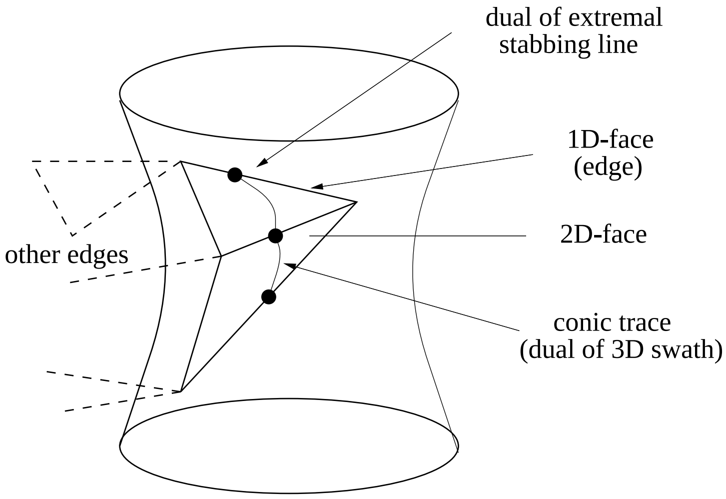

Figure 3.14: Duals of 3D swaths as a traces on the Plücker surface in 5D.

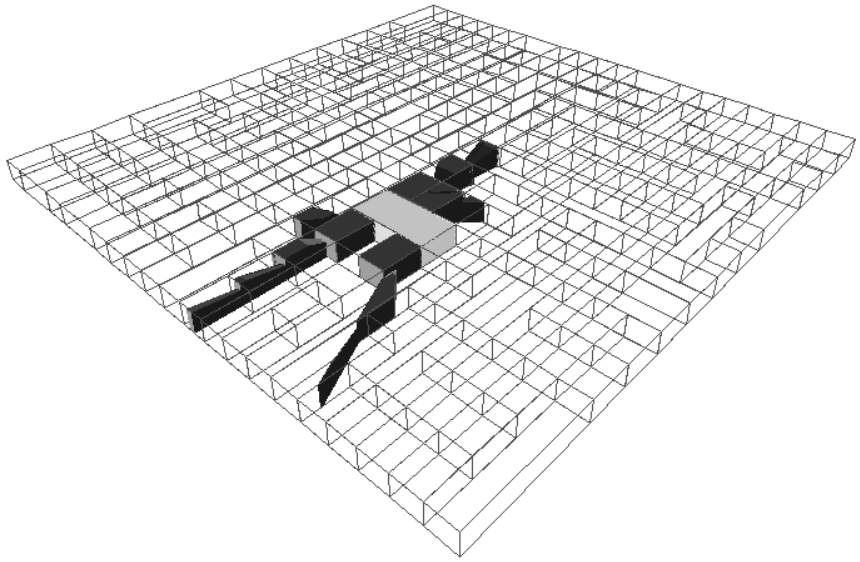

Figure 3.15: Cell-to-frustrum visibility. Potentially visible regions shown in black

3.2.6 Conservative Estimations of the Visible Regions

We proposed and implemented an algorithm to conservatively estimate the antipenumbra, or the visible region, in contrast to the exact computation based on classifying the internal and penumbral swaths in the 5D Plücker space.

We project all the extremal stabbing lines to the "far plane" in 3D, to get a set of two dimensional points. We compute the convex hull of these points and classify the extremal stabbing lines corresponding to the points lying on the 2D convex hull, as *penumbral stabbing lines*. The two adjacent penumbral stabbing lines, can define a planar plane or a quadratic surface in 3D. Since we compute the conservative estimation of the visible frustrum, we want to express it as a set of bounding planar planes (faces). If two adjacent penumbral lines define a plane, we compute its four coefficients and report the plane as a boundary of the visible region. In the other case we approximate the quadratic surface swath conservatively, by two planar planes. The "far plane" used to project the extremal stabbing lines to 2D is the plane laying behind the last portal in the sequence, with its normal equal to the normal of the first portal (light source).

Another method how to conservatively express the visible region is described in [38]. It is based on sort of gift-wrapping algorithm, known from the 3D convex hull computation.

3.3 Dynamic Visibility Queries

The static visibility determination generally overestimates the visible portion of the model to a particular observer. This section details *dynamic visibility* queries that exploit both the precomputed static visibility information and the instantaneous knowledge of the observer's view variables, to get more discriminating visibility information. We characterize the dynamic visibility as the process of real time determination of visible entities, which is *dependent on location and on viewing direction* of the observer. The visibility tree referenced in each cell defines the set of cells potentially visible from the source cell. The set of distinct cells of any visibility tree defines a superset of the visible portion of the model, which can be further processed to establish a tighter approximation of the currently visible entities. It is obvious, that only objects enclosed in the cells referenced in the visibility tree are considered for further visibility processing.

The dynamic visibility algorithms go further in utilizing the visibility tree in compare to enumeration of all the referenced cells, which could be done in preprocessing. Two ways how to reduce the currently visible set of entities will be presented, using the information about the current position of the observer in the cell, the viewing direction and the viewing angle.

3.3.1 Observer View Variables

In three dimensions, the observer's viewing frustrum can be entirely defined by a view direction, azimuthal and altitudal opening half-angles, and an up-vector that specifies the remaining "twist" degree of freedom. However, mainly the frustrum is represented as four implicit plane equations whose positive half-spaces intersect in the viewing volume. Near and far planes bounding the frustrum may also be specified, what restricts the observer's potentially visible region to a closed volume.

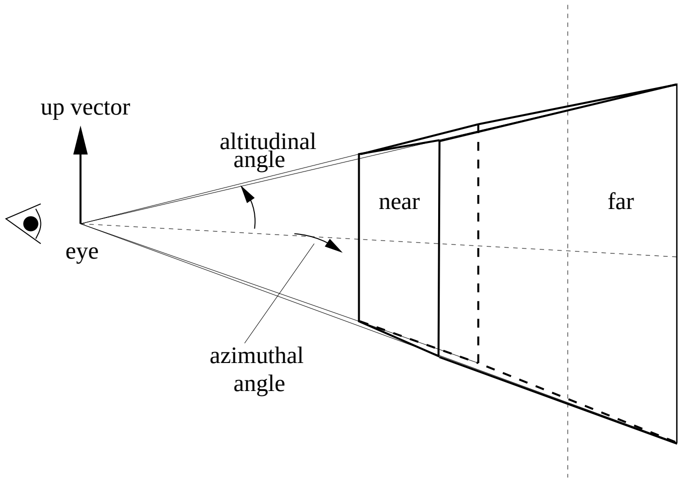

Figure 3.16: Observer view variables in 3D.

3.3.2 Eye-to-Cell and Eye-to-Region Visibility

In this section we explain the way, how to conservatively estimate the eye-to-cell and eye-to-frustrum visibility, using a simple search in the visibility tree.

To establish the eye-to-cell visibility, an algorithm based on portal culling is being used. We perform the depth first search on the visibility tree and at each encountered portal we check its incidence with the observers viewing frustrum. The frustrum is expected to be defined by four plane equations. We compute a dot product of the first portal vertex with each of the equations. If the vertex is infeasible with respect to the frustrum we continue, considering the next vertex of the portal. If no feasible vertex is found, the portal lies out of the viewing frustrum and the current branch DFS is terminated. This approach has complexity $O(n)$ on a portal sequence with n edges. Another way how to cull away the portals, is to use the precomputed center and radius of the portal, thus approximating the portal with a sphere. For each portal encountered by the DFS, the minimal dot product with the four frustrum equations must be greater than the negative of the portal extent to classify the current sequence as *passable*. This lowers the time complexity of the algorithm to $O(p)$ on a sequence of p portals.

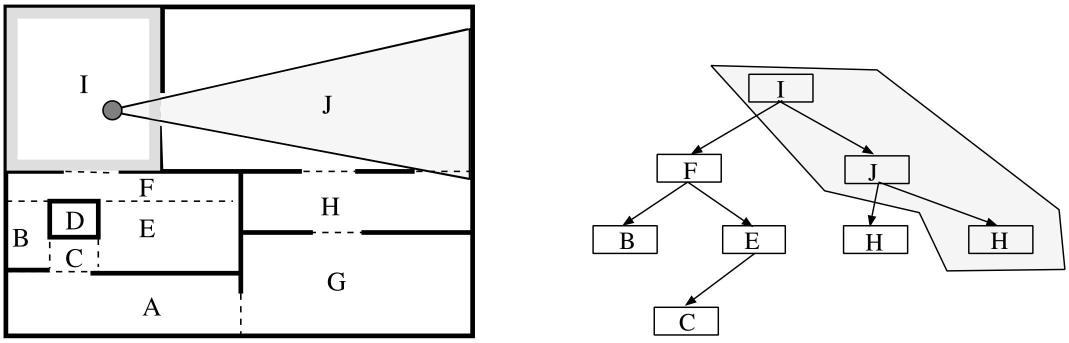

Figure 3.17: The pruning of the visibility tree during the eye-to-cell visibility query.

The *eye-to-region*, or the *eye-to-frustrum* visibility is basically very similar to the eye-to-cell visibility, but it provides us more precise results. The difference is that each edge of the visibility tree is assumed to contain the information about the area-of-interest and the visible frustrum induced by the portal sequence and this information is used in the on-line query. Thus the DFS on the visibility tree is constrained not just by the observer's view frustrum, but also the equations defining the area-of-interest with respect to the particular portal sequence are concerned. If the observer is located outside this area, the portal sequence is impassable and the active branch of the DFS is terminates. Otherwise, all the objects enclosed in the potentially visible frustrum of the current portal sequence, are classified as visible and send to the rendering subsystem.

Both of the described methods of dynamic visibility are conservative, what means that they can misclasify a cell as visible, although it is currently invisible to the observer. This arises from the fact, that these algorithms are made as simple as possible to lower their time complexity, thus they do not consider the "on-line" occlusions, but just the observer's view frustrum.

Chapter 4

Convex Polyhedra Enumeration

In the previous chapter we saw, that the enumeration of convex polyhedra in 5D, is the crucial operation of extreme stabbing lines computation and thus describing the potentially visible regions. This chapter focuses on the arbitrary dimension convex polyhedra enumeration based on the reverse search vertex enumeration algorithm of Avis and Fukuda. This implementation was extended with the additional preprocessing and post-processing of the input data to meet requirements of variety of problems. The attention was paid especially to the enumeration of convex polyhedra leading to a full description of vertices, edges, rays and faces and their adjacency. Regarding to the visibility computations a special instance of the algorithm was made to perform all the necessary calculations in the Plücker Space.

4.1 Background

An enumeration method called *Reverse Search* was introduced by Avis and Fukuda in [23] and applied to the problem of finding all vertices of a convex polyhedron [24]. Their implementation uses multiple precision rational arithmetic and handles problems of quite large size, especially for the simple polyhedra.

However, to satisfy requirements of the Global Visibility project, some additional features of the implementation were necessary. First of all the algorithm has to be able to determine not just vertices of the convex polyhedra, but also the other topological features such as edges and rays and their adjacency. It is preferred to use real arithmetic, since it leads to significant improvement in speed of the implementation, although the exact solution is no longer insured. In contrast to implementation of Avis and Fukuda this algorithm uses some additional storage of intermediate results, what leads to another speed improvement and in the case of real arithmetic products lower error.

4.2 Introduction

For the formal description of the reverse search algorithm reader is referred to [23]. Briefly and informally, the algorithm works as follows. Suppose we have a system of linear inequalities defining a polyhedron in $\mathcal{R}^d$ and vertex of that polyhedron. A vertex is specified by giving the indices of d inequalities whose bounding hyper-planes intersect at the vertex. For any given linear objective function, the *simplex method* [17] generates a path along edges of the polyhedron until vertex maximizing this objective function is found. For simplicity, let us assume for the moment that the optimum vertex is contained on exactly d bounding hyper-planes. The path is found by pivoting, which involves interchanging one of the equations defining the vertex with one not currently used. The path chosen from an initial given vertex depends on the pivot rule used. In fact, care must be taken because some pivot rules generate cycles and do not lead to the optimum vertex. However, a particularly simple rule, known as *Bland's rule* or the least subscript rule [25], guarantees a unique path from any starting vertex to the optimum vertex. If we look at the set of all such paths from all vertices of the polyhedron, we get a spanning tree of the edge graph of the polyhedron rooted at the optimum vertex. The reverse search algorithm simply starts at an "optimum vertex" and traces out the tree in depth first order by "reversing" the Bland's rule.

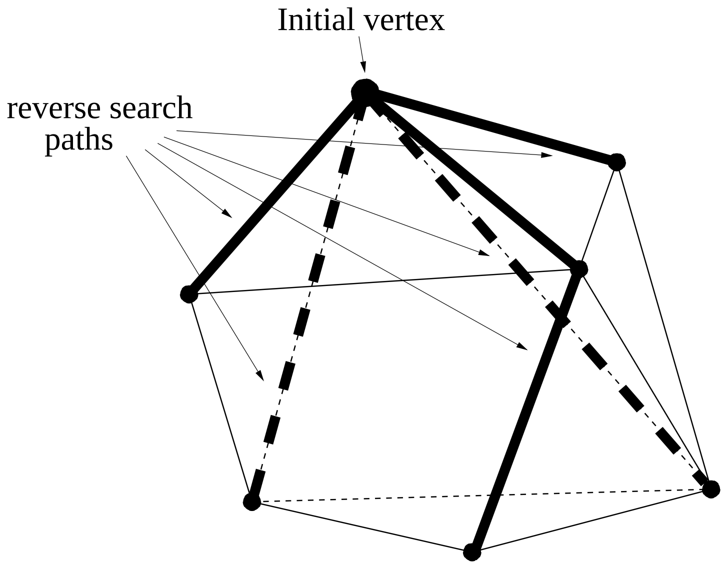

Figure 4.1: The spanning tree of reverse search, when enumerating vertices of polyhedra.

Going down the tree all valid "reverse" pivots are explored in lexicographical order from any given intermediate node, while the state (matrix) of the "parent" node is stored on the state stack. Going back up the tree, the state is restored from the stack along with the current pivot indices. From there it is simple to continue by considering the next lexicographic "reverse" pivot. One possible difficulty arises at so called degenerate vertices, vertices which lie on more than d bounding hyper-planes (d is the dimension of the space). By using duality, the algorithm can be used for convex hull enumeration of set of points in $\mathcal{R}^d$. It can also be used for enumerating all the vertices of the Voronoi diagram of a set of points in $\mathcal{R}^d$, since this can be done by solving a convex hull problem in $\mathcal{R}^{d+1}$ [26].

4.3 Dictionaries

Let A be a $m \times n$ matrix, with columns indexed by the set $E = \{0, 1, 2, \ldots, n-1\}$. Fix distinct indices f and g of E. Consider system of equations:

$$A \cdot x = 0, x_g = 1. \tag{4.1}$$

For any $J \subseteq E, x_j$ denotes the sub-vector of x indexed by J, and A_j denotes the sub-matrix of A consisting of columns indexed by J. A *basis* B for 4.1 is a subset of E of cardinality m containing f but not g, for which A_b is nonsingular. We will only be concerned with systems 4.1 that have at least one basis, and will assume this for the rest of the paper[1]. Given any basis B, we can transform 4.1 into the *dictionary*:

$$x_B = -A_B^{-1} A_N \cdot x_N = \bar{A} \cdot x_N, \tag{4.2}$$

where $N = E - B$ is the $co - basis$, and $\bar{A}$ denotes $-A_B^{-1}A_N$. $\bar{A}$ is called the *coefficient matrix* of the dictionary, with rows indexed by B and columns indexed by N, so that $\bar{A} = \{\bar{a}_{ij} : i \in B, j \in N\}$. Note that the co-basis always contains the index g.

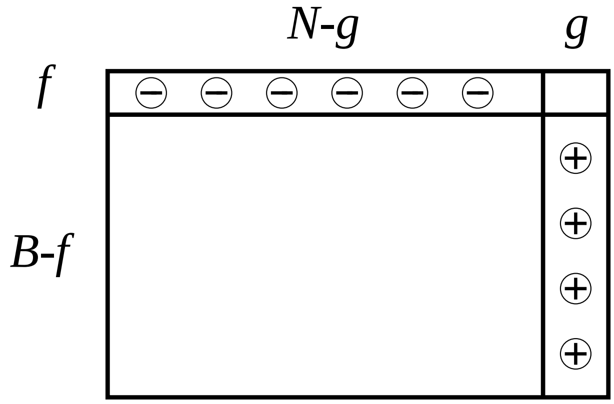

Figure 4.2: An optimal dictionary.

A variable x_i is primal feasible if $i \in B - f$ and $\bar{a}_{ig} \geq 0$. A variable x_j is dual feasible if $j \in N - g$ and $\bar{a}_{fj} \leq 0$. A dictionary is *primal feasible* if x_i is primal feasible for all $i \in B - f$ and *dual feasible* for all $j \in N - g$. A dictionary is *optimal* if it is primal and dual feasible 4.2. A *basic solution* to 4.2 is obtained from a dictionary by setting $x_{N-g} = 0, x_g = 1$. If any basic variable has a value zero, we call the basic solution and the corresponding dictionary *degenerate*.

Degenerate optimal dictionaries pose a problem for a reverse search, since there may be many optimal dictionaries, and each has to be enumerated. This problem can be solved by lexicographic pivot selection rule.

[1]The existence of a basis can be found by linear programming in $O(n)$

4.4 Vertex Enumeration for Polyhedra

A (convex) polyhedron P is the solution set to a system of n_0 inequalities in d non-negative variables:

$$P = \{x \in \mathcal{R}^d | A' \cdot y \leq b\}, \tag{4.3}$$

where A' is an $n_0 \times d$ matrix and b is a n_0-vector. A *vertex* of the polyhedron is a vector $y \in P$ that satisfies a linearly independent set of d of the inequalities as equations. The *vertex enumeration problem* for P is to enumerate all of its vertices.

Let $m = d+1, n = n_0 + d + 2, f = 0, g = n-1, B = \{0, n-m, n-m+1, \ldots, n-2\}$ and $N = \{1, \ldots, n-m-1, n-1\}$. Form the dictionary $\bar{A}$ from A' and b as follows:

1. all signs of A' are reversed
2. vector b is appended as an additional column
3. a row zero is appended with all entries -1

Once a solution is found, the program performs the reverse search on the dictionary $x_B = \bar{A} \cdot x_N$.

4.5 The convex hull of a set of points

The problem of enumerating all of the facets of the convex hull of a set of points can be easily transformed to a vertex enumeration problem for a polyhedron. Let $P^* = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in $\mathcal{R}^d$. We assume, by translating the points if necessary, that the origin lies in the interior of the convex hull of P^*. An inequality $ay \leq 1$ defines a facet of the convex hull of P^* if $ap_i \leq 1, i = 1 \ldots n$, and equality is obtained for an affinely independent subset of at least d of the points of P^*.

Form the $n \times d$ matrix $\bar{A}$ by letting row i be p_i. Then standard arguments show that the vertices of

$$\bar{P} = \{x \in \mathcal{R}^d | \bar{A}x \leq 1\}, \tag{4.4}$$

correspond to the facets of the convex hull of P^*. In particular, if $\bar{x}$ is a vertex of $\bar{P}$, then $\bar{x}y \leq 1$ defines a facet of the convex hull of P^*.

4.6 Pivoting

A pivot (r, s) on a basis B, and corresponding dictionary $x_B = \bar{A} \cdot x_N$, is an interchange of some $r \in B - f$ with some index $s \in N - g$ giving a new basis B'. The new coefficient matrix $A' = (a'_{ij})$ is given by

$$a'_{sr} = \frac{1}{\bar{a}_{rs}}, \; a'_{ir} = \frac{\bar{a}_{is}}{\bar{a}_{rs}}, \; a'_{sj} = -\frac{\bar{a}_{rj}}{\bar{a}_{rs}}, \; a'_{ij} = \bar{a}_{ij} - \frac{\bar{a}_{is}\bar{a}_{rj}}{\bar{a}_{rs}}, \; (i \in B - r, j \in N - s). \tag{4.5}$$

A pivot is *primal feasible* if the new basic solution obtained from 4.5 is non-negative. Note that the pivot maintains the property that $f \in B$ and $g \in N$. A basic solution to 4.5 is degenerate, if $x_i = 0$ for some $i \in B - f$. Let B be a basis such that the dictionary 4.2 is primal feasible. We call a pivot that is chosen by the following rule a *Bland* pivot.

4.6.1 Bland's Rule

1. Let s be the smallest index such that x_s is dual infeasible, that is, $\bar{a}_{fs} > 0$.
2. Set $\lambda = min\{-\frac{\bar{a}_{ig}}{\bar{a}_{is}} : i \in B - f, \bar{a}_{is} < 0\}$. Let r be the smallest index obtaining this minimum.

The pivot (r, s) maintains the primal feasibility of the dictionary. If step (1) does not apply, the dictionary is also dual feasible and hence optimal.

4.6.2 Reverse Pivoting

Let $\bar{A}$ and A' be two dictionaries with the same dimensions. Suppose application of some pivot rule X to A' yields pivot (r, s) which takes A' to $\bar{A}$. Then we say that (r, s) is a *reverse pivot* for $\bar{A}$ with respect to pivot rule X. In this section necessary and sufficient conditions for a reverse pivot with respect to modified Bland's rule are shown.

Lemma 2 *(r, s) is a reverse pivot with respect to Bland's rule for $\bar{A}$ if and only if the following five conditions hold:*

(i) $\bar{a}_{fs} < 0$.

(ii) $\bar{a}_{rs} < 0$.

(iii) $-\frac{\bar{a}_{rg}}{\bar{a}_{rs}} = min\ \{-\frac{\bar{a}_{ig}}{\bar{a}_{is}} : i \in B - f, \bar{a}_{is} < 0\}$.

(iv) $\bar{a}_{ig} \neq 0$ *or* $\bar{a}_{is} \leq 0$ *for* $i \in B - f - r, i < s$.

(v) $\bar{a}_{fj} \leq \frac{\bar{a}_{fs}\bar{a}_{rj}}{\bar{a}_{rs}}$ *for* $j \in N - g - s, j < r$.

Proof can be found in [24].

4.7 Algorithm Overview

4.7.1 Initial Vertex Search

To perform reverse search as described in [24] it is necessary to find some vertex of the convex polyhedra formed by input inequalities. This can be done by solving the linear programming problem. In some cases no feasible vertex exists and the polyhedra is empty. We can easily recognize such situation from the result of the linear programming.

Briefly and informally the search for a vertex starts with pivoting all the cobasis variables N to the basis G. In the case that such pivoting cannot be performed, the inequality system is linearly independent and the Gaussian elimination can be done, to find maximum affine space. In all other cases we obtain a vertex V as intersection of planes of N. The first d indices of G now correspond to variables of d dimensional space. However vertex V is not necessarily feasible vertex (the dictionary is not primal feasible), thus further computations must be performed. We augment the dictionary by k artificial variables [27], where k is the number of "infeasible rows" of the dictionary.

The modified Bland's rule for pivot selection considers just last $m-d$ indices of B as possible candidates for pivot step, and values of column f of these rows of the dictionary are no longer considered to hold primal feasibility. In other words, negative values of the column f in the first d rows of the dictionary are "allowed" to get round the non-negative constraints of the space.

To avoid loops in the reverse search lexicographical pivot selection is used, hence it is very important to properly update indices while searching for the starting vertex and discarding the artificial variables.

4.7.2 Reverse Search

In reverse search all the valid "reverse" pivots are explored in lexicographical order from any given intermediate node, while the state (dictionary) of the "parent" node is stored on the special stack. At each step we find a vertex and both topology (incident planes) and geometric informations are stored in a vertex list. We also store any rays, which are examined by searching all the dictionary columns for zero entries. Both vertex- and ray- lists are used later for searching for edges and faces of the polyhedra.

Going back up the tree, the state is restored from the stack along with the current pivot indices. Although it is possible to simply perform pivoting to go up the tree, the dictionary-storing was used mainly because of the reduction of the cumulative error. From there it is simple to continue by considering the next lexicographic "reverse" pivot.

4.8 Extended Topology Output

The special attention in our implementation was paid to determine all the edges of the polyhedra. Edges or lines in d dimensional space arise from intersection of $d-1$ hyper-planes. In the polyhedra description edge is a connection of two vertices of the polyhedra or a ray incident with some vertex.

A special case of absence of any vertex occurs for the non-empty polyhedra. Then all the rays can be found by transforming the problem to a lower dimensional one. The maximal affine space is found by Gaussian elimination, corresponding dictionary is transformed appropriately and $d-1$ dimensional reverse search is executed.

Performing the search through the lists of vertices, we can easily enumerate all the line-segment-edges, since there is an edge E between the two vertices U,V, when some set S of $d-1$

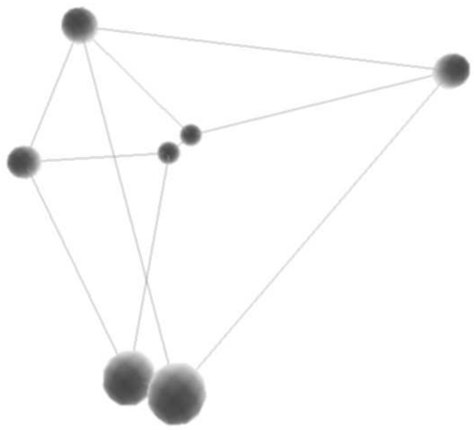

Figure 4.3: Results of convex polyhedra enumeration in $\mathcal{R}^3$. Five constraints (plane equations) were used to bound the polyhedra.

planes incident with U incidents also with vertex V. Than the edge E is the line segment on the intersection of planes S. Similarly searching through the ray list, we try to find a couple of vertex V and ray R, incident on the same set of $d-1$ planes.

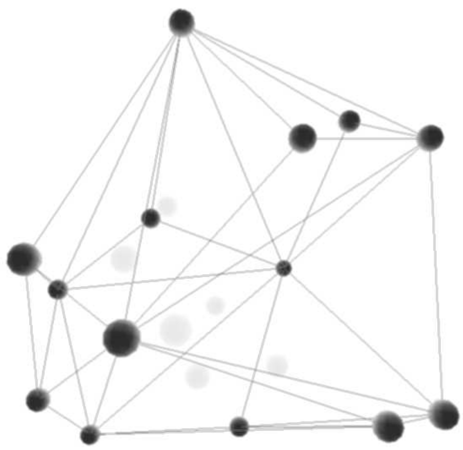

Figure 4.4: Convex hull of a set of 25 points in $\mathcal{R}^3$. Black points bound the convex hull, the grey ones lie inside the convex set.

Chapter 5

Implementation Issues

In the previous chapters the theory and background related to this project were described. Now, the actual implementation of the algorithms mentioned there will be discussed, together with the current limitations and differences to what was described before.

5.1 Programming Methodology

In any larger project it is very important to perform an analysis of the problems, which are going to be solved, to produce efficient solutions of transformation of the input data. It is preferable to implement as many parts of the algorithms as possible, reusing the already implemented ones and thus minimize the overhead in terms of the size of the necessary code and work.

An *object programming* brings powerful tools and methods, to use an elegant and efficient way to meet the above mentioned criteria. However, it is not just the use of object programming, what simplifies the actual implementation of larger project. It is important to use some kind of *standard labeling* and name conventions for the programming language, especially when more people is going to participate on the project.

The decision was made to use the C++ programming language, because of several reasons. First, it is supported on most of the computer platforms and systems, and it is generally platform independent. Second, it contains the object programming and template structures. Third important reason the use C++ was, that it became very popular (probably because of the its two previously mentioned features) and nowadays treated as a “standard” programming language among very large part of computer community.

5.1.1 Implementation of Elementary Data Structures

Most of the implemented algorithms, very often process just a few types of data structures (DS). An elegant way to implement such DS is using the *template objects*, which are a powerful tool of C++ programming language. An example of such data structure is a list of elements. Using advantages of template classes, we can implement all necessary methods to manipulate the list and its elements, without knowledge of what type of data the list is going to store.

Similarly, we can treat another data types as an array or a stack. An example of code with a definition of a variable of type list follows.

```
ListC<double>   floatList;
ListC<Vector3C> vectorList;

vectorList.add(Vector3C(0,0,1));
vectorList.add(Vector3C(0,1,0));
vectorList.add(Vector3C(4,5,7));

ListIteratorC<Vector3C> vector(vectorList);
for(;vector();++vector) intList.add(vector()->size());

floatList.add(3.14159);
ListIteratorC<float>  floatI(floatList);
for(;floatI();++floatI) cout<<*floatI();
```

The simple peace of code shows the basic operations defined on variables of type list and the way how to iterate the list and how to access its elements.

5.1.2 Representation of Geometrical Primitives

A very important task is a proper, robust and flexible implementation of geometrical primitives such as point(vector), face or polygon. Implementing this data structures wrongly (e.g. in terms of flexibility) can lead to significant troubles in later extensions of the project.

An object analysis has been done as a first step of the implementation. The class hierarchy, inheritance and containment was derived from the analysis and previous experience in the field of implementation of visualization systems.

5.1.3 Platform Independence

Since the software being implemented should serve as a basis for further research, the goal was to make it as much independent on the target platform as possible. Since not all the systems, support each of the features utilized in the project, several "layers" of independence exist. Basically, the main part of the code should be completely platform independent, using no special libraries and communicating just through standard input and output. Additional options can be specified in the time of compilation, to enable platform specific features, such as Open-GL browser or X-Windows on-line graphical debugging. Mostly, an "automatic" check to determine the target platform and its features should work, requiring no change in the source code.

We have tested the compilation of the program and the program it self on several platforms, with a different operating systems, compilers and processor types. The table 5.1 gives an overview of resources used in various cases.

System	Linux	Irix	SunOS	DOS
Compiler	GNU g++	CC	GNU g++	Watcom-C
VRML-debugging	yes	yes	yes	yes
On-line debugging	X-Window	X-Window	X-Window	graphlib
Browsing	Mesa-GL	Open-GL	—	—
Visual Visibility	—	Inventor	—	—

Table 5.1: Overview of supported features on different platforms.

5.1.4 Graphical Debugging using VRML

During the development of many of algorithms of computational geometry, it is hard to follow all the consequences and to explore all possible errors in the output, studying just the numerical expression of data structures. To simplify the debugging process, several methods to visualize geometrical data structures were used. These methods, were applied to output temporary results as well as the final solutions of the algorithms, such as some of the pictures in this thesis.

As a first step implementation of visualization of geometrical structures was made, using some graphical libraries of functions, supported by some of the C++ compilers. This method of visualizing of the current state of computation can be made interactive, so an user can for example stop further processing when necessary or use another debugging tool in the same time. However, this approach has a significant drawback in the fact, that it is generally hard to make such interactive visualization completely platform independent. Another disadvantage of this method is that to implement the convenient user interface to allow to explore each detail in presented data structures can be quite complex task. Our solution to the problem, is usage of the *Virtual Reality Modeling Language* (VRML), namely VRML 1.0, to output the geometry information of the data structures to files. These structures can be visualized by passing the corresponding files to some interactive program with a support of visualization of VRML. Such programs are widely available for variety of platforms, what makes this approach of graphical debugging and results visualization independent of the particular computer operating system and architecture. We used VRWeb program for Linux, Netscape Navigator with the Live-3D plug-in in Windows-95 and WebSpace on Silicon Graphics machines.

When exploiting VRML as a tool to visualize the state of computation, we can utilize the advantages of VRML format to describe even complex geometric elements in a simple way. Further the use of transparent, emissive or reflective materials can considerably simplify understanding of visualized structures.

5.2 Spatial Subdivision

The spatial subdivision is a first step taking place, before visibility computations can start. It is important that a robust algorithm is used to build the SSD. The spatial subdivision, must be conforming, that is it must have all the properties mentioned in chapter 2.

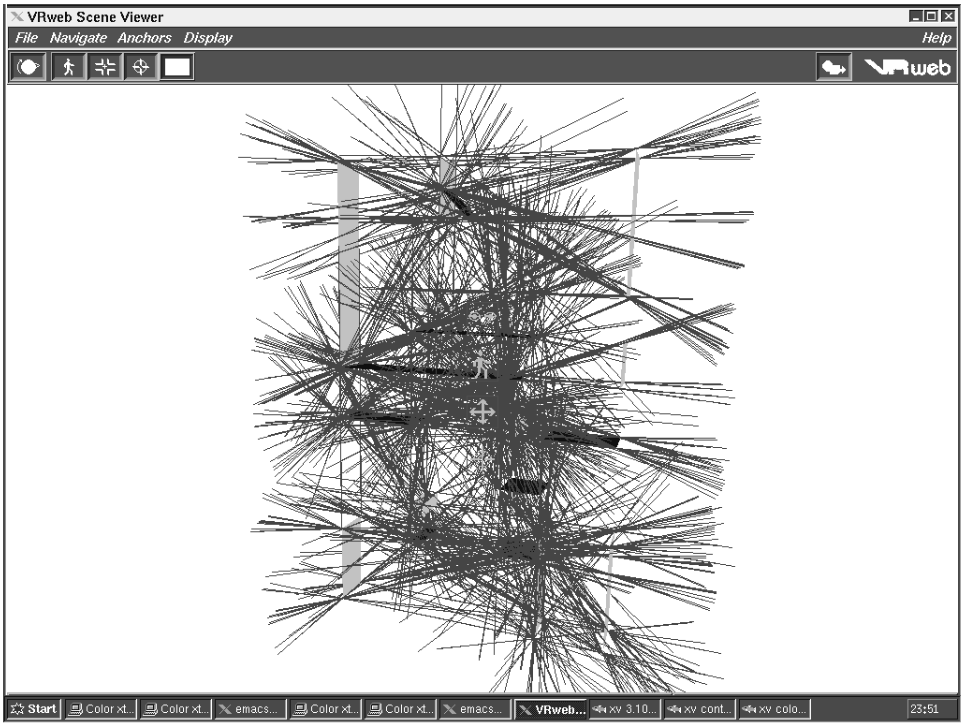

Figure 5.1: An example of the visualization of data structures using VRML.

Some of the "real life" scenes, are corrupt, hence some non-manifolds, degenerated polygons, or unwanted gaps can be present. To handle such cases, a preprocessing of input geometrical elements is performed. In the current implementation, we check just for the correctness and convexity of polygons. Many of CAD programs generate meshes of triangles even in situations that they form a simple convex polygon. The useful step regarding to the further computations would be preprocessing of the input data to form as large simple convex polygons as possible, subdividing the non-convex ones.

During the subdivision process, we have to ensure the correct properties of the polygons which are created as a result of splitting of some other ones. Moreover, since we explicitly create boundaries (virtual polygons), we have to check the correctness of the cells of the spatial subdivision. The cells must be closed, that is completely bounded by the corresponding boundaries.

An algorithm to check and possibly clean-up a polygon data structure was developed. This algorithm is invoked each time when a new polygon is created. It inspects the convexity of the polygon, removes all degenerate vertices, and checks if all the vertices of the polygon are embedded to the same support plane. The properties of cell data structures are checked by an algorithm based on winged edge data structures. Basically, this algorithm was made just to find mistakes in the process of spatial subdivision, thus it does not attempt to repair a possibly wrong cell.

During the subdivision we have to distinguish between any *detail objects* and *occluders.*

When this information is not specified previously known, some heuristics must be applied to classify input polygons as parts of detail objects, and therefore non-occluding, or occluders, such as walls. In the current implementation, we establish a threshold in size of polygon, classifying all the larger polygons as occluders. The threshold calculation is based on the volume of the box bounding the model of virtual environment.

5.3 Visibility Computations

Having a conforming spatial subdivision of an environment, we attempt to establish restrictions in visibility, computing the static visibility relations. As described in chapter 3 we perform constrained traversal of the adjacency graph, testing the existence of a stabbing line. If such line exists we compute the visible frustrum induced by the portal sequence.

5.3.1 Linear Programming

Linear programming, namely the *simplex method*, was used to solve the existence query in Plücker coordinates. The following three different variants of the simplex method were implemented.

1. simplex method with a special pivot rule in $\mathcal{R}^6$
2. simplex method with the variable extension and the ordinary pivot rule in $\mathcal{R}^{12}$
3. dual of the simplex method of (2)

The "ordinary" simplex method assumes the input in normal form and moreover the result it produces must lie in positive half-spaces defined by each of the axis of d-dimensional space. However, in Plücker coordinates there are no restrictions in a sign of the coordinates of the mapping of a line. Therefore we implemented methods to overcome this problem. First, a special pivot rule was created to allow the selected variables to become negative during pivoting. Generalization of this rule to the dual simplex method seamed to be difficult, so change in the dimension of the problem was made (see 3.1.4). Than a dual simplex method could be applied, what gave the best results in terms of time spend by processing the linear program.

5.3.2 Determination of Visible Frustrums

In the case, that we want to compute the antipenumbra (potentially visible region) defined by a sequence of portals, some more calculations must be performed than just checking of the existence of a sightline. However, the linear programming, can still be used first and just in the case that we proved the existence of a stabbing line through a portal sequence, we continue with further processing.

In the current implementation, we generate and clip the portal sequences as described in chapter 3. We apply the Plücker mapping of lines of the portals and invoke the convex polyhedra enumeration algorithm described in the section 5.4. We compute the intersection of polyhedra

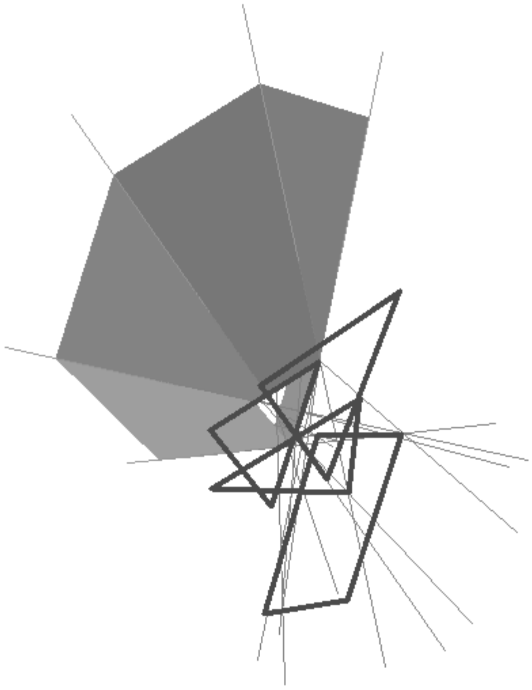

Figure 5.2: The visible frustrum (antipenumbra) through the sequence of four portals.

edges with the Plücker quadric and apply inverse mapping to get extreme stabbing lines. The lines are projected to points in $\mathcal{R}^2$ as described in section 3.2.6 and the $O(n \log n)$ *divide and conquer* convex hull algorithm is invoked. We must apply filtering of the input and output data to ensure the stability of the algorithms. Lines (or points after the projection), with a distance smaller than some threshold are replaced by just one representative line (point). Similarly several coaffine adjacent planes, bounding the antipenumbra approximation are replaced by just one representative plane.

To visualize the frustrums, the enumeration of convex polyhedra in 3D is used, with the set of half-spaces, which is the unification of boundaries of the appropriate cell and the planes bounding the frustrum of visibility in that cell. The output of the algorithm are convex polygons, which are saved to the VRML file.

5.4 Polyhedra Enumeration

The algorithm of convex polyhedra enumeration in arbitrary dimension was implemented as a part of the visibility computations project as well as a stand-alone program. In the later mentioned case, it can be used also to evaluate convex hull of a set of points in arbitrary dimension. The output of the results in 3D form is possible using the VRML file format. The program is written in C++, using advantages of object programming methods.

One generic class called `HullC` was created and all the special cases of input structure, output information, as well as simple modifications to the task can be easily implemented as a class inherited from the generic one. Two such examples are `VRMLHullC` and `CHullC` classes. The `CHullC` class implements the dual transformation of the convex hull problem to the polyhedra enumeration task. The input is a set S of points in d-dimensional space and the output are the points of the convex hull of S. For 3D tasks the output is stored in the VRML file, where also the point adjacency (edges) are described. The `VRMLHullC` is just an extension to `HullC` class to produce the 3D description of the output.

For all computations floating point arithmetic was used, especially because of high speed demands. However sometimes, because of cumulative error, the exact solution is no longer guaranteed. Since we define a zero threshold of 10^{-6}, the normalization of the input for some tasks is preferable.

5.5 The Browser

An important part of the project is an *interactive program* utilizing the precomputed information, that is the spatial subdivision and the visibility information. Such a program should be able to take the preprocessed model of a virtual environment as an input. Further, it should allow an user, or better an *avatar*, to move through the environment, getting a proper feedback to his commands. We call this program a *browser*.

The attention was paid to use as much of the information, created by the preprocessor as possible and to compute on-line visibility queries as described in chapter 3. The browser should be easily extensible in terms of better visualizing of the environment and realism of user interaction. Another important criterion is, that it should be portable among different platforms. The way how to deal with such requirements is to create a generic and platform independent kind of browser, which will support queries defined on the spatial subdivision and visibility structures. Using the object programming such a browser can form a *generic class* which is further extended focusing on different platforms or purposes of usage.

5.5.1 Generic Browser Class

The generic browser class was made to process the queries regarding to the visibility computations, so practically no special handling of the visibility preprocessing is necessary to create extensions of the program. Since it is platform independent, no graphical output is considered in methods of this class. The only possible way of output is a text output on the computer-terminal, however it calls "virtual" functions which can be easily redefined later. Similarly no handling of input devices is implemented here. The generic browser implements the dynamic visibility queries such as eye-to-cell or eye-to-frustrum visibility. It supports the point location query, which is the current cell containing the observer.

5.5.2 Hit Detection

The hit detection is a very important feature of the browser, when the user is moving through the virtual environment. Although the visibility queries do not help us in terms of hit detection, we can still utilize the spatial subdivision, namely the adjacency graph. On the other hand, it is important to know the location of the observer, especially the *current cell* where the observer occurs to be able to process the dynamic visibility queries.

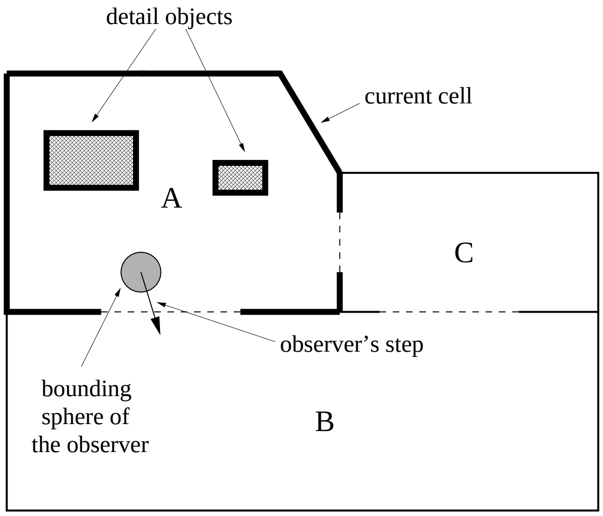

Figure 5.3: A step of the observer invoking the boundary crossing event.

As mentioned in chapter 2, the location of the avatar is mostly highly coherent to one of previous location query. Thus, we can detect just qualitative changes in the location of the observers. In terms of the spatial subdivision, such change means, change of the current cell. This occurs crossing some of the boundaries of the current cell. From the adjacency graph we can easily recover the information where the boundary leads to, hence a new cell to become the current one. Moreover, we can restrict avatar not to be able to "go through" solid elements of the environment. Therefore we allow it to cross the boundary, just through its transparent portions – portals. To ensure such property of the avatar in relationship to detail objects, we have to check the avatar's position regarding to all the objects contained in the current cell.

It is clear, that till the observer does not cross a boundary, the hit detection can be restricted to the current cell and its interior. We can further speedup the hit detection procedure by some additional data structure, such as BSP tree, build of boundaries of the current cell and bounding boxes of the populated objects. In the current implementation we bound the avatar by a sphere, although generally a function can be used to express the shape of the avatar's body.

5.5.3 Visibility Preprocessing Aided walk-through

The browser uses the preprocessed information, which is stored in the file together with the scene description. When observer changes the position, it performs dynamic visibility queries based on the visibility tree. It is preferable that the type of dynamic visibility determination can be changed by the user. For testing purposes different types of dynamic visibility processing are performed, to compare the their efficiency and time complexity.

Geometrical primitives (polygons, objects) classified as visible by the browser are then send

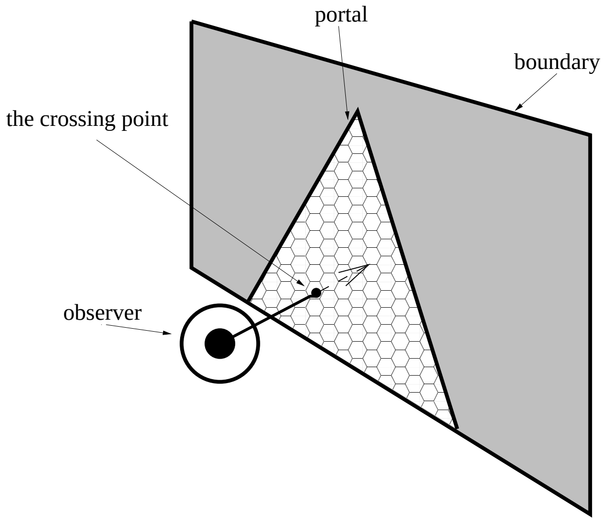

Figure 5.4: Testing the position of the observer against a portal.

to the rendering subsystem. Since we use the Open-GL , we assume a low level visibility to be solved at the render. The main advantage of usage of the Open-GL is that it is supported of variety of platforms. Furthermore, if some hardware accelerators are present it employs them in a way, which is transparent the rest of the software.

5.5.4 On-line Debugging and Testing

It is very important to have a possibility to perform testing and possibly some debugging of the software during the "walk-through" process. Since the software should remain portable, it is difficult to use some kind of window-based graphical user interface. Instead of this just a keyboard input was used to invoke different tests or debugging procedures. We implemented the *keystroke-queue*, to store several last keystrokes. A simple search in this queue is used to find pre-specified keywords. When such a keyword is found in the queue, corresponding command is invoked and the queue is flushed. This simple handling of the keyboard input allows us to use many commands while browsing the environment. Such commands were used to start the automatic walk-through, to show the estimated the speedups of the visibility processing, to visualize the regions of interest directly in the environment, to output the currently visible portions of the model to the VRML file or to show the current information about visible regions.

Chapter 6

Testing and Results

In this chapter we present an empirical evidence of the asset of visibility computations to applications of virtual reality. We mention the time spend by visibility preprocessing for several different environments and the demands for the storage of visibility information. We also show the theoretical bounds of time and storage complexities of algorithms used in visibility computations.

6.1 Algorithmic Complexity

In previous chapters some time and storage complexities of the described algorithms were presented. These complexities were expressed in terms of elements (occluders, portals) they are applied to. However, the question remains how many of such elements is passed to these algorithms for a particular model of virtual environment. The most important features of the environment are number of occluders and portals. The are portals basically induced by the occluders and their topology(density). The occluders define the "shape" of the spatial subdivision, thus the number of cells in the adjacency graph. The portals and their spatial relationship determine the maximal length of portal sequences admitting a sightline, hence the size of visibility tree assigned to each cell.

For architectural models visibility computations were applied to, the length of portal sequences appeared to be a small constant between 5 and 10 in average. This means that the

Algorithm	Elementar Entity	Time Complexity	Space Complexity
Spatial subdivision	occluder	$O(n^3)$	$O(n^2)$
Cell unification	cell	$O(n)$	$O(n)$
Portal stabbing	portal edge	$O(n^2)$	$O(n^2)$
Antipenumbra computation	portal edge	$O(n^2)$	$O(n^2)$

Table 6.1: Summarization of the time and storage complexities of the crucial algorithms of visibility computations.

storage of visibility information in such cases is practically linear in relation to number of cells in the SSD. However, this "feature" of visibility strongly depends on the type of the environment. Clearly, for very sparsely occluded environments the search for portal sequences in the adjacency graph is not constrained properly, what leads to combinatorial explosion. Then the stabbing line algorithm as well as the antipenumbra computation are applied to a very large number of portals and their results does not lead to any speedup of later visualization, since most of the model is classified as visible anyway. This observations define the *spectrum of applicability* of this approach of visibility preprocessing, that are the densely occluded architectural environments. The time and storage complexities of the crucial algorithms are summarized in table 6.1.

6.2 Testing Methodology

To guarantee more or less objective results of testing, several criteria have to be met. We should use more of representative examples of tested media, in our case models virtual environment, to give an overview about sensitiveness of the algorithm to the type of the environment. Second, we should execute one test several times running no other tasks in the same time. The first tests were made on the Silicon Graphics machine, which was shared by many different users. This fact could lead to significant mistakes in measured running times. This was the reason, why all the other tests were performed on a computer, which was not connected to any network.

6.3 Results of the Spatial Subdivision

As it already mentioned in previous chapters, preprocessing of the visibility information is the most important part of the project. In this section we document the times of preprocessing and their dependence on the size and type of the preprocessed environment. We show the necessary requirements to store the visibility information, using the VRML(see [19]) based file format described in [18].

For purposes of testing and debugging, one scene was created manually by hand. The create other scenes, which were used in the tests, an automatic generation was involved. Different parameters were specified to the scene generation program, to create virtual environments with different density of occlusion.

Table 6.2 shows results of the measurement. We can see, that running time and the size of the output depend not just on the size of the environment. It grows very fast with a lower density of occluders (scene named `6supr`). In such cases the size of visibility trees grows, since there is many possible combinations of different portal sequences, which admit a sightline. An illustration of the problem of "empty" spaces in sparsely occluded environments is shown on figure 6.1.

One possible solution to the problem of "empty" space, is to combine the cells forming such region to *meta-cells*, generally non-convex ones, and to use a special way of handling the visibility determination in such regions.

As we have mentioned in the section 2.4 algorithm for portal enumeration is restricted only

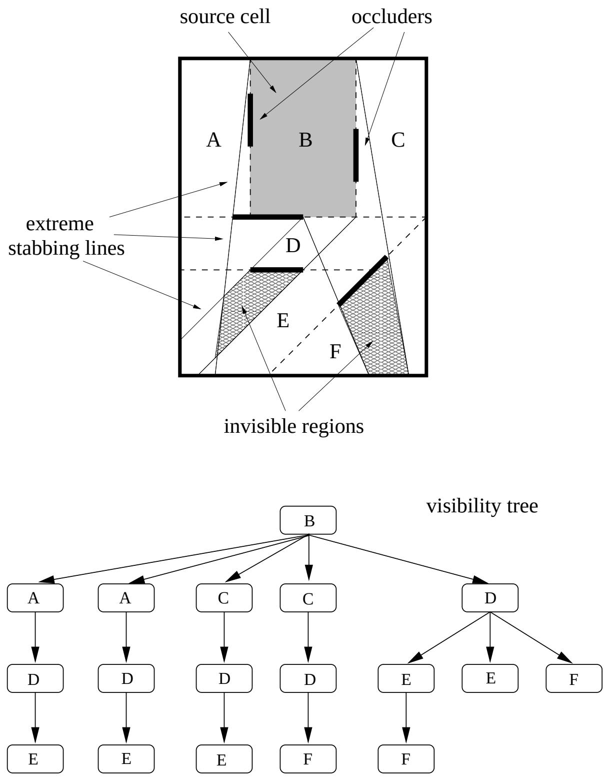

Figure 6.1: Scarcely occluded environment and the effect on the visibility tree.All the cells are classified as potentially visible; the visibility tree is a relatively large data structure.

to rectangular polygons, what limits further testing of our implementation on more sophisticated scenes.

6.3.1 Effects of Visibility Preprocessing

This section presents the effects of the visibility preprocessing in terms of reduction of the size of environment which has to be considered for rendering or other computations based on visibility.

The same environments as in previous section were used to perform the determination of their potentially visible portions by several different methods. For on-line dynamic visibility determination eye-to-cell and eye-to-frustrum visibility calculations were used. Both of them are based on static visibility determination, namely on cell-to-cell (see figure 6.2) and cell-to-frustrum visibility (see figure 3.15). The table 6.3 shows relative numbers of cells necessary to render applying different methods of visibility determination. In other words, the table illustrates percentage of the model which was classified as visible (in average), in relation to the number of cells in the whole model. This percentage directly induces a speedup of the rendering process. Note, that we do not compute the size of the actual size of “volume” reached by visibility overestimation. This computation should show even better results of visibility

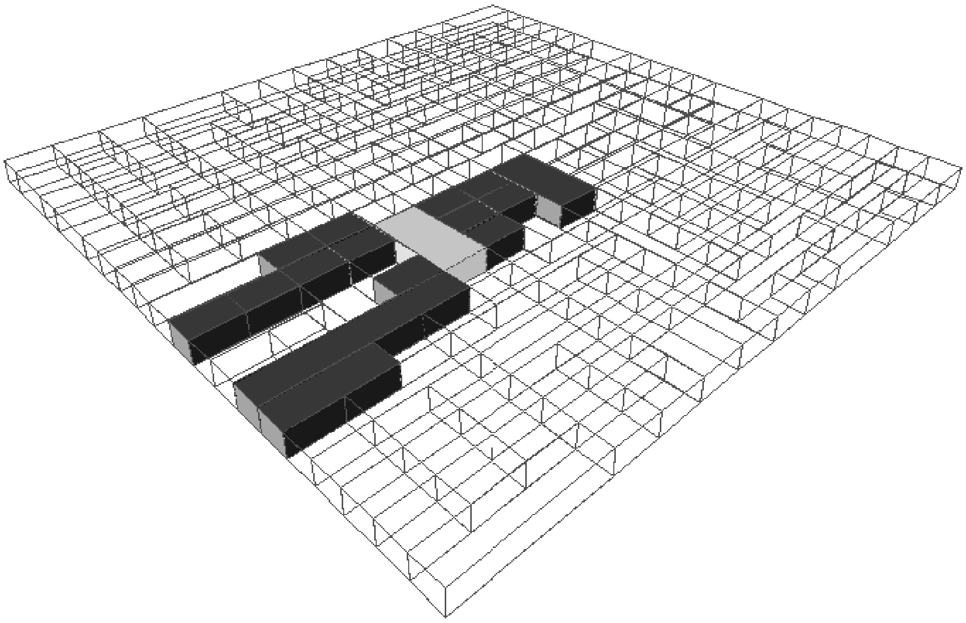

Figure 6.2: Cell-to-cell static visibility. Potentially visible portion of the model is shown in black color.

Scene name	room	6supr	bez2
Number of occluder polygons	30	358	1884
Subdivision time [s]	0.2	1.0	2.2
Number of occluders after SSD	88	1122	2374
Number of cells	24	140	223
Number of boundary polygons	119	622	977
Number of portals	36	120	245
Number of unified cells	2	35	23
VRML file size [kBytes]	85	1200	1200

Table 6.2: Results of the spatial subdivision. Measured on AMD-K5/100Mhz CPU, 32MB RAM,Linux OS.

Scene name	`room`	`6supr`	`bez2`
Number of cells	24	140	223
Number of portals	36	120	245
Static visibility time [s]	2	45	23
Static visibility [% of cells]	32.9	12.0	5.0
Eye-to-cell [% of cells]	8.3	10.0	3.9
Eye-to-frustrum [% of cells]	7.0	5.0	2.1

Table 6.3: Results of the visibility computations. The table shows the time spend by visibility preprocessing as well as the average percentage of the portion of the model classified as visible.

determination based on frustrums. This feature of "frustrum" based visibility preprocessing should be remarkable in the presence of detail objects in the scene, because some of them would lie out of the potentially visible frustrums in corresponding cells, thus classified as invisible.

For densely occluded scenes containing relatively short portal sequences (up to 100 portals), the stabbing line computation is reliable and efficient. Even for extensive datasets consisting of several thousands of polygonal occluders the time requirements do not exceed minutes. However with less occluding scenes, with long portal sequences, we have experienced a few hours long computations, caused by combinatorial nature of portal sequences in such types of environments. Although a long time was spend to discover all the possible mistakes and non-proper handling of cumulative errors, in the case of some complex "real" environments the implementation still appeared to be sticked computing the visibility of many permutations of portal sequences. When spatial subdivision leads to a wrongly divided scene, e.g. with gaps between the walls, the visibility computations take to much time and seem not to terminate in reasonable time. In such cases also the memory requirements to store visibility trees grow very fast.

The table 6.3 gives an overview of the results and measurements obtained on several scenes. The tests has been ran on a PC with Linux operating system, AMD-K5/100MHz CPU and 32MB RAM.

6.4 Other Benefits of the Visibility Computations

In this section, it is proposed how the visibility preprocessing can participate on some other problems related to designing a virtual reality application. The suggestions are results of the experience with the visibility preprocessing and generally related to development multi-user distributed virtual reality system (MUDVRS).

An improvement of the communication scheme for multi-user VR can be achieved by employing location sensitive filtering of the update messages. Each entity in the distributed environment needs to track only those changes in the world database which take a place in a "near" or possibly visible surrounding area of the entity. This visible area (set of cells) defines the *area of interest*(AOI) of the entity relevant to its the current location. It is a straight forward idea to create a *multi-cast group* for each cell and assign a *multi-cast server* to each of the groups. That way if a particular cell belongs to the AOI of the entity, scene updates from the particular multi-cast server are requested. Each entity is then registered with all the multi-cast servers corresponding to cells referenced in the visibility tree of the cell the entity is located in. This method should lead to significant reduction of the communication in the simulation.

6.5 Polyhedra Enumeration

Since the polyhedra enumeration is a very important part of the visibility computations we denote a special section to the performance of the implemented algorithm. All the presented results, were measured using a stand-alone program, which was compiled using the same "kernel" procedures as for the visibility computations. As mentioned in [24] the reverse search reaches a very good performance especially for non-degenerate tasks. The typical job, the program is designed for, is 100 inequalities in 5 dimensions. Table 6.4 shows running times of the convex hull algorithm based on convex polyhedra enumeration, involving the dual transformation.

Points	Dimension	Running time [s]	Points on hull	Edges
100	3	0.4	23	23
200	3	0.9	41	116
300	3	2.0	44	126
800	3	3.5	71	184
100	5	4.1	72	2525

Table 6.4: Convex hull computation : execution times. Measured on AMD K5/100MHz, 32MB RAM.

The bit worse results of polyhedra enumeration for a higher number of input inequalities can be caused namely by the algorithm of initial vertex search based on simplex method of linear programming. Although it reaches performance of $O(n)$ in number of iteration steps, the complexity expressed in number of elementary operations (multiplication,division) is $O(n^3)$ in worse case. Also memory requirements in such situation can grow to $O(n^2)$. In the moment a better algorithm is developed, to close the gap in the polyhedra enumeration.

Chapter 7

Conclusions

7.1 Future Work

Future work and research will be focused on the detail examination of some of the algorithms discussed previously and their possible improvements. The spatial subdivision algorithm should use some better heuristics to subdivide environments of unspecified type. Perhaps it is worth to implement some less complex algorithms to conservatively approximate the antipenumbra, such as one based of gift-wrapping method and then study their efficiency in compare to algorithms already implemented.

Large effort should be spend by closing the gap of the visibility algorithm based on portals, that it is problems of its usage to process sparsely occluded environments. Some hybrid method, based on portals and occluders enclosed in some hierarchical data structure is now the subject of research.

Virtual environments consisting of more than one architectural object, such as town, will be generated,tested and studied. The current browser is going to be extended, to create an experimental multi-user distributed virtual reality system, which will utilize the visibility preprocessing in terms of visibility restrictions of its users and thus to minimize communication requirements .

7.2 Summary

In previous chapters, the description of theoretical background together with practical implementation issues of conservative visibility computations were given. The importance of visibility computations seems to rise quickly with expectations of quality of experience in virtual environments. It was shown that the visibility computations can significantly speedup the visualization process of certain types of environments. The main asset of the thesis, is that the implemented methods give a good background for further research in this field. Moreover, implementation of the visibility preprocessing led to many algorithms of computational geometry and applied mathematics, such as convex hull in arbitrary dimension, linear programming, BSP-tree, which can easily be reused in many future projects.

Appendix A

Contents of the supplied diskette.

There are three archive files on the diskette. The file `diplom.zip` contains this diploma thesis in *Postscript* format. The `source.zip` holds C++ source codes as well as some test models of virtual environments in *Unigraphics* file format. The third file named `results.zip` takes in images and VRML files obtained by visualization of the results.

The list of C++ source code files contained in `source.zip` :

```
2dgraph.cpp
2dgraph.h
Borland.cpp
array.h
base.h
borland.cpp
browser.cpp
browser.h
bsp.cpp
bsp.h
cheats.cpp
cheats.h
common.cpp
common.h
edge.cpp
edge.h
face.cpp
face.h
gbrowser.cpp
gbrowser.h
giftwrap.cpp
giftwrap.h
glmatrix.h
glrender.cpp
glrender.h
hash.cpp
hash.h
hull.cpp
hull.h
hull3d.cpp
hull3d.h
ibrowser.cpp
ibrowser.h
id.h
insert.cpp
inventor.cpp
inventor.h
ko.cpp
ko.h
list.h
makefile
matrix.cpp
matrix.h
object.cpp
object.h
options.cpp
options.h
parser/browser.cxx
parser/convert.cxx
parser/convert.h
parser/dyn_arra.h
parser/nodes.h
parser/p_tab.h
parser/parser.h
parser/ta.cxx
parser/traverse.cxx
parser/traverse.h
plhull.cpp
plhull.h
plucker.cpp
plucker.h
polygon.cpp
polygon.h
portal.cpp
portal.h
portenum.cpp
portenum.h
rectangl.cpp
rectangl.h
scene.cpp
scene.h
scrpoint.cpp
scrpoint.h
stack.cpp
stack.h
stub.cpp
stub.h
t.cpp
template.cpp
test.cpp
unigr.cpp
unigr.h
vector3.cpp
vector3.h
volume.cpp
volume.h
vrml.cpp
vrml.h
watcom.cpp
wingpoly.cpp
wingpoly.h
xwindow.cpp
xwindow.h
```

Acknowledgments

First of all, I would like to thank to Aleš Holeček for his great support. He was an excellent advisor always willing to share my unfinished ideas. I want to thank to Marek Trefný, who implemented the VRML parser and who promptly reacted to all necessary changes. My deep gratitude goes to Martin Cupák for his implementation of the building generator and for creating several models used in the project. Finally let me thank to other students and members of the Computer Graphics Group for their helpful suggestions.

Jiří Bittner
Praha, January 1997

List of Figures

List of Tables

Bibliography

[1] Foley J. D., van Dam A., Feiner S. K., Hughes J. F.: *Computer Graphics Principles and Practice*, Addison–Wesley Publishing Company, 1992.

[2] Werner T., Hersh R., D., Hlaváč V.: *Rendering Real-World Objects Without 3-D Model*, Computer Analysis of Images and Patterns, (Proc. 6th International Conference, CIAP'95), Springer-Verlag, 1995.

[3] Bittner J.,: *Výpočty viditelnosti v hustě stíněných, rozsáhlých 3D scénách, Část I – Dělení prostoru*, Term project at Dept. of Computer Science and Eng. CTU, 1995.

[4] Bittner J.,: *Výpočty viditelnosti v hustě stíněných, rozsáhlých 3D scénách, Část II – Statická a dynamická viditelnost*, Term project at Dept. of Computer Science and Eng. CTU, 1996.

[5] Paterson M. S. Yao F. F.: *Efficient binary space partitioning for hidden-surface removal and solid modeling*, Discrete and Computational Geometry, 5(5):pp. 485–503, 1990.

[6] Rokne J.: *The area of simple polygon*, in Graphics Gems, ed. James Arvo, Academic Press, 1991.

[7] Holeček, A.: *Static Visibility Computation in Virtual Reality Systems*, Doctoral Thesis at Dept. of Computers - CTU, Prague, 1996.

[8] O'Rourke J.: *Computational Geometry in C*, Cambridge University Press, 1994.

[9] Preparata F. P., Shamos M. I.: *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[10] Avis D., Wenger R.: *Algorithms for line traversal in space*, Proc. of 3rd Annual ACM Symposium on Computational Geometry 1987, pp. 300-307, 1987.

[11] McKenna M., O'Rourke J.:*Arrangements of lines in 3-space: A data structure with application*, Proc. of 4th Annual ACM Symposium on Computational Geometry, 1988, pp. 371–380, 1988.

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

[13] Hohmeyer M., Teller S.: *Stabbing Isothetic Rectangles and boxes in* $0(n \lg n)$ *time*, Computational Geometry Theory and Application 4:1992, pp. 201–207, 1992.

[14] Amenta N,: *Finding a line traversal of axial objects in three dimensions*, Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 66–71, 1992.

[15] Teller S., Hohmeyer M.: *Stabbing Oriented Convex Polygons in Randomized* $O(n^2)$ *time*, Contemporary Mathematics, Volume 0, 1994.

[16] Pellegrini M., Shor P.: *Finding stabbing lines in 3-dimensional space*, Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms, 1991, pp. 24–31, 1991.

[17] Škrášek J., Tichý Z.: *Základy aplikované matematiky 3*, SNTL Praha 1990.

[18] Trefný M.: *Formálnípopis 3D scén*,Diploma Thesis, Department of Computer Science and Engineering CTU, 1996.

[19] Bell G., Parisi A., Pesce M.: *The Virtual Reality Modeling Language – Version 1.0 Specification.* http://www.clark.net/theme/vrml/.

[20] Rektorys, K. "Přehled užité matematiky II." *Prometheus*, Praha, 1995.

[21] Teller, S. "Computing the Antipenumbra Cast by an Area Light Source." *Computer Graphics (Proc. SIGGRAPH'92)* 26, 2 (1992), 139-148.

[22] Teller, S., Hanrahan, P. "Global Visibility Algorithms for Illumination Computations." *Computer Graphics (Proc.,Annual conference Series 1993)*, 239-246.

[23] Avis,D.,Fukuda,K. (1992). "A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra," *Discrete and Computational Geometry,* vol. 8, 295-313.

[24] Avis,D. (1993). "A C implementation of the Reverse Search Vertex Enumeration Algorithm," *Technical Report.*

[25] Bland,R.,G. (1977). "A Combinatorial abstraction of linear programming," *J. Combin. Theory B,* vol. 23,33-57.

[26] Edelsbrunner,H. (1987)."Algorithms in Combinatorial Geometry," *Springer-Verlag.*

[27] Gass, S.T. 1969, "Linear Programming," *3rd ed., McGraw-Hill, New York.*

[28] Gigus, Z., Malik, J. "Computing the Aspect Graph for Line Drawings of Polyhedral Objects". *IEEE Transactions on Pattern Analysis and Machine Inteligence 12*, 2 (1990),113-122.

[29] Gigus, Z.,Canny, J., Seidel,R. “Efficiently Computing and Representing Aspect Graphs of Polyhedral Objects”. *IEEE Transactions on Pattern Analysis and Machine Inteligence 13*,6 (1991),542-551.

[30] Plantinga, W., Dyer,C. “Visibility, Occlusion, and the Aspect Graph”. *Int. Journal Computer Vision 5*, 2 (1990),137-160.

[31] Airey, J.M., Rohlf, J.H.,Brooks,Jr.,F.P. “Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments”. *ACM Siggraph Special Issue on 1990 Symposium on Interactive 3D Graphics 24*, 2(1990),41-50.

[32] Funkhouser,T.,Séquin,C.,Teller,S. “Management of Large Amounts of Data in Interactive Building Walkthroughs”. In *Proc. 1992 Workshop on Interactive 3D Graphics* (1992),11-20.

[33] Grove, E., Murali, T.M.,Vitter,J. “The Object Complexity Model for Hidden Surface Elimination”. In Proc. 7th Canadian Conference of Computational Geometry. (Québec City, Canada, 1995),273-278.

[34] Mulmuley, K. “Hidden Surface Removal with Respect to a moving View Point.” In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing* (New Orleans,Lousiana,6-8 May 1991),512-522.

[35] Bern, M.,Dobkin,D.,Eppstein,D.,Grossman,R.“Visibility with a Moving Point of View”.*Algorithmica 11*(1994),360-378.

[36] Greene,N.,Kass,M.,Miller,G. “Hierarchical Z-Buffer Visibility”. In *Computer Graphics Proceedings, Anual Conference Series,1993*,231-240.

[37] Teller,S.,Séquin,C.H. “Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics (Proc. Siggraph '91) 25,* 4 (1991),61-69.

[38] Teller, S.J. “Visibility Computations in Densely Occluded Polyhedral Environments”. *Dissertation work, UC Berkeley*, (1992).

[39] Coorg,S.,Teller,S. “A spatialy and Temporally Coherent Object Space Visibility Algorithm.” *Technical report TM-546,Laboratory for Computer Science,MIT,* (1996).

[40] Grünbaum, B. “Convex Polytopes”. Wiley-Interscience, New York, (1967).

[41] Clarkson, K.L., Shor,P.W. “Applications of random sampling in computational geometry II.” *Discrete computational Geometry,* (1989),387-421.

[42] Teller, S.,Hohmeyer,M. “Computing the lines piercing four lines.” *Technical report UCB/CSD 91/665, Computer Science Department, UC Berkeley*, (1991).