Geometric Data Structures for Computer Graphics

Gabriel Zachmann and Elmar Langetepe

Informatik II/I
University of Bonn
Römerstr. 164
53111 Bonn, Germany
email: {zach,langetep}@cs.uni-bonn.de
`http://web.cs.uni-bonn.de/~zach`
`http://web.cs.uni-bonn.de/I/staff/langetepe.html`

Contents

Abstract

This tutorial aims at presenting a wide range of geometric data structures, algorithms and techniques from computational geometry to computer graphics practitioners. To achieve this goal we introduce several data structures, discuss their complexity, point out construction schemes and the corresponding performance and present standard applications in two and three dimensions.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Data Structures]: Computer Graphics

1. Introduction

In recent years, methods from computational geometry have been widely adopted by the computer graphics community. Many solutions draw their elegance and efficiency from the mutually enriching combination of such geometrical data structures with computer graphics algorithms.

With this tutorial we try to familiarize practitioners in the computer graphics field with several geometric data structures, algorithms and techniques from computational geometry. This should enable the attendants to select the most suitable data structure when developing computer graphics algorithms. In particular, we want to enable them to readily recognize a subproblem if it can be solved by some method known in computational geometry.

The general concept throughout the tutorial is to present each geometric data structure as follows: the data structure will be defined and described in detail; its complexity and some of its fundamental properties will be discussed; construction algorithms and their time bounds are given; one or more simple computational geometry algorithms based upon the data structure will be presented; finally, a number of recent representative and practically relevant algorithms from computer graphics will be described in detail.

Our selection of data structures and algorithms consists of well-known concepts, which are both, powerful *and* easy to implement. However, we do not try to provide a survey over any of the topics touched upon here — this would be far beyond the scope of this tutorial. For the same reason, this tutorial does not provide a comprehensive overview of all techniques and algorithms from computational geometry that might be of interest to computer graphics researchers and developers. However, we do feel that the techniques we present here should be working knowledge of anybody in this field.

The tutorial is organized as follows. The classical quadtrees and k-d-trees are the topics of Section 2. In Section 3 we discuss the concept of Voronoi diagrams and Delaunay triangulations. Furthermore, BSP-trees are presented in Section 4. Section 5 is about volume hierarchies, and finally, in Section 6 we present a method for generic dynamization.

2. Quadtrees and K-d-Trees

Within this section we will present some fundamental geometric data structures.

In section Section 2.1, we introduce the quadtree structure, its definition and complexity, the recursive construction scheme and a standard application are presented. It has applications in mesh generation as shown in Section 2.3, 2.4, 2.5.

A natural generalization of the one-dimensional search tree to k dimensions is shown in Section 2.2. The k-d-tree is efficient for axis-parallel rectangular range queries.

The quadtree description was adapted from de Berg et al. [13] and the k-d-tree introduction was taken from [38].

2.1. Quadtrees and Octrees

2.1.1. Definition

A *quadtree* is a rooted tree so that every internal node has four children. Every node in the tree correspond to a square. If a node v has children, their corresponding squares are the four quadrants, see Figure 1 for an example.

Quadtrees can store many kind of data, we describe the variant that stores a set of points. For the definition a simple recursive splitting of squares is continued until there is only one point in a square. Let P be a set of points.

The definition of a quadtree for a set of points in a square $Q = [x1_Q : x2_Q] \times [y1_Q : y2_Q]$ is as follows:

- If $|P| \leq 1$ then the quadtree is a single leaf where Q and P are stored.
- Otherwise let Q_{NE}, Q_{NW}, Q_{SW} and Q_{SE} denote the four quadrants. Let $x_{mid} := (x1_Q + x2_Q)/2$ and

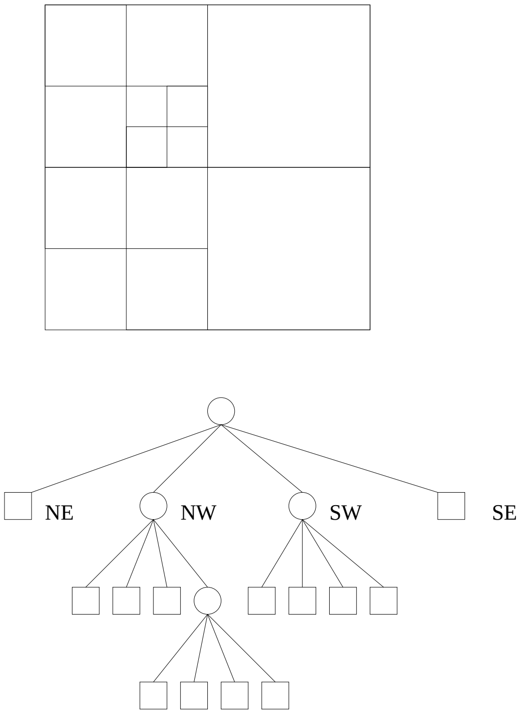

Figure 1: *An example of a quadtree.*

$y_{mid} := (y1_Q + y2_Q)/2$, and define

$P_{NE} := \{p \in P : p_x > x_{mid} \text{ and } p_y > y_{mid}\}$,
$P_{NW} := \{p \in P : p_x \leq x_{mid} \text{ and } p_y > y_{mid}\}$,
$P_{SW} := \{p \in P : p_x \leq x_{mid} \text{ and } p_y \leq y_{mid}\}$ and
$P_{SE} := \{p \in P : p_x > x_{mid} \text{ and } p_y \leq y_{mid}\}$.

The quadtree consists of a root node v, Q is stored at v. In the following, let $Q(v)$ denote the square stored at v. Furthermore v has four children: The X-child is the root of the quadtree of the set P_X for $X \in \{NE, NW, SW, SE\}$.

2.1.2. Complexity and Construction

The recursive definition implies a recursive construction algorithm. Only the starting square has to be chosen adequately. If the split operation cannot be performed well the quadtree is unbalanced. Despite this effect, the depth of the tree is related to the distance between the points.

Theorem 1
The depth of a quadtree for a set P of points in the plane is at most $\log(s/c) + \frac{3}{2}$, where c is the smallest distance between any to points in P and s is the side length of the initial square.

The cost of the recursive construction and the complexity of the quadtree depends on the depth of the tree.

Theorem 2
A quadtree of depth d which stores a set of n points has $O((d+1)n)$ nodes and can be constructed in $O((d+1)n)$ time.

Proof Due to the degree 4 of internal nodes, the total number of leaves is one plus three times the number of internal nodes. Hence it suffices to bound the number of internal nodes.
Any internal node v has one or more points inside $Q(v)$. The squares of the node of a single depth cover the initial square. So at every depth we have at most n internal nodes which gives the node bound.
The most time-consuming task in one step of the recursive approach is the distribution of the points. The amount of time spent is only linear in the number of points and the $O((d+1)n)$ time bound holds. $\square$

The 3D equivalent of quadtrees are *octrees*. The *quadtree* construction can be easily extended to octrees in 3D. The internal nodes of octrees have eight sons and the sons correspond to boxes instead of squares.

2.1.3. Neighbor Finding

A simple application of the quadtree of a point set is *neighbor finding*, i.e., given a node v and a direction, north, east, south or west, find a node v' so that $Q(v)$ is adjacent to $Q(v')$. Normally, v is a leaf and v' should be a leaf as well. The task is equivalent to finding an adjacent square of a given square in the quadtree subdivision.

Obviously one square may have many such neighbors, see Figure 2.

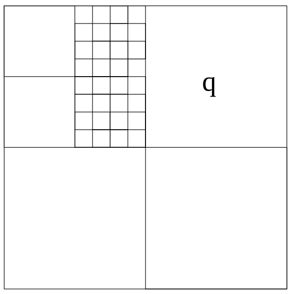

Figure 2: *The square q has many* west *neighbors.*

For convenience, we extend the neighbor search. The given node can also be internal, i.e., v and v' should be adjacent corresponding to the given direction *and* should also have the same depth. If there is no such node, we want to find the deepest node whose square is adjacent.

The algorithm works as follows. Suppose we want to find the north neighbor of v. If v happens to be the SE- or SW-child of its parent, then its north neighbor is easy to find, it is the NE- or NW-child of its parent, respectively. If v itself is the NE- or NW-child of its parent, then we proceed as follows. Recursively find the north neighbor of μ of the parent of v. If μ is an internal node, then the north neighbor of v is a child of μ; if μ is a leaf, the north neighbor we seek for is μ itself.

This simple procedure runs in time $O(d+1)$.

Theorem 3
Let T be quadtree of depth d. The neighbor of a given node v in T a given direction, as defined above, can be found in $O(d+1)$ time.

Furthermore, there is also a simple procedure that constructs a balanced quadtree out of a given quadtree T, this can be done in time $O(d+1)m$ and $O(m)$ space if T has m nodes. For details see Berg et al [13].

Similar results hold for octrees as well.

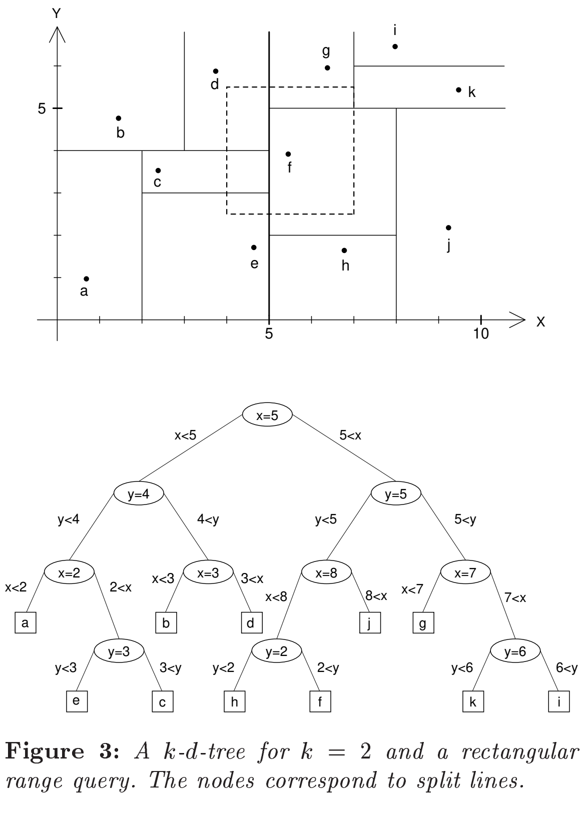

Figure 3: *A k-d-tree for $k = 2$ and a rectangular range query. The nodes correspond to split lines.*

2.2. K-d-Trees

The k-d-tree is a natural generalization of the one-dimensional search tree.

Let D be a set of n points in $\mathbb{R}^k$. For convenience let $k = 2$ and let us assume that all X- and Y-coordinates are different. First, we search for a *split-value* s of the X-coordinates. Then we split D by the *split-line* $X = s$ into subsets

$$\begin{aligned} D_{<s} = \{(x, y) \in D; x < s\} &= D \cap \{X < s\} \\ D_{>s} = \{(x, y) \in D; x > s\} &= D \cap \{X > s\}. \end{aligned}$$

For both sets we proceed with the Y-coordinate and *split-lines* $Y = t_1$ and $Y = t_2$. We repeat the process recursively with the constructed subsets. Thus we obtain a binary tree, namely the 2-d-tree of the point set D, see Figure 3. Each internal node of the tree corresponds to a split-line. For every node v of the 2-d-tree we define the rectangle $R(v)$, which is the intersection of halfplanes corresponding to the path from the root to v. For the root r, $R(r)$ is the plane itself; for the sons of r, say $left$ and $right$, we produce to halfplanes $R(left)$ and $R(right)$ and so on. The set of rectangles $\{R(l) : l \text{ is a leaf}\}$ gives a partition of the plane into rectangles. Every $R(l)$ has exactly one point of D inside.

This structure supports range queries of axis-parallel rectangles, i.e., if Q is an axis-parallel rectangle, the set of sites $v \in D$ with $v \in Q$ can be computed efficiently. We simply have to compute all nodes v with

$$R(v) \cap Q \neq \varnothing.$$

Additionally we have to test whether the points inside the subtree of v are inside Q.

The efficiency of the k-d-tree with respect to range queries depends on the depth of the tree. A balanced k-d-tree can be easily constructed. We sort the X- and Y-coordinates. With this order we recursively split the set into subsets of equal size in time $O(\log n)$. The construction runs in time $O(n \log n)$. Altogether the following theorem holds:

Theorem 4
A balanced k-d-tree for n points in the plane can be constructed in $O(n \log n)$ and needs $O(n)$ space. A range query with an axis-parallel rectangle can be answered in time $O(\sqrt{n} + a)$, where a denotes the size of the answer.

2.3. Height Field Visualization

A special area in 3D visualization is the rendering of large terrains, or more generally, of height fields. A height field is usually given as a uniformly-gridded

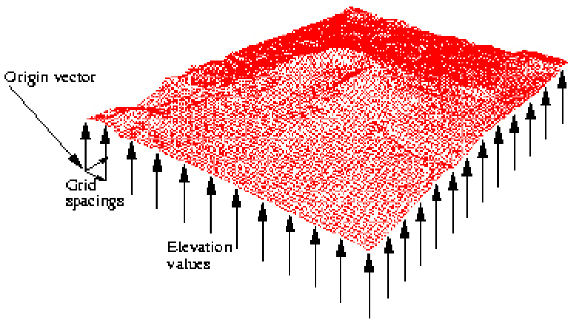

Figure 4: *A height field approximated by a grid* [11].

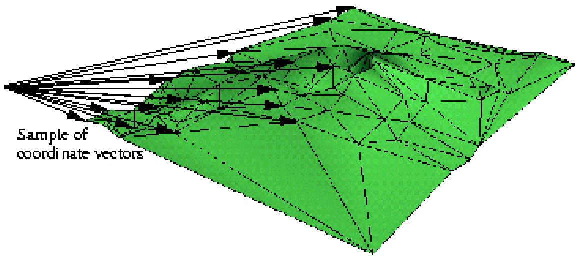

Figure 5: *The same heigt field approximated by a TIN.*

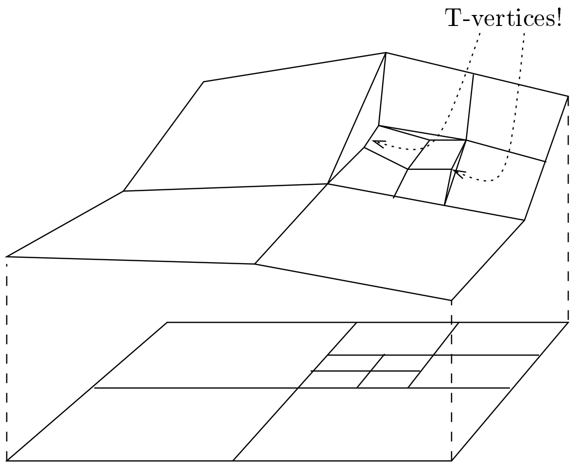

Figure 6: *In order to use quadtres for defining a height field mesh, it should be balanced.*

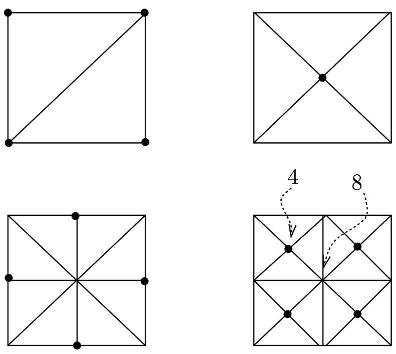

Figure 7: *A quadtree defines a recursive subdivision scheme yielding a 4-8 mesh. The dots denote the newly added vertices. Some vertices have degree 4, some 8 (hence the name).*

square array $h : [0, N-1]^2 \to \mathbb{R}$, $N \in \mathbb{I}$, of height values, where N is typically in the order of 16,384 or more (see Figure 4). In practice, such a raw height field is often stored in some image file format, such as GIF. A regular grid is, for instance, one of the standard forms in which the US Geological Survey publishes their data, known as the Digital Elevation Model (DEM) [22].

Alternatively, height fields can be stored as *triangular irregular networks* (TINs) (see Figure 5). They can adapt much better to the detail and features (or lack thereof) in the height field, so they can approximate any surface at any desired level of accuracy with fewer polygons than any other representation.[44] However, due to their much more complex structure, TINs do not lend themselves as well as more regular representations to interactive visualization.

The problem in terrain visualization is that if the user looks at it from a low viewpoint directed at the horizon, then there are a few parts of the terrain that are very close while the majority of the visible terrain is at a larger distance. Close parts of the terrain should be rendered with high detail, while distant parts should be rendered with very little detail in order to maintain a high frame rate.

In order to solve this problem, a data structure is needed that allows to quickly determine the desired level of detail in each part of the terrain. Quadtrees are such a data structure, in particular, since they seem to be a good compromise between the simplicity of non-hierarchical grid representation and the good adaptivity of TINs.

The general idea is to construct a quadtree over the grid, and then traverse this quadtree top-down in or-

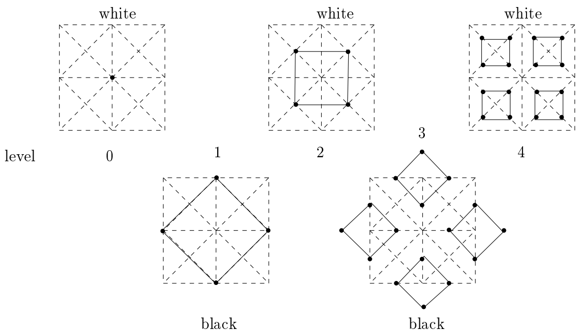

Figure 8: *The 4-8 subdivision can be generated by two interleaved quadtrees. The solid lines connect siblings that share a common father.*

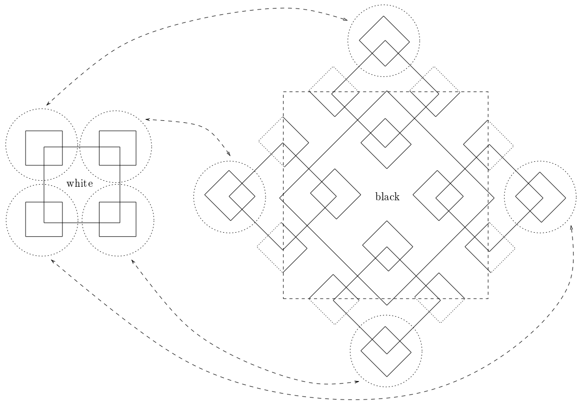

Figure 9: *The white quadtree can be stored in the unused "ghost" nodes of the black quadtree.*

der to render it. At each node, we decide whether the detail offered by rendering the two associated triangles is enough, or if we have to go down further.

One problem with quadtrees is that nodes are not quite independent of each other. Assume we have constructed a quadtree over some terrain as depicted in Figure 6. If we render that as-is, then there will be a gap (a.k.a. *crack*) between the top left square and the fine detail squares inside the top right square. The vertices causing this problem are called *T-vertices*. Triangulating them would help in theory, but in practice this leads to long and thin triangles which have problems on their own.

The solution is, of course, to use a balanced quadtree and triangulate that (see Section 2). Thus, a quadtree offers a recursive subdivision scheme to define a regular grid (see Figure 7): start with a square subdivided into two right-angle triangles; with each recursion step, subdivide the longest side of all triangles (the hypothenuse) yielding two new right-angle triangles each[45] (hence this scheme is sometimes referred to as "longest edge bisection"). This yields a mesh where all vertices have degree 4 or 8 (except the border vertices), which is why such a mesh is often called a 4-8 mesh.

This subdivision scheme induces a directed acyclic graph (DAG) on the set of vertices: vertex j is a child of i if it is created by a split of a right angle at vertex i. This will be denoted by an edge (i, j). Note that almost all vertices are created twice (see Figure 7), so all nodes in the graph have 4 children and 2 parents (except the border vertices).

During rendering, we will choose cells of the subdivision at different levels. Let M^0 be the fully subdivided mesh (which corresponds to the original grid) and M be the current, incompletely subdivided mesh. M corresponds to a subset of the DAG of M^0. The condition of being crack-free can be reformulated in terms of the DAGs associated with M^0 and M:

$$\begin{aligned} &M \text{ is crack-free } \Leftrightarrow \\ &M \text{ does not have any T-vertices } \Leftrightarrow \\ &\forall j \in M : \ (i, j) \in M^0 \Rightarrow (i, j) \in M \qquad (1) \end{aligned}$$

In other words: you cannot subdivide one triangle alone, you also have to subdivide the one on the other side. During rendering, this means that if you render a vertex, then you also have to render all its ancestors (remember: a vertex has 2 parents).

Rendering such a mesh generates (conceptually) a single, long list of vertices that are then fed into the graphics pipeline as a single triangle strip. The pseudocode for the algorithm looks like this (simplified):

```
submesh(i,j)
if error(i) < τ then
  return
end if
if B_i outside viewing frustum then
  return
end if
submesh( j, c_l )
V += p_i
submesh( j, c_r )
```

where error(i) is some error measure for vertex i, and B_i is the sphere around vertex i that completely encloses all descendant triangles.

Note that this algorithm can produce the same vertex multiple times consecutively; this is easy to check, of course. In order to produce one strip, the algorithm

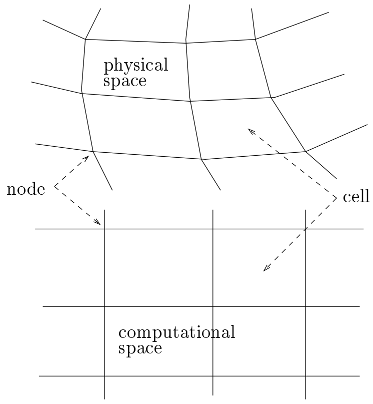

Figure 10: *A scalar field is often given in the form of a curvilinear grid. By doing all calculations in computational space, we can usually save a lot of computational effort.*

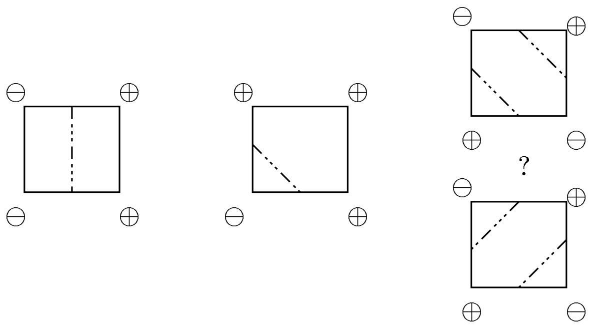

Figure 11: *Cells straddling the isosurface are triangulated accoring to a lookup table. In some cases, several triangulations are possible, which must be resolved by heuristics.*

has to copy older vertices to the current front of the list at places where it makes a "turn"; again, this is easy to detect, and the interested reader is referred to [45].

One can speed up the culling a bit by noticing that if B_i is completely inside the frustum, then we do not need to test the child vertices any more.

We still need to think about the way we store our terrain subdivision mesh. Eventually, we will want to store it as a single linear array for two reasons:

1. The tree is complete, so it really would not make sense to store it using pointers.
2. We want to map the file that holds the tree into memory as-is (for instance, with Unix' `mmap` function), so pointers would not work at all.

We should keep in mind, however, that with current architectures, every memory access that can not be satisfied by the cache is extremely expensive (this is even more so with disk accesses, of course).

The simplest way to organize the terrain vertices is a matrix layout. The disadvantage is that there is no cache locality at all across the major index. In order to improve this, people often introduce some kind of blocking, where each block is stored in matrix and all blocks are arranged in matrix order, too. Unfortunately, Lindstrom and Pascucci[45] report that this is, at least for terrain visualization, worse than the simple matrix layout by a factor 10!

Enter quadtrees. They offer the advantage that vertices on the same level are stored fairly close in memory. The 4-8 subdivision scheme can be viewed as two quadtrees which are interleaved (see Figure 8): we start with the first level of the "white" quadtree that contains just the one vertex in the middle of the grid, which is the one that is generated by the 4-8 subdivision with the first step. Next comes the first level of the "black" quadtree that contains 4 vertices, which are the vertices generated by the second step of the 4-8 subdivision scheme. Etc. Note that the black quadtree is exactly like the white one, except it is rotated by 45°. When you overlay the white and the black quadtree you get exactly the 4-8 mesh.

Notice that the black quadtree contains nodes that are outside the terrain grid; we will call these nodes "ghost nodes". The nice thing about them is that we can store the white quadtree in place of these ghost nodes (see Figure 9). This reduces the number of unused elements in the final linear array down to 33%.

During rendering we need to calculate the indices of the child vertices, given the three vertices of a triangle. It turns out that by cleverly choosing the indices of the top-level vertices this can be done as efficiently as with a matrix layout.

The interested reader can find more about this topic in Lindstrom et al.[44], Lindstrom and Pascucci[45], Balmelli et al.[6], Balmelli et al.[5], and many others.

2.4. Isosurface Generation

One technique (among many others) of visualizing a 3-dimensional volume is to extract isosurfaces and render those as a regular polygonal surface. It can be used to extract the surfaces of bones or organs in medical scans, such as MRI or CT.

Assume for the moment that we are given a scalar

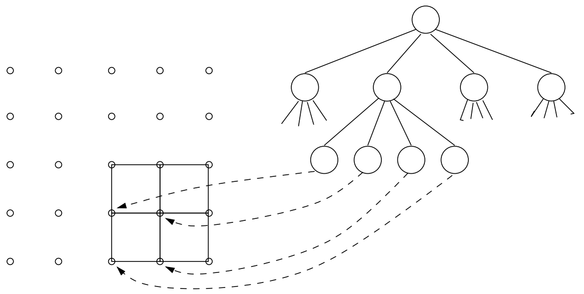

Figure 12: *Octrees offer a simple way to compute isosurfaces efficiently.*

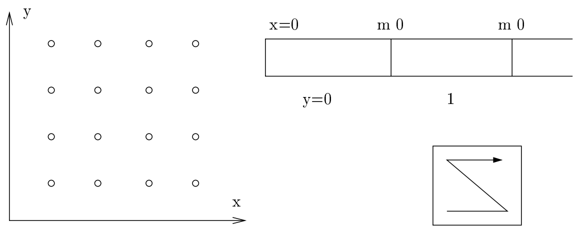

Figure 13: *Volume data layout should match the order of traversal of the octree.*

field $f : \mathbb{R}^3 \to \mathbb{R}$. Then the task of finding an isosurface would "just" be to find all solutions (i.e., all roots) of the equation $f(\vec{x}) = t$.

Since we live in a discrete world (at least in computer graphics), the scalar field is given usually in the form of a *curvilinear grid*: the vertices of the *cells* are called *nodes*, and we have one scalar and a 3D point stored at each node (see Figure 10). Such a curvilinear grid is usually stored as a 3D array, which can be conceived as a regular 3D grid (here, the cells are often called *voxels*).

The task of finding an isosurface for a given value t in a curvilinear grid amounts to finding all cells of which at least one node (i.e., corner) has a value less than t and one node has a value greater than t. Such cells are then triangulated according to a lookup table (see Figure 11). So, a simple algorithm works as follows [46]: compute the sign for all nodes $(\oplus \triangleq > t\,,\ \ominus \triangleq < t)$; then consider each cell in turn, use the eight signs as an index into the lookup table, and triangulate it (if at all).

Notice that in this algorithm we have only used the 3D array — we have not made use at all of the information exactly *where* in space the nodes are (except when actually producing the triangles). We have, in fact, made a transition from *computational space* (i.e., the curvilinear grid) to *computational space* (i.e., the 3D array). So in the following, we can, without loss of generality, restrict ourselves to consider only regular grids, i.e., 3D arrays.

The question is, how we can improve the exhaustive algorithm. One problem is that we must not miss any little part of the isosurface. So we need a data structure that allows us to discard large parts of the volume where the isosurface is guaranteed to *not* be. This calls for octrees.

The idea is to construct a complete octree over the cells of the grid [65] (for the sake of simplicity, we will assume that the grid's size is a power of two). The leaves point to the lower left node of their associated cell (see Figure 12). Each leaf ν stores the minimum $\nu_{\min}$ and the maximum $\nu_{\max}$ of the 8 nodes of the cell. Similarly, each inner node of the octree stores the min/max of its 8 children.

Observe that an isosurface intersects the volume associated with a node ν (inner or leaf node) if and only if $\nu_{\min} \le t \le \nu_{\max}$. This already suggests how the algorithm works: start with the root and visit recursively all the children where the condition holds. At the leaves, construct the triangles as usual.

This can be accelerated further by noticing that if the isosurface crosses an edge of a cell, then that edge will be visited exactly four times during the complete procedure. Therefore, when we visit an edge for the first time, we compute the vertex of the isosurface on that edge, and store the edge together with the vertex in a hash table. So whenever we need a vertex on an edge, we first try to look up that edge in the hash table. Our observation also allows us to keep the size of the hash table fairly low: when an edge has been visited for the fourth time, then we know that it cannot be visited any more; therefore, we remove it from the hash table.

2.5. Ray Shooting

Ray shooting is an elementary task that frequently arises in ray tracing, volume visualization, and in games for collision detection or terrain following. The task is, basically, to find the earliest hit of a given ray when following that ray through a scene composed of polygons or other objects.

A simple idea to avoid checking the ray against all objects is to partition the universe into a regular grid (see Figure 14). With each cell we store a list of objects that occupy that cell (at least partially). Then, we just walk along the ray from cell to cell, and check the ray against all those objects that are stored with that cell.

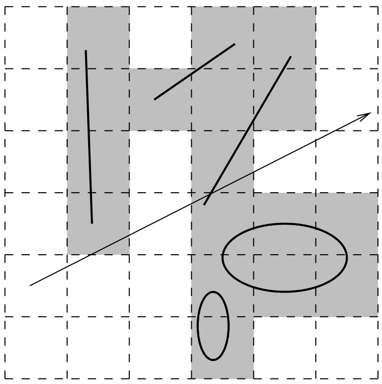

Figure 14: *Ray shooting can be implemented efficiently with an octree.*

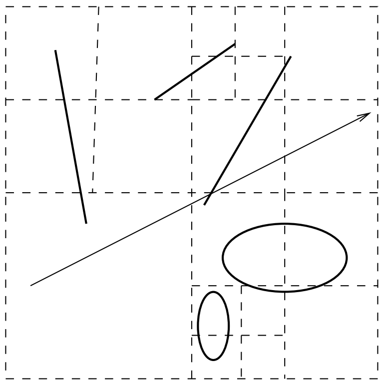

Figure 15: *The same scenario utilizing an octree.*

In this scheme (and others), we need a technique called *mailboxes* that prevents us from checking the ray twice against the same object [27]: every ray gets a unique ID (we just increment a global variable holding that ID whenever we start with a new ray); during traversal, we store the ray's ID with the object whenever we have performed an intersection test with it. But before doing an intersection test with an object, we look into its mailbox whether or not the current ray's ID is already there; if so, then we know that we have already performed the intersection test in an earlier cell.

In the following, we will present two methods which both utilize octrees to further reduce the number of objects considered.

2.5.1. 3D Octree

A canonical way to improve any grid-based method is to construct an octree (see Figure 15). Here, the octree leaves store lists of objects (or, rather, pointers to objects). Since we are dealing now with polygons and other graphical objects, the leaf rule for the octree construction process must be changed slightly:

1. maximum depth reached; or,
2. only one polygon/object occupies the cell.

We can try to better approximate the geometry of the scene by changing the rule to stop only when there are no objects in the cell (or the maximum depth is reached).

How do we traverse an octree along a given ray? Like in the case of a grid, we have to make "horizontal" steps, which actually advance along the ray. With octrees, though, we also need to make "vertical" steps, which traverse the octree up or down.

All algorithms for ray shooting with octrees can be classified into two classes:

- Bottom-up: this method starts at that leaf in the octree that contains the origin of the ray; from there it tries to find that neighbor cell that is stabbed next by the ray, etc.
- Top-down: this method starts at the root of the octree, and tries to recurse down into exactly those nodes and leaves that are stabbed by the ray.

Here, we will describe a top-down method [56]. The idea is to work only with the ray parameter in order to decide which children of a node must be visited.

Let the ray be given by

$$\vec{x} = \vec{p} + t\vec{d}$$

and a voxel v by

$$[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$$

In the following, we will describe the algorithm assuming that all $d_i > 0$; later, we will show that the algorithm works also for all other cases.

First of all, observe that if we already have the line parameters of the intersection of the ray with the borders of a cell, then it is trivial to compute the line intervals half-way in between (see Figure 16):

$$t^m_\alpha = \frac{1}{2}(t^l_\alpha + t^h_\alpha)\ ,\ \alpha \in \{x, y, z\} \qquad (2)$$

So, for 8 children of a cell, we need to compute only three new line parameters. Clearly, the line intersects a cell if and only if $\max\{t^l_i\} < \min\{t^h_j\}$.

The algorithm can be outlined as follows:

<u>**traverse(v, t^l, t^h)**</u>
compute t^m
determine order in which sub-cells are hit by the ray
for all sub-cells v_i that are hit **do**
  traverse(v_i, $t^l|t^m$, $t^m|t^h$)
end for

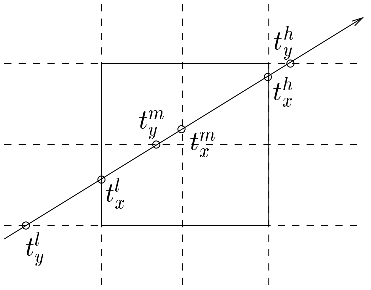

Figure 16: *Line parameters are trivial to compute for children of a node.*

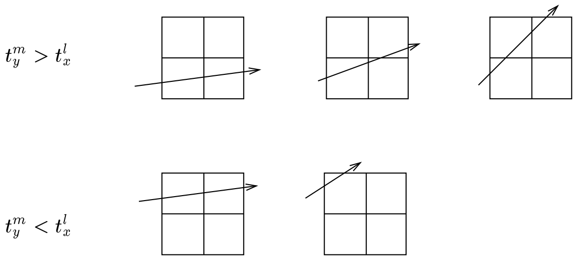

Figure 17: *The sub-cell that must be traversed first can be found by simple comparisons. Here, only the case $t_x^l > t_y^l$ is depicted.*

where $t^l|t^m$ means that we construct the lower boundary for the respective cell by passing the appropriate components from t^l and t^m.

In order to determine the order in which sub-cells should be traversed, we first need to determine which sub-cell is being hit first by the ray. In 2D, this is accomplished by two comparisons (see Figure 17). Then, the comparison of t_x^m with t_y^m tells us which cell is next.

In 3D, this takes a little bit more work, but is essentially the same. First, we determine on which side the ray has been entering the current cell by the following table:

$\max\{t_i^l\}$	Side
t_x^l	YZ
t_y^l	XZ
t_z^l	XY

Next, we determine the first sub-cell to be visited by this table (see Figure 18 for the numbering scheme):

Side	condition	index bits
XY	$t_z^m < t_x^l$	0
	$t_y^m < t_x^l$	1
XZ	$t_x^m < t_y^l$	0
	$t_z^m < t_y^l$	2
YZ	$t_y^m < t_x^l$	1
	$t_z^m < t_x^l$	2

The first column is the entering side determined in the first step. The third column yields the index of the first sub-cell to be visited: start with an index of zero; if one or both of the conditions of the second column hold, then the corresponding bit in the index as indicated by the third column should be set. Finally, we can traverse all sub-cells according to the following table:

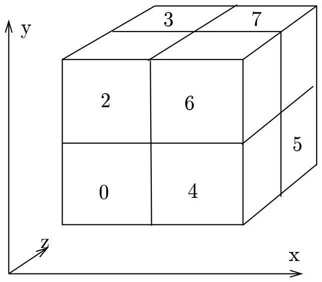

Figure 18: *Sub-cells are numbered according to this scheme.*

current	exit side		
sub-cell	YZ	XZ	XY
0	4	2	1
1	5	3	ex
2	6	ex	3
3	7	ex	ex
4	ex	6	5
5	ex	7	ex
6	ex	ex	7
7	ex	ex	ex

where "exit side" means the exit side of the ray for the *current* sub-cell.

If the ray direction contains a negative component(s), then we just have to mirror all tables along the respective axis (axes) conceptually. This can be implemented efficiently by an XOR operation.

2.5.2. 5D Octree

In the previous, simple algorithm, we still walk along a ray every time we shoot it into the scene. However, rays are essentially static objects, just like the geometry of the scene! This is the basic observation behind

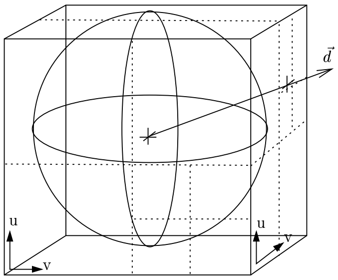

Figure 19: *With the direction cube, we can discretize directions, and organize them with any hierarchical partitioning scheme.*

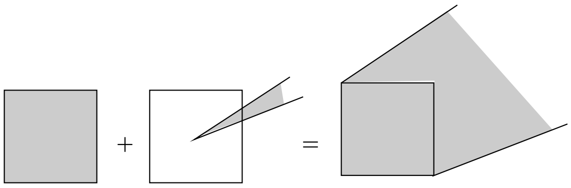

Figure 20: *A uv interval on the direction cube plus a xyz interval in 3-space yield a beam.*

the following algorithm [1, 4]. Again, it makes use of octrees to adaptively decompose the problem.

The underlying technique is a discretization of rays, which are 5-dimensional objects. Consider a cube enclosing the unit sphere of all directions. We can identify any ray's direction with a point on that cube, hence it is called *direction cube* (see Figure 19). The nice thing about it is that we can now perform any hierarchical partitioning scheme that works in the plane, such as an octree: we just apply the scheme individually on each side.

Using the direction cube, we can establish a one-to-one mapping between direction vectors and points on all 6 sides of the cube, i.e.,

$$S^2 \quad \leftrightarrow \quad [-1,+1]^2 \times \{+x,-x,+y,-y,+z,-z\}$$

We will denote the coordinates on the cube's side by u and v.

Within a given universe $B = [0,1]^3$ (we assume it is a box), we can represent all possibly occurring rays by points in

$$R = B \times [-1,+1]^2 \times \{+x,-x,+y,-y,+z,-z\} \quad (3)$$

which can be implemented conveniently by 6 copies of 5-dimensional boxes.

Returning to our goal, we now build six 5-dimensional octrees as follows. Associate (conceptually) all objects with the root. Partition a node in the octree, if

1. there are too many objects associated with it; *and*
2. the node's cell is too large.

If a node is partitioned, we must also partition its set of objects and assign each subset to one of the children.

Observe that each node in the 5D octree defines a beam in 3-space: the xyz-interval of the first three coordinates of the cell define a box in 3-space, and the remaining two uv-intervals define a cone in 3-space. Together (more precisely, their Minkowski sum) they define a beam in 3-space that starts at the cell's box and extends in the general direction of the cone (see Figure 20).

Since we have now defined what a 5D cell of the octree represents, it is almost trivial to define how objects are assigned to sub-cells: we just compare the bounding volume of each object against the sub-cells 3D beam. Note that an object can be assigned to several sub-cells (just like in regular 3D octrees). The test whether or not an object intersects a beam could be simplified further by enclosing a beam with a cone, and then checking the objects bounding sphere against that cone. This just increases the number of false positives a little bit.

Having computed the six 5D octrees for a given scene, ray tracing through that octree is almost trivial: map the ray onto a 5D point via the direction cube; start with the root of that octree which is associated to the side of the direction cube onto which the ray was mapped; find the leaf in that octree that contains the 5D point (i.e., the ray); check the ray against all objects associated with that leaf.

By locating a leaf in one of the six 5D octrees, we have discarded all objects that do *not* lie in the general direction of the ray. But we can optimize the algorithm even further.

First of all, we sort all objects associated with a leaf along the dominant axis of the beam by their minimum (see Figure 21). If the minimum coordinate of an object along the dominant axis is greater than the current intersection point, then we can stop — all other possible intersection points are farther away.

Second, we can utilize ray coherence as follows. We maintain a cache for each level in the ray tree that stores the leaves of the 5D octrees that were visited last time. When following a new ray, we first look into

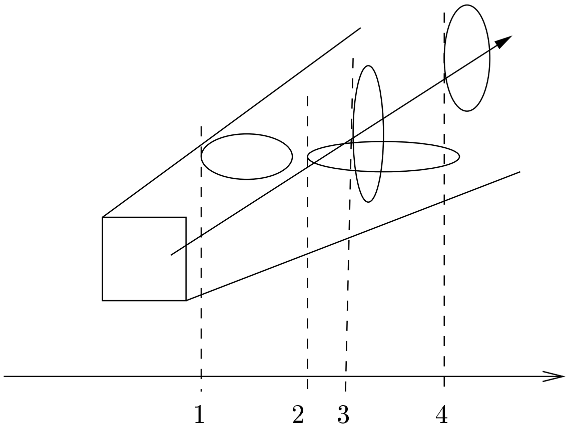

Figure 21: *By sorting objects with in each 5D leaf, we can often stop checking ray intersection quite early.*

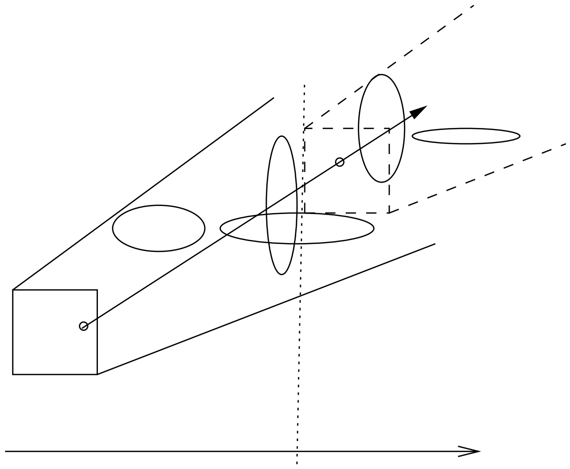

Figure 22: *By truncating the beam (or rather, the list of objects) we can save a lot of memory usage of a 5D octree, while reducing performance only insignificantly.*

the octree leaf in the cache whether it is contained therein, before we start searching for it from the root.

Another trick (that works with other ray acceleration schemes as well) is to exploit the fact that we do not need to know the *first* occluder between a point on a surface and a light source. Any occluder suffices to assert that the point is in shadow. So we also keep a cache with each light source which stores that object (or a small set) which has been an occluder last time.

Finally, we would like to mention a memory optimization technique for 5D octrees, because they can occupy a lot of memory. It is based on the observation that within a beam defined by a leaf of the octree the objects at the back (almost) never intersect with a ray emanating from that cell (see Figure 22). So we store objects with a cell only if they are within a certain distance. Should a ray not hit any object, then we start a new intersection query with another ray that has the same direction and a starting point just behind that maximum distance. Obviously, we have to make a trade-off between space and speed here, but when chosen properly, the cut-off distance should not reduce performance too much while still saving a significant amount of memory.

3. Voronoi Diagrams

For a given set of sites inside an area the Voronoi diagram is a partition of the area into regions of the same neighborship. The Voronoi diagram and its dual have been used for solving numerous problems in many fields of science.

We will concentrate on its application to geometric problems in 2D and 3D. For an overview of the Voronoi diagram and its dual in computational geometry one may consult the surveys by Aurenhammer [2], Bernal [9], Fortune [24] and Aurenhammer and Klein [3]. Additionally, chapters 5 and 6 of Preparata and Shamos [54] and chapter 13 of Edelsbrunner [19] could be consulted.

We start in Section 3.1 with the simple case of the Voronoi diagram and the Delaunay triangulation of n points in the plane, under the Euclidean distance. Additionally we mention some of the elementary structural properties that follow from the definitions.

In Section 3.2 different algorithmic schemes for computing the structures are mentioned. We present a simple *incremental construction* approach which can easily be generalized to 3D, see Section 3.3.1.

Apart from the Euclidean 3D case some other interesting generalizations are mentioned in Section 3.3.2.

In Section 3.4 the relevance of the Voronoi diagram and the Delaunay triangulation in 3D are shown.

Note, that we can only sketch many of the subjects here. For further details and further literature see one of the surveys mentioned above. The figures are taken from Aurenhammer and Klein [3].

3.1. Definitions and Elementary Properties

3.1.1. Voronoi Diagram

Let S a set of $n \geq 3$ point sites $p, q, r, \ldots$ in the plane. In the following we assume that the points are in *general position*, i.e., no four of them lie on the same circle and no three of them on the same line.

For points $p = (p_1, p_2)$ and $x = (x_1, x_2)$ let $d(p, x)$ denote their Euclidean distance. By $\overline{pq}$ we denote the line segment from p to q. The closure of a set A will be denoted by $\overline{A}$.

Definition 5
For $p, q \in S$ let

$$B(p,q) = \{x \mid d(p,x) = d(q,x)\}$$

be the *bisector* of p and q. $B(p,q)$ is the perpendicular line through the center of the line segment $\overline{pq}$. It separates the halfplane

$$D(p,q) = \{x \mid d(p,x) < d(q,x)\}$$

containing p from the halfplane $D(q,p)$ containing q. We call

$$\mathrm{VR}(p,S) = \bigcap_{q \in S, q \neq p} D(p,q)$$

the *Voronoi region* of p with respect to S. Finally, the *Voronoi diagram* of S is defined by

$$V(S) = \bigcup_{p,q \in S, p \neq q} \overline{\mathrm{VR}(p,S)} \cap \overline{\mathrm{VR}(q,S)}.$$

An illustration is given in Figure 23. It shows how the plane is decomposed by $V(S)$ into Voronoi regions. Note that it is convenient to imagine a simple closed curve Γ around the "interesting" part of the Voronoi diagram.

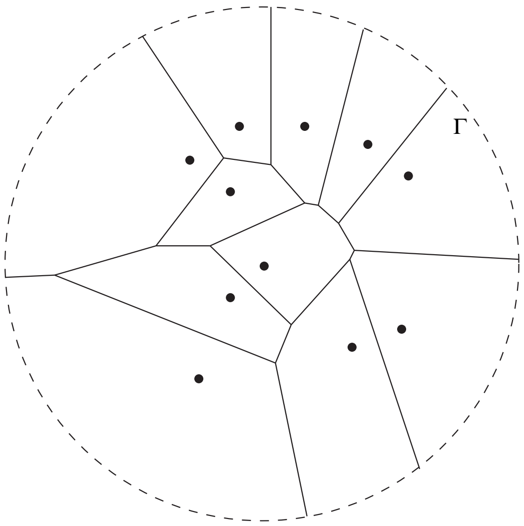

Figure 23: *A Voronoi diagram of points in the Euclidean plane.*

The common boundary of two Voronoi regions belongs to $V(S)$ and is called a *Voronoi edge*, if it contains more than one point. If the Voronoi edge e borders the regions of p and q then $e \subset B(p,q)$ holds. Endpoints of Voronoi edges are called *Voronoi vertices*; they belong to the common boundary of three or more Voronoi regions.

There is an intuitive way of looking at the Voronoi diagram. For any point x in the plane we can expand the circle $C(r)$ with center x and radius r by increasing r continuously. We detect three cases depending on which event occurs first:

- If $C(r)$ hits one of the n sites, say p, then $x \in \mathrm{VR}(p,S)$.
- If $C(r)$ hits two sites p and q simultaneously x belongs to the Voronoi edge of p and q.
- If $C(r)$ hits three sites p, q and r simultaneously x is the Voronoi vertex of p, q and r.

We will enumerate some of the significant properties of Voronoi diagrams.

1. Each Voronoi region $\mathrm{VR}(p,S)$ is the intersection of at most $n-1$ open halfplanes containing the site p. Every $\mathrm{VR}(p,S)$ is open and convex. Different Voronoi regions are disjoint.
2. A point p of S lies on the convex hull of S iff its Voronoi region $\mathrm{VR}(p,S)$ is unbounded.
3. The Voronoi diagram $V(S)$ has $O(n)$ many edges and vertices. The average number of edges in the boundary of a Voronoi region is less than 6.

The Voronoi diagram is a simple linear structure and provides for a partition of the plane into cells of the same neighborship. We omit the proofs and refer to the surveys mentioned in the beginning.

Note, that the Voronoi edges and vertices build a graph. Therefore the diagram normally is represented by a graph of linear size. For example the diagram can be represented by a *doubly connected edge list DCEL*, see de Berg et al. [13], or with the help of an *adjacency matrix*.

3.1.2. Delaunay Triangulation

We consider the dual graph of the Voronoi diagram, the so called *Delaunay triangulation*. In general, a *triangulation* of S is a planar graph with vertex set S and straight line edges, which is maximal in the sense that no further straight line edge can be added without crossing other edges. The triangulation of a point set S has not more than $O(|S|)$ triangles.

Definition 6
The *Delaunay triangulation* $\mathrm{DT}(S)$ is the dual Graph of the Voronoi diagram. The edges of $\mathrm{DT}(S)$ are called *Delaunay edges*.

Obviously, the Delaunay triangulation $\mathrm{DT}(S)$ is a triangulation of S, an example is shown in Figure 24.

We present two equivalent definitions of the Delaunay triangulation. They are applied for the computation of the diagram and give also rise to generalization, for example if the dual of a Voronoi diagram is no longer well-defined.

1. Two points p, q of S give rise to a Delaunay edge iff a circle C exists that passes through p and q and does not contain any other site of S in its interior or boundary.
2. Three points of S give rise to a Delaunay triangle iff their circumcircle does not contain a point of S in its interior.

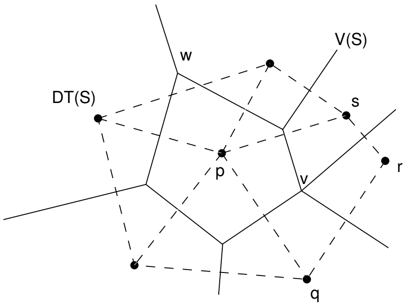

Figure 24: *Voronoi diagram and Delaunay triangulation.*

3.2. Computation

The construction of the Voronoi diagram has time complexity $\Theta(n \log n)$. The lower bound $\Omega(n \log n)$ can be achieved by the following reductions.

- A reduction to the convex hull problem is given by Shamos [58].
- A reduction to the ϵ-closeness problem is given by Djidjev and Lingas [16] and by Zhu and Mirzaian [68].

The well-known computation paradigms

- *Incremental construction*,
- *Divide-and-Conquer* and
- *Sweep*.

are convenient for the construction of the Voronoi diagram or the Delaunay triangulation, respectively. They can also be generalized to other metrics and sites other than points, for example line segments or polygonal chains. The result of the algorithms is stored in a graph of linear size, see above.

All these approaches run in deterministic $O(n \log n)$. We explain a simple *Incremental construction* technique which runs in $O(n \log n)$ expected time and computes the Delaunay triangulation. The presentation is adapted from Klein and Aurenhammer [3]. The technique can easily be generalized to the three dimensional case as we will see in Section 3.3.1.

Simple incremental construction: The insertion process is described as follows: We construct $\mathrm{DT}_i = \mathrm{DT}(\{p_1, \ldots, p_{i-1}, p_i\})$ by inserting the site p_i into DT_{i-1}. We follow Guibas and Stolfi [30] and construct DT_i by exchanging edges, using Lawson's [41] original edge flipping procedure, until all edges invalidated by p_i have been removed.

It is helpful to extend the notion of triangle to the unbounded face of the Delaunay triangulation. If $\overline{pq}$ is an edge of the convex hull of S we call the supporting outer halfplane H not containing S an *infinite triangle* with edge $\overline{pq}$. Its circumcircle is H itself, the limit of all circles through p and q whose center tend to infinity within H. As a consequence, each edge of a Delaunay triangulation is now adjacent to two triangles.

Those triangles of DT_{i-1} whose circumcircles contain the new site, p_i, are said to be *in conflict* with p_i. According to the (equivalent) definition of the DT_i, they will no longer be Delaunay triangles.

Let $\overline{qr}$ be an edge of DT_{i-1}, and let $T(q, r, t)$ be the triangle adjacent to $\overline{qr}$ that lies on the other side of $\overline{qr}$ than p_i; see Figure 25. If its circumcircle $C(q, r, t)$ contains p_i then each circle through q, r contains at least one of p_i, t. Consequently, $\overline{qr}$ cannot belong to DT_i, due to the (equivalent) definition. Instead, $\overline{p_i t}$ will be a new Delaunay edge, because there exists a circle contained in $C(q, r, t)$ that contains only p_i and t in its interior or boundary. This process of replacing edge $\overline{qr}$ by $\overline{p_i t}$ is called an *edge flip*.

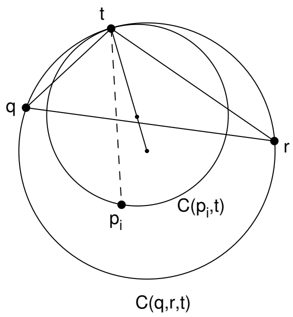

Figure 25: *If triangle $T(q, r, t)$ is in conflict with p_i then former Delaunay edge $\overline{qr}$ must be replaced by $\overline{p_i t}$.*

The necessary edge flips can be carried out efficiently if we know the triangle $T(q, s, r)$ of DT_{i-1} that contains p_i, see fig. Figure 26. The line segments connecting p_i to q, r, and s will be new Delaunay edges, by the same argument from above. Next, we check if e. g. edge $\overline{qr}$ must be flipped. If so, the edges $\overline{qt}$ and $\overline{tr}$ are tested, and so on. We continue until no further edge currently forming a triangle with, but not containing p_i, needs to be flipped, and obtain DT_i.

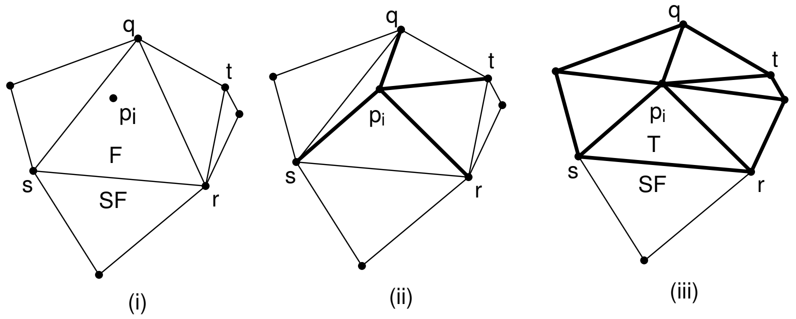

Figure 26: *Updating DT_{i-1} after inserting the new site p_i. In (ii) the new Delaunay edges connecting p_i to q, r, s have been added, and edge $\overline{qr}$ has already been flipped. Two more flips are necessary before the final state shown in (iii) is reached.*

Two task have to be considered:

1. Find the triangle of DT_{i-1} that is in conflict with p_i.
2. Perform all flips starting from this triangle.

It can be shown that the second task is bounded by the degree of p_i in the new triangulation. If the triangle of DT_{i-1} containing p_i is known, the structural work needed for computing DT_i from DT_{i-1} is proportional to the degree d of p_i in DT_i.

So we yield an obvious $O(n^2)$ time algorithm for constructing the Delaunay triangulation of n points: we can determine the triangle of DT_{i-1} containing p_i within linear time, by inspecting all candidates. Moreover, the degree of p_i is trivially bounded by n.

The last argument is too crude. There can be single vertices in DT_i that do have a high degree, but their *average* degree is bounded by 6.

With a special implementation using a directed acyclic graph (DAG), also called *Delaunay tree* due to Boissonnat and Teillaud [10], we can detect the triangles of DT_{i-1} which are in conflict with p_i in $O(\log i)$ expected time.

Altogether we get the following result:

Theorem 7
The Delaunay triangulation of a set of n points in the plane can be easily incrementally constructed incrementally in expected time $O(n \log n)$, using expected linear space. The average is taken over the different orders of inserting the n sites.

3.3. Generalization of the Voronoi Diagram

3.3.1. Voronoi Diagram and Delaunay Triangulation in 3D

We will see that incremental construction is also appropriate for the 3D case. The following description was adapted from Aurenhammer and Klein [3].

Let S be a set of n point sites in 3D. The *bisector* of two sites $p, q \in S$ is the perpendicular plane through the midpoint of the line segment $\overline{pq}$. The region $\mathrm{VR}(p, S)$ of a site $p \in S$ is the intersection of halfspaces bounded by bisectors, and thus is a 3-dimensional convex polyhedron. The boundary of $\mathrm{VR}(p, S)$ consists of *facets* (maximal subsets within the same bisector), of *edges* (maximal line segments in the boundary of facets), and of *vertices* (endpoints of edges). The regions, facets, edges, and vertices of $V(S)$ define a *cell complex* in 3D.

This cell complex is face-to-face: if two regions have a non-empty intersection f, then f is a face (facet, edge, or vertex) of both regions. As an appropriate data structure for storing a 3-dimensional cell complex we mention the facet-edge structure in Dobkin and Laszlo [17].

Complexity: The number of facets of $\mathrm{VR}(p, S)$ is at most $n-1$, at most one for each site $q \in S \setminus \{p\}$. Hence, by the Eulerian polyhedron formula, the number of edges and vertices of $\mathrm{VR}(p, S)$ is $O(n)$, too. This shows that the total number of components of the diagram $V(S)$ in 3D is $O(n^2)$. In fact, there are configurations S that force each pair of regions of $V(S)$ to share a facet, thus achieving their maximum possible number of $\binom{n}{2}$; see, e.g., Dewdney and Vranch [15]. This fact sometimes makes Voronoi diagrams in 3D less useful compared to 2-space. On the other hand, Dwyer [18] showed that the *expected* size of $V(S)$ in

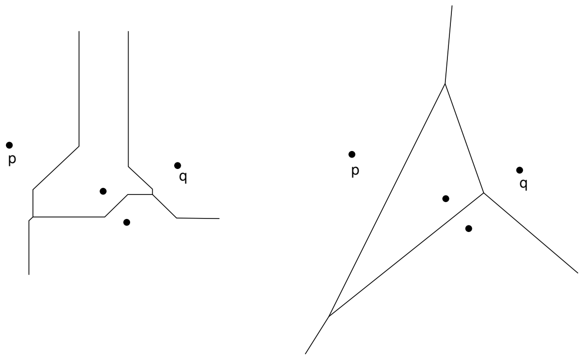

Figure 27: *The Voronoi diagram of a point set in L_1 and L_2. Note, that there are structural differences.*

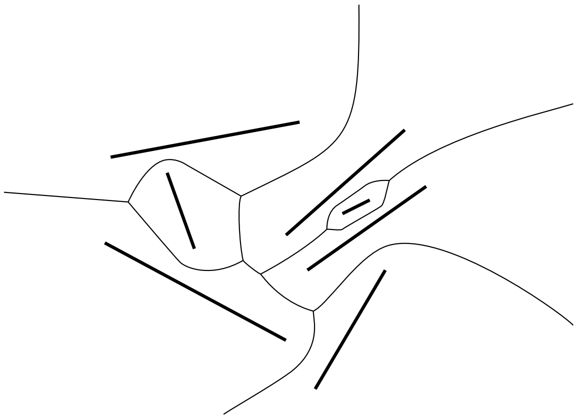

Figure 28: *An Euclidean Voronoi diagram of line segments.*

d-space is only $O(n)$, provided S is drawn uniformly at random in the unit ball. This result indicates that high-dimensional Voronoi diagrams will be small in many practical situations.

In analogy to the 2-dimensional case, the Delaunay triangulation $\mathrm{DT}(S)$ in 3D is defined as the geometric dual of $V(S)$. It contains a tetrahedron for each vertex, a triangle for each edge, and an edge for each facet, of $V(S)$. Equivalently, $\mathrm{DT}(S)$ may be defined using the empty sphere property, by including a tetrahedron spanned by S as Delaunay iff its circumsphere is empty of sites in S. The circumcenters of these empty spheres are just the vertices of $V(S)$. $\mathrm{DT}(S)$ is a partition of the convex hull of S into tetrahedra, provided S is in general position. Note that the edges of $\mathrm{DT}(S)$ may form the complete graph on S.

Simple incremental construction: Among the various proposed methods for constructing $V(S)$ in 3D, *incremental insertion* of sites (compare Section 3.2) is most intuitive and easy to implement. Basically, two different techniques for integrating a new site p into $V(S)$ have been applied. The more obvious method first determines all facets of the region of p in the new diagram, $V(S \cup \{p\})$, and then deletes the parts of $V(S)$ interior to this region; see e.g. Watson [62], Field [23], and Tanemura et al. [59]. Inagaki et al. [33] describe a robust implementation of this method.

In the dual environment, this amounts to detecting and removing all tetrahedra of $\mathrm{DT}(S)$ whose circumspheres contain p, and then filling the 'hole' with empty-sphere tetrahedra with p as apex, to obtain $\mathrm{DT}(S \cup \{p\})$. An example of an edge flip in 3D is shown in Figure 29.

Joe [35], Rajan [55], and Edelsbrunner and Shah [20] follow a different and numerically more stable approach. Like in the planar case, after having added a site to the current Delaunay triangulation, certain flips changing the local tetrahedral structure are performed in order to achieve local "Delaunayhood". The existence of such a sequence of flips is less trivial, however. Joe [34] demonstrated that no flipping sequence might exist that turns an arbitrary tetrahedral triangulation for S into $\mathrm{DT}(S)$.

A complete algorithm with run time $O(n^2)$ can be found in Shah [20].

3.3.2. Other Types of Generalizations

We simply list some of the generalization schemes and show examples of some intuitive ones.

- Different metrics
  - L_1, a comparison of L_1 and L_2 is shown in 27
  - L_∞
  - Convex distance functions
- Different space
  - On trees and graphs
  - Higher dimensions
- Weights
- More general sites
  - Line segments, see 28.
  - Polygonal chains
- Farthest point Voronoi diagram
- K-th order Voronoi digram
- Colored objects

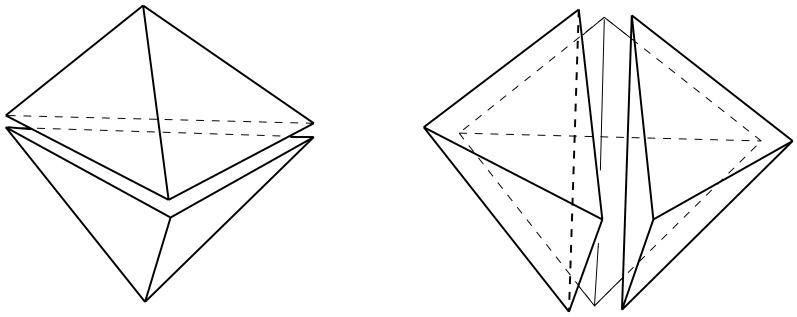

Figure 29: *Two–into–three tetrahedra flip for five sites.*

3.4. Applications of the Voronoi Diagram

3.4.1. Nearest Neighbor or Post Office Problem

We consider the famous post office problem. For a set S of sites in the plane and an arbitrary query point x we want to compute the point of S closest to x efficiently.

In the field of computational geometry there is a general technique for solving such query problems. One tries to decompose the query set into classes so that every class has the same answer. Now for a single answer we only have to determine its class. This technique is called *locus approach*.

The voronoi diagram represents the locus approach for the post office problem. The classes correspond to the regions of the sites. For a query point x we want to determine its class/region and return its owner.

To solve this task a simple technique can be applied. We draw a horizontal line through every vertex of the diagram and sort the lines in $O(n \log n)$ time, see Figure 30. The lines decompose the diagram into *slabs*. For every *slab* we sort the set of crossing edges of the Voronoi diagram in linear time. Altogether we need $O(n^2)$ time for the simple construction.

For a query point x we locate its *slab* in $O(\log n)$ time and afterwards its region in $O(\log n)$ time by binary search.

Theorem 8
Given a set S of n point sites in the plane, one can, within $O(n^2)$ time and storage, construct a data structure that supports nearest neighbor queries: for an arbitrary query point x, its nearest neighbor in S can be found in time $O(\log n)$.

The simple technique can be easily extended to 3D. There are also more efficient approaches, i.e., Edelsbrunner [19] constructs a $O(\log n)$ search structure for the Voronoi diagram in linear time and with linear space.

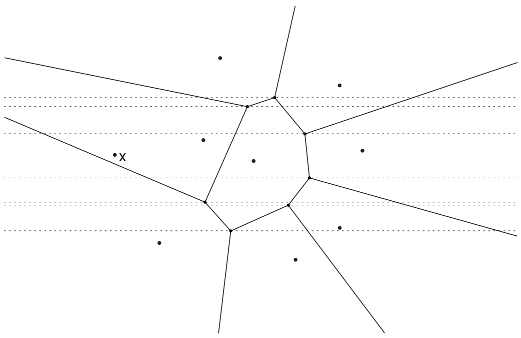

Figure 30: *After constructing the slabs, a query point x can be located quickly.*

3.4.2. Other Applications of the Voronoi Diagram in 2D

There are many different geometrical applications of the Voronoi diagram and its dual. Here we simply list some of them, together with some performance results, provided that the diagram is given:

- Closest Pair of sites, $O(n)$
- Nearest Neighbor Search
  - $O(n)$ for all nearest neighbors of the sites
  - $O(k \log^2 n)$ expected time for k-th nearest neighbors of query point x
- Minimum Spanning Tree and TSP-Heuristic, $O(n \log n)$
- Largest empty circle, $O(n)$
- Smallest enclosing circle (square with fixed orientation), $O(n)$
- Smallest color spanning circle (square with fixed orientation), $O(nk)$, where k is the number of colors
- Localization problems, see Hamacher[31]
- Clustering of objects, see Dehne and Noltemeier [14]

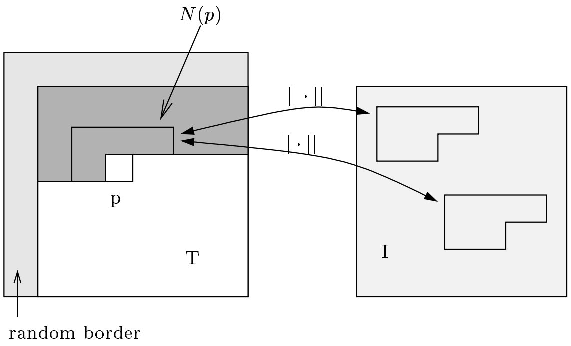

Figure 31: *The texture synthesis algorithm proceeds in scan line order through the texture and considers only the neighborhood around the current pixel as shown.*

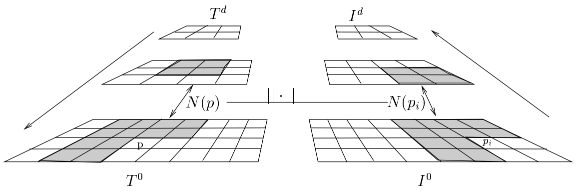

Figure 32: *Using an image pyramid, the texture synthesis process becomes fairly robust against different scales of detail in the sample images.*

All these results stem more or less from the linear complexity of the diagram. As we have already mentioned the complexity of the diagrams in three dimension is also linear in many practical situations. Thus many of the presented problems can be solved in three dimensions with almost the same time bound. We will present some special applications for 3D.

3.5. Texture Synthesis

Textures are visual detail of the rendered geometry, which have become very important in the past few years, because the cost of rendering with texture is the same as the cost without texture. Virtually all real-world objects have texture, so it is extremely important to render them in synthetic worlds, too.

Texture synthesis generally tries to synthesize new textures, either from given images, from a mathematical description, or from a physical model. Mathematical descriptions can be as simple as a number of sine waves to generate water ripples, while physical models try to describe the physical or biological effects and phenomena that lead to some texture (such as patina or fur). In all of these "model-based" methods, the knowledge about the texture is in the model and the algorithm. The other class of methods starts with one or more images; then they try to find some statistical or stochastic description (explicitely or implicitely) of these, and finally it generates a new texture from the statistic.

Basically, textures are images with the following properties:

1. Stationary: if a window with the proper size is moved about the image, the portion inside the window always appears the same.
2. Local: each pixel's color in the image depends only on a relatively small neighborhood.

Of course, images not satisfying these criteria can be used as textures as well (such as façades), but if you want to synthesize such images, then a statistical or stochastic approach is probably not feasible.

In the following, we will describe a stochastic algorithm that is very simple, very efficient, and works remarkably well [64]. Given a sample image, it does not, like most other methods, try to compute explicitly the stochastic model. Instead, it uses the sample image itself, which implicitly contains that model already.

We will use the following terminology:

$$\begin{aligned} I &= \text{Original (sample) image} \\ T &= \text{New texture image} \\ p_i &= \text{Pixel from } I \\ p &= \text{Pixel from } T \text{ to be generated next} \\ N(p) &= \text{Neighborhood of } p \text{ (see Figure 31)} \end{aligned}$$

Initially, T is cleared to black. The algorithm starts by adding a suitably sized border at the left and the top, filled with random pixels (this will be thrown away again at the end). Then, it performs the following simple loop in scan line order (see Figure 31):

for all $p \in T$ **do**
  find the $p_i \in I$ that minimizes $|N(p) - N(p_i)|^2$ {*}
  $p := p_i$
end for

Well, the search in the loop is exactly a nearest-neighbor search! This can be performed efficiently with the algorithm presented in Section 3.4.1: if $N(p)$ contains k pixels, then the points are just $3k$-dimensional vectors of RGB values, and the distance is just the Euclidean distance.

Figure 33: *Some results of the texture synthesis algorithm [64]. In each pair, the image on the left is the original one, the one on the right is the (partly) synthesized one.*

Obviously, all pixels of the new texture are deterministically defined, once the random border has been filled. The shape of the neighborhood $N(p)$ can be chosen arbitrarily, it must just be chosen such that all but the current pixel are already computed. Likewise, other "scans" of the texture are possible and sensible (for instance a spiral scan order), they must just match the shape of $N(p)$.

The quality of the texture depends on the size of the neighborhood $N(p)$. However, the optimal size itself depends on the "granularity" in the sample image. In order to make the algorithm independent, we can synthesize an image pyramid (see Figure 32). First, we generate a pyramid $I^0, I^1, \ldots, I^d$ for the sample image I^0. Then, we synthesize the texture pyramid $T^0, T^1, \ldots, T^d$ level by level with the above algorithm, starting at the coarsest level. The only difference is that we extend the neighborhood $N(p)$ of a pixel p over k levels as depicted by Figure 32. Consequently, we have to build a nearest-neighbor search structure for each level, because as we proceed downwards in the texture pyramid, the size of the neighborhood grows.

Of course, now we have replaced the parameter of the best size of the neighborhood by the parameter of the best size per level and the best number of levels to consider for the neighborhood. However, as [64] report, a neighborhood of 9×9 (at the finest level) across 2 levels seems to be sufficient in almost all cases.

Figure 33 shows two examples of the results that can be achieved with this method.

3.6. Shape Matching

As the availability of 3D models on the net and in databases increases, searching for such models becomes an interesting problem. Such a functionality is needed, for instance, in medical image databases, or CAD databases. One question is how to specify a query. Usually, most researchers pursue the "query by content" approach, where a query is specified by providing a (possibly crude) shape, for which the database is to return best matches. (This idea seems to originate from image database retrieval, where it was called QBIC = "query by image content".) The fundamental step here is the matching of shapes, i.e., the calculation of a *similarity* measure.

Almost all approaches perform the following steps:

1. Define a transformation function that takes a shape and computes a so-called *feature vector* in some high dimensional space, which (hopefully) captures the shape in its essence. Naturally, those transformation functions are preferred that are invariant under rotation and/or translation and tessellation.
2. Define a *similarity measure* d on the feature vectors, such that if $d(f_1, f_2)$ is large, then the associated shapes s_1, s_2 do not *look* similar. Obviously, this is (partly) a human factors issue. In almost all algorithms, d is just the Euclidean distance.
3. Compute a feature vector for each shape in the database and store them in a data structure that allows for fast nearest-neighbor search.
4. Given a query, i.e., a shape, compute its feature vector, and retrieve the nearest neighbor from the database. Usually, the system also retrieves all k nearest neighbors. Often times, you are not interested in the exact k nearest neighbors but only in *approximate* nearest neighbors (because the feature vector is an approximation of the shape anyway).

The main difference among most shape matching algorithms is, therefore, the transformation from shape to feature vector.

So, fast shape retrieval essentially requires a fast (approximate) nearest neighbor search. We could stop our discussion of shape matching here, but for sake of completeness, we will describe a very simple algorithm (from the plethora of others) to compute a feature vector [50].

The general idea is to define some *shape function* $f(P_1, \ldots, P_n) \to \mathbb{R}$, which computes some geometrical property of a number of points, and then evaluate this function for a large number of random points that lie on the surface of the shape. The resulting distribution of f is called a *shape distribution*.

For the shape function, there are a lot of possibilities (your imagination is the limit). Examples are:

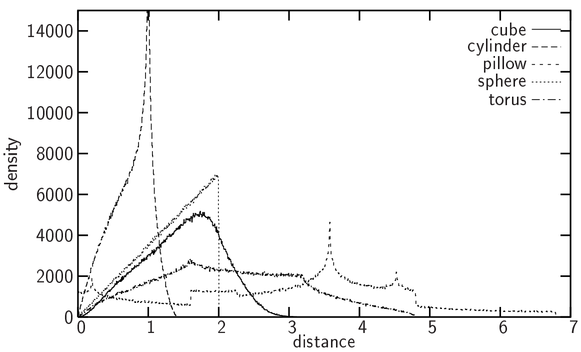

Figure 34: *The shape distribution of a number of different simple objects.*

- $f(P_1, P_2) = |P_1 - P_2|$;
- $f(P_1) = |P_1 - P_0|$, where P_0 is a fixed point, such as the bounding box center;
- $f(P_1, P_2, P_3) = \angle(\overline{P_1 P_2}, \overline{P_1 P_3})$;
- $f(P_1, P_2, P_3, P_4)$ = volume of the tetrahedron between the four points.

Figure 34 shows the shape distributions of a few simple objects with the distance between two points as shape function.

4. BSP Trees

BSP trees (short for *binary space partitioning trees*) can be viewed as a generalization of k-d trees. like k-d trees, BSP trees are binary trees, but now the orientation and position of a splitting plane can be chosen arbitrarily. To get a feeling for a BSP tree, Figure 35 shows an example for a set of objects.

The definition of a BSP (short for BSP tree) is fairly straight-forward. Here, we will present a recursive definition. Let h denote a plane in $\mathbb{R}^d$, h^+ and h^- denote the positive and negative half-space, resp.

Definition 1 (BSP tree)
Let S be a set of objects (points, polygons, groups of polygons, or other spatial objects), and let $S(\nu)$ denote the set of objects associated with a node ν. Then the BSP $T(S)$ is defined by

1. If $|S| \le 1$, then T is a leaf ν which stores $S(\nu) := S$.
2. If $|S| > 1$, then the root of T is a node ν; ν stores a plane h_ν and a set $S(\nu) := \{x \in S | x \subseteq h_\nu\}$ (this is the set of objects that lie completely inside h_ν; in 3D, these can only be polygons, edges, or points). ν also has two children T^- and T^+; T^- is the BSP for the set of objects $S^- := \{x \cap h_\nu^- | x \in S\}$, and T^+ is the BSP for the set of objects $S^+ := \{x \cap h_\nu^+ | x \in S\}$.

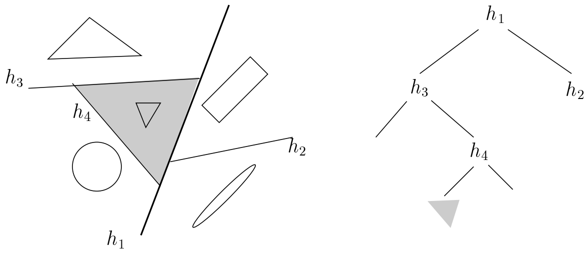

Figure 35: *An example BSP tree for a set of objects.*

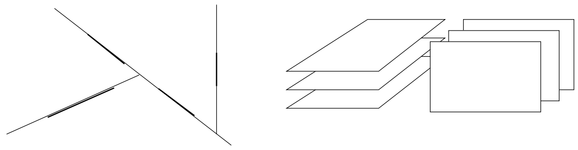

Figure 36: *Left: an auto-partition. Right: an example configuration of which any auto-partition must have quadratic size.*

This can readily be turned into a general algorithm for constructing BSPs. Note that a splitting step (i.e., the construction of an inner node) requires us to split each object into two disjoint *fragments* if it straddles the splitting plane of that node. In some applications though (such as ray shooting), this is not really necessary; instead, we can just put those objects into both subsets.

Note that with each node of the BSP a convex cell is associated (which is possibly unbounded): the "cell" associated with the root is the whole space, which is convex; splitting a convex region into two parts yields two convex regions. In Figure 35, the convex region of one of the leaves has been highlighted as an example.

With BSPs, we have much more freedom to place the splitting planes than with k-d trees. However, this also makes that decision much harder (as almost always in life). If our input is a set of polygons, then a very common approach is to choose one of the polygons from the input set and use this as the splitting plane. This is called an *auto-partition* (see Figure 36).

While an auto-partition can have $\Omega(n^2)$ fragments, it is possible to show the following in 2D [13, 53].

Lemma 1
Given a set S of n line segments in the plane, the expected number of fragments in an auto-partition $T(S)$ is in $O(n \log n)$; it can be constructed in time $O(n^2 \log n)$.

In higher dimensions, it is not possible to show a

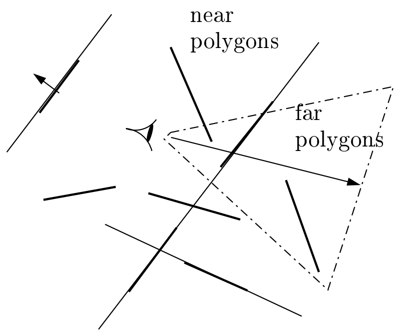

Figure 37: *BSP trees are an efficient data structure encoding visibility order of a set of polygons.*

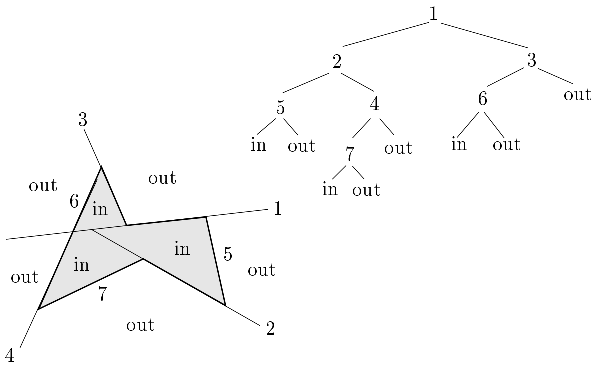

Figure 38: *Each leaf cell of BSP representation of an object is completely inside or completely outside.*

similar result. In fact, one can construct sets of polygons such that any BSP tree (not just auto-partitions) must have $\Omega(n^2)$ many fragments (see Figure 36 for a "bad" example for auto-partitions).

However, all of these examples producing quadratic BSPs violate the *principle of locality*: polygons are small compared to the extent of the whole set. In practice, no BSPs have been observed that exhibit the worst-case quadratic behavior[49].

4.1. Rendering Without a Z-Buffer

BSP trees were introduced to computer graphics by Fuchs et al.[25]. At the time, hidden-surface removal was still a major obstacle towards interactive computer graphics, because a z-buffer was just too costly in terms of memory.

In this section, we will describe how to solve this problem, not so much because the application itself is relevant today, but because it nicely exhibits one of the fundamental "features" of BSP trees: they enable efficient enumeration of all polygons in *visibility order* from any point in any direction. (Actually, the first version of Doom used exactly this algorithm to achieve its fantastic frame rate (at the time) on PCs even without any graphics accelerator.)

A simple algorithm to render a set of polygons with correct hidden-surface removal, and without a z-buffer, is the *painter's algorithm*: render the scene from back to front as seen from the current viewpoint. Front polygons will just overwrite the contents of the frame buffer, thus effectively hiding the polygons in the back. There are polygon configurations where this kind of sorting is not always possible, but we will deal with that later.

How can we efficiently obtain such a visibility order of all polygons? Using BSP trees, this is almost trivial: starting from the root, first traverse the branch that does *not* contain the viewpoint, then render the polygon stored with the node, then traverse the other branch containing the viewpoint (see Figure 37).

For sake of completeness, we would like to mention a few strategies to optimize this algorithm. First of all, we should make use of the viewing direction by skipping BSP branches that lie completely behind the viewpoint.

Furthermore, we can perform back-face culling as usual (which does not cause any extra costs). We can also perform view-frustum culling by testing all vertices of the frustum against the plane of a BSP node.

Another problem with the simple algorithm is that a pixel is potentially written to many times (this is exactly the *pixel complexity*), although only the last write "survives". To remedy this, we must traverse the BSP from front to back. But in order to actually save work, we also need to maintain a 2D BSP for the screen that allows us to quickly discard those parts of a polygon that fall onto a screen area that is already occupied. In that 2D screen BSP, we mark all cells either "free" or "occupied". Initially, it consists only of a "free" root node. When a new polygon is to be rendered, it is first run through the screen BSP, splitting it into smaller and smaller convex parts until it reaches the leaves. If a part reaches a leaf that is already occupied, nothing happens; if it reaches a free leaf, then it is inserted beneath that leaf, and this part is drawn on the screen.

4.2. Representing Objects with BSPs

BSPs offer a nice way to represent volumetric polygonal objects, which are objects consisting of polygons that are closed, i.e., they have an "inside" and an "outside". Such a BSP representation of an object is just like an ordinary BSP for the set of polygons (we can,

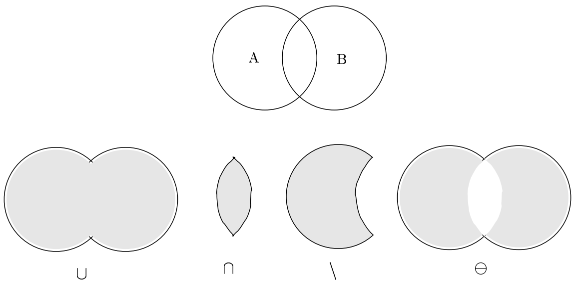

Figure 39: *Using BSPs, we can efficiently compute these boolean operations on solids.*

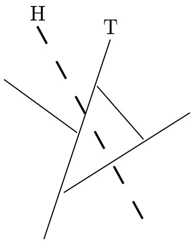

Figure 40: *The fundamental step of the construction is this simple operation, which merges a BSP and a plane.*

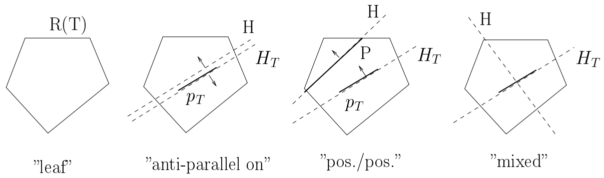

Figure 41: *The main building block of the algorithm consists of these four cases (plus analogous ones).*

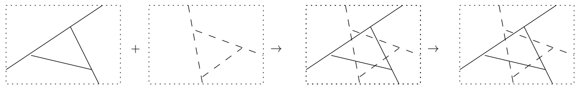

Figure 42: *Computation of boolean operations is based on a general merge operation.*

for instance, build an auto-partition), except that here we stop the construction process (see Definition 1) only when the set is empty. These leaves represent homogeneous convex cells of the space partition, i.e., they are completely "in" our "out".

Figure 38 shows an example for such a BSP representation. In this section, we will follow the convention that normals point to the "outside", and that the right child of a BSP node lies in the positive half-space and the left child in the negative half-space. So, in a real implementation that adheres to these conventions, we can still stop the construction when only one polygon is left, because we know that the left child of such a pseudo-leaf will be "in" and the right one will be "out".

Given such a representation, it is very easy and efficient, for instance, to determine whether or not a given a point is inside an object. In the next section, we will describe an algorithm for solving a slightly more difficult problem.

4.3. Boolean Operations

In solid modeling, a very frequent task is to compute the intersection or union of a pair of objects. More generally, given two objects A and B, we want to compute $C := A$ op B, where op $\in \{\cup, \cap, \setminus, \ominus\}$ (see Figure 39). This can be computed efficiently using the BSP representation of objects [48, 49]. Furthermore, the algorithm is almost the same for all of these operations: only the elementary step that processes two leaves of the BSPs is different.

We will present the algorithm for boolean operations bottom-up in three steps. The first step is a sub-procedure for computing the following simple operation: given a BSP T and a plane H, construct a new BSP $\hat{T}$ whose root is H, such that $\hat{T}^- \triangleq T \cap H^-$, $\hat{T}^+ \triangleq T \cap H^+$ (see Figure 40). This basically splits a BSP tree by a plane and then puts that plane at the root of the two halves. Since we will not need the new tree $\hat{T}$ explicitly, we will describe only the splitting procedure (which is the bulk of the work anyway).

First, we need to define some nomenclature:

$T^-, T^+ =$ left and rigt child of T, resp.

$R(T) =$ region of the cell of node T (which is convex)

$T^\oplus, T^\ominus =$portion of T on the positive/negative side of H, resp.

Finally, we would like to define a node T by the tuple (H_T, p_T, T^-, T^+), where H is the splitting plane, p is the polygon associated with T (with $p \subset H$).

The pseudo-code below is organized into 8 cases (see Figure 41):

<u>**split-tree(** T, H, P **)** $\to (T^\ominus, T^\oplus)$</u>
$\{P = H \cap R(T)\}$
case T is a leaf :
  return $(T^\ominus, T^\oplus) := (T, T)$
case "anti-parallel" and "on" :
  return $(T^\ominus, T^\oplus) := (T^+, T^-)$
case "parallel" and "on" :
  ...
case "pos./pos." :
  $(T^{+\ominus}, T^{+\oplus}) :=$ split-tree(T^+, H)

$T^{\ominus} := (H_T, p_T, T^-, T^{+\ominus})$
$T^{\oplus} := T^{+\oplus}$
case "pos./neg." :
...
case "neg./pos." :
...
case "neg./neg." :
...
case "mixed" :
$(T^{+\ominus}, T^{+\oplus}) :=$ split-tree$(T^+, H, P \cap R(T^+))$
$(T^{-\ominus}, T^{-\oplus}) :=$ split-tree$(T^-, H, P \cap R(T^-))$
$T^{\ominus} := (H_T, p_T \cap H^-, T^{-\ominus}, T^{+\ominus})$
$T^{\oplus} := (H_T, p_T \cap H^+, T^{-\oplus}, T^{+\oplus})$
return $(T^{\ominus}, T^{\oplus})$
end case

This might look a little bit confusing at first sight, but it is really pretty simple. A few notes might be in order.

The polygon P is only needed in order to find the case applying at each recursion. Computing $P \cap R(T^+)$ might seem very expensive. However, it can be computed quite efficiently by computing $P \cap H_T^+$, which basically amounts to finding the two edges that intersect with H_T. Please see [12] for more details on how to detect the correct case.

It seems surprising at first sight that function `split-tree` does almost no work — it just traverses the BSP tree, classifies the case found at each recursion, and computes $p \cap H^+$ and $p \cap H^-$.

The previous algorithm is already the main building block of the overall boolean operation algorithm. The next step towards that end is an algorithm that performs a so-called *merge* operation on two BSP trees T_1 and T_2. Let $\mathcal{C}_i$ denote the set of elementary cells of a BSP, i.e., all regions $R(L_j)$ of tree T_i where L_j are all the leaves. Then the merge of T_1, T_2 yields a new BSP tree T_3 such that $\mathcal{C}_3 = \{c_1 \cap c_2 | c_1 \in \mathcal{C}_1, c_2 \in \mathcal{C}_2, c_1 \cap c_2 \neq \varnothing\}$ (see Figure 42).

This merge operation is performed by the following pseudo-code:

<u>merge(T_1, T_2) $\to T_3$</u>
if T_1 or T_2 is a leaf **then**
perform the **cell-op** as required by the boolean operation to be constructed (see below)
else
$(T_2^{\ominus}, T_2^{\oplus}) :=$ split-tree$(T_2, H_1, \ldots)$
$T_3^- :=$ merge$(T_1^-, T_2^{\ominus})$
$T_3^+ :=$ merge$(T_1^+, T_2^{\oplus})$
$T_3 := (H_1, T_3^-, T_3^+)$
end if

The function `cell-op` is the only place where the semantic of the general merge operation is specialized. When we have reached that point, then we know that one of the two cells is homogeneous, so we can just replace it by the other node's sub-tree suitably modified according to the boolean operation. The following table lists the details of this function (assuming that T_1 is the leaf):

Operation	T_1	Result
$\cup$	in	T_1
	out	T_2
$\cap$	in	T_2
	out	T_1
$\setminus$	in	T_2^c
	out	T_1
$\ominus$	in	T_2^c
	out	T_2

Furthermore, we would like to point out that the `merge` function is symmetric: it does not matter whether we partition T_2 with H_1 or, the other way round, T_1 with H_2 — the result will be the same.

5. Bounding Volume Hierarchies

Like the previous hierarchical data structures, bounding volume (BV) hierarchies are mostly used to prevent performing an operation exhaustively on all objects. Often times, bounding volume (BV) hierarchies are described as the opposite of spatial partitioning schemes, such as quadtrees or BSP trees: instead of partitioning space, the idea is to partition the set of objects recursively until some leaf criterion is met. (However, we will argue at the end that BV hierarchies are just at the other end of a whole spectrum of hierarchical data structures.) Here, objects can be anything from points to complete graphical objects. With BV hierarchies, almost all queries, which can be implemented with space partitioning schemes, can also be answered, too. Example queries and operations are ray shooting, frustum culling, occlusion culling, point location, nearest neighbor, collision detection.

Definition 2 (BV hierarchy)
Let $O = \{o_1, \ldots, o_n\}$ be a set of elementary objects. A bounding volume hierarchy for O, BVH(O), is defined by

1. If $|O| = e$, then BVH(O) := a leaf node that stores O and a BV of O;
2. If $|O| > e$, then BVH(O) := a node ν with $n(\nu)$ children $\nu_1, \ldots, \nu_n$, where each child ν_i is a BV hierarchy BVH(O_i) over a subset $O_i \subset O$, such that $\bigcup O_i = O$. In addition, ν stores a BV of O.

The definition mentions two parameters. The threshold e is often set to 1, but depending on the application, the optimal e can be much larger. Just

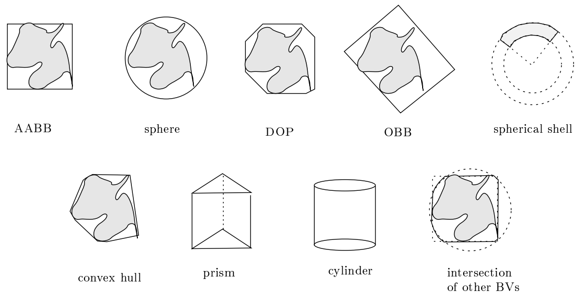

Figure 43: *Some of the most commonly used BVs, and some less often used ones.*

like sorting, when the set of objects is small, it is often cheaper to perform the operation on all of them, because recursive algorithms always incur some overhead.

Another parameter in the definition is the arity. Mostly, BV hierarchies are constructed as binary trees, but again, the optimum can be larger. And what is more, as the definition suggests, the out-degree of nodes in a BV hierarchy does not necessarily have to be constant, although this often simplifies implementations considerably.

Effectively, these two parameters, e and $n(\nu)$, control the balance between linear, exhaustive search/operation, and a maximally recursive algorithm.

There are more design choices possible according to the definition. For inner nodes, it only requires that $\bigcup O_i = O$; this means, that the same object $o \in O$ could be associated with several children. Depending on the application, the type of BVs, and the construction process, this may not always be avoidable. But if possible, you should always split the set of objects into disjoint subsets.

Finally, there is, at least, one more design choice: the type of BV used at each node. Again, this does not necessarily mean that each node uses the same type of BV. Figure 43 shows a number of the most commonly used BVs. The difference between OBBs [4] and AABBs is that OBBs can be oriented arbitrarily (hence "oriented bounding boxes"). DOPs [67, 39, 37] are a generalization of AABBs: basically, they are the intersection of k slabs. Prisms and cylinders have been proposed by [7, 63], but they seem to be too expensive computationally. A spherical shell is the intersection of a shell and a cone (the cone's apex coincides with the sphere's center), and a shell is the space between two concentric spheres. Finally, one can always take the intersection of two or more different types of BVs [36].

There are three characteristic properties of BVs:

- tightness,
- memory usage,
- number of operations needed to test the query object against a BV.

Often, one has to make a trade-off between these properties: generally, the type of BV that offers better tightness also requires more operations per query and more memory.

Regarding the tightness, one can establish a theoretical advantage of OBBs. But first, we need to define tightness [29].

Definition 3 (Tightness by Hausdorff distance)
Let B be a BV, G some geometry bounded by B, i.e., $g \subset B$. Let

$$h(B,G) = \max_{b\in B} \min_{g\in G} d(b,g)$$

be the *directed Hausdorff distance*, i.e., the maximum distance of B to the nearest point in G. (Here, d is any metric, very often just the Euclidean distance.) Let

$$\operatorname{diam}(G) = \max_{g,f\in G} d(g,f)$$

be the *diameter* of G.

Then we can define *tightness*

$$\tau := \frac{h(B,G)}{\operatorname{diam}(G)}.$$

See Figure 44 for an illustration.

Since the Hausdorff distance is very sensitive to outliers, one could also think of other definitions such as the following one:

Definition 4 (Tightness by volume)
Let $C(\nu)$ b the set of children of a node ν of the BV hierarchy. Let $\operatorname{Vol}(\nu)$ be the volume of the BV stored with ν.

Then, we can define the tightness as

$$\tau := \frac{\operatorname{Vol}(\nu)}{\sum_{\nu' \in C(\nu)} \operatorname{Vol}(\nu')}.$$

Alternatively, we can define it as

$$\tau := \frac{\operatorname{Vol}(\nu)}{\sum_{\nu' \in L(\nu)} \operatorname{Vol}(\nu')},$$

where $L(\nu)$ is the set of leaves beneath ν.

Getting back to the tightness definition based on the Hausdorff distance, we observe a fundamental difference between AABBs and OBBs [29]:

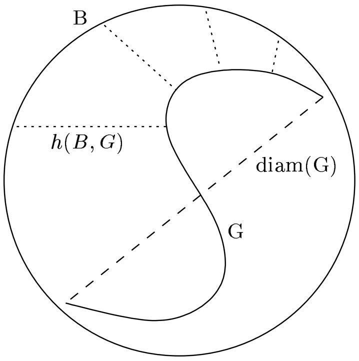

Figure 44: *One way to define tightness is via the directed Hausdorff distance.*

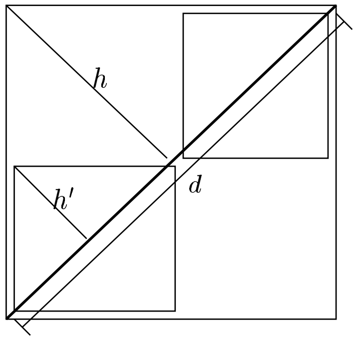

Figure 45: *The tightness of an AABB remains more or less constant throughout the levels of a AABB hierarchy for surfaces of small curvature.*

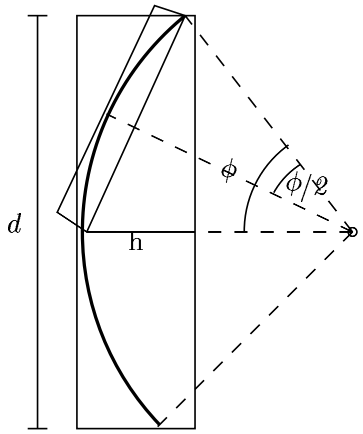

Figure 46: *The tightness of an OBB decreases for deeper levels in a OBB hierarchy for small curvature surfaces.*

- The tightness of AABBs depends on the orientation of the enclosed geometry. What is worse is that the tightness of the children of an AABB enclosing a surface of small curvature is almost the same as that of the father.
  The worst case is depicted in Figure 45. The tightness of the father is $\tau = {}^{h}/d$, while the tightness of a child is $\tau' = \frac{h'}{d/2} = \frac{h/2}{d/2} = \tau$.
- The tightness of OBBs does not depend on the orientation of the enclosed geometry. Instead, it depends on its curvature, and it decreases approximately linearly with the depth in the hierarchy.
  Figure 46 depicts the situation for a sphere. The Hausdorff distance from an OBB to an enclosed spherical arc is $h = r(1 - \cos\phi)$, while the diameter of the arc is $d = 2r\sin\phi$. Thus, the tightness for an OBB bounding a spherical arc of degree ϕ is $\tau = \frac{1-\cos\phi}{2\sin\phi}$, which approaches 0 linearly as $\phi \to 0$.

This makes OBBs seem much more attractive than AABBs. The price of the much improved tightness is, of course, the higher computational effort needed for most queries per node when traversing an OBB tree with a query.

5.1. Construction of BV Hierarchies

Essentially, there are 3 strategies to build BV trees:

- bottom-up,
- top-down,
- insertion

From a theoretical point of view, one could pursue a simple top-down strategy, which just splits the set of objects into two equally sized parts, where the objects are assigned randomly to either subset. Asymptotically, this yields usually the same query time as any other strategy. However, in practice, the query times offered by such a BV hierarchy are by a large factor worse.

During construction of a BV hierarchy, it is convenient to forget about the graphical objects or primitives, and instead deal with their BVs and consider those as the atoms. Sometimes, another simplification is to just approximate each object by its center (baryenter or bounding box center), and then deal only with sets of points during the construction. Of course, when the BVs are finally computed for the nodes, then the true extents of the objects must be considered.

In the following we will describe algorithms for each construction strategy.

5.1.1. Bottom-up

In this class, we will actually describe two algorithms.

Let B be the set of BVs on the top-most level of the BV hierarchy that has been constructed so far [57]. For each $b_i \in B$ find the nearest neighbor $b'_i \in B$; let d_i be the distance between b_i and b'_i. Sort B with respect to d_i. Then, combine the first k nodes in B under a common father; do the same with the next k elements from B, etc. This yields a new set B', and the process is repeated.

Note that this strategy does not necessarily produce BVs with a small "dead space": in Figure 47, the strategy would choose to combine the left pair (distance =

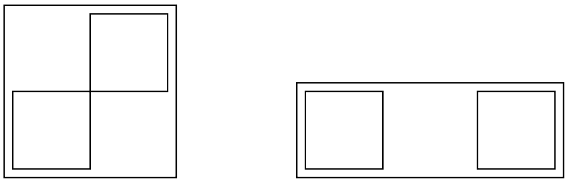

Figure 47: *A simple greedy strategy can produce much "dead space".*

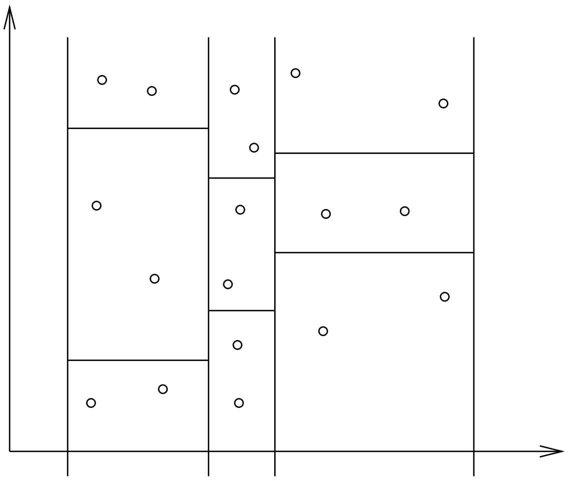

Figure 48: *A less greedy strategy combines BVs by computing a "tiling".*

0), while choosing the right pair would result in much less dead space.

The second strategy [42] is less greedy in that it computes a tiling for each level. We will describe it first in 2D. Again, let B be the set of BVs on the top-most level so far constructed, with $|B| = n$. The algorithm first computes the center c^i for each $b^i \in B$. Then, it sorts B along the x-axis with respect to c^i_x. Now, the set B is split into $\sqrt{n/k}$ vertical "slices" (again with respect to c^i_x). Now, each slice is sorted according to c^i_y and subsequently split into $\sqrt{n/k}$ "tiles", so that we end up with k tiles (see Figure 48). Finally, all nodes in a tile are combined under one common father, its BV is combined, and the process repeats with a new set B'.

In $\mathbb{R}^d$ it works quite similarly: we just split each slice repeatedly by $\sqrt[d]{n/k}$ along all coordinate axes.

5.1.2. Insertion

This construction scheme starts with an empty tree. Let B be the set of elementary BVs. The following pseudo-code describes the general procedure:

```
1: while |B| > 0 do
2:    choose next b ∈ B
3:    ν := root
4:    while ν ≠ leaf do
5:       choose child ν',
            so that insertion of b into ν' causes minimal
         increase in the costs of the total tree
6:       ν := ν'
7:    end while
8: end while
```

All insertion algorithms only vary step 2 and/or 5. Step 2 is important because a "bad" choice in the beginning can probably never be made right afterwards. Step 5 depends on the type of query that is to be performed on the BV tree. See below for a few criteria.

Usually, algorithms in this class have complexity $O(n \log n)$.

5.1.3. Top-down

This scheme is the most popular one. It seems to produce very good hierarchies while still being very efficient, and usually it can be implemented easily.

The general idea is to start with the complete set of elementary BVs, split that into k parts, and create a BV tree for each part recursively. The splitting is guided by some heuristic or criterion that (hopefully) produces good hierarchies.

5.1.4. Criteria

In the literature, there is a vast number of criteria for guiding the splitting, insertion, or merging, during BV tree construction. (Often, the authors endow the thus constructed BV hierarchy with a new name, even though the BVs utilized are well known.) Obviously, the criterion depends on the application for which the BV tree is to be used. In the following, we will present a few of these criteria.

For ray tracing, if we can estimate the probability that a ray will hit a child box when it has hit the father box, then we know how likely it is, that we need to visit the child node when we have visited the father node. Let us assume that all rays emanate from the same

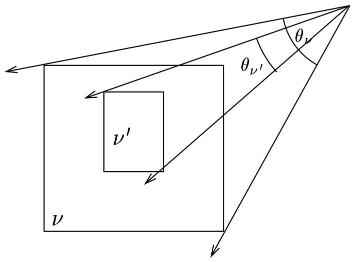

Figure 49: *The probability of a ray hitting a child box can be extimated by the surface area.*

origin (see Figure 49). Then, we can observe that the probability that a ray s hits a child box ν' under the condition that it has hit the father box ν is

$$P(s \text{ hits } \nu' | s \text{ hits } \nu) = \frac{\theta_{\nu'}}{\theta_\nu} \approx \frac{\text{Area}(\nu')}{\text{Area}(\nu)} \qquad (4)$$

where Area denotes the surface area of the BV, and θ denotes the solid angle subtended by the BV. This is because for a convex object, the solid angle subtended by it, when seen from large distances, is approximately proportional to its surface area [28]. So, a simple strategy is to just minimize the surface area of the BVs of the children that are produced by a split. (For the insertion scheme, the strategy is to choose that child node whose area is increased least [28].)

A more elaborate criterion tries to establish a *cost function* for a split and minimize that. For ray tracing, this cost function can be approximated by

$$C(\nu_1, \nu_2) = \frac{\text{Area}(\nu_1)}{\text{Area}(\nu)} C(\nu_1) + \frac{\text{Area}(\nu_2)}{\text{Area}(\nu)} C(\nu_2) \qquad (5)$$

where ν_1, ν_2 are the children of ν. The optimal split $B = B_1 \cup B_2$ minimizes this cost function:

$$C(B_1, B_2) = \min_{B' \in \mathcal{P}(B)} C(B', B \setminus B')$$

where B_1, B_2 are the subsets of elementary BVs (or objects) assigned to the children. Here, we have assumed a binary tree, but this can be extended to other arities analogously.

Of course, such a minimization is too expensive in practice, in particular, because of the recursive definition of the cost function. So, Fussell and Subramanian[26], Müller et al.[47], and Beckmann et al.[8] have proposed the following approximation algorithm:

for $\alpha = x, y, z$ **do**
  sort B along axis α with respect to the BV centers
  find

$$k^\alpha = \min_{j=0\ldots n} \left\{ \frac{\text{Area}(b_1, \ldots, b_j)}{\text{Area}(B)} j + \frac{\text{Area}(b_{j+1}, \ldots, b_n)}{\text{Area}(B)} (n - j) \right\}$$

end for
choose the best k^α

where $\text{Area}(b_1, \ldots, b_j)$ denotes the surface area of the BV enclosing $b_1, \ldots, b_j$.

If the query is a point location query (e.g., is a given point inside or outside the object), then the volume instead of the surface area should be used. This is because the probability that a point is contained in a child BV, under the condition that it is contained in the father BV, is proportional to the ratio of the two volumes.

For range queries, and for collision detection, the volume seems to be a good probability estimation, too.

A quite different splitting algorithm does not (explicitely) try to estimate any probabilities. It just approximates each elementary BV/object by its center point. It then proceeds as follows. For a given set B of such points, compute its principal components (the Eigenvectors of the covariance matrix); choose the largest of them (i.e., the one exhibiting the largest variance); place a plane orthogonal to that principal axis and through the barycenter of all points in B; finally, split B into two subsets according to the side on which the point lies. (This description is a slightly modified version of Gottschalk et al. [29].) Alternatively, one can place the splitting plane through the median of all points, instead of the barycenter. This would lead to balanced trees, but not necessarily better ones.

5.2. Collision Detection

Fast and exact collision detection of polygonal objects undergoing rigid motions is at the core of many simulation algorithms in computer graphics. In particular, all kinds of highly interactive applications such as virtual prototyping need exact collision detection at interactive speed for very complex, arbitrary "polygon soups". It is a fundamental problem of dynamic simulation of rigid bodies, simulation of natural interaction with objects, and haptic rendering.

Bounding volume trees seem to be a very efficient data structure to tackle the problem of collision detection for rigid bodies. All kinds of different types of BVs have been explored in the past: sphere trees [32, 52], OBB trees [29], DOP trees [39, 67], AABB trees [66, 60, 40], and convex hull hierarchies [21], to name but a few.

Given two hierarchical BV volume data structures

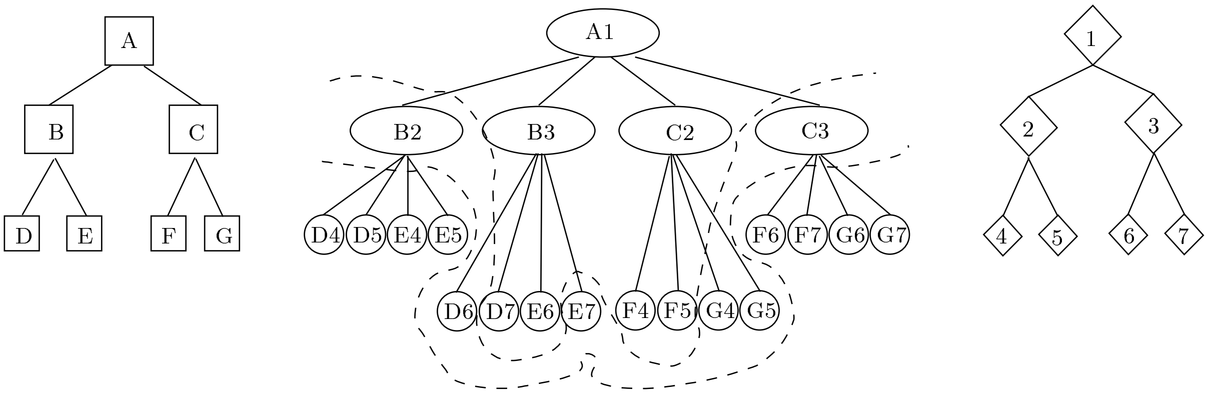

Figure 50: *The recursion tree is induced by the simultaneous traversal of two BV trees.*

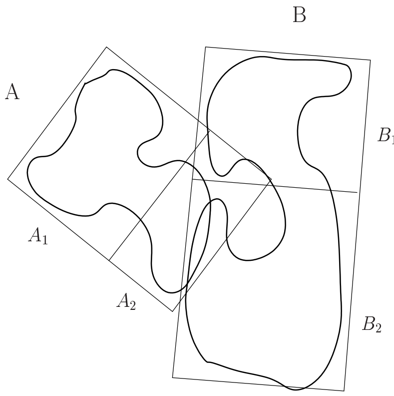

Figure 51: *Hierarchical collision detection can discard many pairs of polygons with one BV check. Here, all pairs of polygons in A_1 and B_2 can be discarded.*

for two objects A and B, almost all hierarchical collision detection algorithms implement the following general algorithm scheme:

```
traverse(A,B)
if A and B do not overlap then
  return
end if
if A and B are leaves then
  return intersection of primitives
         enclosed by A and B
else
  for all children A[i] and B[j] do
    traverse(A[i],B[j])
  end for
end if
```

This algorithm quickly "zooms in" on pairs of close polygons. The characteristics of different hierarchical collision detection algorithms lie in the type of BV used, the overlap test for a pair of nodes, and the algorithm for construction of the BV trees.

The algorithm outlined above is essentially a simultaneous traversal of two hierarchies, which induces a so-called *recursion tree* (see Figure 50). Each node in this tree denotes a BV overlap test. Leaves in the recursion tree denote an intersection test of the enclosed primitives (polygons); whether or not a BV test is done at the leaves depends on how expensive it is, compared to the intersection test of primitives.

During collision detection, the simultaneous traversal will stop at some nodes in the recursion tree. Let us call the set of nodes, of which some children are not visited (because their BVs do not overlap), the "bottom slice" through the recursion tree (see the dashed lines in Figure 50).

One idea is to save this set for a given pair of objects [43]. When this pair is to be checked next time, we can start from this set, going either up or down. Hopefully, if the objects have moved only a little relative to each other, the number of nodes that need to be added or removed from the bottom slice is small. This scheme is called *incremental hierarchical collision detection*.

6. Dynamization of Geometric Data Structures

We present a generic approach for the dynamization of an *arbitrary* static geometric data structure. Often a simple *static* data structure is sufficient if the set of represented geometric objects will have few changes over time. Once created, the static structure mostly has to cope with data queries due to its geometric intention. If the set of objects varies very much over time, there is need for more complex dynamic structures which allow efficient *insertion and deletion* of objects.

For example a one dimensional sorted array of a fixed set M is sufficient for x *is Element of* M queries. But if the set M has many changes over time, a dynamic AVL-tree would be more likely. The AVL-tree implementation is more complex since rotations of the tree has to be considered for insertion and deletion of objects. Additionally, the AVL-tree dynamization was *invented* for the special case of the one-dimensional search. We want to show that it is possible to dynamize a simple static data structure indirectly but also efficiently in a general setting. Once this generic approach is implemented it can be used for many static data structures.

The generic approaches presented here are not optimal against a dynamization adapted directly to a single data structure, but they are easy to implement and efficient for many applications.

In Section 6.1 we formalize the given problem and define some requirements. In Section 6.2 we present methods allowing insertion and deletion in amortized efficient time. For many applications this is already efficient enough. Within this section the dynamization technique is explained in detail and the amortized cost of the new operations are shown. Similar ideas for the worst-case sensitive approach are sketched in Section 6.3. The *effort* of the dynamization itself is amortized over time. For details see Klein [38] or the work of Overmars [51] and van Kreveld [61]. We present a simple example in Section 6.4.

6.1. Model of the Dynamization

Let us assume that *TStat* is a static abstract (geometric) data type. Since we have a geometric data structure, we assume that the essential motivation of *TStat* is a query operation on the set of stored objects D. i.e., for a query object q the answer is always a subset of D which might be empty.

We want to define the generic dynamization by a module that imports the following operations from *TStat*:

$build(V, D)$:	Build the structure V of type *TStat* with all data objects in the set D.
$query(V, q)$:	Gives the answer (objects of D) to a query to V with query object q.
$extract(V, D)$:	Collects all data objects D of V in a single set an returns a pointer to this set.
$erase(V)$:	Delete the complete data structure V from the storage.

The dynamization module should export a dynamic abstract (geometric) data type *TDyn* with the following operations:

$Build(W, D)$:	Build the structure W of type *TDyn* with data objects in the set D.
$Query(W, q)$:	Gives the answer (objects of D) to a query to W with query object q.
$Extract(W, D)$:	Collects all data objects D of W in a single set and returns a pointer this set.
$Erase(W)$:	Delete the complete data structure W from the storage.
$Insert(W, d)$:	Insert object d into W.
$Delete(W, d)$:	Delete d out of W.

Note, that the *new* operations *Delete* and *Insert* are necessary since we have a dynamic data type now.

Additionally, we introduce some cost functions for the operations of the abstract dynamic and the abstract static data type. For example $B_V(n)$ denotes the time function for the operation $Build(V, D)$ of *TStat*. The notations are fully presented in Figure 52. The cost functions depend on the implementation of *TStat*. Note, that the cost function of *TDyn* will depend on the cost functions of *TStat* together with the efficiency of the our general dynamization.

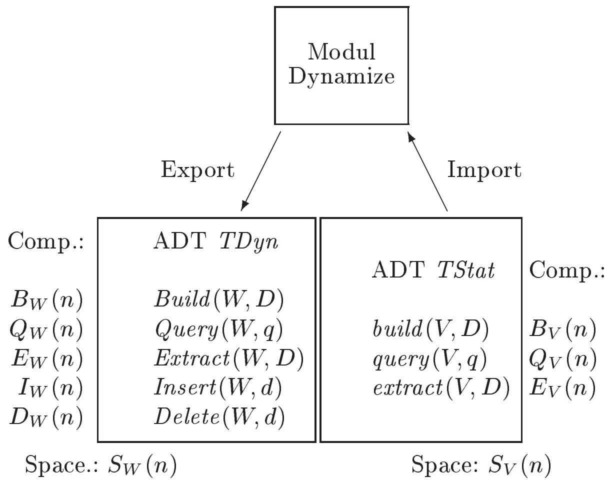

Figure 52: *Dynamization in the generic sense.*

In order to guarantee some bounds for the corresponding cost functions of *TDyn* the cost functions of *TStat* must not increase arbitrarily. On the other hand for the proof of some time bounds we need some kind of monotonic behavior in the functions, they should not oscillate. Altogether we define the following requirements which are fulfilled in many cases:

1. $Q_V(n)$ and $E_V(n)$ increase monotonically in n; examples: 1, $\log n$, $\sqrt{n}$, n, $n \log n$, n^2, 2^n.
2. $\frac{B_V(n)}{n}$ and $\frac{S_V(n)}{n}$ increase monotonically in n; examples n, $n \log n$, n^2, 2^n.
3. For all $f \in \{Q_V, B_V, E_V, S_V\}$ there is a constant $C \geq 1$, so that $f(2n) \leq C\ f(n)$; examples: 1, $\sqrt{n}$, n, n^2, and also $\log n$ with $n > 1$, as well as the products of this functions, but not 2^n.
4. $E_V(n) \leq 2 \cdot B_V(n)$.

Moreover, we assume that the that the *query* operation can be decomposed, i.e., for a decomposition $V = V_1 \cup V_2 \cup \cdots \cup V_j$ of the data set V the results of the single operations $query(V_i, d)$ lead to the solution of $query(V, d)$. This is true for many kinds of range queries.

6.2. Amortized Insert and Delete

6.2.1. Amortized Insert: Binary Structure

The very first idea is to implement $Insert(W, d)$ and $Delete(W, d)$ directly by

$$\begin{aligned} &\mathit{Insert}(W,d): && \mathit{Extract}(W,D);\ \mathit{Build}(W, D \cup \{d\}) \\ &\mathit{Delete}(W,d): && \mathit{Extract}(W,D);\ \mathit{Build}(W, D \setminus \{d\}). \end{aligned}$$

This *throw-away* implementation is not very efficient. Therefore we distribute the n data objects of the static structure V among several structures V_i. If a new element has to be inserted we hope that only a single structure V_i may be concerned. Let

$$n = a_l 2^l + a_{l-1} 2^{l-1} + \ldots + a_1 2 + a_0 \text{ mit } a_i \in \{0, 1\}.$$

Then $a_l a_{l-1} \ldots a_1 a_0$ is the *binary representation* of n. For every $a_i = 1$ we build a structure V_i which has 2^i elements. The collection of these structures is a representation of W_n which is called *binary structure*, see Figure 53. To build up the binary structure W_n we proceed as follows:

Build(W, D): Compute *binary representation* of $n = |D|$.
Decompose D into sets D_i with $|D_i| = 2^i$ w.r.t. the representation of n.
Compute *build*(V_i, D_i) for every D_i.

In principle, the binary structure W_n can be constructed as quick as the corresponding structure V.

Lemma 9

$$B_W(n) \in O(B_V(n)).$$

Proof Computing the binary representation of n and the decomposition into D_i can be done in linear time $O(n)$.
The operation *build*(V_i, D_i) needs $B_V(2^i)$ time. We have $i \leq l = \lfloor \log n \rfloor$ and therefore we conclude:

$$\begin{aligned} \sum_{i=0}^{\lfloor \log n \rfloor} B_V(2^i) &= \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \frac{B_V(2^i)}{2^i} \\ &\leq \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \frac{B_V(n)}{n} \\ &\leq 2^{\log n} \frac{B_V(n)}{n} \\ &\in O(B_V(n)). \end{aligned}$$

We used the fact that $\frac{B_V(n)}{n}$ increases monotonically. Altogether we have

$$B_W(n) \in O(n + B_V(n)) = O(B_V(n))$$

since $B_V(n)$ is at least linear. □

Similar results hold for some other operations. We can prove

$$E_W(n) \leq \log n\ E_V(n)$$

since we have to collect the results of $extraxt(V_i)$ for at most $\log n$ structures V_i.

Additionally,

$$Q_W(n) \leq \log n\ Q_v(n)$$

holds if we assume that the query can be decompose as well, see the requirements.

It is also easy to see that

$$S_W(n) \leq \sum_{i=0}^{\lfloor \log n \rfloor} S_V(2^i) \in O(S_V(n)).$$

Therefore it remains to analyse $I_W(n)$.

As we have seen from Figure 53 sometimes the whole structure of W_n is destroyed when W_{n+1} was build up. In this example we had to perform the following tasks:

$extract(V_0, D_0); extract(V_1, D_1); extract(V_2, D_2);$
$D := D_0 \cup D_1 \cup D_2 \cup \{d\};$
$build(V_3, D);$

In general we have to build up V_j and extract and erase V_{j-1}, V_{j-2}, …, V_0 only if $a_i = 1$ holds for $i = 0, 1 \ldots, j-1$ and $a_j = 0$ holds (in the binary representation of the current n).

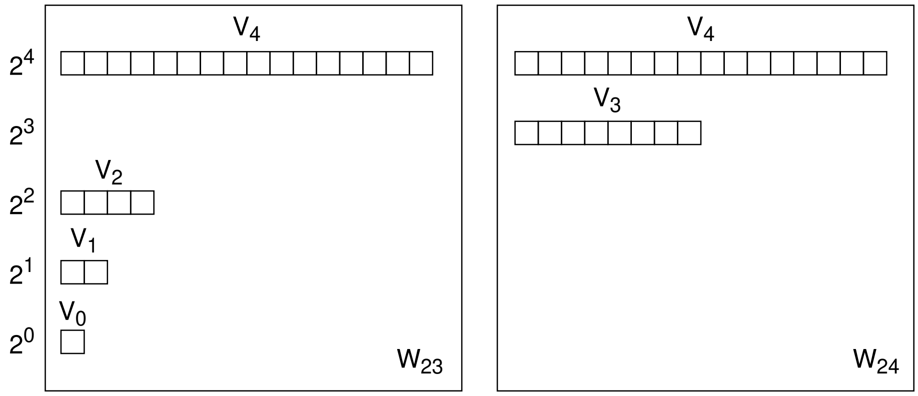

Figure 53: *The binary structure W_n contains the structure V_i if $a_i = 1$ holds for the binary representation of n. For examples see $n = 23$ (left) and $n = 24$ (right).*

In this special case we have

$$\begin{aligned} I_W(n) &\leq \left(\sum_{i=0}^{j-1} E_V(2^i)\right) + Cj + B_V(2^j) \\ &\leq E_V(2^j) + Cj + B_V(2^j) \\ &\in O\left(B_V(2^j)\right). \end{aligned}$$

For a long sequence of insertions many of them are performed without extreme reconstructions. Thus the effort for all $Insert(W, d)$ is amortized over time.

Generally, let *Work* be an arbitrary operation with cost function W. For a sequence of s different operations let *Work* be applied k times. If

$$\frac{\text{total cost of } k \ \textit{Work} \text{ operationen}}{k} \leq \overline{W}(s)$$

holds for a monotonically increasing cost function $\overline{W}$, we say that the operation *Work* is performed in *amortized time* $\overline{W}(s)$.

Note, that this is not an expected value and that $\overline{W}$ is a function of s, i.e., the length of the operation sequence. The current data set may have a number of elements $n \leq s$.

For the $Insert(W, d)$ operation one can prove

$$\overline{I}_W(s) \in O\left(\frac{\log s}{s} B_V(s)\right).$$

Note, that except insertions there are only queries in s. Queries do not change the size of the data set and so we can also replace s by the number of insertions here.

Altogether, the results are presented in the following theorem.

Theorem 10
A static abstract data type as presented in Figure 52 can be dynamized by means of the *binary structure* in a dynamic abstract data type $TDyn$ so that the operation *Insert*(W, d) is performed in amortized time

$$\overline{I}_W(s) \in O\left(\frac{\log s}{s} B_V(s)\right).$$

Let n be the size of the current data set. We have

$$\begin{aligned} S_W(n) &\in \log n \ S_v(n) \\ B_W(n) &\in B_V(n)) \\ Q_W(n) &\in \log n \ Q_v(n). \end{aligned}$$

6.2.2. Amortized Delete: Occasional Reconstruction

Assume that we did not have implemented the $Insert$ operation, yet.

If we have to delete an object we can not choose its location beforehand. Therefore the deletion of an objects is much more difficult than the insertion. Deletion may cause fundamental reconstruction.

For many data structures it is easier to simply mark an object as deleted. Physically the object remains in the structure but does no longer belong to the data set D. These objects have bad influence on the running time of all operations although they are no longer necessary. Therefore from time to time we have to reconstruct the data structure for the actual data set.

First of all for $TStat$ we introduce an additional operation $weak.delete(V, d)$ with cost function $WD_V(n)$. We simply want to construct a *strong* delete function with an acceptable amortized time bound for $TDyn$.

Therefore we use $weak.delete(V, d)$ until D has only the half size of V. Then we erase V and build a new structure V out of D. The cost of the *occasional reconstruction* is amortized over the preceeding *delete*-operations. This gives the following result.

Theorem 11
A static abstract data type as presented in Figure 52 with an additional $weak.delete(V, d)$ operation and with additional cost function $WD_V(n)$ can be dynamized by means of *occasional reconstruction* in a dynamic abstract data type $TDyn$ so that

$$\begin{aligned} B_W(r) &= B_V(r) \\ E_W(r) &\in O(E_V(r)) \\ Q_W(r) &\in O(Q_V(r)) \\ S_W(r) &\in O(S_V(r)) \\ \overline{D}_W(s) &\in O\left(WD_V(s) + \frac{B_V(s)}{s}\right), \end{aligned}$$

holds. The size of the current actual data set is denoted with r and s denotes the length of the operation sequence.

We omit the proof here.

6.2.3. Amortized Insert *and* Amortized Delete

In the preceeding sections we have discussed Insert and Delete separately. Now we want to show how to combine the two approaches.

A static abstract data type with a weak delete implementation is given. As in Section 6.2.1 we use the binary structure for the insertion. The operation *weak.delete* is only available for the structures V_i and we have to extend it to W in order to apply the result of Section 6.2.2. If $Weak.Delete(W, d)$ is applied, d should be marked as deleted in W. But we do not know in which of the structures V_i the element d lies. Therefore in addition to the binary structure we construct a balanced *searchtree* T that stores this information. For every $d \in W$ there is a pointer to the structure V_i with $d \in V_i$, see Figure 54 for an example.

The additional cost of the search tree T is covered as follows. *Query* operations are not involved. For $Weak.Delete(W, d)$ there is an additional $O(\log n)$ for searching the corresponding V_i and for marking d as deleted in V_i.

If an object d has to be inserted we have to update T. The object d gets an entry for its structure V_j in T, this is done in time $O(\log n)$ and it will not affect the time bound for the insertion. But furthermore if $V_0, \dots, V_{j-1}$ has to be erased the corresponding objects should point to V_j afterwards. This can be efficiently realized by collecting the pointers of T to V_i in a list for every V_i. We collect the pointers and change them to "'V_j"'. This operation is already covered by time $O(B_V(2^j))$ for constructing V_j.

Altogether we conclude:

Theorem 12
A static abstract data type as presented in Figure 52 with an additional $weak.delete(V, d)$ operation and with additional cost function $WD_V(n)$ can be dynamized by means of *binary structure*, *searchtree* T and *occasional reconstruction* in a dynamic abstract data type $TDyn$ so that the amortized time for insertion reads

$$\overline{I}_W(s) \in O\left(\log s \frac{B_V(s)}{s}\right),$$

and the amortized time for deletion reads

$$\overline{D}_W(s) \in O\left(\log s + WD_V(s) + \frac{B_V(s)}{s}\right).$$

For the rest of the operations we have

$$\begin{aligned} B_W(r) &= B_V(r) \\ E_W(r) &\in O(\log r \; E_V(r)) \\ Q_W(r) &\in O(\log r \; Q_V(r)) \\ S_W(r) &\in O(S_V(r)). \end{aligned}$$

The size of the current actual data set is denoted with r and s denotes the length of the operation sequence.

6.3. Worst-Case sensitive Insert and Delete

In the last section we have seen that it is easy to amortize the *cost* of Insert and Delete analytically over time. The main idea for the construction of the dynamic data structure was given by the *binary structure* of W_n which has fundamental changes from time to time but the corresponding costs were amortized. Now we are looking for the worst-case cost of Insert and Delete. The idea is to distribute the construction of V_j *itself* over time, i.e., the structure V_j should be finished if V_{j-1}, V_{j-2}, ... V_0 has to be erased.

We only refer to the result of this approach. The ideas are very similar to the ideas of the preceeding sections. Technically, a *modified binary representation* is used in order to distribute the effort of the reconstruction over time. For the interested reader we refer to Klein [38] or van Kreveld [61] and Overmars [51].

Theorem 13
A static abstract data type as presented in Figure 52 with an additional $weak.delete(V, d)$ operation and with additional cost function $WD_V(n)$ can be dynamized in a dynamic abstract data type $TDyn$ so that

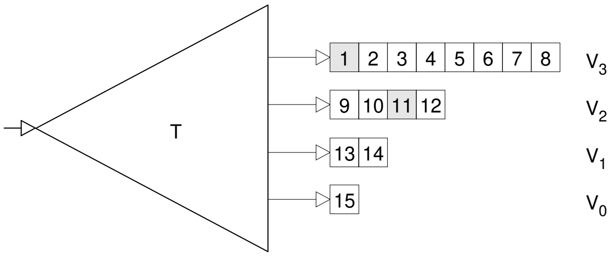

Figure 54: *A structure W_{15} with a searchtree T storing pointers to V_i for every $d \in V_i$.*

$$\begin{array}{ll}
\mathit{Build}(W, D) & \in O(B_V(n)) \\
\mathit{Query}(W, q) & \in O(\log n \cdot Q_V(n)) \\
\mathit{Insert}(W, d) & \in O\left(\frac{\log n}{n} B_V(n)\right) \\
\mathit{Delete}(W, d) & \in O\left(\log n + WD_V(n) + \frac{B_V(n)}{n}\right) \\
\text{Space} & \quad O(S_V(n)).
\end{array}$$

Here n denotes the number of relevant, stored data objects.

6.4. A Simple Example

For convenience, we take a simple example from Section 2 and apply Theorem 13, thus implementing worst-case sensitive insertion and deletion.

In Section 2.2 an easy implementation of the *static* k-d-tree was presented with $S_{k\text{-}d}(n) = O(n)$ and query time $Q_{k\text{-}d}(n) = O(\sqrt{n} + a)$, where a represents the size of the answer, see Theorem 4. Obviously, $\mathit{weak.delete}(k\text{-}d, x)$ can be implemented in $O(\log n)$ time, thus $WD_{k\text{-}d}(n) = O(\log n)$. Additionally we have $B_{k\text{-}d}(n) = O(n \log n)$.

Let $\mathit{Dyn}(k\text{-}d)$ denote the dynamic variant based upon the *statically* implemented k-d-tree.

Application of Theorem 13 results in:

$$\begin{array}{ll}
\mathit{Build}(\mathit{Dyn}(k\text{-}d), D) & \in O(n \log n) \\
\mathit{Query}(\mathit{Dyn}(k\text{-}d), q) & \in O(\sqrt{n} \log n + a) \\
\mathit{Insert}(\mathit{Dyn}(k\text{-}d), d) & \in O\left(\log^2 n\right) \\
\mathit{Delete}(\mathit{Dyn}(k\text{-}d), d) & \in O\left(\log n\right) \\
\text{Space} & \quad O(n).
\end{array}$$

References

1. J. Arvo and D. B. Kirk, *Fast Ray Tracing by Ray Classification*, in Computer Graphics (SIGGRAPH '87 Proceedings), M. C. Stone, ed., vol. 21, July 1987, pp. 55–64.

2. F. Aurenhammer, *Voronoi diagrams: A survey of a fundamental geometric data structure*, ACM Comput. Surv., 23 (1991), pp. 345–405.

3. F. Aurenhammer and R. Klein, *Voronoi Diagrams*, in Handbook of Computational Geometry, J.-R. Sack and J. Urrutia, eds., Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000, pp. 201–290. URL `http://wwwpi6.fernuni-hagen.de/Publikationen/tr198.pdf`.

4. J. Avro and D. Kirk, *A survey of ray tracing acceleration techniques*, in An Introduction to Ray Tracing, A. Glassner, ed., Academic Press, San Diego, CA, 1989, pp. 201–262. ISBN 0-12-286160-4.

5. L. Balmelli, J. Kovacevic, and M. Vetterli, *Quadtrees for Embedded Surface Visualization: Constraints and Efficient Data Structures*, in Proc. of IEEE International Conference on Image Processing (ICIP), vol. 2, Oct. 1999, pp. 487–491.

6. L. Balmelli, T. Liebling, and M. Vetterli, *Computational Analysis of 4-8 Meshes with Application to Surface Simplification using global Error*, in Proc. of the 13th Canadian Conference on Computational Geometry (CCCG), Aug. 2001.

7. G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal, *BOXTREE: A Hierarchical Representation for Surfaces in 3D*, Computer Graphics Forum, 15 (1996), pp. 387–396. Proceedings of Eurographics '96. ISSN 1067-7055.

8. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, *The R^*-tree: An efficient and robust access method for points and rectangles*, in Proc. ACM SIGMOD Conf. on Management of Data, 1990, pp. 322–331.

9. J. Bernal, *Bibliographic notes on Voronoi diagrams*, tech. rep., National Institute of Standards and Technology, Gaithersburg, MD 20899, 1992.

10. J.-D. Boissonnat and M. Teillaud, *On the randomized construction of the Delaunay tree*,

Theoret. Comput. Sci., 112 (1993), pp. 339–354. URL `http://www.inria.fr/cgi-bin/wais_ra_sophia?question=1140`.

11. P. J. C. Brown, *Selective Mesh Refinement for Rendering*, PhD dissertation, Emmanuel College, University of Cambridge, Feb. 1998. URL `http://www.cl.cam.ac.uk/Research/Rainbow/publications/pjcb/thesis/`.

12. N. Chin, *Partitioning a 3D Convex Polygon with an Arbitrary Plane*, in Graphics Gems III, D. Kirk, ed., Academic Press, 1992, chapter V.2, pp. 219–222.

13. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin, Germany, 2nd ed., 2000.

14. F. Dehne and H. Noltemeier, *A computational geometry approach to clustering problems*, in Proc. 1st Annu. ACM Sympos. Comput. Geom., 1985, pp. 245–250.

15. A. K. Dewdney and J. K. Vranch, *A convex partition of R^3 with applications to Crum's problem and Knuth's post-office problem*, Utilitas Math., 12 (1977), pp. 193–199.

16. H. Djidjev and A. Lingas, *On computing the Voronoi diagram for restricted planar figures*, in Proc. 2nd Workshop Algorithms Data Struct., vol. 519 of Lecture Notes Comput. Sci., Springer-Verlag, 1991, pp. 54–64.

17. D. P. Dobkin and M. J. Laszlo, *Primitives for the manipulation of three-dimensional subdivisions*, Algorithmica, 4 (1989), pp. 3–32.

18. R. A. Dwyer, *Higher-dimensional Voronoi diagrams in linear expected time*, Discrete Comput. Geom., 6 (1991), pp. 343–367.

19. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, vol. 10 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Heidelberg, West Germany, 1987.

20. H. Edelsbrunner and N. R. Shah, *Incremental topological flipping works for regular triangulations*, Algorithmica, 15 (1996), pp. 223–241.

21. S. A. Ehmann and M. C. Lin, *Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition*, in Computer Graphics Forum, vol. 20, 2001, pp. 500–510. ISSN 1067-7055.

22. A. A. Elassal and V. M. Caruso, *USGS digital cartographic data standards - Digital Elevation Models*, Technical Report Geological Survey Circular 895-B, US Geological Survey, 1984.

23. D. A. Field, *Implementing Watson's algorithm in three dimensions*, in Proc. 2nd Annu. ACM Sympos. Comput. Geom., 1986, pp. 246–259.

24. S. Fortune, *Voronoi diagrams and Delaunay triangulations*, in Computing in Euclidean Geometry, D.-Z. Du and F. K. Hwang, eds., vol. 1 of Lecture Notes Series on Computing, World Scientific, Singapore, 1st ed., 1992, pp. 193–233.

25. H. Fuchs, Z. M. Kedem, and B. F. Naylor, *On Visible Surface Generation by a Priori Tree Structures*, in Computer Graphics (SIGGRAPH '80 Proceedings), vol. 14, July 1980, pp. 124–133.

26. D. Fussell and K. R. Subramanian, *Fast Ray Tracing Using K-D Trees*, Technical Report TR-88-07, U. of Texas, Austin, Dept. Of Computer Science, Mar. 1988.

27. A. S. Glassner, ed., *An Introduction to Ray Tracing*, Academic Press, 1989.

28. J. Goldsmith and J. Salmon, *Automatic Creation of Object Hierarchies for Ray Tracing*, IEEE Computer Graphics and Applications, 7 (1987), pp. 14–20.

29. S. Gottschalk, M. Lin, and D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*, in SIGGRAPH 96 Conference Proceedings, H. Rushmeier, ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, Aug. 1996, pp. 171–180. held in New Orleans, Louisiana, 04-09 August 1996.

30. L. J. Guibas and J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Trans. Graph., 4 (1985), pp. 74–123.

31. H. W. Hamacher, *Mathematische Lösungsverfahren für planare Standortprobleme*, Verlag Vieweg, Wiesbaden, 1995.

32. P. M. Hubbard, *Real-Time Collision Detection and Time-Critical Computing*, in SIVE 95, The First Worjshop on Simulation and Interaction in Virtual Environments, no. 1, Iowa City, Iowa, July 1995, University of Iowa, informal proceedings, pp. 92–96.

33. H. Inagaki, K. Sugihara, and N. Sugie, *Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams*, in Proc. 4th Canad. Conf. Comput. Geom., 1992, pp. 334–339.

34. B. Joe, *3-Dimensional Triangulations from Local Transformations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 718–741.

35. ———, *Construction of Three-Dimensional Delaunay Triangulations Using Local Transformations*, Comput. Aided Geom. Design, 8 (1991), pp. 123–142.

36. N. Katayama and S. Satoh, *The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries*, in Proc. ACM SIGMOD Conf. on Management of Data, 1997, pp. 369–380.

37. T. L. Kay and J. T. Kajiya, *Ray Tracing Complex Scenes*, in Computer Graphics (SIGGRAPH '86 Proceedings), D. C. Evans and R. J. Athay, eds., vol. 20, Aug. 1986, pp. 269–278.

38. R. Klein, *Algorithmische Geometrie*, Addison-Wesley, Bonn, 1997. URL `http://www.oldenbourg.de/cgi-bin/rotitel?T=24382`.

39. J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowrizal, and K. Zikan, *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs*, IEEE Transactions on Visualization and Computer Graphics, 4 (1998), pp. 21–36.

40. T. Larsson and T. Akenine-Mller, *Collision Detection for Continuously Deforming Bodies*, in Eurographics, 2001, pp. 325–333. short presentation.

41. C. L. Lawson, *Software for C^1 surface interpolation*, in Math. Software III, J. R. Rice, ed., Academic Press, New York, NY, 1977, pp. 161–194.

42. S. Leutenegger, J. Edgington, and M. Lopez, *STR : A Simple and Efficient Algorithm for R-Tree Packing*, in Proceedings of the 13th International Conference on Data Engineering (ICDE'97), Washington - Brussels - Tokyo, Apr. 1997, IEEE, pp. 497–507. ISBN 0-8186-7807-0.

43. T.-Y. Li and J.-S. Chen, *Incremental 3D Collision Detection with Hierarchical Data Structures*, in Proc. VRST '98, Taipei, Taiwan, Nov. 1998, ACM, pp. 139–144.

44. P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hughes, N. Faust, and G. Turner, *Real-Time, Continuous Level of Detail Rendering of Height Fields*, in SIGGRAPH 96 Conference Proceedings, H. Rushmeier, ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, Aug. 1996, pp. 109–118. held in New Orleans, Louisiana, 04-09 August 1996.

45. P. Lindstrom and V. Pascucci, *Visualization of Large Terrains Made Easy*, in Proc. IEEE Visualization, San Diego, 2001.

46. W. E. Lorensen and H. E. Cline, *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, in Computer Graphics (SIGGRAPH '87 Proceedings), M. C. Stone, ed., vol. 21, July 1987, pp. 163–169.

47. G. Mller, S. Schfer, and W. D. Fellner, *Automatic Creation of Object Hierarchies for Radiosity Clustering*, Computer Graphics Forum, 19 (2000). CODEN CGFODY. ISSN 0167-7055.

48. B. Naylor, J. Amanatides, and W. Thibault, *Merging BSP Trees Yields Polyhedral Set Operations*, in Computer Graphics (SIGGRAPH '90 Proceedings), F. Baskett, ed., vol. 24, Aug. 1990, pp. 115–124.

49. B. F. Naylor, *A Tutorial on Binary Space Partitioning Trees*, ACM SIGGRAPH '96 Course Notes 29, (1996).

50. R. Osada, T. Funkhouser, B. Chazelle, and D. Dobkin, *Matching 3D models with shape distributions*, in Proceedings of the International Conference on Shape Modeling and Applications (SMI-01), B. Werner, ed., Los Alamitos, CA, May 7–11 2001, IEEE Computer Society, pp. 154–166.

51. M. H. Overmars, *The Design of Dynamic Data Structures*, vol. 156 of Lecture Notes Comput. Sci., Springer-Verlag, Heidelberg, West Germany, 1983.

52. I. J. Palmer and R. L. Grimsdale, *Collision Detection for Animation using Sphere-Trees*, Computer Graphics Forum, 14 (1995), pp. 105–116. CODEN CGFODY. ISSN 0167-7055.

53. M. S. Paterson and F. F. Yao, *Efficient binary space partitions for hidden-surface removal and solid modeling*, Discrete Comput. Geom., 5 (1990), pp. 485–503.

54. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, NY, 1985.

55. V. T. Rajan, *Optimality of the Delaunay triangulation in R^d*, in Proc. 7th Annu. ACM Sympos. Comput. Geom., 1991, pp. 357–363.

56. J. Revelles, C. Urena, and M. Lastra, *An Efficient Parametric Algorithm for Octree Traversal*, in WSCG 2000 Conference Proceedings, 2000. URL `http://visinfo.zib.de/EVlib/Show?EVL-2000-307`.

57. N. Roussopoulos and D. Leifker, *Direct spatial search on pictorial databases using packed R-trees*, in Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data, May 28–31, 1985, LaMansion Hotel, Austin, Texas, S. Navathe, ed., New York, NY 10036, USA, 1985, ACM Press, pp. 17–31. ISBN 0-89791-160-1. LCCN QA 76.9 D3 I59 1985; QA1 .A87. URL http://www.acm.org/pubs/articles/proceedings/mod/318898/p17-roussopoulos/p17-roussopoulos.pdf;http://www.acm.org/pubs/citations/proceedings/mod/318898/p17-roussopoulos/.

58. M. I. Shamos, *Computational Geometry*, ph.D. thesis, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1978.

59. M. Tanemura, T. Ogawa, and W. Ogita, *A New Algorithm for Three-Dimensional Voronoi Tesselation*, J. Comput. Phys., 51 (1983), pp. 191–207.

60. G. van den Bergen, *Efficient Collision Detection of Complex Deformable Models using AABB Trees*, Journal of Graphics Tools, 2 (1997), pp. 1–14.

61. M. J. van Kreveld, *New Results on Data Structures in Computational Geometry*, ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1992.

62. D. F. Watson, *Computing the n-Dimensional Delaunay Tesselation with Applications to Voronoi Polytopes*, Comput. J., 24 (1981), pp. 167–172.

63. H. Weghorst, G. Hooper, and D. P. Greenberg, *Improved Computational Methods for Ray Tracing*, ACM Transactions on Graphics, 3 (1984), pp. 52–69.

64. L.-Y. Wei and M. Levoy, *Fast Texture Synthesis Using Tree-Structured Vector Quantization*, in Siggraph 2000, Computer Graphics Proceedings, K. Akeley, ed., Annual Conference Series, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000, pp. 479–488. URL http://visinfo.zib.de/EVlib/Show?EVL-2000-87.

65. J. Wilhelms and A. V. Gelder, *Octrees for Faster Isosurface Generation Extended Abstract*, in Computer Graphics (San Diego Workshop on Volume Visualization), vol. 24, Nov. 1990, pp. 57–62.

66. G. Zachmann, *Real-time and Exact Collision Detection for Interactive Virtual Prototyping*, in Proc. of the 1997 ASME Design Engineering Technical Conferences, Sacramento, California, Sept. 1997. Paper no. CIE-4306.

67. ———, *Rapid Collision Detection by Dynamically Aligned DOP-Trees*, in Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98, Atlanta, Georgia, Mar. 1998, pp. 90–97.

68. B. Zhu and A. Mirzaian, *Sorting does not always help in computational geometry*, in Proc. 3rd Canad. Conf. Comput. Geom., Aug. 1991, pp. 239–242.