A Fast Polygonization Method for Quasi-convolutionally Smoothed Polyhedra

Burkhard Claus Wünsche

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

University of Auckland
September 24, 1996

Abstract

This thesis introduces Triage Polygonization, a new fast polygonization method for quasi-convolutionally smoothed polyhedra. The polygonization method exploits the property that quasi-convolutionally smoothed polyhedra usually have predominantly planar surfaces with only edges and corners rounded.

A quasi-convolutionally smoothed polyhedron is represented implicitly as a density field iso-surface. Triage Polygonization subdivides the density field in a BSP-like manner and classifies the resulting cells as inside, outside, or intersected by the iso-surface. Planar surface areas usually lie on the boundary of cells and are extracted directly from the subdivided density field with minimal fragmentation. For cells intersected by the iso-surface a more general polygonization is performed. For quasi-convolutionally smoothed scenes with a small rounding radius Triage Polygonization is 20–30 times faster and outputs only 1–2% of the polygons of the Marching Cubes algorithm without compromising the approximation. The approach taken for Triage Polygonization can be extended to related problems. Currently research is underway to extend Triage Polygonization to handle truly convolutionally smoothed scenes.

// Acknowledgments

The following people are sincerely thanked for their direct and indirect contributions to this thesis:

- My Supervisor, Dr. Richard Lobb, who first proposed the topic for this thesis and provided essential advice, guidance, and encouragement.
- My family for their support during my years in Auckland University.
- My girl-friend and friends for their encouragement, patience, and understanding.
- Prof. Dr. Hans Hagen and Dr. Dieter Lasser from the University of Kaiserslautern for introducing me to the fields of Computer Graphics and CAGD.
- Peter Dance for technical support.
- Dr. Hans Güsgen and Dr. Jeremy Gibbons for their help with several LaTeX problems.
- Erika Albury and Peter Dansted for their proofreading efforts.
- The staff in the Computer Science Department for providing advice and needed resources and making the workplace environment most pleasant.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Polygonization is the process of approximating a surface by a set of polygons. Polygons are an important geometric entity in Computer Graphics. Many algorithms have been developed to manipulate and render polygons and polygonal scenes. The available algorithms stretch from finite element analysis, computer-aided manufacturing to image rendering. Furthermore todays graphic machines have many of these algorithms hardware implemented. This makes polygonal scenes a popular choice in Computer Graphics, especially if speed and simplicity are important.

This master thesis introduces first quasi-convolutional smoothing, a method to model smoothed polyhedral scenes developed by Richard Lobb [Lob95]. We then present a new polygonization scheme called Triage Polygonization. The method was primarily designed for quasi-convolutionally smoothed polyhedral scenes but currently research is underway to extend Triage Polygonization to related problems.

1.1 Quasi-convolutional Smoothing

Quasi-convolutional smoothing was developed from the observation that a large proportion of computer graphics scenes are modeled with polyhedra. However, natural polyhedra usually have smoothed edges and corners connecting adjacent planes. Many methods exist to replace sharp edges and corners with rounded surfaces. However, most prove computationally difficult or are clumsy to handle. A solution to this problem is quasi-convolutional smoothing.

Lobb [Lob95] restricts the scene description to polyhedra. Polyhedral objects are represented by CSG-like structures with arithmetic operators as internal nodes rather then set membership operators. An object is rounded by approximating the process of true convolutionally filtering. The smoothed object's surface is defined as iso-surface in the resulting density field.

Quasi-convolutionally smoothed objects have some special properties. The most important is that they usually consist of large planar surfaces connected by relatively small parts of smooth and curved surfaces.

In the original work by Richard Lobb these scenes are rendered by ray-tracing. The results are good but may easily need several hours computation time. Especially during the modeling process this is not acceptable. A solution is to polygonize the scene. The existing polygonization algorithms, however, do not exploit the special properties of a quasi-convolutionally smoothed scene and hence lead to fragmentation and an unnecessary long computation time.

1.2 Triage Polygonization

Triage Polygonization is a fast, high-quality polygonization method especially designed for quasi-convolutionally smoothed objects. It first generates a special BSP-like space subdivision of the object. Potential regions of curvature are identified and most planar regions of the rounded object are extracted in a subsequent step. A subspace polygonization process polygonizes the remaining parts of the object surface.

Triage Polygonization is fast, produces arbitrary convex polygons, and results in only a small fragmentation of the surface. Triage Polygonization is guaranteed to find rounded edges and corners of an objects. For small rounding radii the resulting polygonizations are equivalent.

1.3 Why "Triage" ?

> **triage** *n.* **1** the act of sorting according to quality. **2** the assignment of degrees of urgency to decide the order of treatment of wounds, illnesses, etc. [F f. *trier*: cf. TRY]
>
> *The concise Oxford dictionary of current English [All90]*

"Triage Polygonization" is named after its two main principles: first note that Triage Polygonization partitions the space into regions of three different classes: regions which lie outside the object, regions inside the object, and regions which contain the surface of the object. The number "three" is reflected in the syllable "tri" [**tri-**, three or three times, L & Gk f. L *tres*, Gk *treis* three].

Secondly the word "triage" reflects that Triage Polygonization subdivides the surface into areas of different qualities and "urgency". Planar surface patches are polygonized immediately, whereas curved surface patches are polygonized in a later subspace polygonization step.

1.4 Guide to this Thesis

This thesis is divided into three parts. The first part encompasses chapters 2–4 and introduces concepts, data structures, and includes a literature review. The second part presents Triage Polygonization in chapter 5 and select implementation details in the following chapter. The third and final part of this thesis gives a performance analysis of Triage Polygonization in chapter 7 and concludes with a summary of this thesis and the achievements. For easier reading we give here a short preview of the following chapters:

Chapter 2 introduces the concepts of CSG objects and quasi-convolutionally smoothed polyhedra, the object model used in this thesis. The chapter concludes with a presentation of the corresponding data structures and some example scenes which we use for testing. A scene is composed from quasi-convolutionally smoothed and unsmoothed polyhedra.

Chapter 3 introduces BSP trees. We also present two b-rep algorithms and analyze their time complexity. The reader familiar with the concept of BSP trees may want to skip this chapter altogether. The first time reader can skip the complexity analysis (section 3.6) and the mathematical derivations (subsection 3.3.3). The implementation details (section 3.5) are only interesting for the actual implementation.

Chapter 4 gives an overview of existing polygonization methods. We present four methods in detail and extract some common features from them. Some details of the polygonization methods occur in similar form in Triage Polygonization. The first time reader might skip the overview of existing polygonization methods (subsections 4.2.1 – 4.2.4). We recommend however reading the analysis of polygonization methods (section 4.3) and the list of quality criteria (section 4.4) because they form the motivation for Triage Polygonization.

Chapter 5 presents Triage Polygonization. Some refinements contained in sections 5.5 – 5.7 are interesting but maybe cumbersome to read. It is not essential to understand every detail in these sections.

Chapter 6 gives some implementation details. We describe numerical problems and their solution and deal with the aspect of continuity of the polygonized surface. Also two extensions to the object model are presented. Though interesting these issues are not important for the understanding of Triage Polygonization and the first time reader can leave them out.

Chapter 7 gives statistical results and analyzes the performance of the algorithm. Some results are only important for a deeper understanding of the different steps of Triage Polygonization. The complexity analysis builds on results obtained for b-reps (see sections 3.6 – 3.7) and might be hard to understand without reading the latter sections. The reader with short time on hand can immediately proceed to the conclusion where the main results are summarized.

Chapter 8 summarizes Triage Polygonization and the results we obtained. We suggest some improvements and mention areas of future research.

This thesis contains many mathematical details and proofs. Though they prove useful for an exact understanding of some concepts used in this thesis they are often cumbersome to read. Most of the ideas are also intuitively clear. The reader might want to read the theorems in appendix A but can skip the proofs, which provide little additional insight.

Appendix B summarizes the data structures introduced in this thesis. It also defines some library functions frequently used in the algorithms given.

Appendix C contains a brief glossary and appendix D gives color images of some example scenes.

CHAPTER 2

The Object Model

This thesis presents a new polygonization method called Triage Polygonization. The method is specifically designed for quasi-convolutionally smoothed objects. We refer to the data structure for quasi-convolutionally smoothed polyhedra as the *object model*. Basic knowledge of the object model is necessary to understand Triage Polygonization.

This chapter introduces the object model by first presenting a polyhedral model defined as a CSG object. Many computer graphic scenes are modeled with polyhedra. However, natural polyhedra usually have rounded edges and corners. Richard Lobb [Lob95] suggests as solution a global rounding method called quasi-convolutional smoothing. The smoothing process transforms the polyhedral object to an arithmetic tree representing a density field. The surface of the smoothed object is given as an iso-surface of the density field.

The first section of this chapter introduces the concept of CSG objects. The quasi-convolutional smoothing process is presented next. We then mention restrictions and extensions to the object model. The chapter concludes with the general scene description, the data types used, and a presentation of some example scenes used as test data.

2.1 CSG Objects

Constructive Solid Geometry (CSG) objects are a popular model in CAD/CAM technology. They are represented as trees whose internal nodes represent set operations and whose leaves represent primitive solids. The object is defined by applying the set operations recursively to the child objects.

Example 2.1 Figure 2.1 shows a table modeled as union of a table top and four legs. The top and legs are cuboids and are defined as primitive objects.

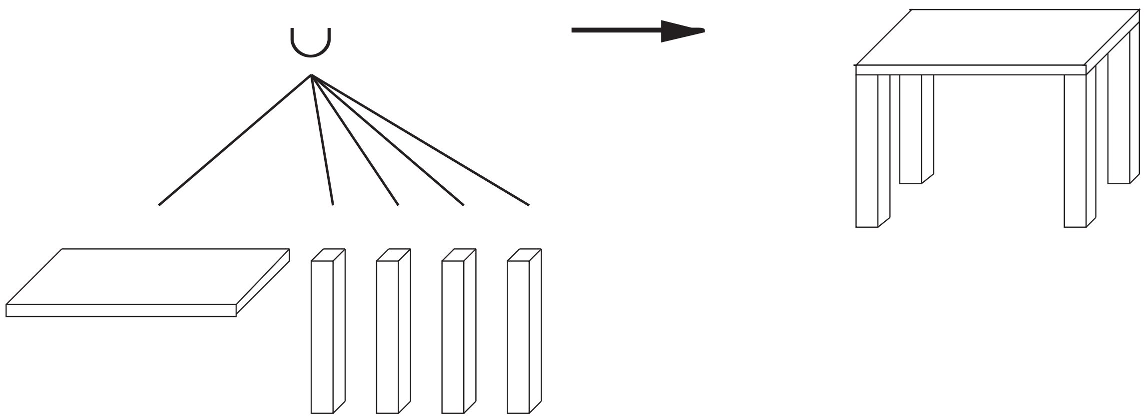

Figure 2.1. *A table as union a of five cuboids.*

In our object model the primitive objects are convex polyhedra. In the following paragraphs we introduce a sound mathematical concept for CSG objects which is necessary for understanding the algorithms presented later.

First realize that not all subsets of $\mathbb{R}^3$ are acceptable idealizations for the shape of physical (real world) objects. Requicha has proposed a set of properties and conditions that characterize the notion of "real-world solidity" [Req80].

He observed that a physical solid is homogeneously three dimensional; that is, every part of it has a volume. A solid object has no dangling vertices, faces, or edges, and no infinitely thin holes or cracks; it has a bounded volume and occupies a finite portion of space. The boundary of a solid object is a *2-manifold* and separates its inside from its outside. Requicha proposes *regularized sets* (r-sets) as suitable models for solids.

For a set A denote its boundary by ∂A, its (open) interior by $int(A)$, its closure by $\overline{A}$, and its complement by A^c.

Definition 2.1 (R-Set) *An r-set is a bounded, closed regular, semi-analytic set (see appendix C for definitions).*

The closed regularity reflects the property that the boundary of an object separates its inside from its environment. For example the objects in figure 2.3 are not closed regular because of the dangling edges and hence are not r-sets. Half-spaces are not valid r-sets because they are not bounded. Fractal objects are excluded from r-sets because they are not semi-analytic.

The intuitive mathematical operators for combining r-sets into more complex objects are the standard set operations union, intersection, set difference and complement. However the result of applying a standard set operation to two r-sets may not be an r-set.

Color Images 150

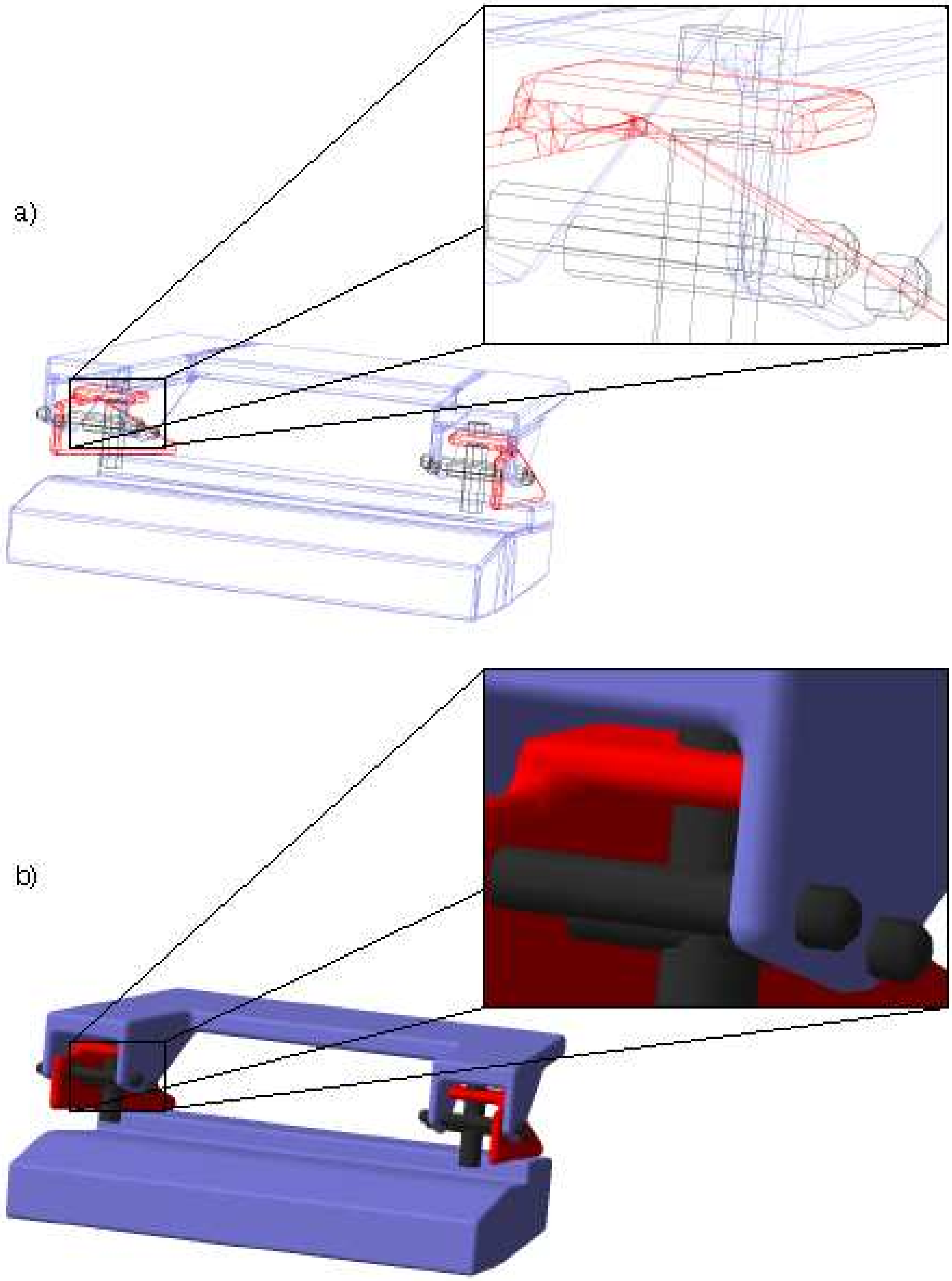

Figure D.1. *The "Hole Punch" scene polygonized with Triage Polygonization. The result is shown as a wireframe representation (a) and as Gouraud shaded polygons (b).*

Figure 2.2. .

Figure 2.3. *Not closed regular sets.*

Example 2.2 Figure 2.4 shows two rectangles sharing an edge. Their intersection is the shared edge which is not closed regular and hence not an r-set.

Modified versions of these operators, that preserve closed regularity and semi-analyticity are given by regularized set operations [Req80, TR80].

Definition 2.2 (Regularized Set Operations) *The regularized union ($\cup^\star$), regularized intersection ($\cap^\star$), regularized set difference ($\setminus^\star$) of two sets X and Y, and the regularized complement ($c^\star$) of a set X are defined as*

$$\begin{aligned} X \cup^\star Y &= \overline{int(X \cup Y)} \\ X \cap^\star Y &= \overline{int(X \cap Y)} \\ X \setminus^\star Y &= \overline{int(X \setminus Y)} \\ X^{c^\star} &= \overline{int(X^c)} \end{aligned}$$

Note that

$$\overline{int(X)} \subseteq \overline{X} = X \cup \partial X \tag{2.1}$$

and

$$\partial(\overline{int(X)}) = \partial(int(X)) \subseteq \partial X \tag{2.2}$$

Hence a regularized set is only a subset of the union with its boundary and the boundary of a regularized set is only a subset of its original boundary. With these mathematical basics and noting that we choose convex polyhedra as primitive objects a CSG object is defined in definition 2.3.

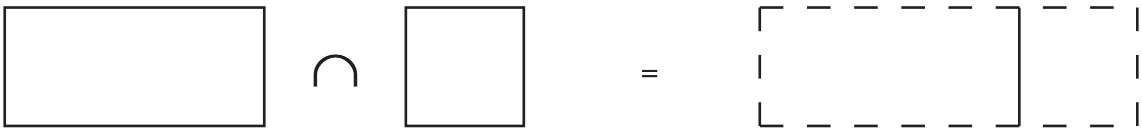

Figure 2.4. *Standard set operations form non-regular sets.*

Definition 2.3 (CSG object[a]) *A set X is a CSG object if*

$$X = P$$

where P is a convex polyhedron, or

$$X = Y \oplus^{\star} Z, \ \ (\oplus^{\star} \in \{\cup^{\star}, \cap^{\star}, \setminus^{\star}\})$$

where Y and Z are CSG objects

[a]This definition is restricted to our application. The general definition allows a greater variety of set operations and primitive objects.

In the following chapters, if not noted differently, we change freely between the expressions "set", "regularized set" and "CSG object".

The object model at this stage is a polyhedral CSG object. The following section explains how the model is globally smoothed.

2.2 Quasi-Convolutional Smoothing

The previous section introduced the initial object model as a polyhedral CSG object. This section extends a CSG object with a rounding attribute and shows how to define a quasi-convolutionally smoothed object.

We will first give an overview of popular general smoothing methods. Their inherent complexity is overcome by a scheme called convolutional smoothing. From this we derive quasi-convolutional smoothing. The quasi-convolutionally smoothed objects represent the object model used in this thesis.

2.2.1 Blending Techniques

Natural objects usually have rounded edges and corners. If a polyhedral natural object is modeled it is often found that its rounded surfaces are functionally not important for the model. In CAD/CAM technology these surfaces are called *blends*. The principle difficulty is to shape and position blends so as to achieve tangency to primary (unrounded) surfaces.

Blends can be defined as implicit or parametric surfaces. Implicit blends are usually given as algebraic surfaces. In general they are easier to derive but offer less shape control than parametric blends. Also the rendering of an implicitly defined surface proves more complex. Parametric blends are usually given as free-form surfaces. They offer various design parameters, but are usually harder to define. A polynomial parametric surface is of a higher degree than the corresponding algebraic surface.

Blending methods are either *local* or *global*. A local blending method is restricted to a bounded area, usually a single edge or corner. In contrast a global blending method smoothes a whole object at once. Blending methods can be grouped into three concepts.

The most general concept is *surface blending*, which creates a new piece of surface by smoothly joining two existing surfaces. Algebraic surface blends are described in [HH87, Kos91, War89, MS85, Sed85]. Koparkar designs a parametric blend between two parametrically defined surfaces [Kop91a, Kop91b]. Chiyokura et al. devise free-form surfaces to interpolate over irregular meshes [CTKH91].

Volumetric blending is based on a solid model (usually represented as CSG object or b-rep). The blending operation is just one operation of the modeler. Rockwood and Owen [RO87] establish a pattern to devise implicitly described blending surfaces in solid modeling and describe super elliptic blends. Middleditch and Sears [MS85] use spherical blends. Várady et al. define parametric blending for a b-rep modeler. Rossignac and Requicha [RR84] extend the CSG concept by constant radius blends defined in terms of offset solids.

Polyhedral blending involves generating free-form surfaces from polyhedra. This reduces complex calculations and improves reliability. Catmull and Clark [CC78] describe a recursive subdivision scheme. The method has been extended by various authors to local rounding operations for integrated solid modeling. Chiyokura [Chi87] replaces edges by Gregory patches, whose tangent planes are continuous on the bounds of the meshes generated. Beeker [Bee86] uses Bernstein patches, whereas Fjällström [Fjä86] uses a generalization of the Brown square [BBK78]. Finally Hoschek and Hartmann [HH91] introduce implicit blends which can be interpreted as G^{n-1} functional splines.

All methods described above perform local blending and allow a variety of design parameters. Though this is preferable for the designer, it makes the method hard to understand and requires user interaction. Additionally the above methods are all computationally expensive. A good overview of blending methods is given by Woodwark [Woo87].

An alternative approach is described by Blinn [Bli82]. He smoothly blends articulated models by using implicit functions defined by the summation of point potentials. Wyvill et al. [WMW86b, WMW86a, WWM87] and Bloomenthal et al. [Blo88, BW90, BS91] extend the approach to potentials of skeletons and present fast rendering schemes. These models are generally known as *Soft Objects* or *Blobby Models* (see also [Mur91]). The blending method is local and does not perform optimally for smoothed polyhedral objects.

A simple global blending method is described in the next subsection.

2.2.2 Convolutional Smoothing

The previous subsection gave an overview of popular local blending methods. We mentioned that they are hard to understand, and often require user interaction.

Furthermore they have practical limitations when more than three surfaces influence the shape of the blend at any given point. Colburn [Col90] states that this limitation is related to the model complexity. He presents a solution by introducing a spherical test volume with a radius equal to the desired blend radius. The surface of the smoothed object is defined as all points such that when a sphere of the specified radius is positioned at the point, exactly half of the volume of the sphere is inside the polyhedron and half is outside. Figure 2.5 shows a polygon and its smoothed version in two dimensions.

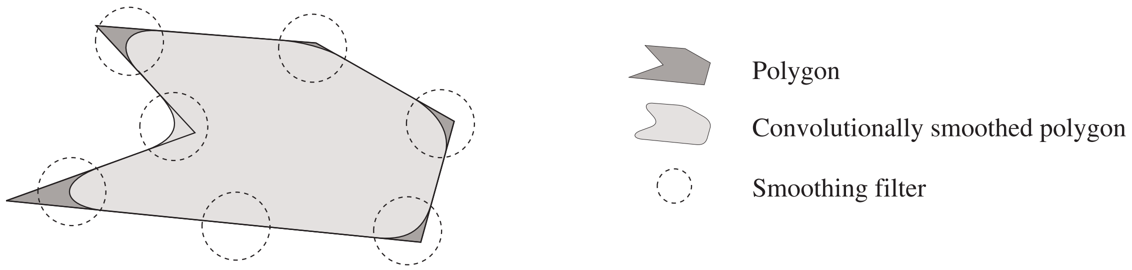

Figure 2.5. *Convolutionally smoothed polygon.*

Note that now the shape of the blend at any given point is only influenced by the portion of the unblended model that falls inside the sphere.

The process can be understood as a low pass filtering with a spherical filter of radius r and is mathematically expressed by a convolution.

Definition 2.4 (Convolutional Smoothing) *Given a smoothing filter of radius r, a set (object) $Obj \subset I\!R^3$, and a density field $\rho_{Obj} : I\!R^3 \rightarrow \{0,1\}$ such that the density has a value of 1 inside the set Obj and a value of 0 outside it, define a density field $\hat{\rho}^r_{Obj}$ as*

$$\hat{\rho}^r_{Obj}(x,y,z) = \frac{1}{4\pi r^3} \int\!\!\int\!\!\int \rho_{Obj}(u,v,w)\, h(x-u, y-v, z-w)\, du\, dv\, dw \qquad (2.3)$$

where the integral is over all space, and

$$h(x,y,z) = \begin{cases} 0 & x^2+y^2+z^2 < r^2 \\ 1 & \text{otherwise} \end{cases}$$

The convolutionally smoothed object $\widehat{Obj}^r$ is then defined as

$$\widehat{Obj}^r = \{p \in I\!R^3 \mid \hat{\rho}^r_{Obj}(p) \geq 0.5\} \qquad (2.4)$$

and its surface is all points $(x,y,z) \in R^3$ for which

$$\hat{\rho}^r_{Obj}(x,y,z) = 0.5 \qquad (2.5)$$

In this definition the function $h : \mathbb{R}^3 \rightarrow \{0, 1\}$ represents the spherical filter of radius r. The factor before the integral normalizes the result of the convolution. If the spherical filter is completely inside the object the right hand side of equation 2.3 equals one.

Though the smoothing process is easy to understand and mathematically simple, it is computationally expensive to solve equation 2.5. To render or to polygonize the smoothed object a large number of solutions must be found. Also for a non-convex object the surface of the smoothed volume can lie outside the volume. This makes it difficult to know where to search for solutions of equation 2.5.

Lobb addresses these problems with an approximation to convolutional smoothing called quasi-convolutional smoothing. The method is presented in the following section.

2.2.3 Quasi-convolutional Smoothing

The preceding subsection presented convolutional smoothing as a simple solution to the global blending problem. The solution to equation 2.5, though, proves computationally complicated because of the convolution integral.

Lobb [Lob95] introduces quasi-convolutional smoothing as a computationally easy approximation. Recall that the object model is a CSG object with convex polyhedral primitives and a rounding attribute specifying a global rounding radius r. Lobb represents convex polyhedra by (bounded) intersections of half-spaces (see figure 2.6).

He then defines the smoothed object similarly to Colburn's description but replaces the density field $\hat{\rho}^r_{Obj}$ of the convolutionally smoothed CSG object $\widehat{Obj}^r$ with an approximation ρ^r_{Obj}. Instead of convolving the CSG object Obj with a spherical filter Lobb replaces the set operations union, intersection, and set difference by the arithmetic operations addition, multiplication, and subtraction. He then convolves only the half-spaces with a spherical filter of radius r.

The density field of a smoothed half-space H^r can be described exactly by computation of the convolution integral with the density field ρ_H. The result of the convolution is equivalent to the volume of a sphere intersected with a half-space, for which the answer is

$$\rho^r_H(p) = \begin{cases} 0 & \alpha \geq 1 \\ 1 & \alpha \leq -1 \\ (1-\alpha)^2 * (2+\alpha)/4 & \text{otherwise} \end{cases} \tag{2.6}$$

$$\text{where } \alpha = \frac{d}{r}$$

and d is the distance of point p to the half-space H. Figure 2.7 shows the resulting density distribution.

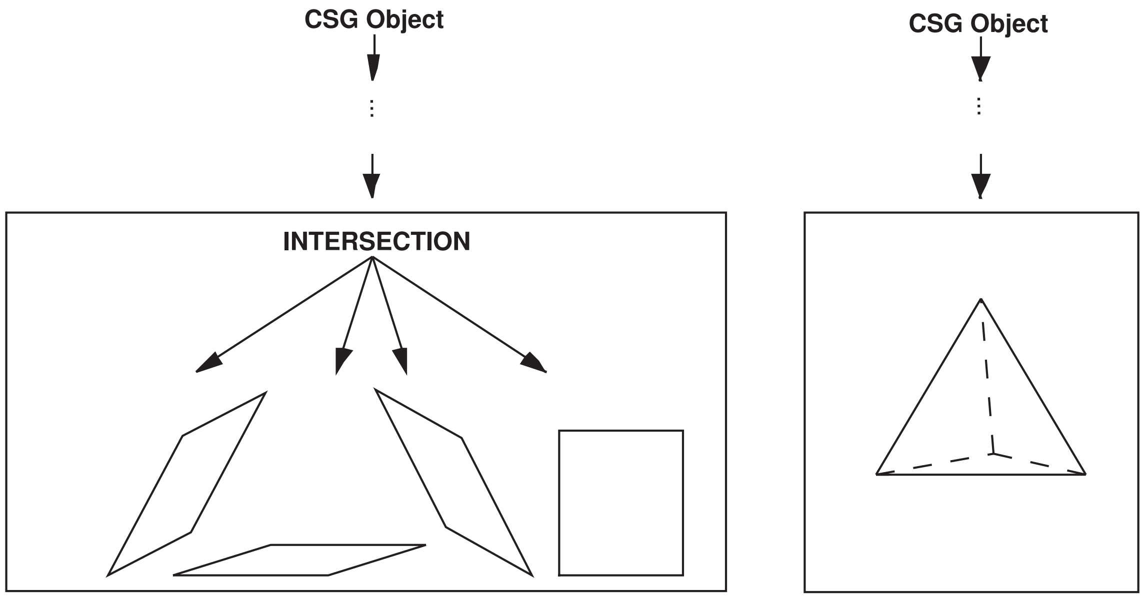

Figure 2.6. *Primitive objects are convex polyhedra which are represented as intersections of half-spaces.*

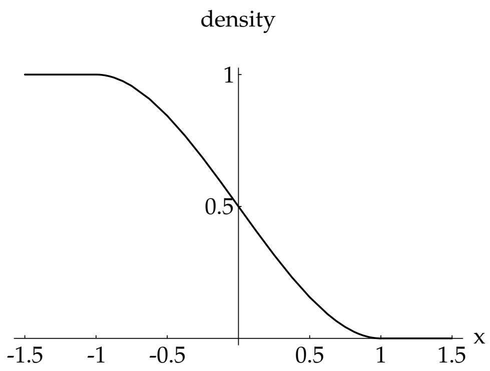

Figure 2.7. *Density distribution for a smoothed half-space.*

We can now define a quasi-convolutionally smoothed object as:

Definition 2.5 (Quasi-convolutional smoothing) *Given a smoothing filter of radius r, a CSG object $Obj \subset \mathbb{R}^3$ with intersections of half-spaces as primitives (i.e., convex polyhedra), and density fields $\rho_H : \mathbb{R}^3 \to \{0,1\}$ such that the density has a value of 1 inside a half-space H and a value of 0 outside it, define a density field ρ^r_{Obj} as*

$$\rho^r_H(x) = \begin{cases} 0 & \alpha \geq 1 \\ 1 & \alpha \leq -1 \\ (1-\alpha)^2(2+\alpha)/4 & otherwise \end{cases} \tag{2.7}$$

$$\rho^r_{A\cup B}(x) = \rho^r_A(x) + \rho^r_B(x) \tag{2.8}$$

$$\rho^r_{A\cap B}(x) = \rho^r_A(x) * \rho^r_B(x) \tag{2.9}$$

$$\rho^r_{A\setminus B}(x) = \rho^r_A(x) - \rho^r_B(x) \tag{2.10}$$

where $\alpha = d/r$, $d =< x, \vec{n}_H >$ is the distance of point x to the plane of the half-space H, $\vec{n}_H$ is H's surface normal, $< .,. >$ is the dot product (inner product) of two vectors, and A and B are CSG objects.

Then the quasi-convolutionally smoothed object Obj^r is defined as

$$Obj^r = \{x \in \mathbb{R}^3 \mid \rho^r_{Obj}(x) \geq 0.5\}$$

and its surface is all points $x \in R^3$ for which

$$\rho^r_{Obj}(x) = 0.5$$

Since the integral is a linear operator the approximation ρ^r_{Obj} is identical to $\hat{\rho}^r_{Obj}$ for the union and set difference operation. For the intersection operation, though, the density field ρ^r_{Obj} of a truly convolutionally smoothed object is generally different from $\hat{\rho}^r_{Obj}$. We only mention that the density field is also correct for a quasi-convolutionally smoothed intersection of two orthogonal half-spaces. For a further discussion on the properties of quasi-convolutional smoothing see the original paper [Lob95].

2.3 Restrictions and Extensions to the Object Model

In this section we mention restrictions and extensions of the object model. We introduced the object model as a CSG object with polyhedral primitives and a global rounding attribute. In the original paper Lobb assumes bounded primitives. For simplicity we do the same.

An important restriction is that for a smoothed CSG object Obj all density values of the density field ρ^r_{Obj} must lie in the intervall $[0,1]$. A necessary condition for this is that all unions are disjoint and that for the set difference $A \setminus B$ of two sets A and B the condition $A \supseteq B$ is valid.

Note that above conditions are not sufficient to guarantee density values in the interval $[0, 1]$. Even if all density values for the unsmoothed object are between 0 and 1, the smoothed object may have arbitrary high/low density values.

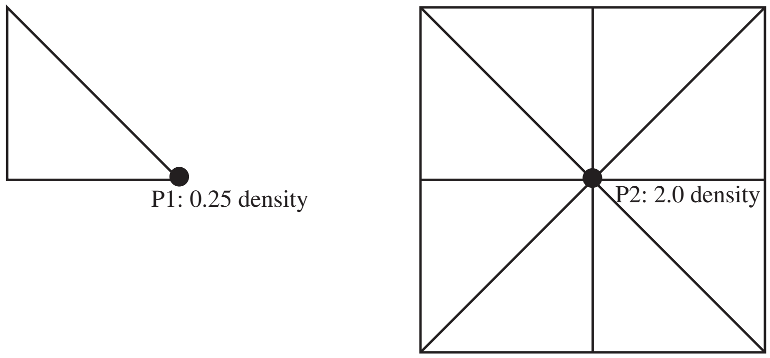

Figure 2.8. *Quasi-convolutional smoothing generates unbounded density values.*

Example 2.3 This effect is seen in figure 2.8, where a square is defined as the union of eight disjoint triangles. If the square is quasi-convolutionally smoothed, the density value at the square's centre is the sum of the density values at the corresponding vertices of the eight smoothed triangles (see equation 2.8). Assuming the rounding radius is sufficiently small, each vertex is formed as the intersection of two half-spaces. Equation 2.9 shows that the density value for a triangle vertex is 0.25. This results in a density value of 2 at the square's centre.

We also assume that the object is smooth. This is not as trivial as it sounds, since it is possible to construct a smoothed object which has arbitrary many intersections with a straight line. Also it is not clear that the variation diminishing property holds. However, since the density field is an arithmetic tree with smooth leaf functions, the maximum gradient of the density field at any point is bounded. The density field fulfills a Lipschitz property.

A final assumption is that the rounding radius of a quasi-convolutionally smoothed polyhedron is in general small to the size of the object. Then the object's surface is predominantly planar and most of the curved surfaces of the quasi-convolutionally smoothed object are simple, i.e., either rounded edges or corners. These properties are crucial for the efficiency of Triage Polygonization.

In the current form the modeling capabilities for our object model are quite restricted. Here we give a preview of two extensions which we have implemented for our polygonization method (see section 6.6). First recall that a polyhedral primitive is represented as an intersection of half-spaces. In definition 2.5 we smooth a CSG object by smoothing all half-spaces of its primitive objects with spherical filters of identical radius. Subsection 6.6.2 suggests that it can be desirable to apply filter with different radii to different half-spaces. Some of the resulting effects are also described in [Lob95]. Modeling capabilities are further improved by the introduction of clipping planes (see subsection 6.6.1). They allow the design of objects with both rounded and sharp corners and edges.

2.4 Scene Definition and Data Types

The preceding sections presented the object model used in this thesis. All that remains is to define the scene and to introduce the data types used.

The type descriptions are given in the language CLEAN [PvE93, PvE95] and are all summarized in appendix B. The syntax is similar to modern functional languages such as Haskell [HJP+92] or Miranda [Tur85], but with a few explanatory comments it should be understandable for the inexperienced reader, too.

```
::  Scene   :== CSGObject
::  Radius  :== REAL
::  Plane   :== (Vector,REAL) // (normal,distance)

::  CSGObject
            = Union CSGObject CSGObject
            | Intersection CSGObject CSGObject
            | SetDifference CSGObject CSGObject
            | Rounded Radius CSGObject
            | Primitive PolyhedralPrimitive

::  PolyhedralPrimitive
            = Intersection [HalfSpace]

::  HalfSpace :== Plane
```

Figure 2.9. *Data types of scene definition.*

The scene description is given in figure 2.9. The type definitions, which are indicated by a double colon, specify that a scene is just a CSG object with polyhedral or rounded primitives. We assume that a rounded object does not consist of other rounded objects[1]. A polyhedral primitive is given as the intersection of a list of half-spaces. A half-space is defined by a plane with an outwards pointing normal.

[1] The actual implementation allows a rounded object as a child object of a rounded object. The interpretation is that in the CSG tree the rounding attribute of a child object overrides any rounding attribute higher up defined in the tree. This extension to the object model is presented in section 6.6.2.

A polyhedral primitive is given internally as a list of its boundary faces:

```
::  PolyhedralPrimitive = Polyhedron [Face]
```

and a face is a polygon[2] defined as a list of points:

```
::  Face    :== Polygon
::  Polygon :== [Point]
```

Applying quasi-convolutional smoothing with a spherical filter of radius r transforms a CSG object with rounding attribute into an arithmetic tree representing a density field. The corresponding data type is shown in figure 2.10.

```
::  DensityField = Sum DensityField DensityField
                 | Product DensityField DensityField
                 | Difference DensityField DensityField
                 | DensityFieldOfHalfSpace Radius HalfSpace
```

Figure 2.10. *Data type of a density field.*

A density field is either the sum, product or difference of two density fields or is the density field of a half-space convolutionally smoothed with a specified radius.

With the help of equation 2.7 the density field can be evaluated in an arbitrary point. Figure 2.11 shows the functional code for the density evaluation in Clean syntax. We use this notation partly because the prototype implementation is in Clean, but mainly because it offers a much more compact and precise definition than normal pseudocode.

The function `Density` takes as input a point and a density field and returns a real value representing the density value in the given point. The function is defined by a set of rules, exactly one of which should "pattern match" any particular invocation. For example, if the arithmetic tree defining the density field is constructed as the sum of two density fields then the function `Density` returns the sum of the density values of the given point in both density fields.

We use the Clean syntax for all algorithms except for high-level algorithms. From time to time we use library functions which have no equivalent in imperative languages. The library functions are explained in appendix B.

2.5 Example Scenes

This section briefly describes a few example scenes we refer to throughout the thesis as a basis for discussion. Also they are used in chapter 3 and chapter 7 as test data for a b-rep algorithm and Triage Polygonization, respectively.

[2] In the implementation we add the face plane to the data structure. This increases numeric stability if testing two faces for coplanarity.

```
Density :: Point DensityField -> Real
Density     p     (Sum field1 field2)
        = (Density p field1) + (Density p field2)
Density     p     (Product field1 field2)
        = (Density p field1) * (Density p field2)
Density     p     (Difference field1 field2)
        = (Density p field1) - (Density p field2)
Density     p     (DensityFieldOfHalfSpace r plane)
        | distance > r  = 0.0
        | distance < -r = 1.0
        | otherwise     = truncatedSphereVolume
where
        distance              = DistanceOfPointFromPlane p plane
        truncatedSphereVolume = (1 − α)^2 ∗ (2 + α)/4
        α                     = distance / r
```

Figure 2.11. *Algorithm to evaluate a density field.*

Cube A unit cube smoothed with a spherical filter of radius 0.1 represents a simple test object for quasi-convolutional smoothing.

Cube In Cube A simple case for a complex object combining a rounded and an unrounded object. The rounded object is given as a set difference of a big cube and a small cube and is pictured in figure D.1. The full scene has a small unrounded cube inside the hole of the rounded object. This scene proved useful for debugging.

Stapler Complex real world object with many clipping planes. Figure D.3 shows a color image of the scene.

CSG Example Shows the effects of applying set operations and rounding operations in different order. Figure D.5 shows a color image of the scene.

Variable Radius Shows the effects of increasing the rounding radius for a smoothed object. A color image of the scene is given in figure D.6.

Hole Punch Very complex real world object which uses all set operations and applies smoothing to non-primitive objects. Some objects (the metal pins) are smoothed with varying rounding radii. The figure D.2 shows a color image of the scene.

Many Stapler 24 Stapler form our most complex example scene.

n^3 **Blended Cubes** Rounded union of n^3 cubes such that the cubes are blended together. A color image of 3^3 blended cubes is shown in figure D.4.

The "n^3 Blended Cubes" scene, if not smoothed, is just an array of cubes and is in this case referred to as "n^3 cubes".

To evaluate the complexity of the later presented algorithms it is useful to have some statistical information about the example scenes. Table 2.1 gives for each scene the number of primitive objects, half-spaces, clipping planes and set operations necessary to construct the scene.

Scene	# objects	# half-spaces	# clipping planes	# ($\cup^\star$)	# ($\cap^\star$)	# ($\setminus^\star$)
Cube	1	6	0	0	0	0
Cube In Cube	3	18	0	1	0	1
Stapler	7	47	6	6	0	0
CSG Example	12	72	1	7	0	4
Variable Radius	12	72	0	5	0	6
Hole Punch	33	203	32	28	2	2
Many Staplers	168	1128	144	167	0	0
n^3 Blended Cubes	n^3	$6n^3$	0	n^3-1	0	0

Table 2.1. *Example Scenes.*

CHAPTER 3

Binary Space Partitioning and Boundary Representation

This chapter introduces the concept of *Binary Space Partitioning* (BSP) trees and presents two *boundary representation* (b-rep) algorithms based on the data structure of BSP trees. The b-rep of a polyhedral CSG object represents a polygonization of the object surface.

3.1 Introduction

We are interested in the boundary representation of a CSG object, because the object model for Triage Polygonization is a rounded CSG object. Since by assumption the rounding radius is small in comparison to the object the unrounded CSG object represents a first approximation to the rounded object. This implies the following lemma:

Lemma 3.1 *The b-rep of an unrounded CSG object approximates the polygonization of a rounded CSG object.*

Lemma 3.1 forms the motivation for Triage Polygonization introduced in chapter 5. The concepts of binary space partitioning and boundary extraction are essential for the polygonization. The b-rep algorithm is also used to improve the efficiency of Triage Polygonization in the actual implementation (section 6.1).

Input of a b-rep algorithms is a polyhedral CSG object. The b-rep is represented explicitly as an augmented BSP tree. The first b-rep algorithm is called the *lazy b-rep algorithm* because the faces representing the object boundary are extracted only in a post-processing step. The second algorithm extracts the object boundary by merging BSP trees and is called the *merged b-rep algorithm*.

After presenting the b-rep algorithms some implementation details common to both algorithms follow. The chapter concludes with a complexity analysis and a presentation of the results achieved with our implementation of the algorithms.

3.2 BSP Trees

Historically the methodology underlying b-reps is that of the direct representation of the topology of a surface. The topological approach requires the decomposition of a 3-space polytope (may be generalized for d-space) into all dimensions 3,2,1,0, i.e., into polytopes, faces, edges, and points. Then the b-rep explicitly encodes the connectivity/incidence among these components.

This representation, while widely used, possesses a number of inherent limitations as is pointed out in [NAT90]. For example sets whose boundary is unbounded can not be represented. Algorithmically performing set operations with b-reps requires explicit detection of the coincidence of all combinations of the variously dimensioned elements (e.g., face-face, face-edge, edge-vertex) along with some appropriate action for each [RV85].

Also efficiency considerations demand some kind of spatial search structure. Typically an axis-aligned spatial decomposition is chosen. This, however, does not transform with the representation and must be reconstructed after each transformation.

An increasingly popular alternative is the binary space partitioning (BSP) tree [SBGS69, SSS74, FKB80, Nay81] . The fundamental methodology underlying BSP trees is spatial partitioning. Hyper-planes are used to recursively subdivide d-space to create a disjoint set of d-dimensional cells. Each cell is then designated as either interior or exterior to the set. The boundary need not be represented explicitly as it is derivable from the cells.

In this thesis we use only BSP trees in three dimensions. The hyper-planes forming the spatial partitioning are then ordinary planes. The best way to explain a BSP tree is through the process that constructs one, as illustrated in figure 3.1. One begins with a region of space R, chooses some plane h that intersects R, and then uses h to induce a binary partitioning of R. If H_{in} and H_{out} denote the inside and outside open half-spaces of h two new regions are derived:

$$R_{in} = R \cap H_{in}$$
$$R_{out} = R \cap H_{out}$$

Each of these unpartitioned children can in turn be partitioned, and so on, to produce a binary tree of regions. We make the convention that R_{in} and R_{out} always are the left and right child (*IN tree* and *OUT tree*), respectively, of the current node.

Each polyhedral object can be represented as a union of regions of a BSP tree. The regions inside and outside the object are called *IN cells* and *OUT cells*, respectively. Figure 3.2 shows a polyhedral object and a possible BSP tree representation.

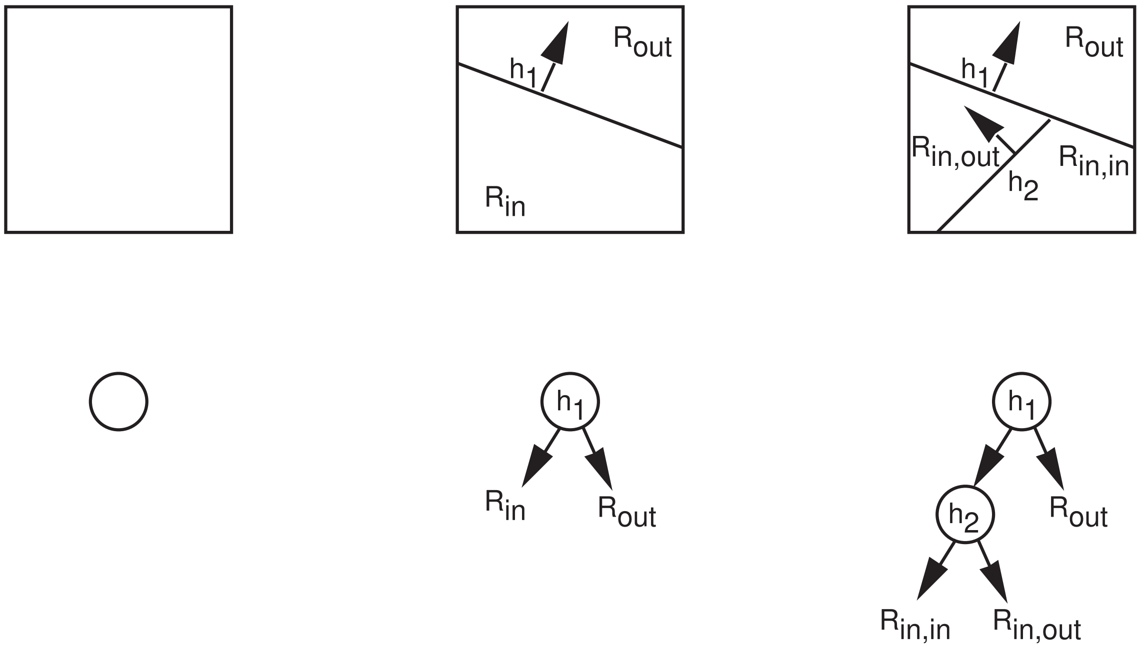

Figure 3.1. *Constructing a BSP tree.*

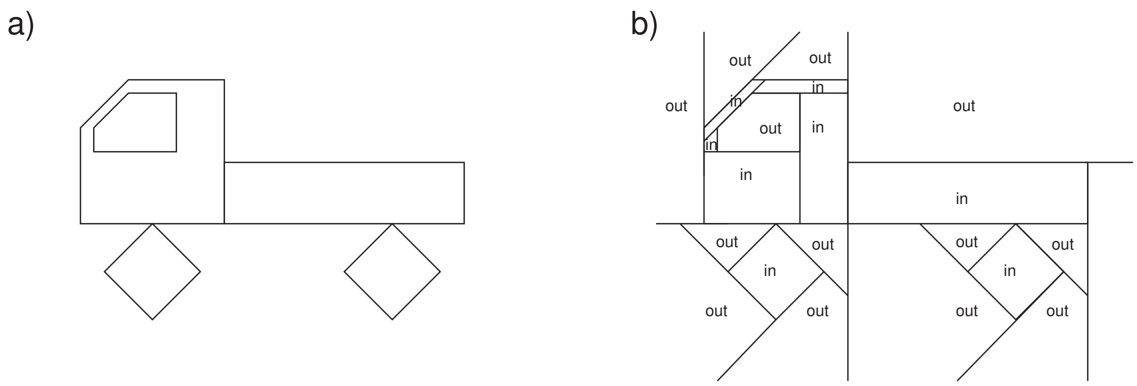

Figure 3.2. *A polyhedral object (a) and the corresponding BSP tree (b).*

Since BSP trees ignore the topological properties of a set no distinction is made between convex and non-convex sets. Thus the entire representational domain is treated uniformly, providing a considerable improvement in the simplicity of the algorithm. In addition, the spatial search structure is intrinsic to the representation and so transforms with it. Also note that a BSP tree solves the hidden surface problem. The linearity of both planes and viewing rays means that if a ray intersects a plane it does so at only one point. Hence the ray is divided into a near and far section. This permits inducing a visibility priority ordering on the three subspaces formed by the plane: near half-space, plane, and far half-space. Given a BSP tree τ, determining this ordering at every node of the tree in a recursive manner provides a total ordering of the elements of the region partitioned by τ (e.g., [SSS74, Nay81]).

For computational reasons it is often desired to represent the boundary of an object explicitly in the BSP tree structure. This is achieved by augmenting each BSP node with the boundary faces lying in its plane (e.g., [SBGS69, FKB80]). We allow instead to augment a BSP node with an arbitrary set of faces lying in its plane. This is done for computational reasons and becomes clear in the following section. With these remarks a BSP tree is defined formally as

Definition 3.1 *BSP tree*

A BSP tree $\tau(A)$, defining an object A, is recursively defined as

$$\tau(A) = (h, faces_h, \tau(A_{h,in}), \tau(A_{h,out})) \mid IN \mid OUT$$

where
h is a partitioning plane,
$faces_h$ is a set of faces lying in the partitioning plane h,
$A_{h,in}$ and $A_{h,out}$ are the parts of object A lying inside and outside the partitioning plane h, respectively,
$\tau(A_{h,in})$ and $\tau(A_{h,out})$ are the IN tree and OUT tree, respectively,
IN and OUT represent an IN cell and an OUT cell, respectively.

We denote with $faces(\tau(A))$ the set of all faces augmenting the BSP tree $\tau(A)$.

Definition 3.2 *Boundary BSP tree*

A BSP tree $\tau(A)$ contains explicitly the boundary representation of an object A if $faces(\tau(A)) = \partial A$. In this case the BSP tree is called a "boundary BSP tree" and is denoted by $\beta(A)$.

The latter interpretation of the BSP structure corresponds to the classical one [NAT90] that a BSP tree represents a set of boundary points with neighborhood information. In our case the neighborhood information is given by classifying the BSP cells into IN and OUT. So it corresponds to the direct representation of b-reps

```
::  LeafClass  = IN | OUT
::  BSPTree    = BSPNode Plane [Face] BSPTree BSPTree | BSPLeaf LeafClass
```

Figure 3.3. *Data type of a BSP tree.*

[RV85]. The data structure for a BSP tree in CLEAN reflects directly definition 3.1 and is given in figure 3.3.

In the following sections we introduce two b-rep algorithms. The derivation will show that it is easier to construct a BSP tree for a non-regularized object than for a regularized object. This does not seem to be of much help because with section 2.1 a CSG object is always defined as a regularized object. However, with equations 2.1 and 2.2 we know

$$\forall A \forall \tau(A_{non-regularized}) \exists \tau(A_{regularized}) : \tau(A_{regularized}) = \tau(A_{non-regularized}) \quad (3.1)$$

i.e., for every polyhedral CSG object A there is always a BSP tree $\tau(A_{regularized})$ equal to the BSP tree $\tau(A_{non-regularized})$ of the corresponding not-regularized object A. This result allows us to compute the b-rep of a CSG object A with a BSP tree for the corresponding non-regularized object.

Figure 3.4 shows the BSP partition for an object A before and after regularization. Clearly, the BSP partition for $A_{non-regularized}$ is valid for $A_{regularized}$,too. Note though, that the opposite is *not* true, since there is no partitioning plane for the dangling face.

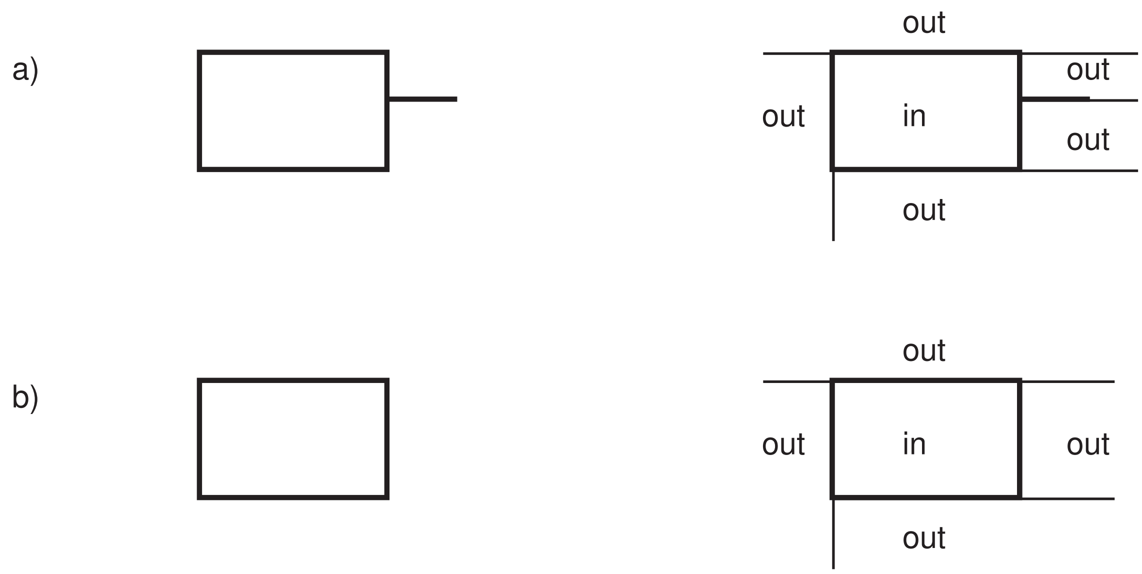

Figure 3.4. *BSP partition for a non-regularized (a) and a regularized object (b).*

3.3 Lazy B-rep Algorithm

This section introduces a b-rep algorithm popular in contemporary literature. The following b-rep algorithm was first introduced by Thibault and Naylor [TN87]. The central idea is to separate the problem of computing the boundary of an object and the problem of transforming a CSG object into a BSP tree. In the original paper the boundary is represented implicitly in the BSP tree. It can be extracted in a post-processing step. However, implementational advantages (see note 2 on page 45) suggest representing the b-rep explicitly in the BSP tree.

We first compute a BSP tree $\tau(A)$ representing a CSG object A. During computation a superset $faces(\tau(A))$ of the object's boundary ∂A is produced. The superset forms a candidate set from which the boundary representation is extracted in a post-processing step.

The algorithm transforming the CSG object A into a BSP tree $\tau(A)$ is called the *lazy BSP tree algorithm.* The lazy BSP tree algorithm together with the post-processing step of extracting the actual boundary faces yields a boundary BSP tree explicitly containing the b-rep. The resulting algorithm is called the *lazy b-rep algorithm* and is summarized in figure 3.5

```
BRep_lazy ::  CSGObject -> BSPTree

BRep_lazy csgObject = (BoundaryBSPTree o BSPTree_lazy) csgObject
```

Figure 3.5. *Lazy b-rep algorithm.*

The function `BSPTree_lazy`, introduced in the next section and given in figure 3.7, implements the lazy BSP tree algorithm. The post-processing step of extracting the object boundary is implemented by the function `BoundaryBSPTree`, which is derived in section 3.3.4 and defined in figure 3.12. The final BSP tree contains explicitly the boundary faces and is hence a boundary BSP tree. The symbol "$\circ$" denotes functional composition.

3.3.1 Lazy BSP Tree Algorithm

The lazy BSP tree for a CSG object A, denoted by $\tau_{lazy}(A)$, is a BSP tree such that $faces(\tau_{lazy}(A)) \supseteq \partial A$. The corresponding lazy BSP tree algorithm is developed by induction and makes use of equation 3.1. This means the lazy BSP tree algorithm constructs a BSP tree for a non-regularized version of the CSG object A. The resulting BSP tree is always a valid (but usually not minimal) BSP tree for the regularized CSG object A.

The induction basis is given for the primitive objects of a CSG object, which are convex polyhedra, each represented by a list of faces. A BSP tree for a primitive object is therefore given by the BSP tree for a convex polyhedron, which is a linear tree formed by the face planes of the polyhedron (see figure 3.6). Each BSP node of the linear tree is augmented with the face of the polyhedron lying on the corresponding partitioning plane. The resulting BSP tree fulfills $\tau_{lazy}(P) = \beta(P)$, i.e., it is a boundary BSP tree (see definition 3.2).

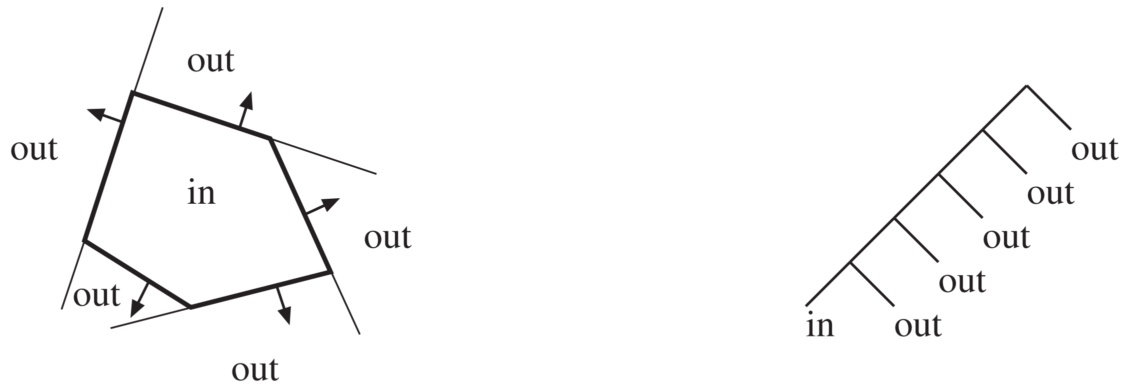

Figure 3.6. *A convex set and its BSP tree.*

For the induction step assume it is known how to compute a BSP tree τ_{lazy} for a CSG object A such that $faces(\tau_{lazy}(A)) \supseteq \partial A$. Suppose we are given a non-primitive CSG object $A_1 \oplus^\star A_2$ ($\oplus \in \{\cup, \cap, \setminus\}$). To compute $\tau_{lazy}(A_1 \oplus^\star A_2)$ insert the CSG object A_2 into the BSP tree $\tau_{lazy}(A_1)$ according to the corresponding set operation $\oplus$. The insertion operation can be considered as a (non-regularized) set operation between a BSP tree and a CSG object yielding a BSP tree. We call the operation a *lazy set operation* because it does not yield a regularized object and denote it by the subscript "lazy". The lazy set operations must fulfill $faces(\tau_{lazy}(A_1) \oplus_{lazy} A_2) \supseteq \partial(A_1 \oplus^\star A_2)$ and are described in the next subsection. Figure 3.7 summarizes the lazy BSP tree algorithm.

```
BSPTree_lazy ::  CSGObject -> BSPTree

BSPTree_lazy (Union obj1 obj2) = Union_lazy (BSPTree_lazy obj1) obj2
BSPTree_lazy (Intersection obj1 obj2) = Intersection_lazy (BSPTree_lazy obj1) obj2
BSPTree_lazy (SetDifference obj1 obj2) = SetDiff_lazy (BSPTree_lazy obj1) obj2
BSPTree_lazy (Primitive (Polyhedron faces)) = LinearBSPTree faces

LinearBSPTree ::  [Face] -> BSPTree
LinearBSPTree [face:faces]
   = BSPNode (PlaneOf face) (LinearBSPTree faces) (BSPLeaf OUT)
LinearBSPTree [] = BSPLeaf IN
```

Figure 3.7. *Lazy BSP tree algorithm.*

3.3.2 Lazy Set Operations

It remains to define the lazy set operations. We present only the lazy union operator $\cup_{lazy}$ referred to as `Union_lazy` in figure 3.7. The lazy intersection and lazy set difference are defined similarly. This subsection introduces the lazy union operation rather informally. For a better understanding the next subsection presents a mathematical derivation, which can be skipped at a first reading.

For any two polyhedral CSG objects A and B the lazy union operator must yield a BSP tree $\tau_{lazy}(A \cup^\star B) = \tau_{lazy}(A) \cup_{lazy} B$. Two conditions are sufficient:

1. $\tau_{lazy}(A \cup^\star B)$ is a BSP tree for $A \cup B$
2. $faces(\tau_{lazy}(A \cup^\star B)) \supseteq \partial(A \cup^\star B)$

To fulfill these conditions we develop the algorithm for the lazy union operation in two steps. First build a BSP tree $\tau_{lazy}(A \cup^\star B)$ by inserting the CSG object B into the BSP tree $\tau_{lazy}(A)$. Then augment the BSP tree $\tau_{lazy}(A \cup^\star B)$ with a superset of $\partial(A \cup^\star B)$.

We define the insertion of B into the BSP tree $\tau_{lazy}(A)$ by induction on the BSP tree structure. For the base case assume $\tau_{lazy}(A)$ is a cell and B lies completely inside the cell. If $\tau_{lazy}(A)$ is an IN cell, i.e., completely inside the represented object, then the union with object B does not change the classification of the cell. Conversely, if $\tau_{lazy}(A)$ is an OUT cell, i.e., completely outside the represented object, then the union with object B is given by the BSP tree representing B.

For the induction step take a BSP node with partitioning plane h and child trees $\tau_{lazy}(A_{h,in})$ and $\tau_{lazy}(A_{h,out})$. Splitting the CSG object B on the partitioning plane h yields the two CSG objects $B_{h,in}$ and $B_{h,out}$. The resulting parts are inserted in the corresponding child BSP trees on either side of the partitioning plane by recursively applying the lazy union operation:

$$\begin{aligned} \tau_{lazy}(A_{h,in} \cup^\star B_{h,in}) &= \tau_{lazy}(A_{h,in}) \cup_{lazy} B_{h,in} \\ \tau_{lazy}(A_{h,out} \cup^\star B_{h,out}) &= \tau_{lazy}(A_{h,out}) \cup_{lazy} B_{h,out} \end{aligned}$$

Forming a new tree with $\tau_{lazy}(A_{h,in} \cup^\star B_{h,in})$ and $\tau_{lazy}(A_{h,out} \cup^\star B_{h,out})$ as the IN and OUT tree, respectively, yields the desired result $\tau_{lazy}(A \cup^\star B) = \tau_{lazy}(A) \cup_{lazy} B$.

It remains to find a superset $faces(\tau_{lazy}(A \cup^\star B))$ of $\partial(A \cup^\star B)$. Such a superset is given by taking all faces of the lazy BSP tree for object A, denoted $faces(\tau_{lazy}(A))$, and all faces of all primitive polyhedra of object B. The set has the desired property because $faces(\tau_{lazy}(A)) \supseteq \partial A$ by definition of a lazy BSP tree and because the boundary of a CSG object is a subset of the boundary of all its primitives. Figure 3.8 gives the complete algorithm for the lazy union operation in functional code.

The algorithm is simplified by computing the superset of the boundary faces during the BSP tree construction. This is achieved by guaranteeing condition 2 (on page 28) for each partitioning plane h separately. Condition 2 is rewritten as

```
Union_lazy ::  BSPTree CSGObject -> BSPTree
Union_lazy (BSPNode plane facesOnPlane inTree outTree) csgObj
          = BSPNode plane newFacesOnPlane newInTree newOutTree
where
          newInTree          = Union_lazy inTree inCSG
          newOutTree         = Union_lazy outTree outCSG
          (inCSG,outCSG)     = SplitCSGObj plane csgObj
          objectFacesOnPlane = Intersection plane csgObj
          newFacesOnPlane    = facesOnPlane ++ objectFacesOnPlane

Union_lazy (BSPLeaf IN) csgObj   = BSPLeaf IN
Union_lazy (BSPLeaf OUT) csgObj  = BSPTree_lazy csgObj
```

Figure 3.8. *Lazy union of a BSP tree and a CSG object.*

2′. $\forall h : faces_h(\tau_{lazy}(A \cup^\star B)) \supseteq \partial(A \cup^\star B) \cap h$

where $faces_h(\tau_{lazy}(A \cup^\star B))$ denotes the faces of the BSP tree $\tau_{lazy}(A \cup^\star B)$ which lie on the partitioning plane h. The faces are given by the faces of the BSP tree $\tau_{lazy}(A)$ lying on the partitioning plane h and by all boundary faces of the primitive objects of the CSG object B lying on the partitioning plane h.

The lazy union algorithm in figure 3.8 refers to the faces of the BSP tree on the partitioning plane as `facesOnPlane` and to the boundary faces of the primitive objects of the CSG object, which lie on the partitioning plane, as `objectFacesOnPlane`. In the actual implementation the latter set of faces is efficiently determined during splitting the CSG object B with the partitioning plane h (see subsection 3.5.1).

Example 3.1 Figure 3.9 shows the union of a BSP Tree with a CSG object (a polyhedron). After inserting the polyhedron in the tree only that portion of its boundary shown in bold remains. These boundary faces are inserted in the OUT cell and subdivide it as shown. The algorithmic details of the subdivision are discussed in subsection 3.5.2.

3.3.3 Mathematical Derivation of the Lazy Union Operation

The lazy union operator introduced in the previous subsection can be be derived formally by observing that both BSP trees and CSG objects represent mathematical sets. Since this subsection is a little bit cumbersome to read we recommend to skip it at first reading.

Let A and B be two sets with domain Ω, and h a plane with inside and outside half-spaces H_{in} and H_{out}, respectively. Let $A_{h,in}$ and $A_{h,out}$ give the intersection of

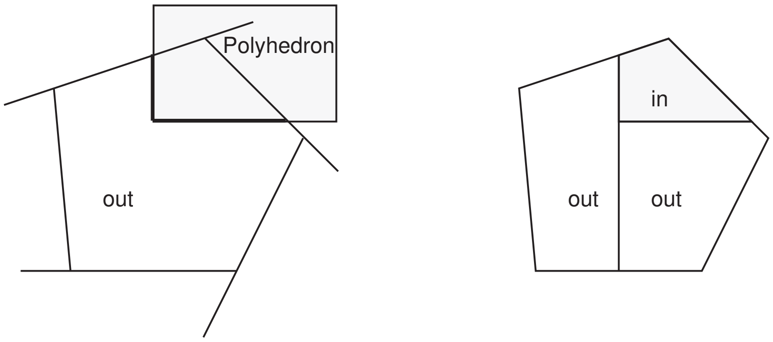

Figure 3.9. *Inserting a CSG object into an OUT cell according to an union operation.*

the set A with the half-spaces H_{in} and H_{out}, respectively, and analogously for $B_{h,in}$ and $B_{h,out}$. The symbol $\dot\cup$ denotes a disjoint union. Standard algebra[1] yields

$$\begin{aligned}
A \cup B &= (A \cup B) \cap (H_{in} \dot\cup H_{out}) \\
&= ((A \cup B) \cap H_{in}) \dot\cup ((A \cup B) \cap H_{out}) \\
&= ((A \cap H_{in}) \cup (B \cap H_{in})) \dot\cup ((A \cap H_{out}) \cup (B \cap H_{out})) \\
&= (A_{h,in} \cup B_{h,in}) \dot\cup (A_{h,out} \cup B_{h,out}) \qquad (3.2)
\end{aligned}$$

and

$$\emptyset \cup B = B \qquad (3.3)$$
$$\Omega \cup B = \Omega \qquad (3.4)$$

Equations 3.2 – 3.4 can be combined as

$$A \cup B = \begin{cases} \Omega & \text{if } A = \Omega \\ B & \text{if } A = \emptyset \\ (A_{h,in} \cup B_{h,in}) \dot\cup (A_{h,out} \cup B_{h,out}) & \text{if } A = A_{h,in} \dot\cup A_{h,out} \end{cases} \qquad (3.5)$$

where

$$B_{h,in} = B \cap H_{in}$$
$$B_{h,out} = B \cap H_{out}$$

The recursive definition of the union operation in equation 3.5 provides a basis for the lazy union operation. Before giving a more detailed explanation we want to compute a superset of the boundary of $A \cup^\star B$. Again A denotes a set and $faces(A)$ is a superset of its boundary.

[1] Equality with respect to a set of Lebesgue measure zero.

Assume the set B is defined by applying set operations to a set of basic sets P (e.g., B is a CSG object) and note that

$$\partial(B_1 \oplus B_2) \subseteq \partial B_1 \cup \partial B_2 \qquad (\oplus \in \{\cup, \cap, \setminus\})$$

Let B_1 and B_2 be components of B and define a function ∂_{all} recursively by

$$\begin{aligned} \partial_{all}(B_1 \oplus B_2) &= \partial_{all} B_1 \cup \partial_{all} B_2 \qquad (\oplus \in \{\cup, \cap, \setminus\}) \\ \partial_{all} P &= \partial P \end{aligned}$$

Then $\partial_{all} B$ is the set of all boundaries of all basic sets from which B is composed. We obtain

$$\begin{aligned} \partial(A \cup^{\star} B) &\subseteq \partial A \cup \partial B \\ &\subseteq faces(A) \cup \partial_{all} B \end{aligned}$$

and for an arbitrary plane h

$$\partial(A \cup^{\star} B) \cap h \subseteq (faces(A) \cap h) \cup (\partial_{all} B \cap h) \tag{3.6}$$

Now let any set A be equivalent to the corresponding BSP tree $\tau_{lazy}(A)$, i.e.,

$$\begin{aligned} A &\equiv \tau_{lazy}(A) \\ A_{h,in} &\equiv \tau_{lazy}(A_{h,in}) \\ A_{h,out} &\equiv \tau_{lazy}(A_{h,out}) \\ \emptyset &\equiv OUT \\ \Omega &\equiv IN \end{aligned}$$

and replace $faces(A) \cap h$ with $faces_h(\tau(A))$. Especially note that because of above equivalence and equation 3.1

$$A \cup B \equiv \tau_{lazy}(A \cup B) \equiv \tau_{lazy}(A \cup^{\star} B)$$

Then equation 3.5 and 3.6 can be combined to form the following recursive algorithm:

Algorithm 3.1 (Lazy union algorithm)

$$\tau_{lazy}(A) \cup_{lazy} B = \begin{cases} IN & \text{if } \tau_{lazy}(A) = IN \\ \tau_{lazy}(B) & \text{if } \tau_{lazy}(A) = OUT \\ \tau_{lazy}(A \cup^{\star} B) & \text{if } \tau_{lazy}(A) = (h, faces_h, \tau_{lazy}(A_{h,in}), \tau_{lazy}(A_{h,out})) \end{cases}$$

where

$$\begin{aligned} \tau_{lazy}(A \cup^{\star} B) &= (h, faces_h \cup (\partial_{all} B \cap h), \tau_{in}, \tau_{out}) \\ \tau_{in} &= \tau_{lazy}(A_{h,in}) \cup_{lazy} B_{h,in} \\ \tau_{out} &= \tau_{lazy}(A_{h,out}) \cup_{lazy} B_{h,out} \\ B_{h,in} &= B \cap H_{in} \\ B_{h,out} &= B \cap H_{out} \end{aligned}$$

This is exactly the lazy union algorithm given in figure 3.8. Especially $\partial_{all} B \cap h$ denotes all faces of the primitive polyhedra of CSG object B which lie on the partitioning plane h.

3.3.4 Boundary Extraction

To complete the lazy b-rep algorithm in figure 3.5 it remains to define the function `BoundaryBSPTree`. Assume as input a BSP tree $\tau_{lazy}(A)$ augmented with a superset $faces(\tau_{lazy}(A))$ of the boundary ∂A. As result of the boundary extraction we want a BSP tree $\beta(A)$ augmented with the boundary faces of object A, i.e., a boundary BSP tree.

Observe that the boundary of a regularized set (a "real world solid") separates its inside from its outside. Hence a face is part of the object boundary, if and only if, it separates IN cells from OUT cells. Subsection 3.5.3 shows that we can ensure additionally that any face of $faces(\tau_{lazy}(A))$ which is a boundary face of A has an outward pointing normal. With these preconditions the following Lemma gives the solution to the problem of boundary extraction:

Lemma 3.2 *Let A be a polyhedral CSG object, $\tau_{lazy}(A)$ the corresponding lazy BSP tree, and $f \in faces(\tau_{lazy}(A))$ having an outwards pointing normal if $f \in \partial A$.*

Then f is a boundary face of A, if and only if, in the BSP tree $\tau_{lazy}(A)$ the face f faces only IN cells on its inside and only OUT cells on its outside.

The boundary faces are found by traversing the BSP tree. For every BSP node with partitioning plane h, IN tree $\tau_{lazy}(A_{h,in})$, OUT tree $\tau_{lazy}(A_{h,out})$ and a set of faces $faces_h(\tau_{lazy}(A))$ on the plane h perform the following process:

Insert every face $f \in faces_h(\tau_{lazy}(A))$ into the IN tree of the corresponding BSP node and split it on all partitioning planes. The parts of the face f which lie at the end of this process in an IN cell are exactly the parts which face an IN cell on their inside. Insert only these faces in the tree on the outside of face f and collect the parts which lie in an OUT cell. These parts are exactly the parts of the input faces with an OUT cell on their outside. Hence with lemma 3.2 the resulting parts of the original face f lie on the object boundary. The algorithm is described in functional code in figure 3.12. Before explaining the code we want to clarify the algorithm with two examples.

Note first that the algorithm can be understood as a double filtering process: first remove all parts of face f that do not face an IN cell at the inside, and from the remaining parts remove all parts that do not face an OUT cell at the outside.

Example 3.2 Figure 3.10 clarifies the double filtering process. Part (a) of the figure shows a CSG object defined as $Object = (Object1 \setminus^{\star} Object2) \cup^{\star} Object3$. In (b) the corresponding BSP tree is pictured. Consider the face labeled $Face$. To extract parts from it lying on the boundary the face is first filtered down the subtree on its inside. In this case this is the IN tree since the face and the partitioning plane h_1 have the same orientation. Figure 3.10 (c) shows that the part labeled $Face_1$ ends in an IN cell and the part labeled $Face_2$ ends in an OUT cell. Hence only $Face_1$ is used for the second filtering step. Now $Face_1$ is inserted into the subtree on its outside (here the OUT tree) and the face is divided into three parts. Only two of them end in an OUT cell, namely $Face_{1.1}$ and $Face_{1.3}$. Indeed, these are exactly

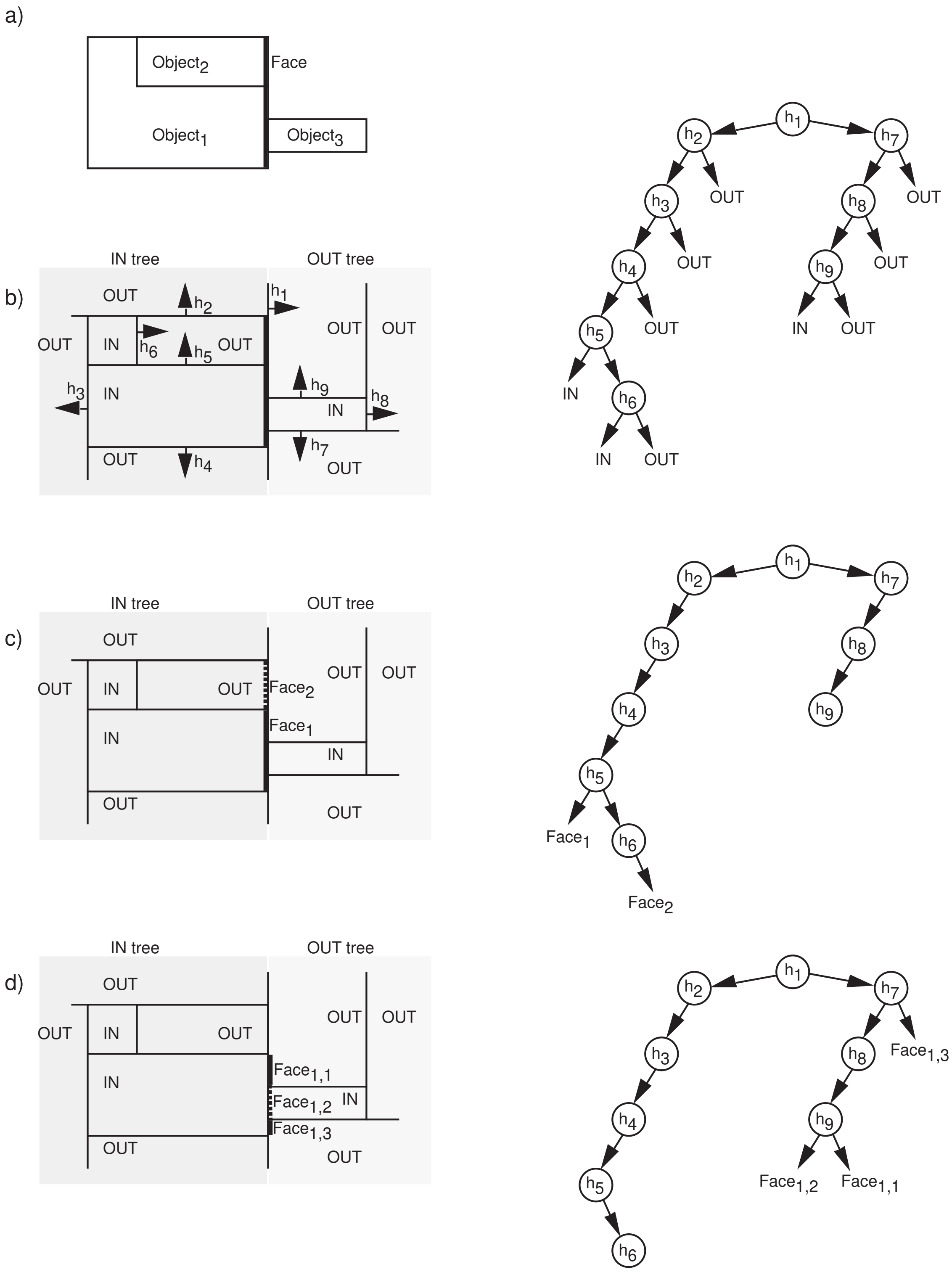

Figure 3.10. *Extracting part of the object boundary from a face.*

the fragments of the original face, which lie on the boundary of *Object* as lemma 3.2 claimed.

Example 3.3 Figure 3.11 illustrates the boundary extraction step for a face with normal orientation opposite to the partitioning plane. Then the OUT tree of the corresponding BSP node lies inside the face and the IN tree outside the face. Hence the face fragments on the object boundary are those parts which face an IN cell in the OUT tree and an OUT cell in the IN tree.

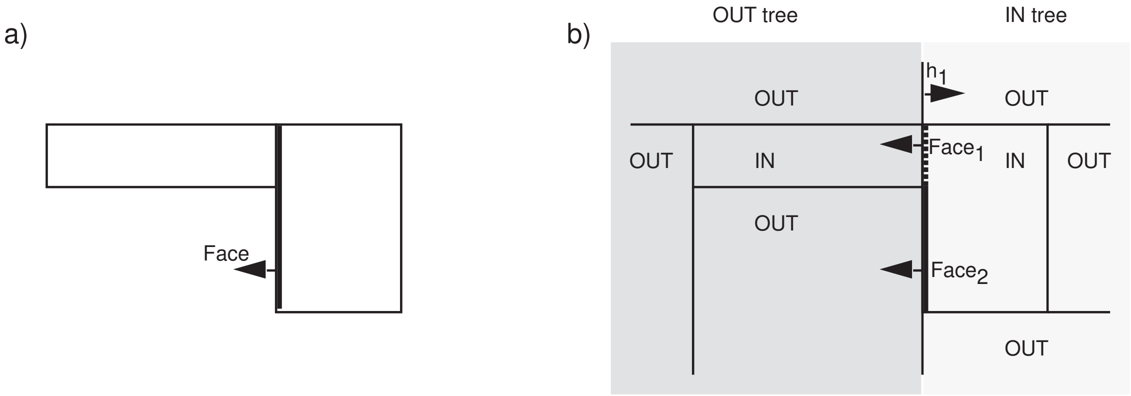

Figure 3.11. *Extracting part of the object boundary from a face with normal orientation opposite to the partitioning plane normal.*

Figure 3.12 summarizes the algorithm for boundary extraction. It is implemented by the function `BoundaryBSPTree` which traverses the BSP tree in in-order. For each BSP node the superset `faces` of the boundary faces is split into two sets `facesSameNormal` and `facesOpNormal` of faces with normal orientation equal and opposite, respectively, to the plane normal. The function `SingleSideExtractBoundary` determines then for all faces of these sets the fragments lying on the object boundary.

Input to the function `SingleSideExtractBoundary` are a list of coplanar faces with identical normal orientation and two BSP trees lying on the inside and outside of the faces, respectively. The result of the function are all parts of the input faces which lie on the object boundary. The function `SingleSideExtractBoundary` filters first all faces down the BSP tree on their inside and retains the face fragments reaching an IN cell. This is done by the function `InsertInCells` with the leaf class `IN` as first argument. The remaining parts of the face are then filtered down the BSP tree on their outside (function `InsertInCells` with first argument `OUT`) and the face fragments landing in OUT cells are returned.

The straightforward function `InsertInCells` is defined in figure 3.13. Insert a list of faces into a BSP tree by splitting them with the partitioning planes of the BSP tree. The parts of the faces lying inside and outside the partitioning plane are inserted recursively into the corresponding IN tree and OUT tree, respectively. If a

```
BoundaryBSPTree ::  BSPTree -> BSPTree

BoundaryBSPTree (BSPLeaf class) = BSPLeaf class
BoundaryBSPTree (BSPNode plane faces inTree outTree)
    = BSPNode plane (boundsSameNormal ++ boundsOpNormal) newInTree newOutTree
where
    SameNormal plane face           = (PlaneNormal plane) == (FaceNormal face)
    (facesSameNormal,facesOpNormal) = SplitListWith (SameNormal plane) faces
    boundsSameNormal = SingleSideExtractBoundary inTree outTree facesSameNormal
    boundsOpNormal   = SingleSideExtractBoundary outTree inTree facesOpNormal
    newInTree        = BoundaryBSPTree inTree
    newOutTree       = BoundaryBSPTree outTree


SingleSideExtractBoundary ::  BSPTree BSPTree [Face] -> [Face]

SingleSideExtractBoundary insideTree outsideTree faces = facesOnBoundary
where
    facesInIN_Cells = InsertInCells IN insideTree faces
                    // Face fragments which lie in IN cell of insideTree
    facesOnBoundary = InsertInCells OUT outsideTree facesInIN_Cells
                    // Face fragments which lie in OUT cell of outsideTree
```

Figure 3.12. *Extracting a boundary BSP tree from a BSP tree augmented with a candidate set of the boundary.*

```
InsertInCells ::  LeafClass BSPTree [Face] -> [Face]

InsertInCells _ _ [] = []            // no face reaches this (sub)tree
InsertInCells wantedClass (BSPLeaf class) faces
   | class == wantedClass = faces
   = []
InsertInCells wantedClass (BSPNode plane _ inTree outTree) faces
   = faceFragmentsInside ++ faceFragmentsOutside
where
   (facesInside,facesOutside) = UnZipWith (SplitFace plane) faces
   faceFragementsInside = InsertInCells wantedClass inTree facesInside
   faceFragmentsOutside = InsertInCells wantedClass outTree facesOutside
```

Figure 3.13. *Inserting a set of faces into a BSP tree and returning fragments which reach a cell of the specified leaf class.*

subtree is not reached by any face the recursion stops. The faces reaching a cell of the specified leaf class are returned.

3.4 Merging B-rep Algorithm

A second approach for transforming a CSG object $A = A_1 \oplus^\star A_2$ into a boundary BSP tree $\beta(A)$ is to generate recursively BSP trees from the child objects A_1 and A_2 and to merge them with the set operation $\oplus^\star$.

Naylor, Amanatides, and Thibault [NAT90] describe an algorithm to merge two BSP trees $\tau(A_1)$ and $\tau(A_2)$ by inserting the tree $\tau(A_2)$ into the tree $\tau(A_1)$. Binary partitioning the tree $\tau(A_2)$ in turn involves inserting the binary partitioners of $\tau(A_1)$ into $\tau(A_2)$. The algorithm is efficient but fairly complex.

We present here a simpler though less efficient solution. An optimal solution is not necessary, since Triage Polygonization itself does not merge BSP trees to polygonize a quasi-convolutionally smoothed object, but uses the more efficient lazy b-rep algorithm instead. However, the merging of BSP trees is necessary to polygonize the example scenes from section 2.5.

A straightforward solution to the problem of merging BSP trees is achieved by recognizing that a BSP tree $\tau(A)$ represents the object A as a union of its IN cells. Furthermore each IN cell is a (convex) polyhedral object. Hence with our model a BSP tree is a CSG object formed as a union of primitive convex polyhedra.

The merging operations are handled by the lazy set operations. The union of two BSP trees, e.g., is computed with the lazy union operation as shown in figure 3.14.

```
Union_merged' ::  BSPTree BSPTree -> BSPTree

Union_merged' tree1 tree2 = foldl Union_lazy tree1 (GetInCells tree2).
```

Figure 3.14. *Inefficient union of two BSP trees.*

Here we assume that the function `GetInCells` returns all IN cells of a BSP tree as polyhedra (i.e., primitive CSG objects). The `foldl` function iterates the lazy union operator `Union_lazy` with the initial BSP tree `tree1` over a list of CSG objects. In that way all IN cells of the second BSP tree are inserted in the first BSP tree yielding the union of both BSP trees.

However, we recognized that a complex CSG object usually gets heavily fragmented into IN cells if represented as a BSP tree. Recall that the lazy union operation inserts every face of a primitive polyhedral object into the tree as a candidate face for the boundary of the represented object. Therefore the resulting BSP tree is augmented with a disproportional large superset of the boundary of the represented object. We obtain a more efficient algorithm by merging the BSP trees without explicitly representing faces. Instead a candidate set for the object boundary is inserted in the tree in a post-processing step. Finally the boundary BSP tree is produced by extracting all boundary faces. The resulting *merged b-rep algorithm* is given in figure 3.15.

```
BRep_merged :: CSGObject -> BSPTree

BRep_merged csgObject
   = (BoundaryBSPTree o InsertCandidateFaces o BSPTree_merged) csgObject
```

Figure 3.15. *Merged b-rep algorithm.*

```
BSPTree_merged :: CSGObject -> BSPTree

BSPTree_merged (Union obj1 obj2)
   = Union_merged (BSPTree_merged obj1) (BSPTree_merged obj2)
BSPTree_merged (Intersection obj1 obj2)
   = Intersection_merged (BSPTree_merged obj1) (BSPTree_merged obj2)
BSPTree_merged (SetDifference obj1 obj2)
   = SetDiff_merged (BSPTree_merged obj1) (BSPTree_merged obj2)
BSPTree_merged (Primitive (Polyhedron faces)) = LinearBSPTree faces
```

Figure 3.16. *Merged BSP tree algorithm.*

Here the function `BSPTree_merged`, given in figure 3.16, computes an unaugmented BSP tree. The set operations `Union_merged`, `Intersection_merged`, and `SetDiff_merged` on BSP trees are defined analogously to the merged union in figure 3.14 except that they use lazy set operations producing unaugmented BSP trees. The function `LinearBSPTree` was already given in figure 3.7.

Lazy set operations producing unaugmented BSP trees are easily achieved by taking the corresponding lazy set operation (e.g., the lazy union operation in figure 3.8) without defining the list `newFacesOnPlane` of boundary faces for the new tree. The parts of the algorithm computing the superset of boundary faces are then not anymore needed.

It remains to define the function `InsertCandidateFaces`, which computes a candidate set of boundary faces for the boundary extraction. Note that every boundary face must lie on a partitioning plane and every CSG object is bounded. We get all possible boundary faces of a CSG object by pushing its bounding box (a polyhedron) down the tree. Every time the polyhedron is intersected by a partitioning plane the intersection forms a candidate face for the boundary. The parts of the split polyhedron inside and outside the partitioning plane are inserted recursively into the left and right subtree, respectively. The recursion ends if the bounding box reaches a leaf in the tree. Figure 3.17 gives the algorithm in functional code.

```
InsertCandidateFaces ::  BSPTree Polyhedron -> BSPTree
InsertCandidateFaces leaf=:(BSPLeaf _) _ = leaf
InsertCandidateFaces (BSPNode plane _ inTree outTree) polyhedron
   = BSPNode plane [newFace] newInTree newOutTree
where
   (inPolyhedron,outPolyhedron) = SplitPolyhedron plane polyhedron
   newFace                      = Intersection plane polyhedron
   newInTree                    = InsertCandidateFaces inTree inPolyhedron
   newOutTree                   = InsertCandidateFaces outTree outPolyhedron
```

Figure 3.17. *Producing candidate faces for the boundary representation.*

As a final remark note that even though all IN cells of a BSP tree are disjoint, it is not trivial to form a new BSP tree from a subset of them.

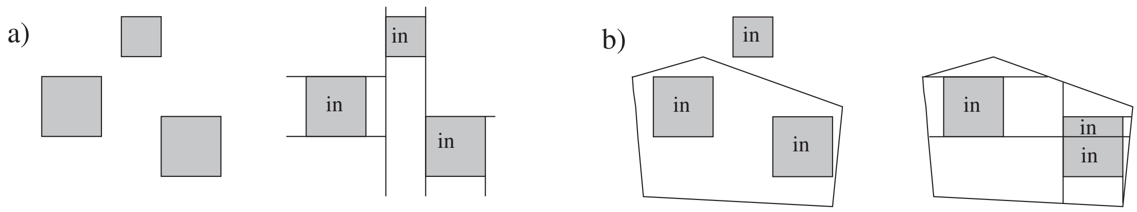

Figure 3.18. *Forming a BSP tree from a subset of its IN cells may lead to additional fragmentation.*

Example 3.4 Figure 3.18 (a) shows a union of three cubes and the corresponding BSP tree. In (b) only a subset of two cubes is taken. If their union is represented as BSP tree the rightmost square is fragmented into two IN cells. Hence taking a subset of a CSG object may yield a different binary space partitioning for the subset than if taking the object as a whole.

3.5 Implementation

The previous sections introduced two b-rep algorithms. For a better understanding of the actual algorithms we have postponed the definition of the various splitting algorithms used to this section. We also give some additional implementation details.

3.5.1 Splitting a CSG Object

A common operation of the b-rep algorithm is the splitting of a CSG object with a partitioning plane. To recognize the subtleties of the splitting operation note that it corresponds to the intersection of an object with the inside and outside half-spaces of the partitioning plane. Also recall that all objects are "real-world solids". This means that for example an object which only touches the partitioning plane must not be split into two objects. To produce regularized results the splitting operation must correspond to a regularized intersection with a half-space.

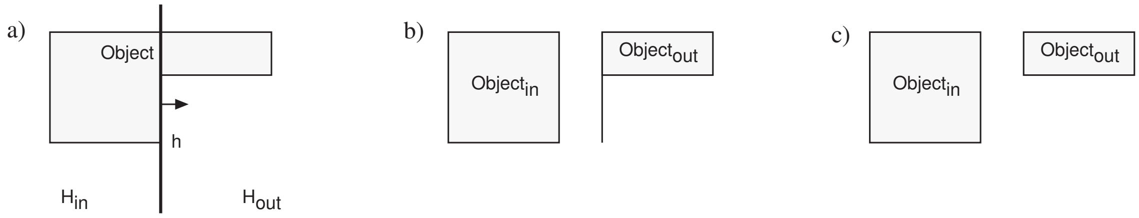

Figure 3.19. *Splitting an object with a partitioning plane implemented as a non-regularized operation (b) and as a regularized operation (c).*

Example 3.5 Figure 3.19 (a) shows an object $Object$ and a partitioning plane h with inside and outside half-spaces H_{in} and H_{out} respectively. The object is split into two parts $Object_{in}$ and $Object_{out}$ lying inside and outside the partitioning plane, respectively. In (b) the splitting operation is performed according to a non-regularized intersection with H_{in} and H_{out}. As result $Object_{out}$ has a dangling face. This error is corrected in (c) by using a regularized intersection instead.

The splitting operation for a CSG object can be reduced to splitting its primitive objects. As example consider a set $B = B_1 \cap^\star B_2$ and a partitioning plane h with inside and outside half-spaces H_{in} and H_{out}, respectively. Define $B_{in} = B \cap^\star H_{in}$, $B_{1,in} = B_1 \cap^\star H_{in}$, and $B_{2,in} = B_2 \cap^\star H_{in}$. Basic set algebra yields

$$\begin{aligned}
B_{in} &= B \cap^\star H_{in} \\
&= (B_1 \cap^\star B_2) \cap^\star H_{in} \\
&= (B_1 \cap^\star H_{in}) \cap^\star (B_2 \cap^\star H_{in}) \\
&= B_{1,in} \cap^\star B_{2,in} \\
&= \begin{cases} \emptyset & \text{if } B_{1,in} = \emptyset \vee B_{2,in} = \emptyset \\ B_{1,in} \cap^\star B_{2,in} & \text{otherwise} \end{cases}
\end{aligned}$$

Hence B_{in}, the part of object B inside the partitioning plane, is found by splitting B's child objects B_1 and B_2 on the partitioning plane. If either of them lies outside the partitioning plane then B lies outside the partitioning plane as well. In this case

B_{in} is void. Otherwise B_{in} is the intersection of the inside the partitioning plane lying parts of the child objects. B_{out} is determined in the same manner. Figure 3.20 uses above equation to give an algorithm for splitting an intersection of two CSG objects.

```
SplitCSGObject ::  Plane CSGObject -> (CSGObject,CSGObject)

SplitCSGObject plane (Intersection bit1 bit2)
   = (inBit,outBit)
where
   (inBit1,outBit1) = SplitCSGObject plane bit1
   (inBit2,outBit2) = SplitCSGObject plane bit2
   inBit            = if ((IsVoid inBit1) || (IsVoid inBit2)) VoidObject
                         (Intersection inBit1 inBit2)
   outBit           = if ((IsVoid outBit1) || (IsVoid outBit2)) VoidObject
                         (Intersection outBit1 outBit2)
```

Figure 3.20. *Splitting an intersection of two CSG objects.*

The splitting of a union or set difference of CSG objects is performed similarly. The problem of splitting a CSG object is therefore reduced to that of splitting its primitive objects. In our object model these are convex polyhedra.

Section 2.4 mentioned that the internal representation of a polyhedral primitive is a list of its boundary faces. If all of its faces lie on one side of the partitioning plane the polyhedron is not split. The same is valid if one of the polyhedron's faces lies on the partitioning plane (since a polyhedral primitive is convex). Otherwise the polyhedron is split by partitioning all its faces. A new boundary face for the two bits of the split polyhedron is formed by the intersection of the partitioning plane with the polyhedron. The corresponding algorithm is shown in figure 3.21.

The algorithm for splitting a (convex) polyhedron first classifies all faces with the partitioning plane. A face is either inside, outside, or on the plane or it is intersected by the plane. All faces intersected by the plane are split into inside fragments (`inBits`) and outside fragments (`outBits`). The intersection of the polyhedron with the partitioning plane (`newFace`) is calculated by intersecting all the polyhedron's edges with the plane. The resulting new face is a boundary face of both the `inPolyhedron` (the other faces of which are all the inside faces and inside fragments) and the `outPolyhedron` (the other faces of which are all the outside faces and outside fragments).

We conclude with a remark: In subsection 3.3.2 we claimed that the boundary faces of all primitive objects of a CSG object lying on a partitioning plane can be determined during splitting the CSG object. This is indeed the case. Inspecting figure 3.21 reveals that the desired faces are given as `onF`. In the actual implementation the function `SplitPolyhedron` returns additionally these faces.

```
SplitPolyhedron ::  Plane Polyhedron -> (Polyhedron,Polyhedron)

SplitPolyhedron plane polyhedron=:(Polyhedron faces)
   | IsEmpty insideF  = (VoidPolyhedron,polyhedron)
   | IsEmpty outsideF = (polyhedron,VoidPolyhedron)
   = (inPolyhedron,outPolyhedron)
where
   (insideF,outsideF,onF,intersectedF) = ClassifyWith plane faces
   (inBits,outBits) = UnZipWith (SplitFace plane) intersectedF
   newFace          = Intersection plane polyhedron
   inPolyhedron     = Polyhedron [newFace:(inBits ++ insideF)]
   outPolyhedron    = Polyhedron [FlippedFace newFace:(outBits ++ outsideF)]
```

Figure 3.21. *Splitting a polyhedron.*

3.5.2 Inserting a CSG Object into a BSP tree

The heart of the b-rep algorithms introduced in this chapter is the insertion of a CSG object into a BSP tree according to a set operation. Figure 3.8 gives as an example the lazy union operation. A crucial part of the insertion operation is to retain and classify the boundary information of the inserted CSG object. We explain why it is necessary to retain and classify boundary information and show how this is achieved in our implementation.

A CSG object inserted in a BSP tree is split with partitioning planes. In this section we often consider only one of the resulting fragments and say the polyhedron is clipped with the partitioning plane. Depending whether we choose the fragment inside or outside the partitioning plane, clipping corresponds to the regularized intersection with the inside or outside half-space, respectively.

To see the importance of retaining and classifying the boundary information of a CSG object, insert it into a BSP tree. Once the CSG object reaches a cell, dependent on the cell type and the set operation, the cell is partitioned with the boundary faces of the object. This means the boundary faces completely inside the cell must be known (see figure 3.9). However, the remaining boundary information of the CSG object must not be forgotten. This fact is illustrated in figure 3.22, which shows a polyhedron clipped with all partitioning planes defining a cell. In the first case (a) the polyhedron encloses the cell and in the second case (b) it is disjoint from the cell. However, in both cases all its boundary faces are clipped off. If a polyhedron is only represented by its boundary faces these two cases can not be differentiated. Hence the result of a set operation between the object and the cell can not be determined.

The solution is to define with each clipping operation a new boundary face lying on the partitioning plane. These new boundary faces must be distinctive from the original boundary faces since they are not used to partition a cell. Hence an object with two types of boundary faces is required. The boundary faces of the clipped

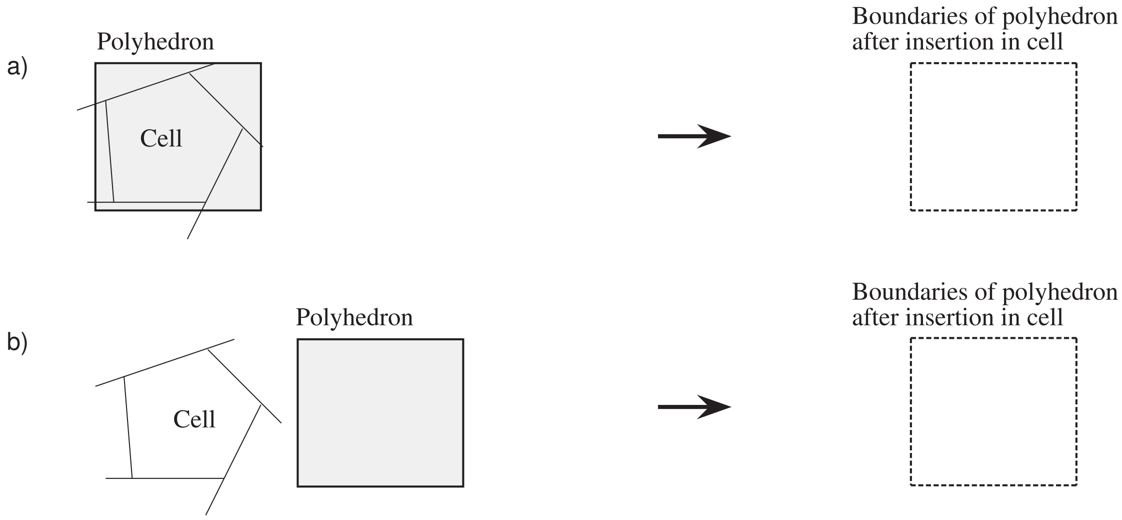

Figure 3.22. *Inserting a polyhedron into a cell (without defining domain boundaries). In (a) the polyhedron encloses the cell, in (b) it is disjoint from the cell. In both cases all its boundary faces are clipped off during insertion.*

object which lie on the partitioning plane are called *domain boundaries*. The original boundary faces are called *object boundaries*. Since a CSG object is composed from convex polyhedra it is enough to change the data type for a polyhedron accordingly.

We call a polyhedron with the above two types of boundary faces a *restricted polyhedron*, since it is restricted to a domain specified by a set of partitioning planes. The corresponding data structure is:

```
:: RPolyhedron = RPolyh [Face] [Face]
                // RPolyh objectBoundaries domainBoundaries
```

With theorem A.8 the boundary of the inside fragment of a polyhedron P split with a partitioning plane h is given as all boundary faces lying inside the corresponding half-space H_{in}, the part of the partitioning plane h lying inside the polyhedron, and those faces of the polyhedron lying on the partitioning plane h with the same normal orientation as h. Figure 3.23 gives two examples how object and domain boundaries are affected by this result.

It can be seen that the type of domain boundaries does not change by splitting. However, figure 3.23 (b) shows that an object boundary lying on the partitioning plane becomes a domain boundary. The intersection of the partitioning plane with the interior of the polyhedron (in part (a) of the figure) is always a domain boundary. These results are translated into an algorithm to split a restricted polyhedron shown in figure 3.24.

Similar to the algorithm for splitting a polyhedron (see figure 3.21) we classify first all boundary faces of a restricted polyhedron (i.e., object and domain boundaries). If there are no boundaries on the outside of the partitioning plane the whole polyhedron lies inside of it. Boundary faces on the partitioning plane (`onOB`) become

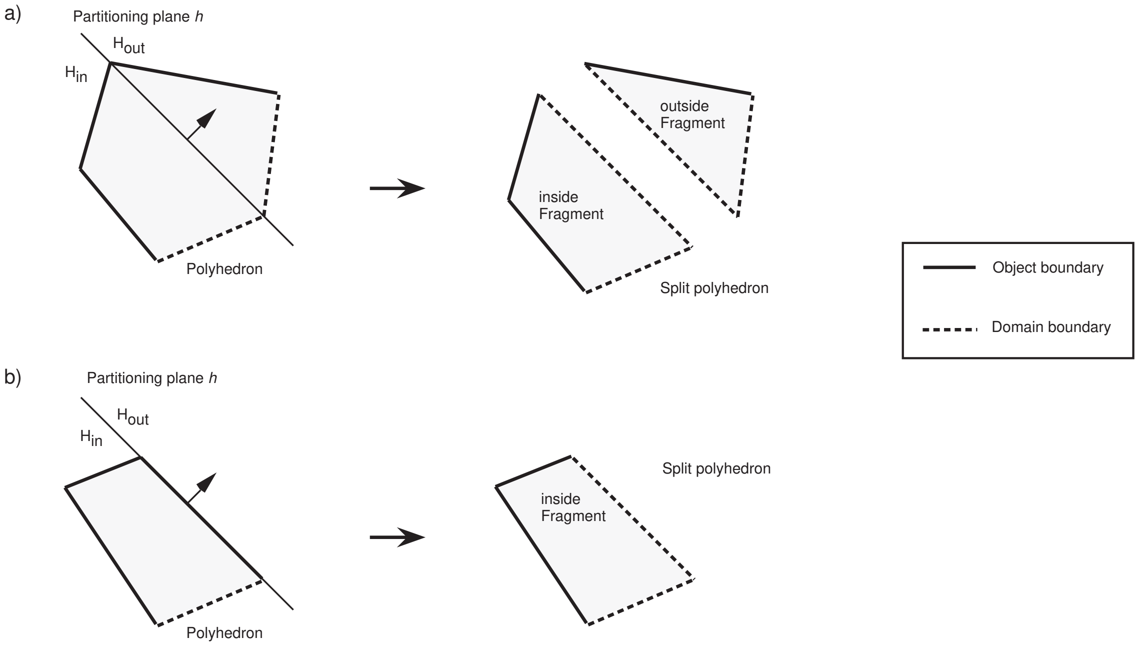

Figure 3.23. *Splitting a restricted polyhedron with a partitioning plane. The intersection of the partitioning plane with the polyhedron forms a domain boundary for the resulting fragments (a). If an object boundary of the polyhedron lies on the partitioning plane it becomes a domain boundary (b).*

domain boundaries (see figure 3.23 (b) for an example). A similar case arises if there are no boundary faces inside the partitioning plane.

In the remaining case the polyhedron is split on the partitioning plane. We split all its boundary faces and form with them two new polyhedra. The object boundaries of the polyhedron inside the partitioning plane are given by the object boundaries `insideOB` completely inside the plane and the bits `inBitsOB` of the split object boundaries. The same is valid for the domain boundaries. An additional domain boundary `newFace` is formed from the intersection of the partitioning plane with the polyhedron. The outside bit of the split polyhedron is defined in a similar manner.

3.5.3 Surface Normals

Subsection 3.3.4 presented an algorithm to extract the boundary ∂A of an object A from a BSP tree $\tau_{lazy}(A)$ augmented with a superset $faces(\tau_{lazy}(A))$ of ∂A. The algorithm assumed that all surface normals of boundary faces point to the outside of the object. Here we explain briefly how this is achieved.

The lazy BSP tree $\tau_{lazy}(A)$ is constructed with lazy set operations (see figure 3.5). This means that for example for the union of two objects A_1 and A_2 a superset

```
SplitRPolyhedron ::  Plane RPolyhedron -> (RPolyhedron,RPolyhedron)

SplitRPolyhedron plane rPolyhedron=:(RPolyhedron objBounds domBounds)
   | IsEmpty (outsideOB ++ outsideDB)  = (RPolyhedronOnInside,VoidRPolyhedron)
   | IsEmpty (insideOB ++ insideDB)    = (VoidRPolyhedron,RPolyhedronOnOutside)
   = (inRPolyhedron,outRPolyhedron)
where
   (insideOB,outsideOB,onOB,intersectedOB) = ClassifyWith plane objBounds
   (insideDB,outsideDB,onDB,intersectedDB) = ClassifyWith plane domBounds
   (inBitsOB,outBitsOB) = UnZipWith (SplitFace plane) intersectedOB
   (inBitsDB,outBitsDB) = UnZipWith (SplitFace plane) intersectedDB
   RPolyhedronOnInside  = RPolyhedron insideOB (onOB ++ domBounds)
   RPolyhedronOnOutside = RPolyhedron outsideOB (onOB ++ domBounds)
   newFace       = Intersection plane rPolyhedron
   inRPolyhedron = RPolyhedron (insideOB ++ inBitsOB)
                               [newFace:(insideDB ++ inBitsDB)]
   outRPolyhedron= RPolyhedron (outsideOB ++ outBitsOB)
                               [FlippedFace newFace:(outsideDB ++ outBitsDB)]
```

Figure 3.24. *Splitting a restricted polyhedron.*

$faces(\tau_{lazy}(A_1 \cup^{\star} A_2))$ of its boundary $\partial(A_1 \cup^{\star} A_2)$ is constructed as the union of the supersets $faces(\tau_{lazy}(A_1))$ and $\partial_{all} A_2$ of the boundaries ∂A_1 and ∂A_2, respectively. The following result applies:

Theorem 3.1 *Let A_1 and A_2 be two polyhedral objects and f a face on the boundary of $A_1 \cup^{\star} A_2$ with surface normal $\vec{n}_f$.*

Then there is an i, $i \in 1, 2$, such that f is on the boundary of A_i and f has an outward normal with respect to the object $A_1 \cup^{\star} A_2$, if and only if, it has an outward normal with respect to the object A_i.

A similar result is valid for the regularized intersection of two sets. For the regularized set difference $A_1 \backslash^{\star} A_2$ above result is valid if all faces on the boundary of A_2 are flipped.

Theorem 3.1 can be proven analytically for arbitrary sets with a 2-manifold boundary (given for polyhedral objects). However, it is easier to illustrate it with a simple example.

Example 3.6 Figure 3.25 (a) shows two CSG objects A_1 and A_2 and their boundary faces with surface normals. In (b) their regularized union $A_1 \cup^{\star} A_2$ is pictured. All boundary faces of the regularized union are boundary faces of either of the two child objects. Their surface normals do not change. Figure 3.25 (c) shows the set

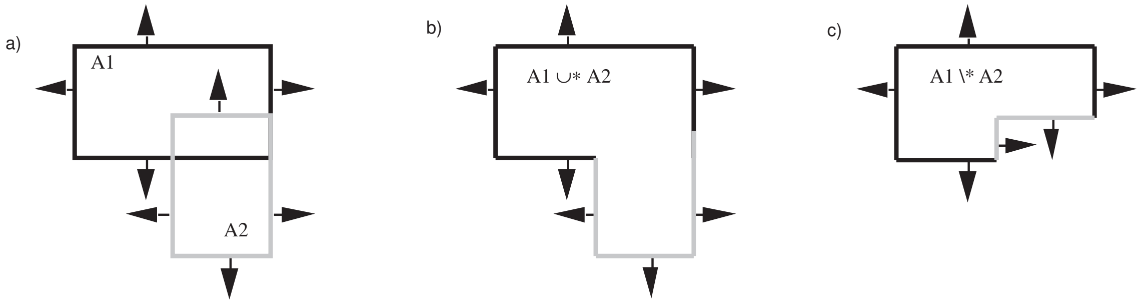

Figure 3.25. *Two CSG objects with boundary faces and surface normals (a), their regularized union (b) and their regularized set difference (c).*

difference of the CSG objects A_1 and A_2 from (a). All boundary faces of the set difference $A_1 \setminus^\star A_2$ belong to either A_1 or A_2. However, the normal orientation of the boundary faces of $A_1 \setminus^\star A_2$ which belong to A_2 is now reversed.

With the above theorem, and assuming that the surface normals of the primitive objects of a CSG object A are outward normals, all normals of the CSG object itself are outward normals as well.

We conclude with the following notes:

NOTE 1. For most algorithms it is computationally advantageous to know the orientation of a face with respect to the partitioning plane it lies on (e.g., the function `BoundaryBSPTree` in figure 3.12). This is achieved by either providing each face with a tag specifying its orientation or using two different face lists for faces parallel or anti-parallel to the partitioning plane.

NOTE 2. If different boundary faces of a CSG object can have different surface properties, the boundary faces must be stored explicitly and retained during BSP operations. The merged b-rep algorithm does not fulfill this condition. Hence it is only suited for objects with constant surface properties.

NOTE 3. The lazy set operations define duplicate faces for common boundaries and do not remove faces that after a set operation no longer belong to the boundary. This is illustrated in figure 3.26

3.6 Complexity Analysis

The previous sections introduced two b-rep algorithms for CSG objects. This section estimates their expected running time. We derive formulas for the best and average case running time of the lazy b-rep algorithm (figure 3.5) and a lower bound for its worst case running time. Since the merged b-rep algorithm (figure 3.15) is essentially

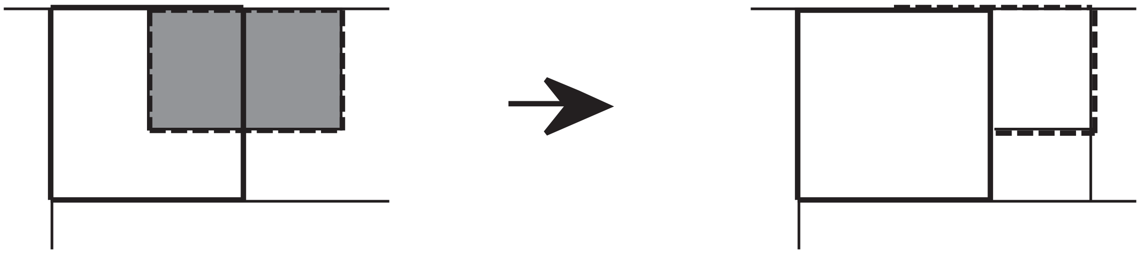

Figure 3.26. *BSP tree faces generated by lazy union.*

based on the lazyb-rep algorithm we expect for it a similar asymptotic behavior. We denote with n the size of a CSG object, i.e., n gives the number of half-spaces (faces) defining the polyhedral primitives of the CSG object. The size of a BSP tree is given by the number m of partitioning planes of the BSP tree. Note that the number of partitioning planes of a BSP tree is both proportional to the number of cells and the total number of nodes of the tree.

Best Case Running Time

The best case running time for the lazy b-rep algorithm is given for a scene which leads to a balanced BSP tree. Furthermore partitioning a CSG object must not occur (since partitioning leads to fragmentation and increases complexity). We assume here and in the following cases that all faces have constant complexity. The splitting algorithm for a CSG object of size n tests all faces of the objects's primitives against the partitioning plane (see section 3.5.1 on page 39). This takes $\Theta(n)$ time irrespective of whether the object is split or not[2].

The lazy set operations (see figure 3.8 for the lazy union operation as an example) between a BSP tree of size m and a CSG object of size n are computed by inserting the CSG object into the BSP tree. The CSG object is split with the partitioning plane of a BSP node and the resulting bits are recursively inserted into the subtrees. Since we assume a balanced tree the subtrees are half the size of the original tree.

If an inserted CSG object reaches a cell we compute for the object a BSP tree representation with the lazy BSP tree algorithm. The complexity of a lazy set operation in the best case is hence given by the recurrence relation

$$\begin{aligned} C^{best}_{\texttt{SetOp\_lazy}}(m,n) &= C^{best}_{\texttt{SetOp\_lazy}}(\frac{m}{2},n)+n \\ C^{best}_{\texttt{SetOp\_lazy}}(1,n) &= C^{best}_{\texttt{BSPTree\_lazy}}(n) \end{aligned} \tag{3.7}$$

which solves with theorem A.5 to

$$C^{best}_{\texttt{SetOp\_lazy}}(m,n) = n\log_2 m + C^{best}_{\texttt{BSPTree\_lazy}}(n)$$

[2]In case the object is not split the running time of the splitting algorithm can be improved to $\Theta(1)$ in most cases by defining a bounding box for each CSG object and testing the bounding box against the partitioning plane.

The lazy BSP tree algorithm (figure 3.7) transforms a CSG object $A = A_1 \oplus^\star A_2$ ($\oplus \in \{\cup, \cap, \setminus\}$) into a BSP tree by transforming the CSG object A_1 into a BSP tree and inserting the CSG object A_2 according to the set operation $\oplus$. In the best case A_1 and A_2 are equal-size and the complexity of the lazy BSP tree algorithm is given by the recurrence relation

$$\begin{aligned} C^{best}_{\texttt{BSPTree\_lazy}}(n) &= C^{best}_{\texttt{BSPTree\_lazy}}(\frac{n}{2}) + C^{best}_{\texttt{SetOp\_lazy}}(\frac{n}{2}, \frac{n}{2}) \\ C^{best}_{\texttt{BSPTree\_lazy}}(1) &= c \end{aligned} \tag{3.8}$$

Inserting equation 3.7 into equation 3.8 yields

$$C^{best}_{\texttt{BSPTree\_lazy}}(n) = 2C^{best}_{\texttt{BSPTree\_lazy}}(\frac{n}{2}) + \frac{n}{2}\log_2(\frac{n}{2})$$

which solves with theorem A.6 to

$$C^{best}_{\texttt{BSPTree\_lazy}}(n) = \Theta(n \log^2 n) \tag{3.9}$$

A similar argument shows that using a splitting operation which needs $\Theta(1)$ time to recognize that an object not intersected by the partitioning plane improves the best case time complexity of the lazy BSP tree algorithm to $\Theta(n \log n)$.

It remains to compute the complexity of the post-processing step for boundary extraction. For the best case we assume that the number of faces stored in the augmented BSP tree does not change. Also assume that all faces are equally distributed on the partitioning planes. This means each partitioning plane is augmented with $\Theta(1)$ faces.

In the best case a face inserted in a tree is not split. Since a tree of size m has a height of $\log m$ the function `InsertInCells` (see figure 3.13) needs $\Theta(n \log m)$ time to insert a list of n faces in a BSP tree of size m. The function `SingleSideExtractBoundary` with two given BSP trees of size m (a tree is balanced in the best case) and a face list of size n has then the same time complexity.

The boundary extraction algorithm `BoundaryBSPTree` (figure 3.12) calls the function `SingleSideExtractBoundary` two times with two BSP trees of half the original size m and a list with a constant number of faces. The recurrence relation

$$\begin{aligned} C^{best}_{\texttt{BoundaryBSPTree}}(m) &= 2C^{best}_{\texttt{BoundaryBSPTree}}(\frac{m}{2}) + 2\log(\frac{m}{2}) \\ C^{best}_{\texttt{BoundaryBSPTree}}(1) &= c \end{aligned}$$

gives the best case time complexity of the boundary extraction algorithm and solves with theorem A.6 to

$$C^{best}_{\texttt{BoundaryBSPTree}}(m) = \Theta(m) \tag{3.10}$$

The lazy b-rep algorithm (figure 3.5) is formed as the composition of the lazy BSP tree algorithm and the boundary extraction algorithm. Adding the complexities of equation 3.9 and equation 3.10 and noting that the size m of a BSP tree is equal to the size n of the corresponding CSG object (since no face splitting takes place) gives for the lazy b-rep algorithm a best case time complexity of

$$C^{best}_{\texttt{BRep\_lazy}}(n) = \Theta(n \log^2 n) \tag{3.11}$$

Worst Case Running Time

A tight upper bound for the worst case time complexity of the lazy b-rep algorithm proves to be difficult to find. However, a lower bound for the worst case time and space complexity of any b-rep algorithm is constructed by considering three orthogonal sets of n parallel flat cuboids. Taking the symmetric difference[3] of the union of each of the three sets of parallel cuboids results in a checkerboard pattern with cells alternately inside and outside the resulting object (see figure 3.27). The BSP tree for the resulting object has $\Theta(n^3)$ faces and cells. Therefore a lower bound for the worst case time and space complexity of the lazy b-rep algorithm is given by $\Omega(n^3)$.

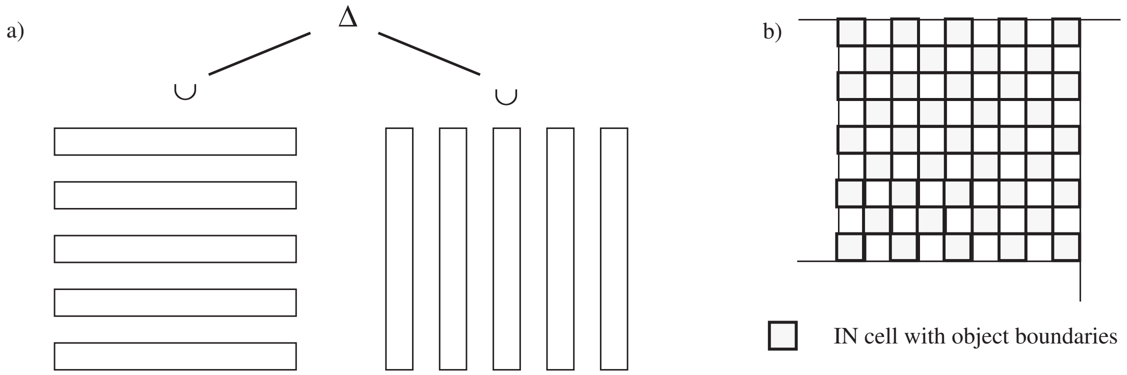

Figure 3.27. *A lower bound for the worst case time complexity of a b-rep algorithm is given for the symmetric difference of* 3 *orthogonal sets of n flat cuboids (a). The resulting BSP tree has* $\Theta(n^3)$ *faces and cells (b).*

Average Case Running Time

For the average case analysis assume that whenever a CSG object is split it is divided into equal-size parts. This means especially that the resulting BSP tree is balanced. This assumption is motivated from binary search trees. Knuth [Knu73] shows that the search of a binary search tree with N keys, inserted in a random order, will

[3]Note that for two sets A and B the symmetric difference is defined as $A \Delta B = A \setminus B \cup B \setminus A$, i.e., we can represent a symmetric difference with our model.

require only about $2 \ln N$ comparisons. Hence well-balanced trees are common and degenerate trees are very rare.

First consider a lazy set operation between a BSP tree of size m and a CSG object of size n. An example is given by the lazy union operation in figure 3.8. To perform a union operation a CSG object is split with the partitioning plane of the BSP tree and the resulting parts are inserted into the child trees of the BSP tree. Splitting a CSG object takes $\Theta(n)$ time. We suggest that in average the size of a CSG object increases at each split by an ad hoc factor[4] of $1 < \gamma < 2$, i.e., the child objects resulting from a splitting operation have an average size of $\frac{n}{2}\gamma$. If a CSG object reaches a cell, we assume it has a constant size. Then the time complexity of a lazy set operation is described by the recurrence relation

$$\begin{aligned} C^{avg}_{\mathtt{SetOp\_lazy}}(m,n) &= 2C^{avg}_{\mathtt{SetOp\_lazy}}(\frac{m}{2},\frac{n}{2}\gamma)+n \\ C^{avg}_{\mathtt{SetOp\_lazy}}(1,n) &= c \end{aligned}$$

which solves with theorem A.5 to

$$C^{avg}_{\mathtt{SetOp\_lazy}}(m,n) = \Theta(m + nm^{\log_2 \gamma})$$

The lazy BSP tree algorithm (figure 3.7) transforms a CSG object $A = A_1 \oplus^\star A_2$ ($\oplus \in \{\cup, \cap, \setminus\}$) into a BSP tree by transforming the CSG object A_1 into a BSP tree and inserting the CSG object A_2 according to the set operation $\oplus$. In the average case A_1 and A_2 are approximately equal-size and the complexity of the lazy BSP tree algorithm is given by the recurrence relation

$$\begin{aligned} C^{avg}_{\mathtt{BSPTree\_lazy}}(n) &= C^{avg}_{\mathtt{BSPTree\_lazy}}(\frac{n}{2}) + C^{avg}_{\mathtt{SetOp\_lazy}}(\frac{n}{2},\frac{n}{2}) \\ &= C^{avg}_{\mathtt{BSPTree\_lazy}}(\frac{n}{2}) + \left(\frac{n}{2}\right)^{\log_2 \gamma + 1} \\ C^{avg}_{\mathtt{BSPTree\_lazy}}(1) &= c \end{aligned}$$

which forms a geometric series and solves to

$$C^{avg}_{\mathtt{BSPTree\_lazy}}(n) = \Theta(n^{\log_2 \gamma + 1}) \tag{3.12}$$

It remains to compute the average time complexity of the post-processing step for boundary extraction. We give here only an upper bound for the time complexity. First note that the size of a CSG object inserted in a BSP tree increases by a factor γ in each step (we consider here the total size of all resulting fragments). If the CSG object is inserted in a BSP tree of size m (and height $\log_2 m$) the number of faces of the CSG object increases therefore by a factor of $\gamma^{\log_2 m}$. Hence the lazy BSP tree for a CSG object of size n is augmented with at most $n\gamma^{log_2 n}$ faces. We assume that most of the new faces define partitioning planes if reaching a cell of the

[4] If an object is split the total number of faces of both parts is at least two greater than the number of faces of the original object. Therefore we assume $\gamma > 1$. On the other hand usually not all faces of an object are split and therefore $\gamma < 2$.

BSP tree. Then the resulting BSP tree has a size of $m = n^{\log_2 \gamma + 1}$ and, assuming the faces are equally distributed over the tree, each partitioning plane is augmented with a constant number of faces. We insert these faces with the function `InsertInCell` into the BSP tree of size m. At each step the number of faces is again increased by a factor of γ. The time complexity of the insertion step[5] with n' faces on each partitioning plane is given by

$$\begin{aligned} C^{avg}_{\texttt{InsertInCell}}(m, n') &= 2C^{avg}_{\texttt{InsertInCell}}(\frac{m}{2}, \frac{n'}{2}\gamma) + n' \\ C^{avg}_{\texttt{InsertInTree}}(1) &= c \end{aligned}$$

which solves with theorem A.5 to

$$C^{avg}_{\texttt{InsertInTree}}(m, n') = \Theta(m + n' m^{\log_2 \gamma}) \tag{3.13}$$

Let n be the number of faces of the initial CSG object. The function `SingleSideExtractBoundary` inserts for each partitioning plane of a BSP tree a constant number of $n' = c$ faces in the IN tree and the resulting faces in the OUT tree. With a similar argument as for the size of the BSP tree the number of faces increases by splitting to at most $n'' = n^{\log_2 \gamma}$. Using equation 3.13 the average case time complexity of the function `SingleSideExtractBoundary` for a BSP tree of size m and $n' = c$ inserted faces is bounded to above by $O(m + n'' m^{\log_2 \gamma}) = O(m)$. For the latter equality we use the fact that the initial BSP tree has the size $m = n^{\log_2 \gamma + 1}$ and that $1 < \gamma < 2$ and therefore $\log_2^2 \gamma < \log_2 \gamma < 1$.

The boundary extraction step evaluates the function `SingleSideExtractBoundary` twice for each node of a BSP tree and calls itself recursively for both subtrees of the node. The recurrence relation

$$\begin{aligned} C^{avg}_{\texttt{BoundaryBSPTree}}(m) &= 2C^{avg}_{\texttt{BoundaryBSPTree}}(\frac{m}{2}) + m \\ C^{avg}_{\texttt{BoundaryBSPTree}}(1) &= c \end{aligned}$$

gives an upper bound for the average case time complexity of the boundary extraction algorithm and solves with theorem A.2 to

$$\begin{aligned} C^{avg}_{\texttt{BoundaryBSPTree}}(m) &= O(m \log_2 m) \\ &= O(n^{\log_2 \gamma + 1} \log_2 n) \end{aligned} \tag{3.14}$$

The lazy b-rep algorithm (figure 3.5) is formed as the composition of the lazy BSP tree algorithm and the boundary extraction algorithm. Combining equation 3.12 and equation 3.14 gives

$$C^{avg}_{\texttt{BRep\_lazy}}(n) = \Omega(n^{\log_2 \gamma + 1}) \tag{3.15}$$

[5] The given time complexity is again only an upper bound since it does not consider the case where the insertion stops before a cell of the BSP tree is reached. This case occurs if the list of inserted faces is empty.

as a tight lower bound for the average time complexity of the lazy b-rep algorithm and

$$C^{avg}_{\texttt{BRep\_lazy}}(n) = O(n^{\log_2 \gamma + 1} \log_2 n) \qquad (3.16)$$

as an upper bound for the average time complexity. Since the upper bound is not tight and since the factor $\log_2 n$ in the upper bound is small compared to the polynomial factor we suggest $\Theta(n^{\log_2 \gamma + 1})$ as average time complexity of the lazy b-rep algorithm.

3.7 Results

This section examines the performance of the lazy b-rep algorithms presented in this chapter. The merged b-rep algorithm is with our implementation based on the lazy b-rep algorithm and exhibits therefore a similar behavior. The main performance measure is the running time. We give evidence that the asymptotic running time is polynomial sub-quadratic as suggested in the complexity analysis in the previous section. Additionally results of interest are the form and quality of the generated BSP tree and of the boundary representation.

We implemented both algorithms in Clean 1.0. The following statistical results were obtained on an Apple Macintosh Quadra 700 with 8 MByte heap space and 1 MByte stack space.

The test data are the example scenes described in section 2.5. Recall that we deal in this section only with unrounded polyhedral objects and we therefore refer to the "n^3 Blended Cubes" as "n^3 Cubes". We examine results for these scenes separately since their geometry is more regular than that of the complex scenes.

Lazy B-rep Algorithm

Table 3.1 summarizes the statistical results obtained with the lazy b-rep algorithm. For each scene the algorithm generates a BSP tree. The table gives the number of half-spaces in the scene and for the generated BSP tree both its height, and the number boundary faces, IN cells, and OUT cells. It also shows the execution time of the lazy b-rep algorithm.

An interesting problem is the dependence of the running time of the b-rep algorithm on the size (number of half-spaces) of a scene. Figure 3.28 gives a scatter chart with the execution time plotted over the number of half-spaces of the scene. In the plot we use a double logarithmic scale because of the large size differences of the scenes. This means a straight line with slope n corresponds to function of the form $f(n) = n^k$. Note that the plots for the "CSG Example" scene and the "Variable Radius" scene fall on the same point. We omit the plot for the "Cube" scene because it is a primitive object and hence not representative for a complex scene.

Complex Scenes	#half-spaces	Height	#boundary faces	#IN cells	#OUT cells	$t_{execution}$ (in *secs*)
Cube	6	6	6	1	6	0.10
Cube In Cube	18	16	44	7	15	0.23
Stapler	47	13	109	19	52	0.81
CSG Example	72	13	118	24	37	0.58
Variable Radius	72	14	132	30	31	0.58
Hole Punch	203	32	460	92	161	4.18
Many Staplers	1128	30	3158	627	1193	30.53

Cube Scenes	#half-spaces	Height	#boundary faces	#IN cells	#OUT cells	$t_{execution}$ (in *secs*)
1 Cube	6	6	6	1	6	0.10
8 Cubes	48	9	48	8	31	0.35
27 Cubes	162	15	162	27	42	1.31
64 Cubes	384	15	384	64	119	2.93
125 Cubes	750	21	750	125	202	7.35
216 Cubes	1296	27	1296	216	261	16.16

Table 3.1. *BSP tree statistics for the lazy b-rep algorithm.*

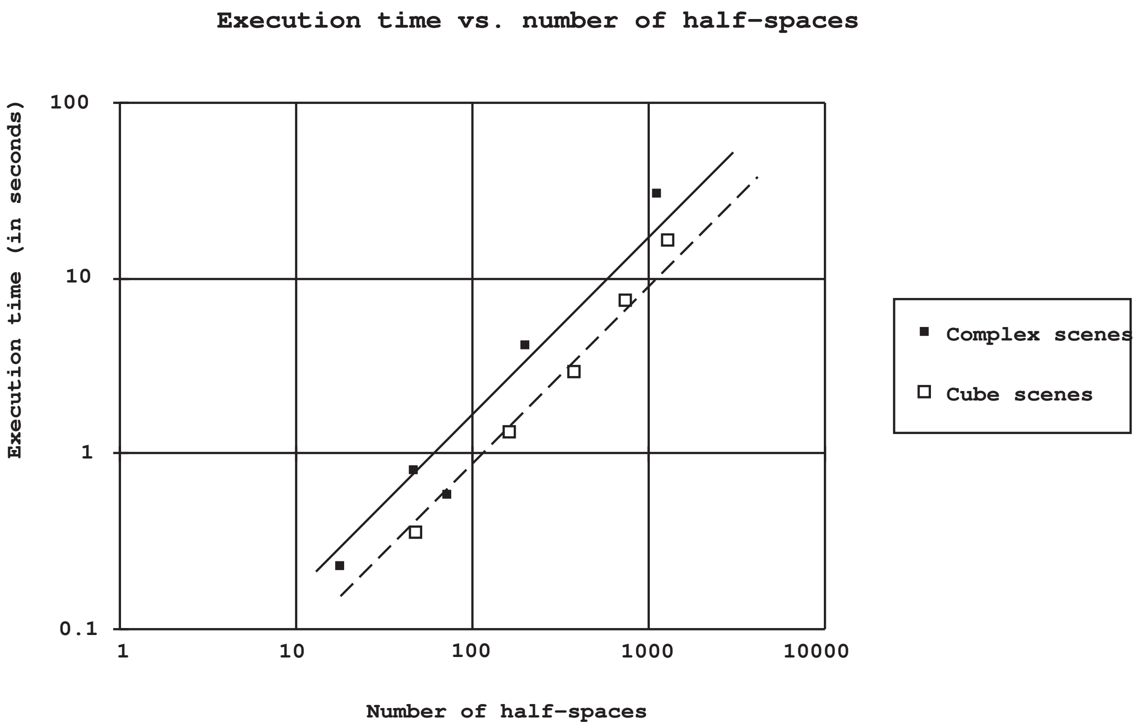

Figure 3.28. *Execution time for the lazy b-rep algorithm vs. number of half-spaces in the scene.*

For easier interpretation we insert for each data set a line with slope one obtained by a least square fit. This line corresponds to a linear function. The plot gives evidence that both data sets rise slightly faster than linear. According to the complexity analysis in the previous section we try to fit a function of the form

$$C : \#\text{half-spaces} \rightarrow t_{\text{execution}}$$
$$C(n) = \alpha n^k + \beta$$

with a least square fit to the plot. The best fitting curve, is the one with the smallest average squared prediction error e_{sq}. An optimal solution is found for $k = 1.1$ as the exponent of the complexity function. This suggests that the time complexity of the lazy b-rep algorithm for complex scenes is $\Theta(n^{1.1})$. Comparing this result with the time complexity $\Theta(n^{\log_2 \gamma + 1})$ obtained with the complexity analysis yields the splitting factor $\gamma = 1.07$.

Another derivation of γ is obtained by extending the argument given for the boundary extraction step of the average case analysis. This gives $n_f = \gamma^{3 \log_2 n}$ as an upper bound for the number of faces in the BSP tree for a CSG object of size n. Using this result computes (a lower bound for) the splitting factor $\gamma = n_f^{1/(3 \log_2 n)}$.

For the example scenes in table 3.1 the number of boundary faces n_f in the final BSP tree is given as "#boundary faces". The corresponding results for γ in the example scenes are given in figure 3.29.

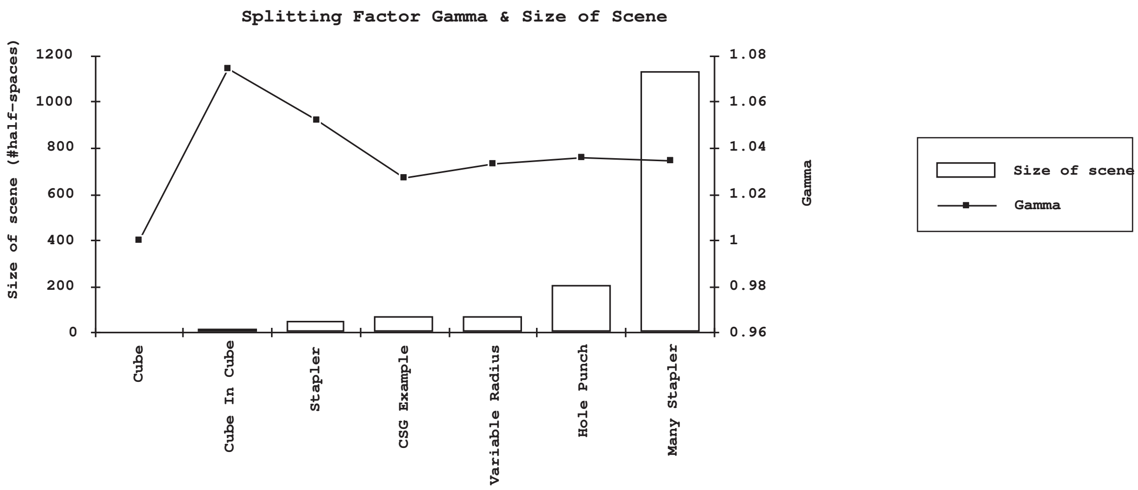

Figure 3.29. *A lower bound for the splitting factor γ.*

The figure suggests that during execution of the lazy b-rep algorithm the total size of a face increases by at least $\gamma = 1.03$ with each split. In other words, the probability that a face is actually split with a partitioning plane seems to be at least 3%. The splitting factor γ does not seem to depend on the size of the scene.

For the cube scenes no splitting of faces occurs. We performed a least square fit to the plot for the cube scenes in figure 3.28. Both a function with complexity class

$\Theta(n \log^2 n)$ and a function with asymptotic complexity $\Theta(n^{1.19})$ yielded a good least square fit. Note that the former function corresponds to the result of our best case analysis. The latter function suggests a wrong splitting factor $\gamma = 1.14$ (recall that no face splitting occurs and hence γ must be one). Possible reasons for this result are the regular (non-random) structure of the cube scenes and the evaluation order of the CSG object (see next subsection). Also is possible to give an example where the lazy b-rep algorithm has a time complexity of $\Theta(n^2)$ even though no splitting of faces occurs.

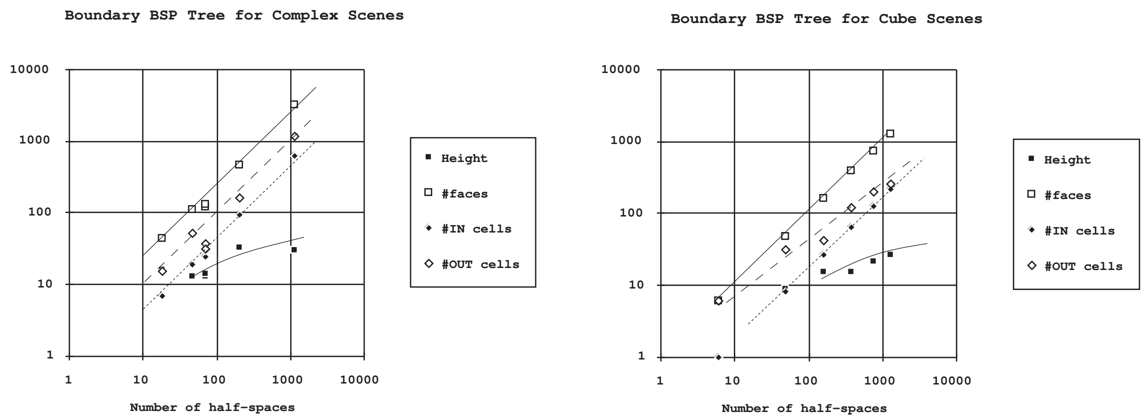

Figure 3.30. *BSP tree statistics for the lazy b-rep algorithm.*

Another interesting problem is the form and quality of the BSP tree produced with the lazy b-rep algorithm. Figure 3.30 gives for each scene the height of the generated BSP tree and the number of its boundary faces, IN cells, and OUT cells. The x-axis is given by the size (number of half-spaces) of the scene. Note that we use again a double logarithmic scale. The plot gives strong evidence that the height of the BSP tree increases less than linearly, possibly logarithmically, in the size of the scene. For the complex scenes the number of boundary faces, IN cells, and OUT cells seems to increase slightly faster than linearly in the size of the scene. For the cube scenes the behavior is linear, and the number of OUT cells grows less than linearly.

Merged B-rep Algorithm

The results for the merged b-rep algorithm are similar to that for the lazy b-rep algorithm. Since the merged b-rep algorithm makes rather inefficient use of the lazy b-rep algorithm we expect that it is at least by a constant factor slower. We found this to be generally true, though for some scenes the merged b-rep algorithm proved to be faster. This behavior can be explained by noticing that the algorithms evaluate the CSG object in different orders. The next subsection explains this phenomenon in more detail.

Order of Evaluation

The last results presented in this section concern the evaluation order of a CSG object by a b-rep algorithm.

Many CSG objects are modeled as a union of a large number of component objects. The b-rep algorithms introduced in this chapter first transform one of these components into a BSP tree and then insert the other component objects. Clearly the form of the BSP tree depends on the order of transformation and insertion, i.e., on the order of evaluation.

We give results only for the lazy b-rep algorithm and the "Hole Punch" scene. This scene is a good example since it is modeled from components of different scales. A simplified description of the "Hole Punch" scene is given in figure 3.31.

The "Hole Punch" is built from four main components:

1. **Base** Two large convex polyhedra.
2. **Top** three large thin convex polyhedra.
3. **Hinges** Two large non convex polyhedra.
4. **Metal Pins** Several small polyhedra of different shapes. Some constructed by intersection and set difference operations.

The order in which the lazy b-rep algorithm evaluates these components influences the performance of the algorithm. Table 3.2 shows the statistics for the final boundary BSP tree for different evaluation orders of the components 1–4.

The symbol "1-4-2-3" gives the evaluation order of the components. In this case the b-rep algorithm transforms the base first into a BSP tree and then inserts the metal pins, the top, and the hinges in this order into the tree. Though 16 permutations are possible we give only 6 evaluation sequences containing the extreme cases.

Evaluation order	Height	#boundary faces	#IN cells	#OUT cells	$t_{execution}$ (in s)
1-2-3-4	32	459	92	161	4.23
4-3-2-1	20	462	80	195	3.35
4-2-3-1	20	453	86	193	3.33
1-3-2-4	29	430	76	155	3.78
3-2-1-4	22	381	56	131	3.00
4-1-2-3	23	490	100	208	3.78

Table 3.2. *BSP tree statistics for the "Hole Punch" scene with different evaluation orders.*

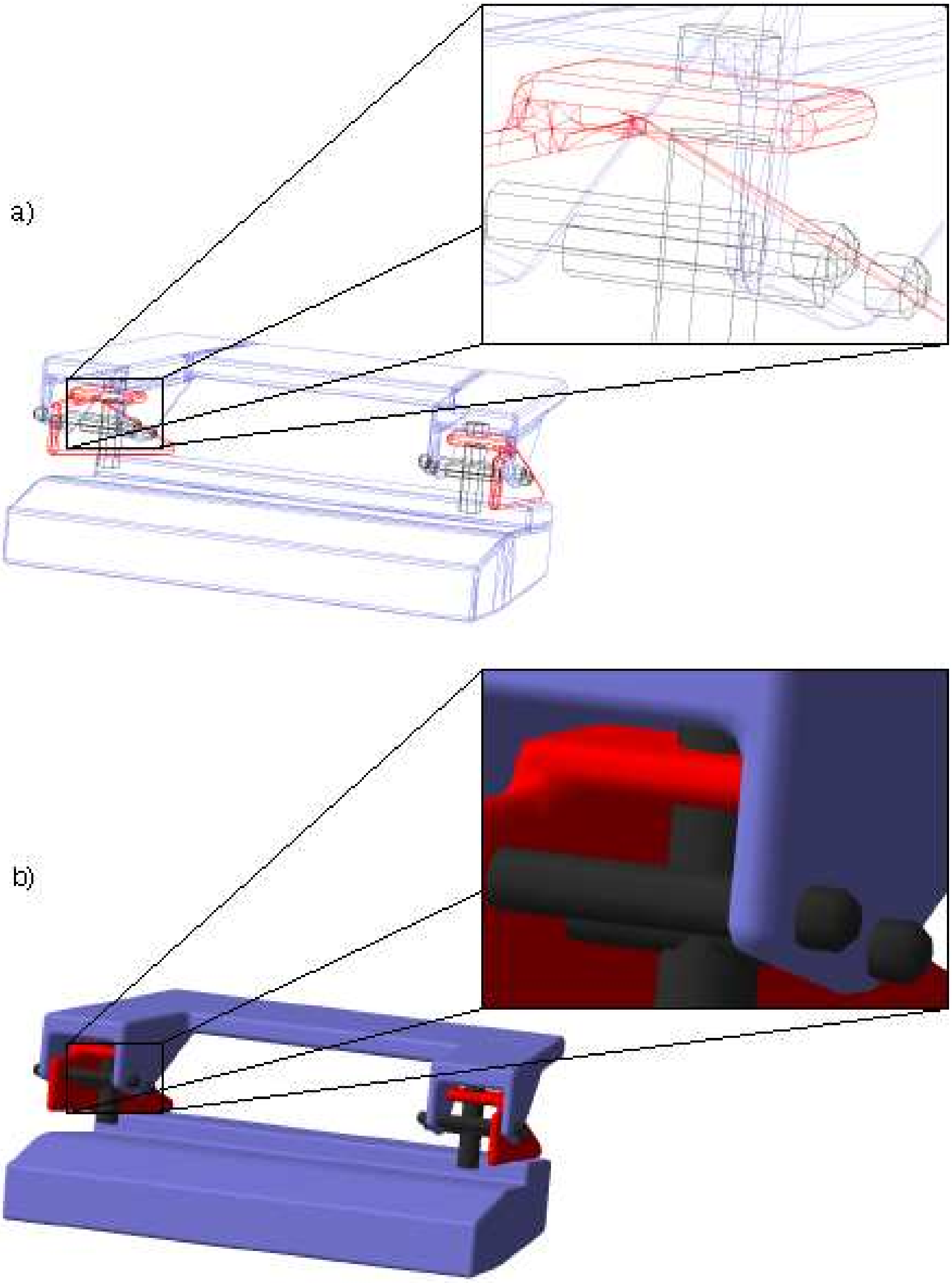

Figure 3.31. *Simplified description of the Hole Punch scene.*

The first table entry gives the tree for the evaluation order “1-2-3-4”. This is the order of the original scene definition. The evaluation starts with the big objects first, from the bottom to the top. The small metal pins are inserted last. Though this order seems intuitively efficient, it results in the highest tree and the slowest evaluation. The reversed order evaluates the smallest objects first and yields the shallowest tree but the highest fragmentation. The fastest evaluation and lowest fragmentation is given for the order “3-2-1-4”. Since the variation of the data is fairly small few conclusions can be drawn. However, we notice the tendencies that inserting the smallest objects last leads to a small fragmentation but also a generally slower evaluation. In above examples this is due to the fact that evaluating the big objects first gives generally a high and unbalanced tree. If we evaluate the big objects first and still can achieve a balanced tree (e.g., order “4-2-3-1”) a fast execution is likely. Starting with the smallest objects leads to high fragmentation but seems to achieve in that way also a better balancing of the tree and hence a fast evaluation of the algorithm.

3.8 Conclusion

This chapter presented two b-rep algorithms, which were used to compute the boundary representation of a polyhedral CSG object. The algorithms compute a BSP tree for the CSG object and augment it with a superset of its boundary. The object boundary is then extracted in a post-processing step leaving a BSP tree augmented with an explicit boundary representation of the CSG object.

As a by-product we developed set operations between two BSP trees and between a BSP tree and a CSG object. The latter set operation is performed by inserting the CSG object in the BSP tree and splitting it on its way down the tree with the partitioning planes.

Both a complexity analysis and test results suggested a polynomial sub-quadratic time complexity for the lazy b-rep algorithm which is dependent on the frequency of the splitting operations. We tested several example scenes and detected that in average the probability that a face is split on a partitioning plane is only about 3–7%. This resulted in a time complexity of about $\Theta(n^{1.1})$. Similar results are valid for the number of faces of the resulting b-rep, i.e., for the space complexity of the lazy b-rep algorithm. We showed, though, that in the worst case at least a cubic time and space complexity must be expected.

As performance enhancing improvements we suggested to compute for each CSG object and its child objects a bounding box. Then, if the CSG object is inserted into a BSP tree, the existence of an intersection with a partitioning plane can be tested in constant time. With a look ahead we mention another improvement suggested in chapter 6. Here fragmentation of the BSP tree is reduced by inserting the bounding box of a CSG object prior to the insertion of the object itself.

CHAPTER 4

Polygonization of Implicit Surfaces

This chapter reviews popular polygonization methods for implicitly defined surfaces. To get a basis for discussion we first introduce some notations. We then briefly review the literature before presenting and explaining four specific methods, which demonstrate useful principles in more detail. We discover a common framework for a general polygonization method for implicitly defined surfaces, which proves helpful in the development of Triage Polygonization in chapter 5. A final section lists some general quality criteria and reviews how the presented methods relate to them.

4.1 Notations & Definitions

An implicit surface is given as all points $x \in \mathbb{R}^3$ such that $\rho(x) = c$ for a function $\rho : \mathbb{R}^3 -> \mathbb{R}$ and a constant $c \in \mathbb{R}$. The resulting surface is called a c *iso-surface*. A polygonization method approximates an implicit surface with a mesh of polygons. The following sections reveal that all polygonization methods reviewed by us take data samples in the volume of interest and compute or approximate from them points on the iso-surface which are connected to form a polygon mesh. To avoid confusion we introduce here a set of notations that we use throughout the chapter. A data sample is referred to as a *voxel*. A convex polyhedral region bounded by voxels is called a computational *cell*, and voxels at the cell's corners are called *vertices*.

The iso-surface is implicitly specified by an underlying scalar function of three variables and a threshold value. For consistency with our polygonization problem in chapter 2 we take 0.5 as the threshold value and call the underlying scalar function ρ a *density field*. For simplicity we assume the function is continuous.

Generation of the iso-surface involves sampling of the density field and the definition of computational cells. For each cell determine whether the underlying function takes on the threshold value within the cell, and if so, approximate where the iso-surface lies. We shall call a vertex value *high* if its value is greater than or equal to the threshold, and *low* if not.

An *intersection point* is the point at which the iso-surface is estimated to cross the edge connecting two adjacent cell vertices that have different classification with respect to the threshold. Such intersection points become vertices of one or more *topological polygons*. These polygons specify the topology of the approximated surface but are usually not planar.

If a face F has n_F vertices $v_1 \dots v_{n_F}$ we define its *center* as the facial average

$$centre_F = \frac{\sum_{i=1}^{n_F} v_i}{n_F}$$

Analog we define the center of a cell.

If a face F has m_F intersection points $p_1 \dots p_{m_F}$ of its edges with the iso-surface we define its *centroid* as the average of the intersection points

$$centroid_F = \frac{\sum_{i=1}^{m_F} p_i}{m_F}$$

Analog we define the centroid of a cell.

We use the word *cell edge* for an edge of a polyhedral cell, and *polygon edge* for an edge of a polygonal approximation of the iso-surface.

4.2 Literature Review

Many published methods exist for finding a polygonal approximation to an implicitly defined surface. Though often written with a specific application in mind all methods that we review here solve the polygonizing problem as defined by us.

The methods of Lorenson and Cline [LC87] and Wyvill et al. [WMW86b] involve creating an array of cubes and evaluating the density field at each vertex. For each cube that exhibits differently classified vertices the method constructs a linear approximation to the surface.

Lorenson and Cline polygonize each cell based on a precomputed table of 15 topologically distinct high-low patterns of cell vertices. This table lookup method is devised for speed, at the occasional expense of a correct topology. In the original implementation the authors did not recognize ambiguities. Düurst [Düu88] showed that this could yield a discontinuity between cells.

Wyvill et al. [WMW86b] recognize ambiguities, and disambiguate by the facial average value; the vertices that agree with the central estimate are considered connected.

Ambiguities can be resolved implicitly by decomposition into simplices. This approach is taken by Koide et al. [KDK86] and Doi and Koide [DK91]. They decompose a cell into tetrahedra and interpolate linearly on each tetrahedral edge. Bloomenthal [Blo88, BW90] extends this method to an adaptive subdivision based on cubes.

Allgower and Gnutzman [AG87] give a more theoretical approach and yield error bounds based on the mesh size.

Petersen, Piper and Worsey [PPW87] use a tetrahedral mesh in conjunction with a Bernstein/Bezier representation. Hall and Warren [HW90] extend their method using an adaptive subdivision technique which maintains a tetrahedral honeycomb at all times.

Finally Gelder and Wilhelms [vGW94] give a thorough discussion of design-objectives of iso-surface algorithms, iso-surface generation and solving of ambiguities.

In the following subsections we present four selected algorithms in more detail. First we choose the Marching Cube method because it is popular and fast. Its implementation will serve as a benchmark for Triage Polygonization introduced in chapter 5.

Next we analyze the Soft Object method from Wyvill et al. [WMW86b]. This method is fast and eliminates the ambiguities of the Marching Cube method.

Hall's and Warren's algorithm [HW90] and Bloomenthal's method [Blo88] are good examples for adaptive solutions. The former algorithm performs a tetrahedral subdivision of space, whereas the latter one is interesting because it uses an octree representation.

4.2.1 Marching Cubes: A High Resolution 3D Surface Construction Algorithm

The Marching Cubes algorithm combines simplicity with high speed. Because of its popularity we chose it as a benchmark program for Triage Polygonization. We first describe the algorithm and then make a few comments about our own implementation.

The algorithm processes 3D data in scan-line order and builds a logical array of cubes. Each cube is created from eight voxels; four each from two adjacent slices. The algorithm determines how the surface intersects this cube, then moves to the next cube.

The iso-surface intersection is determined by first classifying the density values in the cube's vertices as high and low. Each edge with one high and one low vertex value is assumed to intersect the iso-surface once. The intersection point is approximated by linearly interpolating the density values in the vertices.

Since there are eight vertices in each cube and two values, *high* and *low*, there are $2^8 = 256$ ways the surface can intersect the cube. Lorenson and Cline use symmetries to reduce the number of patterns to 15 which are shown in figure 4.1[1].

The algorithm can be summarized as

[1] The cases 12 and 15 are reflective with respect to the xy-plane. This leaves 14 topologically distinct patterns (22 without inversed patterns) [LVG80].

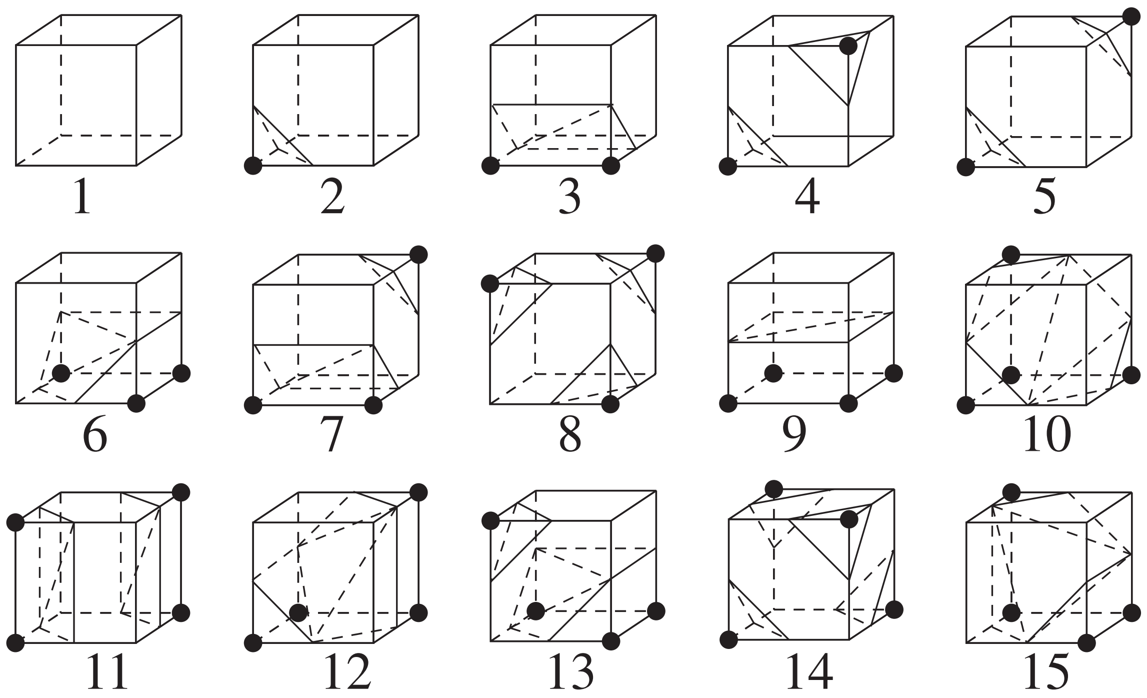

Figure 4.1. *Triangulated Cubes.*

- Scan two adjacent slices and create a cube from any four neighbors on each slice.
- Calculate an 8-bit index for the cube by classifying the eight density values at the cube vertices with respect to the iso-surface density.
- Using the index, look up the list of edges forming triangles from a precalculated table.
- Using the densities at each edge vertex linearly interpolate the iso-surface intersection.

The main disadvantage of the algorithm is that some patterns in figure 4.1 are topologically ambiguous as noted by van Gelder and Wilhelms [vGW94, pages 343 – 344]. This may produce a surface with a hole as pointed out by Düurst [Düu88]. Van Gelder and Wilhelms [vGW94, page 340] cite Baker [Bak89] and Kalvin [Kal91] for modifications that ensure continuity.

To simplify the algorithm in our implementation we only eliminate from the original 256 patterns those with high and low vertices swapped. This leaves us a table with 163 entries, where each pattern has at most four high values. Table 4.1 shows the number of remaining patterns.

We use the Marching Cubes algorithm to polygonize the surface of a quasi-convolutionally smoothed object. In our implementation we chose as cube size half the

Number of *high* vertices	0	1	2	3	4
Number of patterns	1	8	28	56	70

Table 4.1. *Number of patterns with less than 5 high vertices.*

rounding radius of the density object under consideration. This produces about the same detail as our Triage Polygonization method presented in the next chapter. Note that the rounding radius is the minimum resolution necessary to distinct between a rounded and unrounded corner of 90^0 angle.

To polygonize a scene we first polygonize the density objects with the marching cube algorithm. The polygonized objects and the primitive unrounded objects are then transformed into BSP trees. The scene is formed by merging the BSP trees according to the underlying set operations defined by the CSG object. Figure D.6 (b) gives as example a cube smoothed with varying rounding radii. More statistical results for the Marching Cubes algorithm are given in subsection 7.5.1 for comparison with Triage Polygonization.

4.2.2 Data Structure for Soft Objects

Wyvill, McPheeters and Wyvill report a polygonization method designed for soft objects [WMW86b, WMW86a, WWM87, BW90] but which can be readily applied to our problem.

The authors construct a polygon mesh in two distinct stages. In a first step they partition the space occupied by the iso-surface with a three dimensional cubic grid. To find the cubes intersected by the surface without scanning the whole the authors start with a set of seed cubes, at least one for every disconnected component[2]. Starting at the seed cubes, they track the surface by cell propagation. If a cube is intersected by the iso-surface the process continuous for each cube neighboring an intersected face. A hash table is used to prevent cells being revisited during recursion.

In the second stage the authors only deal with cubes which are intersected by the surface. They construct a local polygonal approximation to the iso-surface by linearly interpolating the intersection points of the iso-surface with the edges of the cube. The intersection points on a face are connected to polygon edges. Ambiguities are resolved by considering the center point of a face. Figure 4.2 illustrates the seven possible cases.

This calculation is consistent across adjacent cubes with shared edges. By tracing the natural successors of each polygon edge the authors construct topological polygons. Since the resulting topological polygons are in general not planar the

[2]This is easily achieved for the Soft Object data structure. For our problem a sufficient set of seed cubes can be constructed from the primitive objects of the scene.

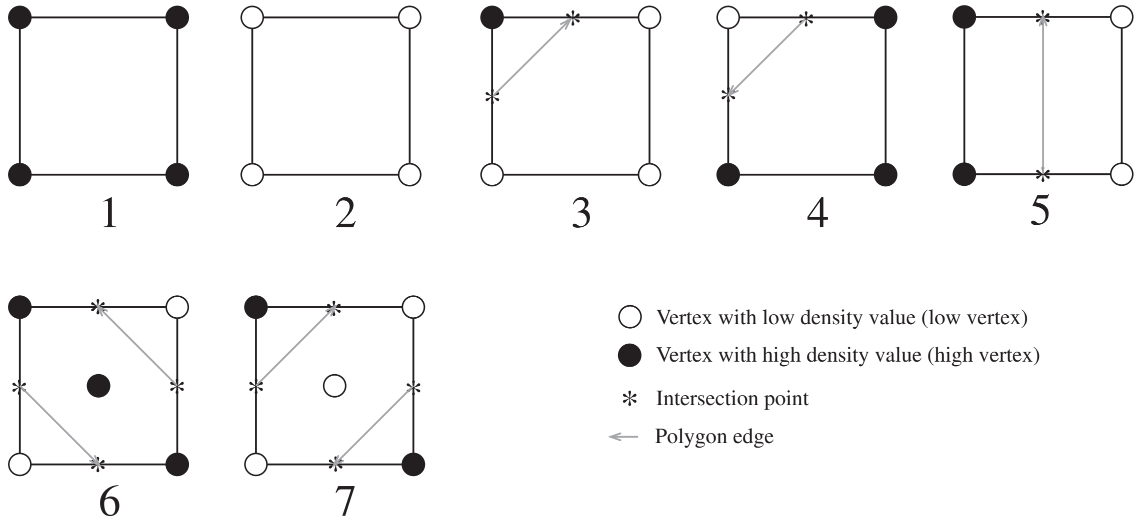

Figure 4.2. *Seven different cases for connecting intersection points.*

authors divide them into triangles by connecting each polygon vertex to the central average of the topological polygon.

4.2.3 Adaptive Polygonization of Implicitly Defined Surfaces

Hall and Warren [HW90] report an adaptive polygonization method that performs a tetrahedral subdivision of space. The authors maintain a tetrahedral honeycomb of space at all time. A polyhedral subdivision of space forms a honeycomb if every face is shared by at most two polyhedra. Because the recursive subdivision of a single tetrahedron might cause the honeycomb property to be lost, the method partially subdivides the neighbors of that tetrahedron to maintain the property. The adaptive subdivision algorithm decides independently for each tetrahedron in the honeycomb whether it should be recursively subdivided. The algorithm defers processing of tetrahedra not recursively subdivided until it has considered all tetrahedra. It then makes a second pass through the list of unsubdivided tetrahedra. For each tetrahedron, the algorithm checks each edge to see if it must be subdivided. The faces and polyhedra are then split according to the number of subdivided edges of each face.

The subspace polygonization algorithm takes as input a set of tetrahedra that forms a honeycomb for the volume of interest. For each edge the authors determine the intersection points with the iso-surface by successive linear interpolation. The resulting intersection points are connected by one or two triangles. To ensure continuity the authors compute the edge intersections once and store them into a hash table.

4.2.4 Polygonization of Implicit Surfaces

Bloomenthal [Blo88] introduces a numerical technique to approximate an implicit surface with a polygonal representation. The implicit function is adaptively sampled as it is surrounded by a spatial partitioning. The partitioning is represented by an octree, which may either converge to the surface or track it. A piecewise polygonal representation is derived from the octree. Bloomenthal reports three steps:

- Spatial partitioning: Bloomenthal considers two methods for sampling the implicit surface. The first method represents the implicit surface as an octree, which is a hierarchical partitioning of space formed by subdivision of cubes, beginning with a cube that bounds the surface. The octree converges to the surface by subdivision of those cubes that intersect the surface. A disadvantage is that small surface details may be missed by a large cube, resulting in a premature termination in the subdivision of the cube. This drawback is overcome by the second method which tracks the surface by cell propagation. This is the same technique as used by Wyvill et al. (see subsection 4.2.2).

  In both cases, the author evaluates the density function ρ at each of the cell's vertices. Only those cells that intersect the surface are retained in the partitioning. Bloomenthal determines the intersection points of the cell's edges with the iso-surface by root search. To ensure continuity between polygons Bloomenthal refers to Wyvill's method of storing the intersection points in a hash table. Alternatively he suggests keeping for each cell eight pointers to its vertices. As new cells are created, they must point correctly to shared vertices.

- Adaptive refinement of the octree: Bloomenthal improves the estimation of the surface by subdividing those cubes containing elements of high curvature.

- Polygonization of the octree nodes: The final surface approximation is obtained by polygonizing the octree nodes (final subdivision cells). For each cube to be processed, the intersection points are ordered, forming a convex polygon whose sides are each embedded in a cube face [WMW86b]; the process is local to each cube. Bloomenthal introduces a simple algorithm, illustrated in figure 4.3 to perform the three-dimensional ordering of intersection points. The ordering begins with any intersection point on the cube and proceeds towards the high vertex and then clockwise about the face to the right until another intersection point is reached.

The (topological) polygons resulting from this method are decomposed into triangles. Note that the adaptive subdivision may destroy the honeycomb property of the spatial partition. Bloomenthal ensures continuity between subspace polygons, by tracking the edges of the topological polygon along the more highly divided face (the light grey vertices in figure 4.4). He resolves ambiguities by taking the central average as additional sample.

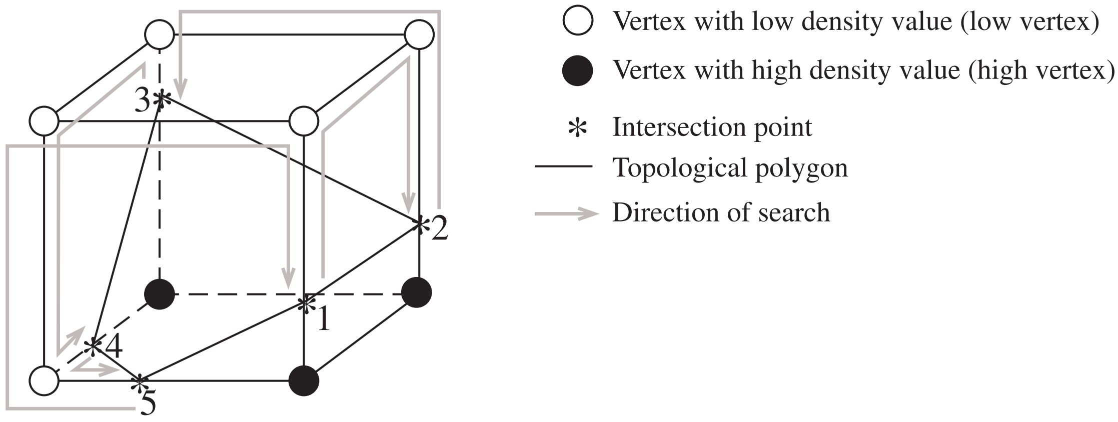

Figure 4.3. *Algorithm to order vertices.*

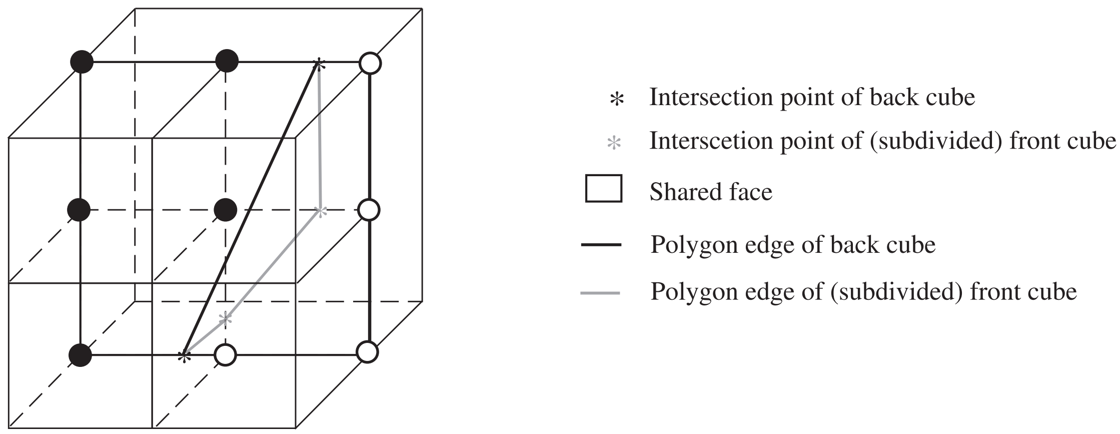

Figure 4.4. *To guarantee continuity Bloomenthal forms polygon edges by always tracking along the more highly divided face.*

4.3 Analysis of Polygonization Algorithms

In this section we analyze the above presented methods and extract a common framework. This will prove helpful for the development of an improved solution for our polygonization problem.

Comparing reviewed polygonization methods it can be seen that they all involve three aspects:

1. Polyhedral subdivision of space
2. Subspace polygonization (approximate iso-surface inside a cell)
3. Ensuring continuity

4.3.1 Space Subdivision

During subdivision of space most methods maintain a honeycomb, the 3D analog of a tessellation. The honeycomb guarantees that linear functions defined over a polyhedron form a continuous surface.

The simplest honeycomb used is an array of cubes. As noted by Bloomenthal [Blo88] vertex locations and face planes are computed more simply if the cells are identical and similarly oriented. In three dimensions, the only such cell that fills space is the cube. Also it enjoys a number of rotational symmetries, and divides into eight similarly oriented cubes. Note though, that a honeycomb can be maintained only by dividing all cubes of the array. An additional disadvantage is that the *high* and *low* vertices of a cubic cell can not be separated by a single plane. This may lead to ambiguities during the second stage of the corresponding polygonization algorithm and may ultimately result in discontinuities for the polygonized surface (see subsection 4.2.1).

In contrast, the vertices of a tetrahedron can always be separated by a single plane, thereby avoiding ambiguities during the polygonization. Also a tetrahedron can be subdivided into tetrahedra without subdividing its faces. This allows for a local subdivision of a tetrahedral honeycomb. The topic of tetrahedral subdivision for an adaptive polygonization is discussed in [HW90]. Tetrahedral subdivision in n-space is commonly called Dirichlet tessellation. Bowyer [Bow81] gives an efficient solution. A comprehensive bibliography of Voronoi diagrams, the dual of Dirichlet tessellation, and related structures is given by Aurenhammer [Aur91].

It is interesting that the existing literature never mentions the use of a space subdivision with general polyhedra. General polyhedra have the advantage that every polyhedral partition of space can easily be transformed in a honeycomb. The transformation is done by subdividing each face that faces more than one polyhedral face. After the subdivision of the face the resulting object is still a polyhedron, apart from that it has several coplanar faces. Figure 4.5 gives an example.

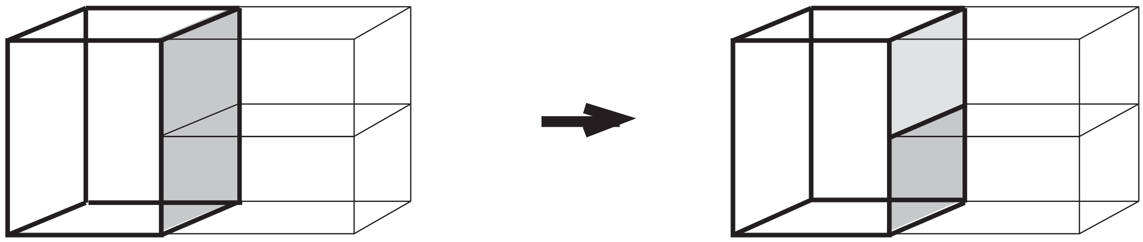

Figure 4.5. *Transforming a polyhedral subdivision into a honeycomb.*

4.3.2 Subspace Polygonization

Two different methods of subspace polygonization are used by the reviewed algorithms. Lorenson and Cline use a binary classification of the cells' vertex values to index a precomputed table yielding a set of polygons for the cell. The polygon vertices are given by the intersection points of the cell's edges with the iso-surface. The other methods first precompute these intersection points and use them to determine the iso-surface intersection with each face. By tracing the resulting edges topological polygons are formed. In a final step these are divided into planar polygons.

The iso-surface intersections of the edges can be determined by interpolation or by a root finder. Linear interpolation has the advantage that it is symmetric, which means neighboring cells, sharing the same edge, share the same intersection point. Omitting this condition leads to discontinuities of the resulting surface approximation. However, we can always achieve the condition by computing each intersection point only once and then referring to it by pointers or by a hash table.

4.3.3 Ensuring Continuity

The third aspect of a polygonization algorithm is to guarantee surface continuity. We identify three aspects: Ensure continuity on shared edges, on shared faces and inside a cell.

All reviewed algorithms proceed by ensuring the following sufficient conditions in this sequence:

1. Cells that meet at a common edge share a common intersection point.
2. Cells that meet along a common face share common polygon edges.
3. The subspace polygonization inside a cell is continuous, i.e., every polygon edge inside a cell is shared by a neighbored polygon.

The first condition can always be fulfilled (for a honeycomb) by computing the edge intersections by linear interpolation (subsection 4.2.1). Another approach, taken by the other three presented algorithms, is to precompute all edge intersections once and use them for all cells sharing that edge.

Given condition one the second condition is trivially fulfilled for a tetrahedral honeycomb (subsection 4.2.3). For a non-tetrahedral honeycomb there may be more than two intersection points leading to ambiguities. Wyvill et al. (subsection 4.2.2) solve the ambiguities explicitly by taking the facial average values (see figure 4.2 case 6 and 7). Lorenson and Cline omitted that point in their original implementation which resulted in possible discontinuities.

Bloomenthal (subsection 4.2.4) offers a different solution. His subdivision does not have the honeycomb property. However, he knows that two faces facing each other are either the same or one is the subdivision of the other. By always computing the iso-surface intersection for the more highly divided face he gains continuity.

At this point all reviewed algorithms result in a set of contours lying on the cell faces. Condition one guarantees that they form closed topological polygons. Above algorithms conclude by dividing the topological polygons into planar polygons (triangles), maintaining the third continuity condition.

The properties of the reviewed algorithms are summarized in table 4.2.

		Lorenson & Cline (subsection 4.2.1)	Wyvill et al. (subsection 4.2.2)	Hall & Warren (subsection 4.2.3)	Bloomenthal (subsection 4.2.4)
1	Type of cells	Cubes	Cubes	Tetrahedra	Cubes
	Honeycomb	Yes	Yes	Yes	No
	Adaptive subdivision	No	No	Yes	Yes
2	Type of polygons	Triangles	Triangles	Triangles	Triangles
	Ambiguities	Yes	No	No	No
	Continuous surface	No	Yes	Yes	Yes
	Computation of intersection points	linear interpolation	linear interpolation	root search (regula falsi)	root search
3	Continuity at shared edge	Interpolates edge intersection linearly	Compute edge intersections only once	Compute edge intersections only once	Compute edge intersections only once
	Continuity at shared face	(no continuity)	Has honeycomb and resolves ambiguities	Has tetrahedral honeycomb	Computes face intersections only once
	Disambiguation	(not resolved)	facial average	(no ambiguities)	central average

Table 4.2. *Comparison of the reviewed polygonization algorithms.*

We conclude this section with a some remarks regarding adaptive subdivision. Adaptive subdivision results from the desire to approximate the iso-surface both accurately and efficiently. This means the polygonization method must sample the function closely. In the process the algorithm may sample heavily in areas where the function is nearly linear. The solution is to sample adaptively, i.e., sampling more closely near highly curved portions of the surface. This is achieved by recursively subdividing the partition of space. Subdivision criteria are:

- terminate subdivision if polyhedron does not contain iso-surface
- terminate subdivision if iso-surface lies within some user-defined tolerance of a linear approximation.

The first criterion terminates refinement away from the iso-surface. The second criterion produces an adaptive surface tessellation based on the curvature of the iso-surface.

4.4 Quality Criteria

Quality criteria for polygonization algorithms are usually dependent on the application. However, van Gelder and Wilhelms [vGW94] suggest a set of desirable features of a general-purpose polygonization method. We will repeat them here because it is interesting to see how our analyzed algorithms fulfill them. Also the quality criteria are relevant to the development of our polygonization algorithm. The quality criteria from van Gelder and Wilhelms are:

1. The algorithm should yield a continuous surface. Each polygon edge should be shared by exactly two polygons or lie in an external face of the entire volume.
2. The iso-surface should be a continuous function of the input data. A small change in the threshold value or some data value should produce a small change in the iso-surface.
3. The iso-surface should be topologically correct when the underlying function is "smooth enough".
4. The iso-surface produced should be neutral with respect to positive and negative sample data values (relative to threshold). Multiplying the samples (and threshold) by -1 should not alter the surface.
5. The algorithm should not create artifacts not implied by the data, such as bums and holes.
6. The algorithm should be fast.

Table 4.3 shows which quality criteria the presented algorithms fulfill.

	Quality criteria	Lorenson & Cline (subsection 4.2.1)	Wyvill et al. (subsection 4.2.2)	Hall & Warren (subsection 4.2.3)	Bloomenthal (subsection 4.2.4)
1.	Continuous surface	No	Yes	Yes	Yes
2.	Continuity	No	No[a]	(unknown)	No
3.	Topologically correct[b]	Yes	Yes	Yes	Yes
4.	Neutral to sample values	No[c]	Yes	(unknown)[d]	
5.	Free from artifacts	No	Yes	Yes	Yes
6.	Speed	Van Gelder and Wilhelms [vGW94] report similar speed		(unknown)	(unknown)

[a] Take an ambiguous case and change facial average value continuously from *high* to *low*.
[b] But what is "smooth enough"?
[c] Case 12 in figure 4.1.
[d] With some extra effort this property can be achieved.

Table 4.3. *Quality criteria of the reviewed polygonization algorithms.*

CHAPTER 5

Triage Polygonization

5.1 Introduction

This chapter presents Triage Polygonization a new polygonization method designed for quasi-convolutionally smoothed objects. Triage Polygonization outperforms conventional implicit surface polygonization methods by increased speed and reduced fragmentation.

Quasi-convolutionally smoothed objects are constructed from CSG objects with polyhedral primitives by approximating a convolution with a spherical filter. A smoothed object is then defined by the 0.5 iso-surface of a density field given as an arithmetic tree. Triage Polygonization exploits the property that quasi-convolutionally smoothed polyhedra usually have predominantly planar surfaces with only edges and corners rounded.

The polygonization is defined in three steps. First the density field defining the quasi-convolutionally smoothed object is partitioned in a BSP-like manner into low and high cells (regions completely outside and inside the iso-surface, respectively). Faces separating a low from a high cell are part of the iso-surface and hence of the polygonal approximation. They are extracted in a second step. The tree defining the partition of the density field is called *density BSP tree* (DBSP tree) and the polygons separating low and high cells are called *tree polygons*.

Some cells can not be classified, i.e., they have points lying both outside and inside the iso-surface. The third and last step of Triage Polygonization performs a subspace polygonization for these cells and approximates the iso-surface inside them. The resulting polygons are called *subspace polygons*. Algorithm 5.1 gives a high-level description of Triage Polygonization.

Triage Polygonization reflects most of the concepts introduced in the previous chapters. The next section summarizes therefore the key ideas and motivates from them Triage Polygonization. We then describe in detail the three steps of the poly-

Algorithm 5.1 (Triage Polygonization)

```
INPUT: Quasi-convolutionally smoothed object
       (defined as iso-surface in a density field)
OUTPUT: List of faces (defined as convex polygons)

1.  Polyhedral subdivision of the density field in a BSP-like manner.
    Identify cells completely inside or outside the iso-surface and
    cells (possibly) intersected by the iso-surface.
2.  Extract polygons separating a cell inside the iso-surface
    from a cell outside the iso-surface.
3.  Perform a subspace polygonization for cells intersected
    by the iso-surface.
```

gonization method. Some necessary refinements are described next. We go on with the introduction of a simplified local description of a density field and use this to improve the polygonization scheme. The chapter concludes with a summary of Triage Polygonization.

5.2 Review & Motivation

In this section we summarize the key ideas from the previous chapters and motivate from them Triage Polygonization.

This thesis introduced first quasi-convolutional smoothing, which approximates convolutional smoothing. The primitive objects of a quasi-convolutionally smoothed CSG object are polyhedra modeled as intersection of half-spaces. We assume that quasi-convolutionally smoothed polyhedra usually have predominantly planar surfaces with only edges and corners rounded. Hence large areas of the object's surface are identical to the surface of the unrounded object. These parts of the surface of a quasi-convolutionally smoothed object are called *unsmoothed* or *unrounded*. Unsmoothed areas of the object's surface are planar and form a subset of all boundary faces of the object's primitive polyhedra. This fact can be recognized by reviewing the b-rep algorithms introduced in chapter 3.

Chapter 3 introduced BSP trees and two b-rep algorithms to extract the boundary of a CSG object. A b-rep algorithm transforms a CSG object into a BSP tree by defining partitioning planes from the half-spaces of the CSG object's primitive polyhedra. The b-rep of a CSG object is defined either implicitly as the faces separating the IN and OUT cells of a BSP tree or explicitly by using an augmented BSP tree with the boundary faces stored in the nodes of the tree.

Finally chapter 4 presented four general polygonization methods with three common aspects:

1. Polyhedral subdivision of space.

2. Subspace polygonization.

3. Ensuring continuity.

Before developing an improved polygonization method for quasi-convolutionally smoothed objects it is useful to analyze why the conventional methods mentioned in the literature review (section 4.2) are not optimal for our polygonization problem.

First note that conventional polygonization methods assume little or no information about the function underlying the surface. They must produce a minimum resolution grid to find essential details. This leads to heavy fragmentation of planar surface areas. The effect can be reduced by using an adaptive method. However, we discovered that quasi-convolutionally smoothed objects may have regions of arbitrarily high curvature. Even with adaptive sampling the minimum resolution of the polyhedral subdivision must be rather high.

If we can identify unsmoothed areas of the iso-surface and can determine regions of curvature (and possibly estimate the curvature), we can expect to produce fewer polygons and a better surface approximation.

Note that conventional polygonization methods only produce an approximation to the implicitly defined surface. Chapter 3 introduced two b-rep algorithms to extract the exact boundary of an unrounded CSG object. The unsmoothed part of a quasi-convolutionally smoothed object is unaffected by the rounding operation and hence, if extracted, approximates the 0.5 iso-surface in that area exactly.

Additionally we found that most curved surfaces of a quasi-convolutionally smoothed object are extremely simple, i.e., either smoothed edges or corners of polyhedra. Identifying these regions gives the possibility of directly computing an optimal solution.

The goals in the design of Triage Polygonization hence are

1. Fast speed.

2. Minimize the number of produced polygons[1].

3. Where possible polygonize the surface exactly.

Also keep the general quality criteria from section 4.4 on page 70 in mind.

[1]Minimizing the number of polygons becomes especially important if performing set operations between rounded (and hence polygonized) objects. Note however, that this might differ with the application. For example Hall and Warren [HW90] report that for finite element analysis a small aspect ratio of the polygonal elements is more important than minimizing their number.

5.3 Polyhedral Subdivision of Space

This section presents the first step of the Triage Polygonization algorithm. We subdivide the density field defining the quasi-convolutionally smoothed object into polyhedral cells and classify the resulting cells. Cells completely inside or outside the 0.5 iso-surface, and cells intersected by the 0.5 iso-surface are identified.

5.3.1 Introduction & Motivation

The previous chapter introduced BSP trees as a useful concept to extract the surface of an unsmoothed CSG object. This suggests using a BSP-like partition as a polyhedral subdivision of the density field defining a quasi-convolutionally smoothed object. Instead of classifying the cells of the partition into IN and OUT cells, we classify them as inside, outside, or intersected by the iso-surface. To facilitate discussion we define:

Definition 5.1 *A density value smaller than 0.5 is called "low", a density value greater than or equal to 0.5 is called "high". A subset S of the density field with all its points having a low (high) density value is called a "low (high) region". Similarly a region of constant density zero or one is called a "zero region" or "one region", respectively. A region which can not be classified as one of the above classes is called "unclassified". The expressions "zero", "low", "unclassified", "high", and "one" represent "density classes".*

Note that the surface of a quasi-convolutionally smoothed object (the 0.5 iso-surface in the corresponding density field) is for all natural objects defined as those points separating low from high regions. This yields the lemma:

Lemma 5.1 *The surface of a quasi-convolutionally smoothed object separates low and high regions. A partitioning of the density field into low and high regions defines implicitly the surface of the quasi-convolutionally smoothed object.*

Hence an approximation to the correct partition of the density field into low and high regions defines an approximation to the surface of the quasi-convolutionally smoothed object. To implement this idea recall the definition of a quasi-convolutionally smoothed object:

Let *Obj* be a CSG object with polyhedral primitives (defined as intersections of half-spaces). Define a density field ρ_{Obj} which is one for all points on or inside the object and zero for all points outside the object. Then

$$Obj = \{p \in \mathbb{R}^3 \mid \rho_{Obj}(p) = 1\}$$

A quasi-convolutionally smoothed object is defined by smoothing the density field ρ_{Obj} of the unsmoothed polyhedral CSG object with a spherical filter of radius r. The new density field $\rho^r_{Obj} : \mathbb{R}^3 \to \mathbb{R}$ is given as an arithmetic tree by convolutionally smoothing the half-spaces of Obj and replacing the set operations union, intersection, and set difference by the arithmetic operations addition, multiplication, and subtraction (see definition 2.5). The surface of a quasi-convolutionally smoothed object is defined as all points $p \in \mathbb{R}^3$ with $\rho^r_{Obj}(p) = 0.5$.

To facilitate the following discussion we introduce a few additional notations.

Definition 5.2 *The plane defining a half-space is called a "half-space plane" (hs-plane). If the half-space is rounded by a spherical filter of radius r we call the hs-plane orthogonally displaced to the inside by the radius r the "r-inner half-space plane" (r-ihs-plane) and denote it by h_{-r}. Similarly we call the hs-plane orthogonally displaced to the outside by r the "r-outer half-space plane" (r-ohs-plane) and denote it by h_r.*

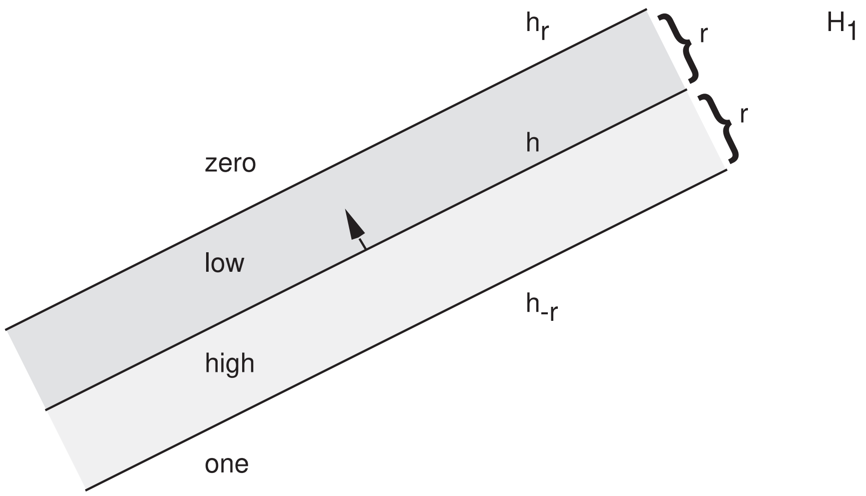

Figure 5.1. *A quasi-convolutionally smoothed half-space partitions the euclidean space into four regions.*

Figure 5.1 shows that the density field ρ^r_H partitions $\mathbb{R}^3$ into four parts. Outside the r-ohs-plane h_r the density field is constant zero. Similarly inside h_{-r} the density field is constant one. Between the two planes are a low region and a high region separated by the half-space plane h. Note that all points on h_{-r} have a density value of one, all points on h_r have a density value of zero, and all points on h have a density value of 0.5.

Figure 5.2 shows the density field of two half-spaces combined with a set operation. The two intersecting half-spaces H_1 and H_2 are smoothed with rounding radii r_1 and r_2, respectively[2].

[2]For better illustration of the method we have chosen two different rounding radii for the half-spaces. Note though that with the original definition the CSG object is smoothed with a constant rounding radius. This restriction is relaxed in section 6.6.

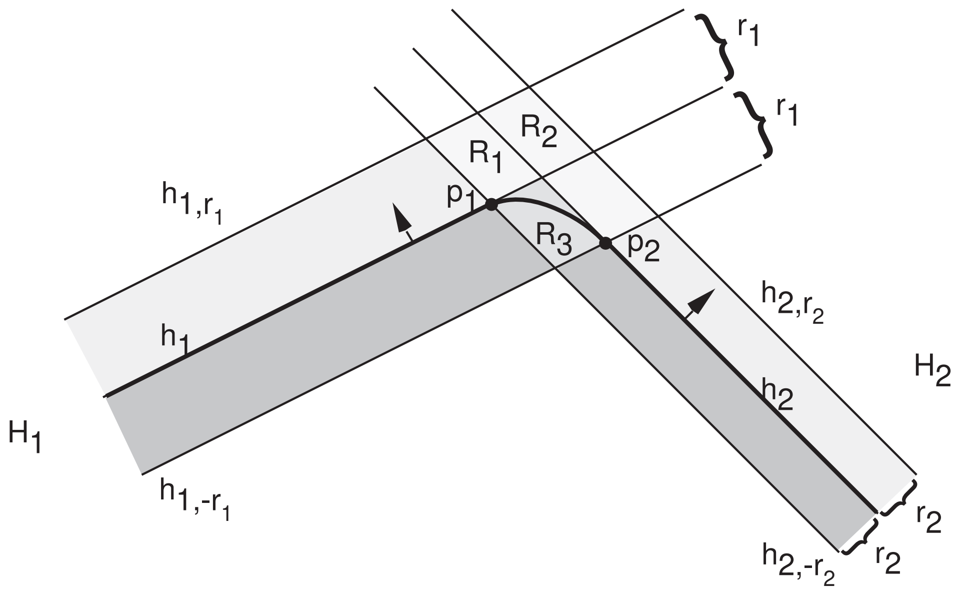

Figure 5.2. *Intersection of two half-spaces.*

According to equation 2.9 the resulting density field is the product of the density fields of the half-spaces. As example consider the points p_1 and p_2. The density value in p_1 is computed as

$$\rho_{H_1 \cap H_2}(p_1) = \rho^{r_1}_{H_1}(p_1) * \rho^{r_2}_{H_2}(p_1) = 0.5 * 1 = 0.5$$

since p_1 lies both on the planes h_1 and $h_{2,-r_2}$. Similarly

$$\rho_{H_1 \cap H_2}(p_2) = \rho^{r_1}_{H_1}(p_2) * \rho^{r_2}_{H_2}(p_2) = 1 * 0.5 = 0.5$$

since p_2 lies both on the planes h_2 and $h_{1,-r_1}$.

5.3.2 Density Classification

The previous section showed how to classify any point of a density field. Similarly a whole region of the density field is classified. Consider again figure 5.2. The planes h_1 and h_2 and the corresponding inner and outer half-space planes partition the $\mathbb{R}^3$ into 16 regions. By using interval arithmetic [Duf92, Sny92] it is possible to compute the density values for a whole region of the density field. As example take region R_1 bounded by the planes h_1, h_{1,r_1}, h_2, and $h_{2,-r_2}$.

Since R_1 is bounded by the planes h_1 and h_{1,r_1} the contribution of the density field of $H_1^{r_1}$ to this region varies between zero and 0.5. We express this as

$$\rho^{r_1}_{H_1}(R_1) = (0.0, 0.5)$$

Similarly, since R_1 is bounded by h_2 and $h_{2,-r_2}$, we have

$$\rho^{r_2}_{H_2}(R_1) = (0.5, 1.0)$$

and obtain

$$\rho_{H_1 \cap H_2}(R_1) = \rho^{r_1}_{H_1}(R_1) *_{[]} \rho^{r_2}_{H_2}(R_1) = (0.0, 0.5) *_{[]} (0.5, 1.0) = (0.0, 0.5)$$

where $*_{[]}$ is the multiplication operator for two intervals. Hence all density values in the region R_1 lie in the (open) interval $(0.0, 0.5)$. The same computation yields for the region R_2

$$\rho_{H_1 \cap H_2}(R_2) = \rho^{r_1}_{H_1}(R_2) *_{[]} \rho^{r_2}_{H_2}(R_2) = (0.0, 0.5) *_{[]} (0.0, 0.5) = (0.0, 0.25) \qquad (5.1)$$

and for the region R_3

$$\rho_{H_1 \cap H_2}(R_3) = \rho^{r_1}_{H_1}(R_3) *_{[]} \rho^{r_2}_{H_2}(R_3) = (0.5, 1.0) *_{[]} (0.5, 1.0) = (0.25, 1.0)$$

The intervals of density values for both R_1 and R_2 have only values smaller then 0.5. Hence both these regions lie outside the 0.5 iso-surface. However, the interval of density values for R_3 contains the value 0.5. This result and similar computations for the other regions reveal that R_3 is the only region in figure 5.2 that may contain the 0.5 iso-surface. Indeed, as indicated by the bold line the iso-surface intersects only R_3. The remaining parts of the iso-surface lie on the planes h_1 and h_2.

In section 2.3 we assumed that the density field of a quasi-convolutionally smoothed object has values only between zero and one. This fact can be used to define special operators $op_{[0,1]}$, $(op \in \{+, -, *, /\})$ for interval arithmetic restricting the result to the $[0.0, 1.0]$ interval.

$$I_1 op_{[0,1]} I_2 = (I_1 op_{[]} I_2) \cap_{[]} [0, 1], \qquad (op \in \{+, -, *, /\})$$

An empty interval as result of the interval operation $op_{[0,1]}$ indicates that the result of the unrestricted interval operation lies outside the $[0.0, 1.0]$ interval and hence shows that a modeling error exists.

Example 5.1 Consider the set difference of two regions R_1 and R_2 with density values $\rho(R_1) = (0.0, 0.5)$ and $\rho(R_2) = (0.5, 1.0)$. We want to compute the density field $\rho(R_1 \setminus R_2) = \rho(R_1) - \rho(R_2)$. With the restricted interval operation the result is an empty interval.

$$(0.0, 0.5) -_{[0,1]} (0.5, 1.0) = []$$

Using the non-restricted interval operator yields

$$(0.0, 0.5) -_{[]} (0.5, 1.0) = (-1.0, 0.0)$$

That means all values in the resulting density field are negative in contradiction to the assumption that the density values lie between zero and one.

Note that the restricted interval operator only detects an error in a density field if all density values of a region are outside the $[0.0, 1.0]$ interval.

The above computations can be simplified by replacing intervals with density classes. Analog to the definition of a point value denote with *low* any subinterval of the half-open interval $[0.0, 0.5)$ and with *high* any subinterval of the closed interval $[0.5, 1.0]$. All other intervals are called *unclassified.* An unclassified interval is a subinterval of $[0.0, 1.0]$. For computational efficiency we introduce a *zero* interval $[0.0, 0.0]$ and a *one* interval $[1.0, 1.0]$.

$*_{[0,1]}$	[0.0,0.0]	[0.0,0.5)	[0.0,1.0]	[0.5,1.0]	[1.0,1.0]
[0.0,0.0]	[0.0,0.0]	[0.0,0.0]	[0.0,0.0]	[0.0,0.0]	[0.0,0.0]
[0.0,0.5)	[0.0,0.0]	[0.0,0.25)	[0.0,0.5)	[0.0,0.5)	[0.0,0.5)
[0.0,1.0]	[0.0,0.0]	[0.0,0.5)	[0.0,1.0]	[0.0,1.0]	[0.0,1.0]
[0.5,1.0]	[0.0,0.0]	[0.0,0.5)	[0.0,1.0]	[0.25,1.0]	[0.5,1.0]
[1.0,1.0]	[0.0,0.0]	[0.0,0.5)	[0.0,1.0]	[0.5,1.0]	[1.0,1.0]

Table 5.1. *Multiplication of intervals.*

Table 5.1 shows the results of interval multiplication for zero, low, unclassified, high, and one intervals. Replacing the intervals with the density classes zero, low, high, one, and unclassified yields table 5.2.

$*$	zero	low	unclassified	high	one
zero	zero	zero	zero	zero	zero
low	zero	low	low	low	low
unclassified	zero	low	unclassified	unclassified	unclassified
high	zero	low	unclassified	unclassified	high
one	zero	low	unclassified	high	one

Table 5.2. *Multiplication of density classes.*

Note that by replacing the intervals with density classes information is lost. As example consider again the computation for region R_2 (equation 5.1):

$$\rho_{H_1 \cap H_2}(R_2) = \rho_{H_1}(R_2) * \rho_{H_1}(R_2) = low * low = low$$

Instead of the exact answer (0.0,0.25) the result is only classified as a low interval, i.e., a subinterval of $[0.0, 0.5)$. Initial tests, however, showed that this information loss makes little difference for the final result, the detection of low, high, and unclassified regions. This justifies using this simple and fast approach.

Similarly we can define tables for addition and subtraction (table 5.3 and 5.4). Prohibited results (empty intervals) are marked with a dash. Additionally we assume that the 0.5 iso-surface is a zero set in $\mathbb{R}^3$, i.e., it is indeed a surface and not a volume.

The table results are then correct with respect to a set of Lebesgue measure zero. The only affected result is the subtraction of a high density class from a high density class, which with a standard equality would be unclassified, but now is a low density class.

$+$	zero	low	unclassified	high	one
zero	zero	low	unclassified	high	one
low	low	unclassified	unclassified	high	one
unclassified	unclassified	unclassified	unclassified	high	one
high	high	high	high	one	-
one	one	one	one	-	-

Table 5.3. *Addition of density classes.*

$-$	zero	low	unclassified	high	one
zero	zero	zero	zero	-	-
low	low	low	low	-	-
unclassified	unclassified	unclassified	unclassified	low	zero
high	high	unclassified	unclassified	low	zero
one	one	high	unclassified	low	zero

Table 5.4. *Subtraction of density classes.*

Figure 5.3 shows the complete classification for our example partition. Note that all faces separating low and high regions are part of the 0.5 iso-surface.

We have shown now how to classify density values in any region of the density field. The next subsection presents a polyhedral subdivision of the density field.

5.3.3 Polyhedral Subdivision

A polyhedral subdivision suitable for the polygonization process must identify planar and curved regions of the surface. Figure 5.1 gave an example how such a subdivision is achieved for the quasi-convolutionally smoothed intersection of two half-spaces. The process can be generalized for an arbitrary quasi-convolutionally smoothed object.

To do this we introduce first a BSP-like data structure to partition and classify a density field into density classes. The data structure is called a *density BSP tree* (DBSP tree) and is defined as follows:

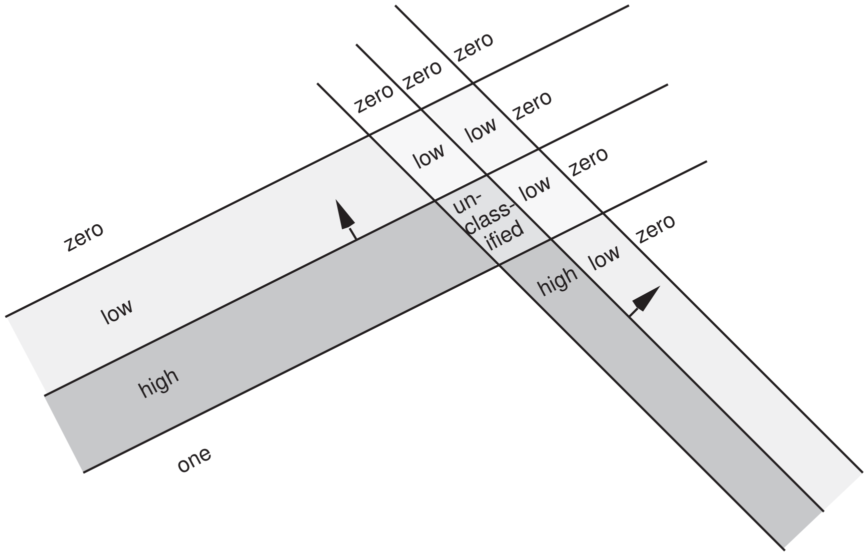

Figure 5.3. *Partitioning and classification of the density field of two intersecting half-spaces.*

Definition 5.3 *DBSP tree*

A DBSP tree $\tau_\rho(A)$ for an object A, quasi-convolutionally smoothed with a spherical filter of radius r and defined by a density field ρ^r_A, is recursively defined as

$$\begin{aligned} \tau_\rho(A) &= (h, \tau_\rho(A_{h,in}), \tau_\rho(A_{h,out})) \\ &\quad \mid zero \mid low \mid unclassified \mid high \mid one \end{aligned}$$

where

h is a hs-plane, r-ihs-plane, or r-ohs-plane of A,
$A_{h,in}$ and $A_{h,out}$ is the part of object A lying inside and outside the partitioning plane h, respectively,
$\tau_\rho(A_{h,in})$ and $\tau_\rho(A_{h,out})$ are DBSP trees inside and outside of h, respectively,
$zero$, low, $unclassified$, $high$, and one are the density classes of the cells in the density field ρ^r_A.

The polyhedral subdivision algorithm transforms the density field of a quasi-convolutionally smoothed CSG object into a DBSP tree. Chapter 3 introduced two algorithms to transform a CSG object into a BSP tree. One of them was the lazy BSP-tree algorithm. It builds a BSP tree by inserting the CSG object step by step into an initially empty BSP tree.

DCSG Objects

To define a similar algorithm for DBSP trees, first convert a quasi-convolutionally smoothed CSG object into a CSG-like structure. This is done by replacing the primitive objects of the unrounded object by convex polyhedral cells classifying their density fields. Since the density field of the primitive objects is zero except for a bounded region we identify only cells with a density class different from zero. The resulting object is called a *density CSG object* (DCSG object). The data types for a DCSG object and a DBSP tree are summarized in figure 5.4.

```
::  DCSGObject = DUnion DCSGObject DCSGObject
               | DIntersection DCSGObject DCSGObject
               | DSetDifference DCSGObject DCSGObject
               | DPrimitive Polyhedron DensityClass

::  DBSPTree = DBSPNode Plane DBSPTree DBSPTree | DBSPLeaf DensityClass

::  DensityClass = Zero | Low | Unclassified | High | One
```

Figure 5.4. *Data types of a DCSG object and a DBSP tree.*

Example 5.2 Figure 5.5 (a) shows a quasi-convolutionally smoothed object. The object is given as a rounded set difference of two cuboids and is pictured in (b). Both cubic primitives define a density field. The DCSG object in (c) is defined by partitioning and classifying the density fields of the cuboids.

To define the partition recall that a primitive object P is modeled as an intersection of half-spaces. Let the object be quasi-convolutionally smoothed with a spherical filter of radius r. Then its density field ρ_P^r is the product of the density fields of the smoothed half-spaces. Hence ρ_P^r is zero outside any r-ohs-plane (a half-space plane displaced orthogonally outwards by the rounding radius r). The region inside all r-ohs-planes is partitioned with all hs-planes and r-ihs-planes. The density class of the resulting cells is determined by classifying them against each half-space and repeatedly applying the multiplication operator for density classes defined by table 5.2.

The algorithm to transform a quasi-convolutionally smoothed CSG object into a DCSG object is most easily formulated in functional code and is given in figure 5.6.

The polyhedron enclosing the non-zero density field is called `outerPolyhedron`. It lies inside all r-ohs-planes. The `outerPolyhedron` is partitioned with all hs-planes and r-ihs-planes. The resulting cells are classified by the function `ClassifiedCell`, which computes the density class of the center of the cell with respect to the density field of each smoothed half-space. The partition guarantees that the density class of the centre is the density class of the entire cell.

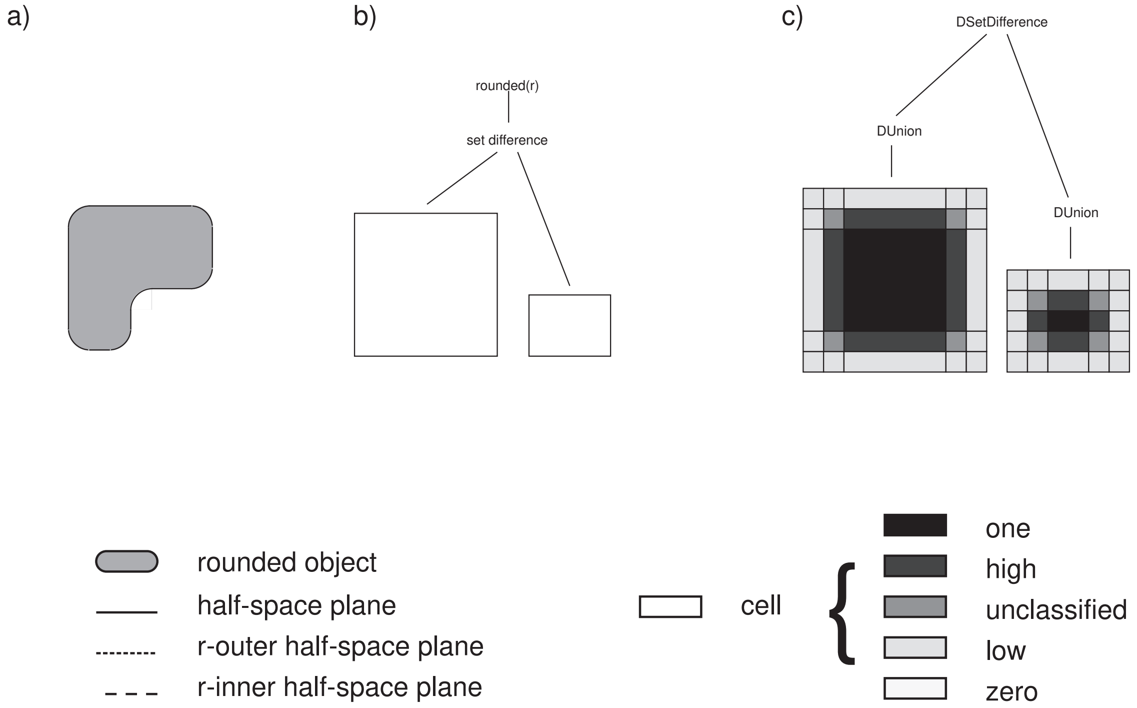

Figure 5.5. *A quasi-convolutionally smoothed object (a) is defined by a CSG object (b). The CSG object is transformed into a DCSG object (c) by partitioning and classifying the density fields of the primitive objects.*

The density field of the cell is the product of the density fields of the half-spaces. Hence its density class is the product of the density classes of the parts of the half-spaces intersected by the cell. The product of the density classes is computed with table 5.2.

Example 5.3 The partitioning of a primitive object is described in figure 5.7. Part (a) shows a quasi-convolutionally smoothed primitive object and the corresponding unrounded object. The density field of the primitive object is partitioned with the object's hs-planes, r-ihs-panes, and r-ohs-planes (b). The resulting cells with non-zero density class are given in (c).

Building a DBSP Tree

It remains to transform the DCSG object into a DBSP tree. As in the lazy b-rep algorithm (in figure 3.5) we start with an initially empty DBSP tree and insert the child objects of the DCSG object according to the involved set operation. A primitive polyhedron is again transformed into a linear tree. The cell representing the polyhedron in the linear tree is classified with the density class of the polyhedron

```
CSG2DCSG ::  Real CSGObject -> DCSGObject
CSG2DCSG r (Union obj1 obj2)
   = DUnion (CSG2DCSG r obj1) (CSG2DCSG r obj2)
CSG2DCSG r (SetDifference obj1 obj2)
   = DSetDifference (CSG2DCSG r obj1) (CSG2DCSG r obj2)
CSG2DCSG r (Intersection obj1 obj2)
   = DIntersection (CSG2DCSG r obj1) (CSG2DCSG r obj2)
CSG2DCSG r (Primitive (Intersection hs_Planes)) = DUnion classifiedCells
where
   r_ohs_Planes      = map (Translation r) hs_Planes
   r_ihs_Planes      = map (Translation (-r)) hs_Planes
   outerPolyhedron   = MakePolyhedron r_ohs_Planes
   polyhedra         = PartitionWith outerPolyhedron (hs_planes ++ r_ihs_Planes)
   classifiedCells   = map (ClassifiedCell r hs_Planes) polyhedra

ClassifiedCell ::  Real [Plane] Polyhedron -> DCSGObject
ClassifiedCell r planes polyhedron = DPrimitive polyhedron densityClass
where
   DensityField plane = DensityFieldOfHalfSpace r plane
   densities = map ((Density (Centre polyhedron)) o DensityField) plane
   densityClass = foldl * One (map DensityClass densities)
```

Figure 5.6. *Transforming a quasi-convolutionally smoothed CSG object into a DCSG object.*

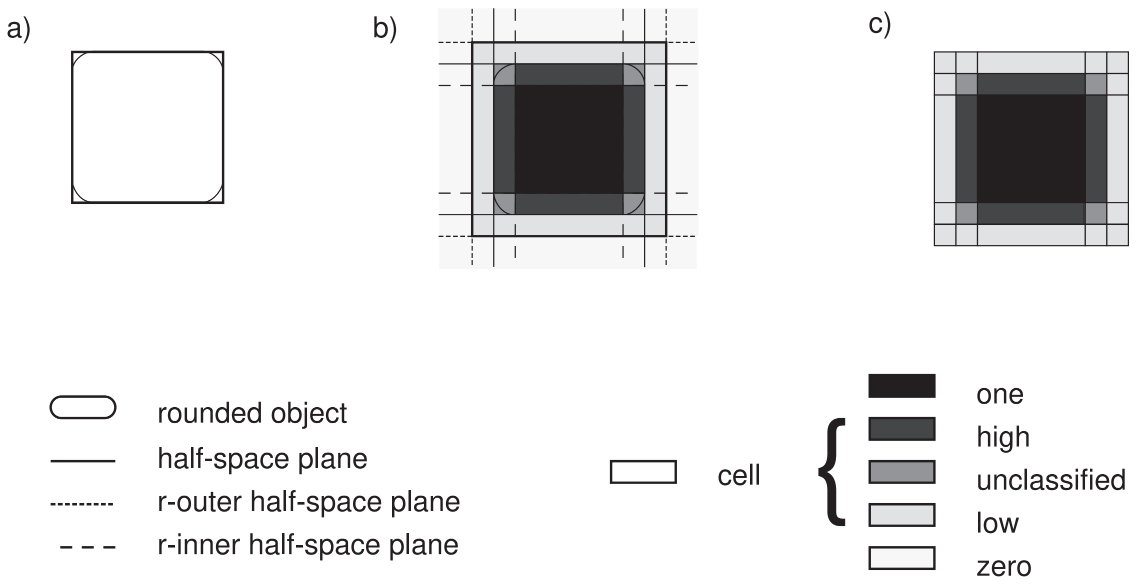

Figure 5.7. *Partitioning of the density field of a quasi-convolutionally smoothed primitive object into cells with non-zero density class.*

and the other cells of the linear DBSP tree are classified as zero. Figure 5.8 presents the algorithm. Note the similarity to the lazy b-rep algorithm.

```
DCSG2DBSP ::  DCSGObject -> DBSPTree

DCSG2DBSP (DUnion obj1 obj2) = UnionDBSP_DCSG (DCSG2DBSP obj1) obj2
DCSG2DBSP (DIntersection obj1 obj2) = IntDBSP_DCSG (DCSG2DBSP obj1) obj2
DCSG2DBSP (DSetDifference obj1 obj2) = SetDifDBSP_DCSG (DCSG2DBSP obj1) obj2
DCSG2DBSP (DPrimitive (Polyhedron faces) densityClass)
   = LinearDBSPTree densityClass faces

LinearDBSPTree ::  DensityClass [Face] -> DBSPTree
LinearDBSPTree densityClass [face:faces]
   = DBSPNode (PlaneOf face) (LinearDBSPTree densityClass faces) (DBSPLeaf Zero)
LinearDBSPTree densityClass [] = DBSPLeaf densityClass
```

Figure 5.8. *Transforming a DCSG object into a DBSP tree.*

As an example for a set operation between a DBSP tree and a DCSG object consider the union operation. The algorithm is again similar to the union operation between a BSP tree and a CSG object (see figure 3.8). The only difference occurs if partitioning a polyhedral cell of a DBSP tree with a DCSG object. If the cell's density class is one then it does not change by the union operation (use the assumption that the density values are bounded by one). If the cell's density class is zero partition the cell with the DCSG object and form a DBSP tree (using the function `DCSG2DBSP`). The same is done if the cell's density class is neither one nor zero. However, in this case the density classes of the resulting subtree are given as the sum of the density class of the partitioned cell and the density class of the corresponding region of the DCSG object. The function `AddDensity` forms the sum by adding the density class of the partitioned cell, according to table 5.3, to all cells of the new subtree.

5.3.4 Tree Polygons

A DBSP tree partitions the density field of a quasi-convolutionally smoothed object into cells with density classes zero, low, unclassified, high, and one. Lemma 5.1 yields that faces between low and high cells form part of the surface of the quasi-convolutionally smoothed object (the 0.5 iso-surface). They are therefore part of the polygonized surface and can be extracted already at this stage. Note that the density field of a quasi-convolutionally smoothed object is continuous and therefore no part of the 0.5 iso-surface neighbors a zero or a one cell.

Section 3.3.4 introduced the function `BoundaryBSPTree` to extract faces, which separate IN and OUT cells, from a BSP tree (see figure 3.12). A similar algorithm

```
UnionDBSP_DCSG ::  DBSPTree DCSGObject -> DBSPTree
UnionDBSP_DCSG (DBSPNode plane inTree outTree) dcsgObj
   = DBSPNode plane newInTree newOutTree
where
   newInTree         = UnionDBSP_DCSG inTree inDCSG
   newOutTree        = UnionDBSP_DCSG outTree outDCSG
   (inDCSG,outDCSG)  = SplitDCSGObj plane dcsgObj
UnionDBSP_DCSG (DBSPLeaf densityClass) dcsgObj
   | densityClass == Zero  = DCSG2DBSP dcsgObj
   | densityClass == One   = DBSPLeaf One
   = AddDensity densityClass (DCSG2DBSP dcsgObj)

AddDensity ::  DensityClass DBSPTree -> DBSPTree
AddDensity densityClass (DBSPNode plane inTree outTree)
   = DBSPNode plane  (AddDensity densityClass inTree)
                     (AddDensity densityClass outTree)
AddDensity densityClass1 (DBSPLeaf densityClass2 )
   = DBSPLeaf  (densityClass1 + densityClass2)
```

Figure 5.9. *Union between a DBSP tree and a DCSG object.*

applies to DBSP trees to extract faces separating low from high cells. The resulting faces are called *tree polygons* and the algorithm to extract them is given in figure 5.10. Since faces are polygons the return value of the function is a list of polygons.

The algorithm `TreePolygons` differs from the algorithm `BoundaryBSPTree` in figure 3.12 in that way that it has a third argument, which is a polyhedron (`boundingBox`) bounding the non-zero density field. The intersection of the bounding box with a partitioning plane forms a candidate face for the parts of the object surface lying on the plane. This is necessary because in contrary to a BSP tree a DBSP tree is not augmented with a set of candidate faces. The definition of a DBSP tree guarantees that all unsmoothed parts of the object's surface lie on a partitioning plane[3].

For each DBSP node the candidate face (`newFace`) is pushed down the IN and OUT tree to find the parts of it facing a high cell on the inside and a low cell on the outside. The resulting polygons then have an outward surface normal. The procees is executed by the function `SingleSideTreePolygons`. To find the polygons on the iso-surface with opposite orientation to the partitioning plane the process is repeated with `newFace` flipped.

After finding the tree polygons of a DBSP node the algorithm is called recursively for the subtrees of the node until a cell is reached.

---

[3]We even know that the unsmoothed part of the object surface must lie on a partitioning plane, which is a half-space plane of the original CSG object, and not a r-ihs-plane or a r-ohs-plane. However, for simplicity we do not make this difference between partitioning planes.

```
TreePolygons ::  DBSPTree Polyhedron -> [Face]

TreePolygons (DBSPNode plane inTree outTree) boundingBox
   = polysOnPlane ++ polysInTree ++ polysOutTree
where
   (inPolyh,outPolyh) = SplitPolyhedron plane boundingBox
   newFace        = Intersection boundingBox plane
   sameFaces      = SingleSideTreePolygons inTree outTree newFace
   oppositeFaces  = SingleSideTreePolygons outTree inTree (FlippedFace newFace)
   polysOnPlane   = sameFaces ++ oppositeFaces
   polysInTree    = TreePolygons inTree inPolyh
   polysOutTree   = TreePolygons outTree outPolyh
TreePolygons (DBSPLeaf _) _ = []


SingleSideTreePolygons ::  DBSPTree DBSPTree Face -> [Face]

SingleSideTreePolygons insideTree outsideTree face = treePolygons
where
   facesInLowCells   = InsertInDCells Low insideTree [face]
            // Face fragments which lie in Low cell of insideTree
   treePolygons      = InsertInDCells High outsideTree facesInLowCells
            // Face fragments whic lie in High cell of outsideTree
```

Figure 5.10. *Extracting tree polygons from a DBSP tree.*

The function `InsertInDCells` inserts a list of faces into a DBSP tree and returns the fragments of the faces landing in a cell with the specified density class. It works analogously to the function `InsertInCells` (figure 3.13) used for BSP trees.

5.4 Subspace Polygonization

The surface of a quasi-convolutionally smoothed object is defined as the 0.5 iso-surface of a density field. The previous section introduced a polyhedral subdivision of the density field. The subdivision identifies cells inside and outside the iso-surface and extracts the sections of the object surface separating them. The remaining parts of the object surface lie inside unclassified cells. These cells can not be classified as lying outside or inside the iso-surface and might be intersected by it. This section presents an algorithm to polygonize the 0.5 iso-surface inside unclassified cells.

```
InsertInDCells ::  DensityClass DBSPTree [Face] -> [Face]

InsertInDCells _ _ [] = []          // no face reaches this (sub)tree
InsertInDCells wantedClass (DBSPLeaf densityClass) faces
   | densityClass == wantedClass = faces
   = []
InsertInDCells wantedClass (DBSPNode plane inTree outTree) faces
  = faceFragmentsInside ++ faceFragmentsOutside
where
  (facesInside,facesOutside) = UnZipWith (SplitFace plane) faces
  faceFragementsInside  = InsertInDCells wantedClass inTree facesInside
  faceFragmentsOutside  = InsertInDCells wantedClass outTree facesOutside
```

Figure 5.11. *Inserting a set of faces into a DBSP tree and returning the fragments which reach a cell of the specified density class.*

5.4.1 Motivation & Definitions

Chapter 4 suggested as a promising approach for a subspace polygonization to approximate the iso-surface in steps of increasing dimension: first compute points on the iso-surface, then connect the points to form edges, and connect the edges to form polygons.

A set of points on the iso-surface is formed by computing for every face of an unclassified cell the intersection points of its edges with the 0.5 iso-surface. The intersection points of the edges of a face can be connected to form a new set of edges, called polygon edges. The polygon edges approximate the 0.5 iso-surface intersection with the face. Connecting the polygon edges to a topological polygon yields an approximation to the 0.5 iso-surface intersection with the cell.

These steps represent already a subspace polygonization. The quality of the polygonization is improved by refining the polygon edges and the topological polygons. In particular a non-planar topological polygon must be subdivided into planar polygons. Some notations are useful for the following discussion:

Definition 5.4 *A face in a DBSP tree which separates two unclassified cells is "fully unclassified". If it separates an unclassified cell and a low or high cell it is "simple unclassified". In all other cases the iso-surface does not intersect the face and it is a "classified" face.*

A face is a polygon defined by "face vertices". A "face edge" connects two face vertices. We will determine "intersection points" of the face edges with the 0.5 iso-surface. The intersection points are also called "polygon vertices". Polygon vertices are connected to form "polygon edges", which are connected to form topological polygons.

The face vertices are given in anti-clockwise order. A polygon vertex lying between *a high and a low face vertex forms a "high-low transition" and a polygon vertex between a low and a high face vertex forms a "low-high transition".*

With these notations algorithm 5.2 gives a high-level description for the subspace polygonization.

Algorithm 5.2 (Subspace Polygonization)

```
INPUT: A density BSP tree (DBSP tree)
OUTPUT: List of convex polygons

for each unclassified cell C do begin
1.  for each face F of cell C do begin
      {Approximate the intersection of F with the 0.5 iso-surface}
      for each edge E of face F do
         Compute the 0.5 iso-surface intersection of edge E
1.1.  Connect intersection points to polygon edges
1.2.  Refine polygon edges
    end {for loop}
2.  Connect polygon edges to form topological polygons
3.  Refine topological polygons and form planar polygons
end {for loop}
```

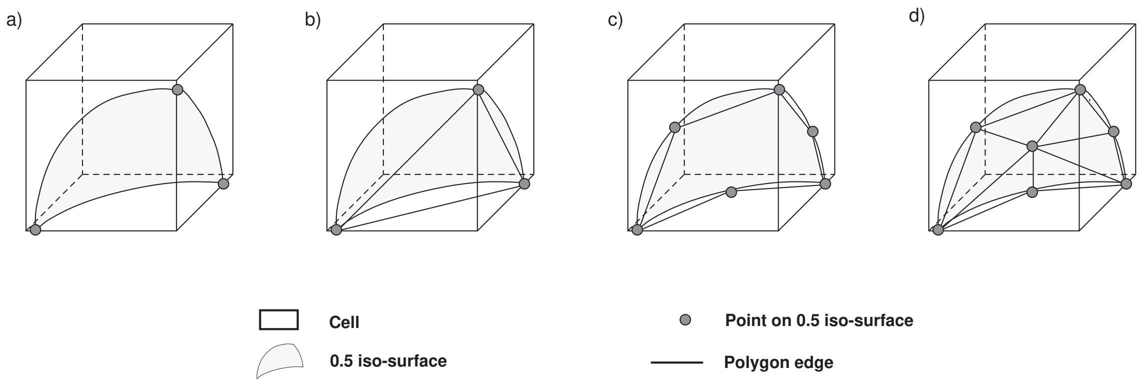

Figure 5.12. *The subspace polygonization is performed in three steps. First approximate the 0.5 iso-surface intersection with a face by computing the iso-surface intersections with the edges (a) and connecting them to polygon edges (b). Polygon edges are refined and connected to form topological polygons (c). Finally the topological polygons are refined and subdivided into planar polygons.*

Example 5.4 The basic steps of the subspace polygonization for a single unclassified cell are described in figure 5.12. The picture shows an unclassified cell and the part of the 0.5 iso-surface intersecting it.

The intersection points of the cell's edges with the 0.5 iso-surface are given in (a) as dots. Part (b) shows the polygon edges connecting two intersection points of a face. The polygon edges are refined in (c). Note that the polygon edges approximate the iso-surface intersection with a face.

Connecting the polygon edges forms a topological polygon. The topological polygon is refined and subdivided into planar polygons in (d) by connecting all polygon vertices to a new point on the 0.5 iso-surface, which is near to the centroid of the topological polygon.

The following subsections describe the subspace polygonization in more detail. Especially we want to achieve two goals:

1. Continuity of the polygonization.

2. Constructing a closed topological polygon.

A discussion of these aspects is presented in section 6.5. At this stage we use these goals only to motivate our solution. The next subsection presents an approach to guarantee continuity by precomputing the polygon edges.

5.4.2 Precomputing Polygon Edges

A necessary condition for a continuous polygonization is continuity at common faces of cells. This means cells that meet at a common face must have the same polygon edges approximating the 0.5 iso-surface intersection with the face. This is most easily achieved by computing polygon edges only once for a face and using them for the subspace polygonization of both cells sharing the face. This subsection introduces an algorithm to precompute polygon edges for all faces of unclassified cells.

First note that two adjacent cells do not necessarily have a common face. This situation is shown in figure 5.13 (a). However, (b) illustrates that faces can be subdivided until each face is common to exactly two cells.

The first step in the precomputation of all polygon edges is to generate a set of faces each of which is common to exactly two cells. For these faces approximate their 0.5 iso-surface intersection by polygon edges.

A simple continuity argument yields that faces of a zero or one cell can not intersect the 0.5 iso-surface[4]. They do not define polygon edges. The set of candidate faces for a 0.5 iso-surface intersection is hence restricted to faces separating a low or high cell from an unclassified cell (simple unclassified faces) and faces separating two unclassified cells (fully unclassified faces).

[4] An exception arises from the application of clipping planes (see section 6.6.1).

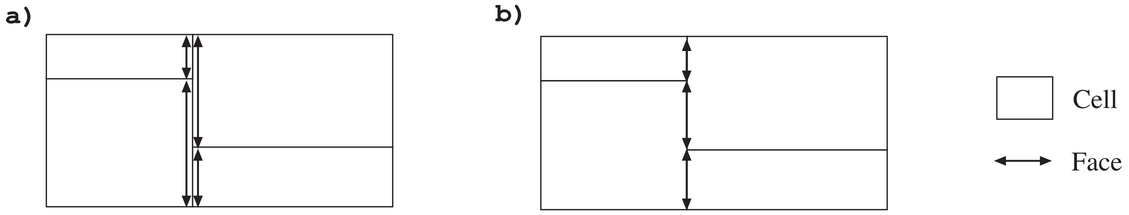

Figure 5.13. *Adjacent cells do not always have common faces (a). Common faces are obtained by subdivision (b).*

The algorithm to precompute polygon edges first extracts faces common to a high or low cell and at least one unclassified cell[5]. It then approximates the 0.5 iso-surface intersection with a face by polygon edges. Because the polygon edges are part of the subspace polygonization of an unclassified cell they are stored in the unclassified cells sharing the corresponding face.

```
PrecomputePolygonEdges ::  DBSPTree DensityField Polyhedron -> DBSPTree

PrecomputePolygonEdges (DBSPNode plane inTree outTree) densityField boundingBox
   = DBSPNode plane newInTree newOutTree
where
   (inBox,outBox)  = SplitPolyhedron plane boundingBox
   newFace         = Intersection boundingBox plane
   polygonEdges    = flatten [PolygonEdgesOnFace densityField face \\
                           face <- UnclassifiedFaces inTree outTree newFace]
   revPolygonEdges = map ReverseEdge polygonEdges
   inTree'         = PrecomputePolygonEdges inTree densityField inBox
   outTree'        = PrecomputePolygonEdges outTree densityField outBox
   newInTree       = StoreEdgesInUnclassifiedCell inTree' polygoneEdges
   newOutTree      = StoreEdgesInUnclassifiedCell outTree' revPolygoneEdges

PrecomputePolygonEdges leaf=:(DBSPLeaf _ densityClass) _ _
   | densityClass == Unclassified = DBSPLeaf (Edges []) densityClass
   = DBSPLeaf NoEdges densityClass
```

Figure 5.14. *Precomputing polygon edges.*

The algorithm is given in functional code in figure 5.14. Input to the algorithm is a DBSP tree and a bounding box given as a polyhedron. The bounding box only serves to create a candidate face `newFace` from which all unclassified faces on

[5]In that respect the function `PrecomputePolygonEdges` is similar to the function `TreePolygons` in figure 5.10, which extracts faces separating low from high cells. In fact, the algorithms can be combined and the results are obtained with only one traversal of the tree.

the partitioning plane are derived[6]. The unclassified faces are determined by the function `UnclassifiedFaces`, which works similar to the function `SingleSideTreePolygons` in figure 5.10 except that it returns face fragments (unclassified faces) common to at least one unclassified cells. For each unclassified face the function `PolygonEdges` returns the polygon edges approximating the 0.5 iso-surface intersection with the face. The function `PrecomputePolygonEdges` is then called recursively for both subtrees of the DBSP node.

The unclassified faces are constructed such that a polygon edge lies on a face shared by exactly two cells. The cell inside and outside the corresponding face lies in the IN tree and OUT tree, respectively. A face is a boundary face of the cell on its inside (since then the normal of the face points outside the cell).

Since polygon edges have an orientation (as explained on page 94 ff.) an edge is stored only in the cell on its inside. For the cell on its outside the reversed edge is taken. To store the edges in unclassified cells the data structure for a DBSP leaf must be extended by a list of edges. The new data structure for a DBSP tree is shown in figure 5.15.

```
::  DBSPTree = DBSPNode Plane DBSPTree DBSPTree
             | DBSPLeaf PolygonEdges DensityClass

::  PolygonEdges = Edges [Edge] | NoEdges
```

Figure 5.15. *New data structure for a DBSP tree with leafs extended by a list of edges.*

The middle point of each edge is then inserted in the subtree in which we want to store the edge. Every unclassified cell having a middle point on its boundary at the end of the insertion process is augmented with the corresponding edge.

It remains to compute the polygon edges approximating the 0.5 iso-surface intersection with a face. We consider simple unclassified faces and fully unclassified faces separately.

Polygon Edges for Simple Unclassified Faces

A continuity argument yields that a simple unclassified face can only be touched by the 0.5 iso-surface. The iso-surface intersection with the face is either a point, an edge, or the face itself. If the intersection is a single point the face does not contain a polygon edge and can be discarded. If the intersection is an edge, the edge itself defines a polygon edge. In the case that the whole face lies on the 0.5 iso-surface all its edges lie on the iso-surface as well, and hence are polygon edges. For a simple unclassified face the polygon edges are therefore the face edges lying on the 0.5 iso-surface[7].

[6] The same principle applied to the function `InsertCandidateFaces` in figure 3.17.

[7] Numerical problems and the emerging of some special cases demanded an alternative solution. The solution is to test for an iso-surface intersection with a displaced iso-surface of value $0.5+\epsilon$

Polygon Edges for Fully Unclassified Faces

In contrast to the above case a fully unclassified face can be intersected anywhere by the 0.5 iso-surface. An approximation to the iso-surface intersection with the face is obtained by computing the iso-surface intersection with the face edges and connecting the intersection points. Since the density field ρ is continuous, an intersection point exists for a given edge if the density values at the edge vertices are on different sides of the 0.5 threshold. The intersection points are computed with a root finder (see section 6.3).

If there are exactly two intersection points they are connected to form an edge approximating the iso-surface intersection with the face. For more than two intersection points their connection is ambiguous.

To resolve ambiguities observe that only neighbored intersection points can belong to an iso-surface intersection (otherwise the iso-surface is self-intersecting or folded). It must be determined which pair of neighbored intersection points are connected. To do this, subdivide a face by connecting each intersection point to the centroid of the face as shown in figure 5.16.

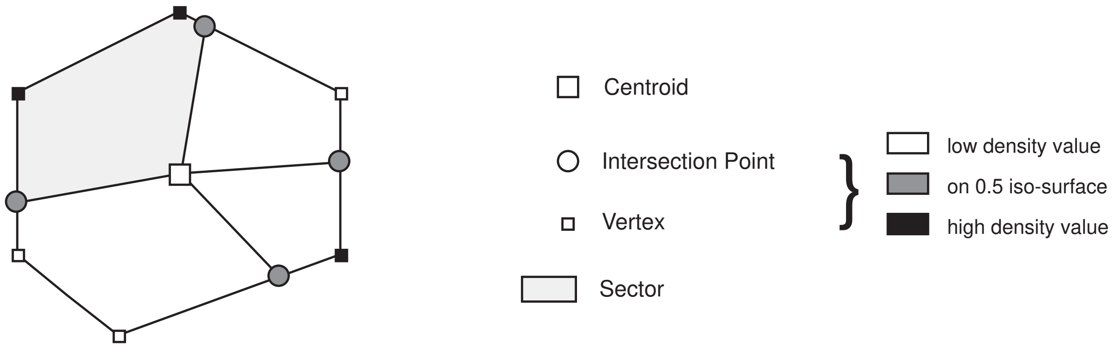

Figure 5.16. *Divide a face into sections by connecting each intersection point to the centroid of the face.*

Each pair of neighbored intersection points forms a sector with the centroid. Since an intersection point lies between a low and a high vertex all face vertices of a sector have the same density class. Observe that the iso-surface, if it intersects the sector, seperates face vertices and the centroid into different density classes. Hence a test for an iso-surface intersection is made by determining the density classes of the centroid and one of the face vertices of the sector. If they have the same density class the sector is not intersected by the iso-surface. Otherwise the intersection points

(we chose $\epsilon = 0.001$). If an iso-surface intersection with the $0.5 + \epsilon$ iso-surface exists the real 0.5 iso-surface intersection is computed. The computation is then similar to the case with a fully unclassified face described in the next chapter. Implementation details for this technique are given in section 6.7.

defining the sector belong to an iso-surface intersection with the face and we connect them to form a polygon edge.

Example 5.5 As example consider figure 5.17. The figure shows a face intersected by different iso-surfaces in four intersection points. In (a) and (b) the centroid is of low and high density, respectively. The algorithm produces different polygon edges and correctly approximates the iso-surface represented as thin curved line. Part (c) of the figure shows the same example as in (b) but with a strongly curved iso-surface. The centroid is now of low density class and the algorithm produces erroneous polygon edges.

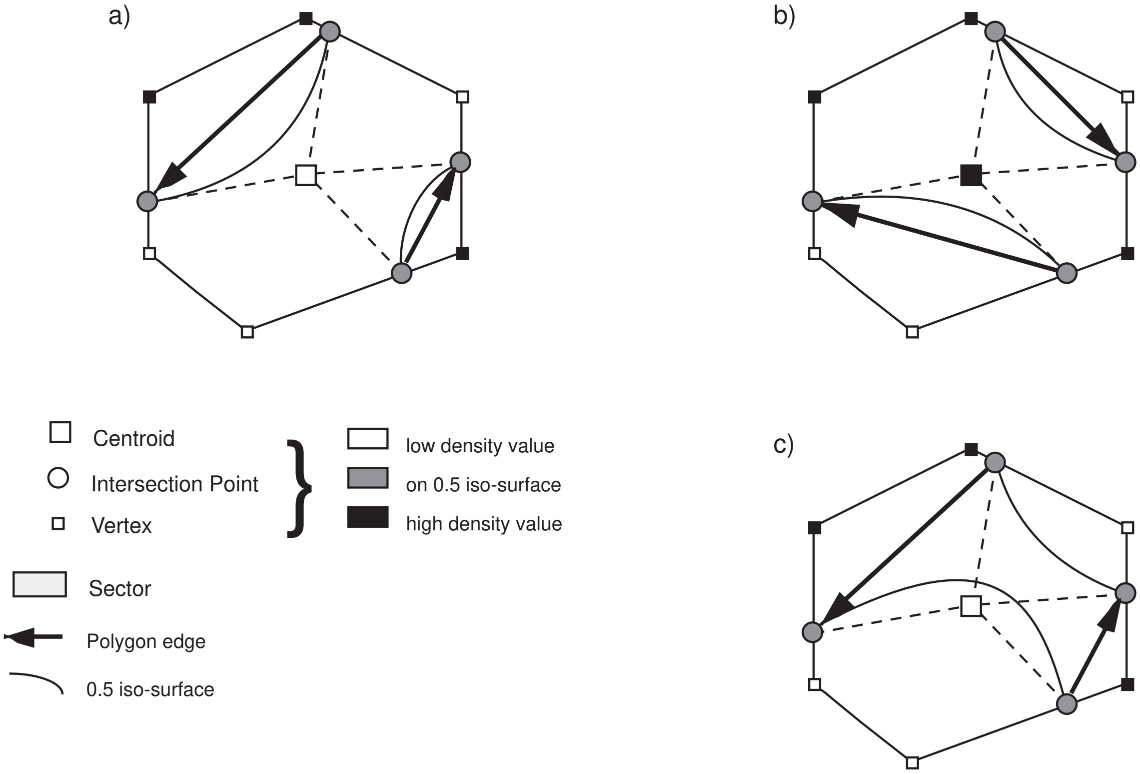

Figure 5.17. *A polygon edge separates low and high density values. If the centroid has a low density value, the polygon edges separate the high vertices from the centroid and hence approximate the iso-surface (a). If the centroid has a high density value the polygon edges separate it from the low vertices (b). For a strongly curved iso-surface the approximation can be wrong (c).*

Above example illustrates that our method to connect intersection points might still produce wrong polygon edges. However, this can only occur if the 0.5 iso-surface intersections with the face are near together, i.e., if the 0.5 iso-surface is strongly curved or fluctuating. Since the surface of a quasi-convolutionally smoothed object is intrinsicly “smooth” this is unlikely[8] and we can expect to compute correct polygon edges.

The quality of the approximation of a polygon edge is improved by refining the polygon edge to a polyline. The refinement process is described in section 5.5. For simplicity we continue with the description of the subspace polygonization algorithm only with unrefined polygon edges.

Orientation of Polygon Edges

A polygon edge is given an orientation by demanding that an edge always points from a low-high transition to a high-low transition. The reason for this is that all points inside the iso-surface have high density values. Every polygon built with polygon edges oriented as above has the high vertices of a cell on its inside, and has therefore an outward surface normal.

Now recall that a fully unclassified face is shared by two cells. The face is then a boundary face (i.e., with an outward normal) of only one of the cells sharing it. The fully unclassified face must be flipped in order to be a boundary face of the other unclassified cell sharing it. A boundary face is flipped by reversing its vertices. This is the reason why the algorithm `PrecomputePolygonEdges` in figure 5.14 inserts the reversed polygon edges in the OUT tree of the corresponding node.

5.4.3 Forming Topological Polygons

The second step of Triage Polygonization connects the precomputed polygon edges of an unclassified cell to form topological polygons. Recall that the polygon edges have an orientation. An algorithm to form topological polygons is given by starting with an arbitrary edge and searching for an edge with a start point identical to the end point of the previous edge. This process is repeated until the topological polygon is closed. Figure 5.18 gives the algorithm in functional code.

The function `MakeTopologicalPolygons` takes into account that the subspace polygonization can produce several topological polygons. `MakeTopologicalPolygons` takes as input a list of edges and returns a list of topological polygons. A topological polygon is generated by the function `MakeTPolygon`. The function connects edges from the list of polygon edges to form a polygon and returns the new polygon and the list of edges not used to form the new polygon. `MakeTPolygon` is repeatedly called until the list of edges is empty.

`MakeTPolygon` takes as arguments a partial (not closed) polygon defined as a polyline, the start point of the polyline, and a list of edges. The polyline is given as

[8] A counterexample, though, is constructed by taking the rounded union of two objects, say cubes. Dependent on the distance of the components the rounded object has either one or two components. Then there exists a fixed distance at which the rounded union of the two cubes is just a single connected object. Varying the distance of the cubes around that threshold may result in a polygonization with erroneous polygon edges.

```
MakeTopologicalPolygons ::  [Edge] -> [Polygon]
MakeTopologicalPolygons edges = [newPoly:MakeTopologicalPolygons restEdges]
where
   (p1,p2)     = hd Edges
   (reversedPolyPoints,restEdges) = MakeTPolygon p1 [p2,p1] (tl edges)
   newPolygon  = reverse reversedPolyPoints
MakeTopologicalPolygons [] = []

MakeTPolygon ::  Point [Point] [Edge] -> [Edge]
MakeTPolygon startPoint polyline edges
   | startPoint == nextPoint = (polyline,edges)
   = MakeTPolygon startPoint [nextPoint:polyline] restEdges
where
   (nextPoint,restEdges) = FindNextPoint (hd polyline) [] edges

FindNextPoint ::  Point [Edge] [Edge] -> (Point,[Edge])
FindNextPoint p _ [] = abort "Cannot close topological polygon"
FindNextPoint p notUsedEdges [(p1,p2):edges]
   | p == p1 = (p2,notUsedEdges ++ edges)
   = FindNextPoint p [(p1,p2):notUsedEdges] edges
```

Figure 5.18. *Algorithm to form topological polygons for subspace polygonization.*

a point list in reversed order, i.e., the first point is the end point of the polyline and the last point is the start point of the polyline. The polyline is point by point extended until it forms a closed polygon. At the initial call of `MakeTPolygon` in `MakeTopologicalPolygons` the polyline consists only of the two points of a single egde.

The polyline is extended if the edge list contains an edge with a start point equal to the end point of the polyline. In this case the edge is removed from the edge list and the polyline is extended by the end point of the edge. The new point and the list of remaining edges are given by the function `FindNextPoint`. The function `MakeTPolygon` is recursively called with the extended polyline and the list of remaining edges. If the end point of the new edge equals the start point of the polyline the polyline defines a closed topological polygon and the recursion stops. The new polygon and the remaining edges are returned.

The function `FindNextPoint` takes a point, a list of edges already examined but not used to extend the polyline, and a list of new edges. It then finds in the list of new edges an edge with a start point equal to the inputted point (the current end point of the polyline). The end point of the found edge and the list of unused edges (i.e., the examined edges and the remaining new edge) are returned.

We conclude this subsection with a few notes:

Note 1. The algorithm might produce degenerate polygons (polygons with an area of zero). Such a polygon is generated if the iso-surface intersects the cell only in a single edge or vertex of the cell. Degenerate polygons are eliminated in a post-processing step

Note 2. It can not be guaranteed that the algorithm always produces closed topological polygons (see also section 6.5). However, this was the case in all scenes we tested.

Note 3. Since no two polygon edges intersect each other, the topological polygons do not intersect either.

Note 4. In general the so formed topological polygons are not planar.

5.4.4 Triangulation of a Topological Polygon

The last step of the subspace polygonization divides each topological polygon into planar polygons. This is achieved by connecting each pair of neighbored polygon vertices to the centroid of the topological polygon and thus triangulating the topological polygon.

The polygonal approximation of the iso-surface is improved by "moving" the centroid towards the iso-surface. This means, that the centroid is replaced by a point on the iso-surface near to it (if such a point can be found). The new point on the iso-surface is called "adjusted centroid" and is given as the 0.5 iso-surface intersection of a linear search space defined by the centroid and its density gradient. If an iso-surface intersection is found the topological polygon is triangulated by connecting its vertices to the adjusted centroid, otherwise the original centroid is used.

Two restrictions apply to the intersection point. First it must lie inside the cell subject to the subspace polygonization. Secondly, if the cell contains more than two topological polygons the triangulated topological polygons must not intersect each other. Both restrictions are achieved by restricting the linear search space to a conical section of the cell. The conical section is defined by connecting the centroid of the cell to the vertices of the topological polygon.

In all of our test scenes we never produced two topological polygons within the same cell. Therefore we do not describe the triangulation algorithm for topological polygons in detail but refer instead to the refinement of polygon edges described in section 5.5. The refinement of polygon edges can be understood as a 2-dimensional version of the problem of triangulating (refining) a topological polygon and is hence both easier to illustrate and to understand.

The final result for the triangulation of a topological polygon is illustrated by figure 5.12 (d). Algorithm 5.3 gives a high level description of the corresponding algorithm.

Algorithm 5.3 (Algorithm to Triangulate a Topological Polygon)

```
INPUT:   Unclassified cell C with density field ρ and
         n topological polygons P_i, i = 1,...,n, given as
            point lists [p_{i,1},...,p_{i,m_i}]
OUTPUT:  Set of triangles T_{i,j} , i = 1,...,n, j = 1,...,m_i
            (given as point lists)

p_cell_centroid := (Σ_{i=1}^{n} Σ_{j=1}^{m_i} p_{i,j}) / (Σ_{i=1}^{n} m_i)
for i := 1 to n do                  {for each topological polygon do}
begin
   p_centroid := (1/m_i) Σ_{j=1}^{m_i} p_{i,j}
   v_search := ∇ρ(p_centroid)        {gradient of density field}
   S_search := {x | x := p_centroid + t v_search, t ∈ ℝ}
                                     {define a linear search space}
   S_search := S_search ∩ C          {restrict search space to cell C}
   if n > 1 then                     {more than one topological polygon}
      begin
      Define conical section C_section from cell C by connecting
         p_cell_centroid to p_{i,1},...,p_{i,m_i}
      S_search := S_search ∩ C_section  {restrict search space to C_section}
      end
   let s_start and s_end be the end points of the search space S_search
   if ¬ equal_density_class(s_start, s_end) then
      begin                          {iso-surface intersection exists}
         Compute the 0.5 iso-surface intersection p_int with S_search
         p_centroid := p_int          {new centroid is on iso-surface }
      end
   for j := 1 to m_i do T_{i,j} := [p_{i,j}, p_{i,(j+1) mod m_i}, p_centroid]
                                     {triangulated topological polygon}
end {for loop}
```

5.4.5 Summary

This section presented the subspace polygonization used in Triage Polygonization. Input is a polyhedral subdivision of a density field identifying low, high, and unclassified cells. Only unclassified cells can be intersected by the 0.5 iso-surface and hence only they yield subspace polygons. The three steps of the subspace polygonization are summarized in figure 5.19.

```
SubspacePolygonization ::  DBSPTree DensityField Polyhedron -> DBSPTree
SubspacePolygonization dbspTree densityField boundingBox = subspacePolygons
where
   dbspTreeWithEdges = PrecomputePolygonEdges dbspTree densityField boundingBox
   subspacePolygons  = TraverseTree dbspTreeWithEdges

TraverseTree ::  DBSPTree -> [Polygon]
TraverseTree (DBSPNode plane inDBSPTree outDBSPTree)
   = (TraverseTree inDBSPTree) ++ (TraverseTree outDBSPTree)
TraverseTree dbspLeaf=:(DBSPLeaf _ _ ) = SubspacePolygonsOfCell dbspLeaf

SubspacePolygonsOfCell ::  DBSPTree -> [Polygon]
SubspacePolygonsOfCell (DBSPLeaf edges densityClass)
   | densityClass != Unclassified = []
   = polygons
where
   topologicalPolygons  = MakeTopologicalPolygons edges
   polygons             = TriangulateTopologicalPolygons topologicalPolygons
```

Figure 5.19. *Subspace polygonization.*

In a first step the function `PrecomputePolygonEdges` approximates the 0.5 iso-surface intersections with the faces of unclassified cells by polygon edges. The polygon edges are stored in the corresponding unclassified cells of the DBSP tree. The tree is traversed and for all unclassified cells subspace polygons are formed. The subspace polygons are obtained by triangulating and refining the topological polygons resulting from connecting the precomputed polygon edges. The function `TriangulateTopologicalPolygons` implements the algorithm 5.3.

The quality of the subspace polygonization is improved by refining the precomputed polygon edges. The corresponding algorithm is explained in the next section. Section 5.6 shows that the density field inside a cell can be simplified. The simplified density field is exploited in section 5.7 to yield an improved subspace polygonization.

5.5 Refinement of Face Intersections

The previous section concluded the description of the subspace polygonization. However, a polygonization obtained with the subspace polygonization in its current form is rather rough. This is mainly due to the fact that the intersection of the iso-surface with a face of a cell is only approximated by one edge. Replacing the edge with a polyline improves the approximation.

The definition of the polyhedral subdivision suggests that in most cases an unclassified cell contains a complete rounded edge or corner of a quasi-convolutionally smoothed object. Experiments indicated that for quasi-convolutionally smoothed scenes a polyline with two segments defines a sufficient approximation. Criteria for a more sophisticated adaptive refinement approach are given by von Herzen and Barr [vHB87]. However, the general principle is best illustrated by refining an edge into two segments. Note that edge refinement is only necessary for a fully unclassified face. Figure 5.20 shows a polygon edge (a) and a possible result of the refinement process (b).

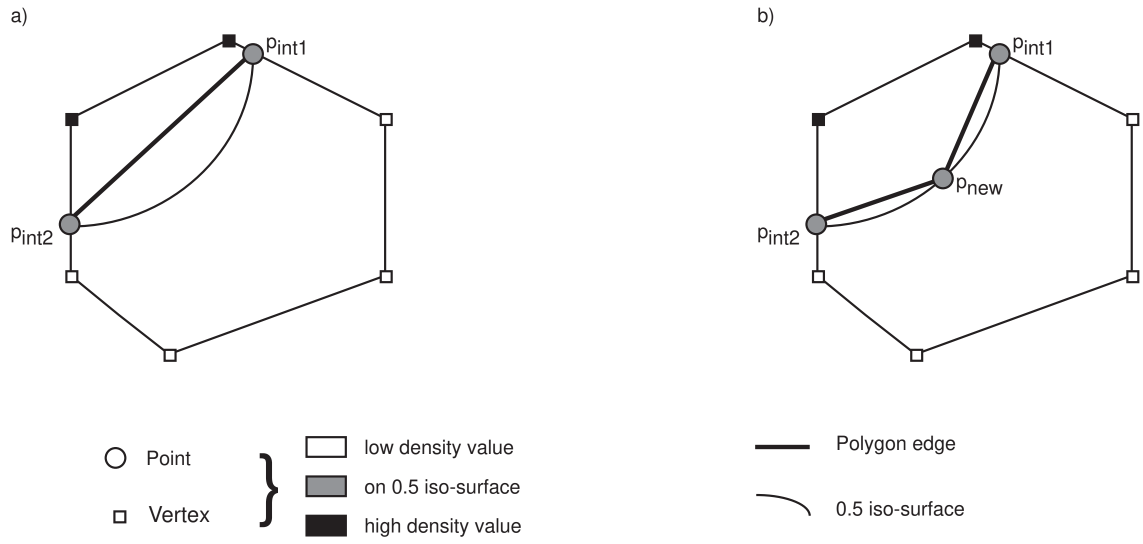

Figure 5.20. *Face intersection before (a) and after (b) refinement.*

It remains to find a suitable refinement point p_{mid}. The point p_{mid} must lie inside the face and, if the face has several 0.5 iso-surface intersections, the corresponding refined polygon edges must not intersect each other. Algorithm 5.4 gives a high-level description of the method to refine polygon edges.

Finding a Refinement Point

A polygon edge on a face is defined by two intersection points p_{int1} and p_{int2} of the face edges with the iso-surface. The edge is refined into two segments by connecting

Algorithm 5.4 (Refining Polygon Edges)

INPUT: A face F with n polygon edges $e_1, \ldots, e_n$
OUTPUT: List of polylines

for i := 1 to n do
begin
  Find for the polygon edge e_i a refinement point $p_{new,i}$, $i = 1, \ldots, n$
    inside face F such that the refined polygon edges
    do not intersect
  Replace the edge $e_i = \overline{p_{start,i} p_{end,i}}$ with the polyline [$p_{start,i}$, $p_{new,i}$, $p_{end,i}$]
end {for loop}

the intersection points to a new point p_{new} on the iso-surface. The quality of the results of Triage Polygonization depends on the choice of the refinement point p_{new} for a polygon edge. Two possible specifications of optimality for p_{new} are

1. The two line segments $\overline{p_{int1} p_{new}}$ and $\overline{p_{new} p_{int2}}$ have equal length
2. The point p_{new} is the point with the highest curvature of the 0.5 iso-surface intersection with the face

A good approximation is achieved by taking both specifications into account. To do this define p_{mid} as the midpoint between the two intersection points (this considers the first specification). From p_{mid} search in direction of the density gradient for an iso-surface intersection. This method reflects the hope that a strong change in the density field indicates a high curvature of the iso-surface (hence taking the second specification of optimality into account).

Since the density gradient in p_{mid} usually does not lie in the face plane we take instead its projection $\nabla_{projF}\rho$ on the face given by

$$\nabla_{proj_F}\rho = \nabla\rho - n_F\langle\nabla\rho, n_F\rangle$$

where n_F is the face normal.

The midpoint p_{mid} of the edge and the projected density gradient in p_{mid} define a linear search space. Since the refinement point p_{new} must lie inside the face the linear search space is restricted to the face, i.e.,

$$S_{search} = \{x \mid x = p_{mid} + t\nabla_{proj_F}\rho(p_{mid}),\ t \in \mathbb{R}\} \cap F$$

The linear search space S_{search} is therefore a line segment. The refinement point p_{new} for the polygon edge (p_{int1}, p_{int2}) is given by the 0.5 iso-surface intersection with the linear search space. If no intersection point exists the edge is not refined, otherwise the edge is replaced with the polyline $[p_{int1}, p_{mid}, p_{int2}]$. Note that the original polygon edge and therefore also the polyline are directed.

The linear search space has an intersection point with the 0.5 iso-surface if its end points s_{start} and s_{end} have different density classes[9]. The intersection point is found by a root search. The search space (and hence the root search) can be shortened by using p_{mid}. If the density class of p_{mid} equals that of s_{start} then p_{mid} becomes the new start point of the linear search space. Otherwise s_{end} becomes the new end point of S_{search}. Figure 5.21 illustrates the two possible cases (without loss of generality the density classes of s_{start} and s_{end} are low and high, respectively).

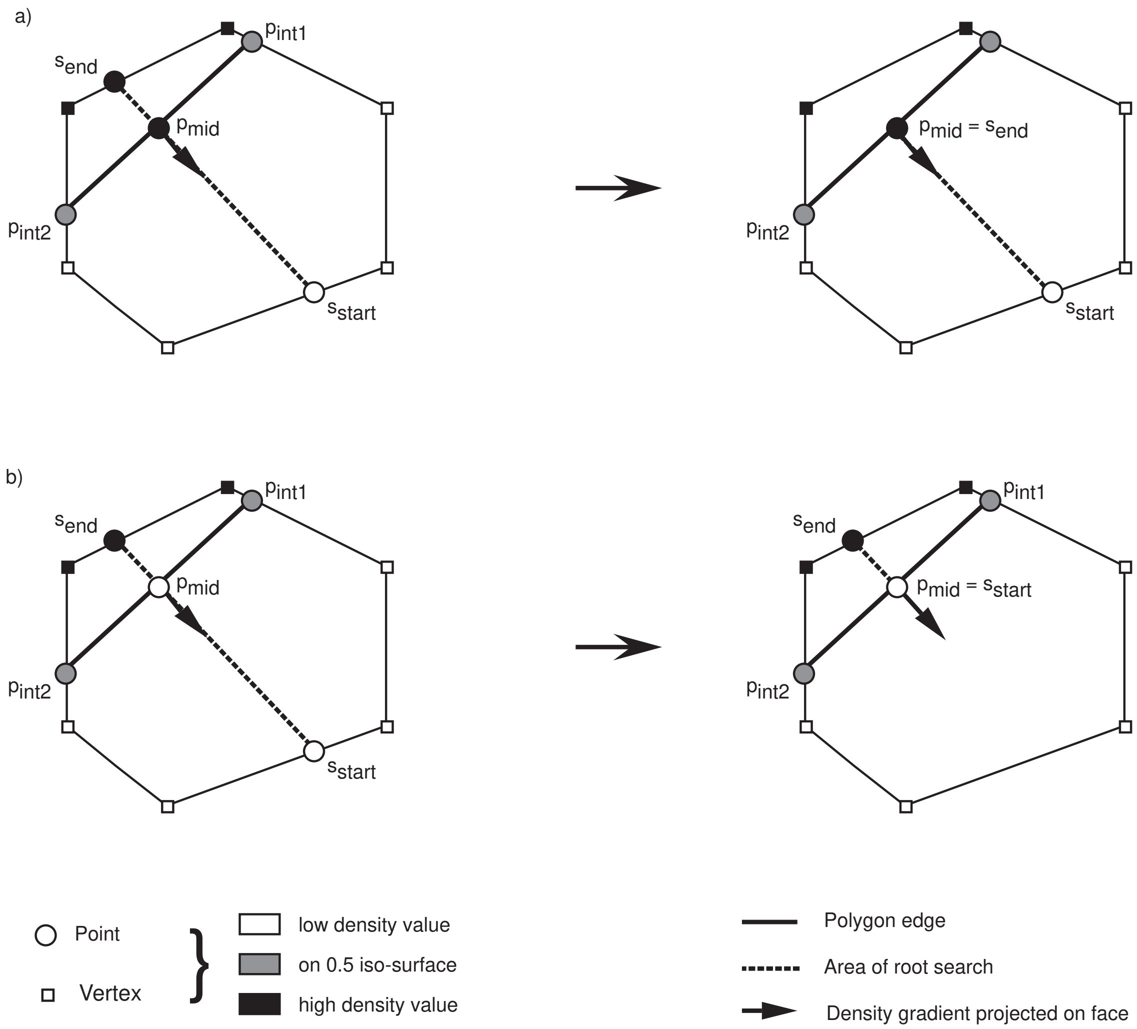

Figure 5.21. *Shortening the linear search space (area of root search) for the edge refinement by using p_{mid}. In (a) and (b) the density class of p_{mid} equals that of the end point and start point of the linear search space, respectively.*

[9]Here we use that the density field is continuous and the assumption that it is "reasonable" smooth.

A root search on the reduced linear search space finds the desired refinement point p_{new}. Connecting the new point p_{new} to the end points of the polygon edge forms the refined edge (see figure 5.22).

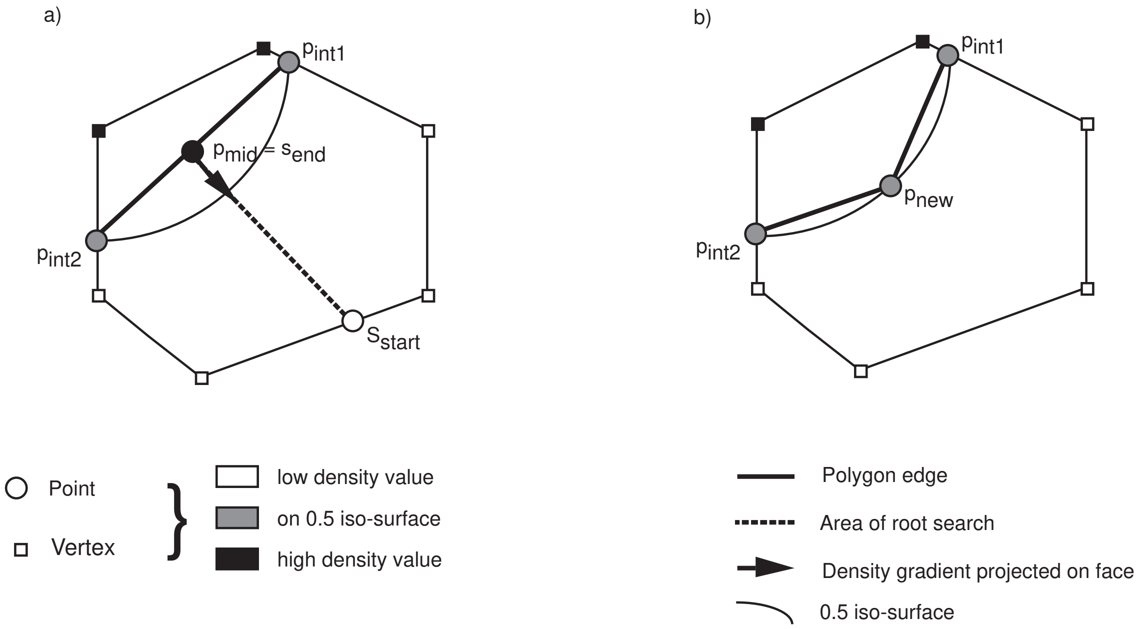

Figure 5.22. *Refinement of a polygon edge.*

Improving the Refinement Process

If a face has more than one iso-surface intersection several problems arise. First note that the end points of the linear search space for the refinement point may have the same density class even though an intersection point exists. This case is illustrated in figure 5.23 (a). If it is known that an iso-surface intersection exists a root search can still be performed. However, part (b) of the figure shows that incorrect intersection points might be found.

The solution to this problem is to restrict the polygon edges to disjoint areas on the face. Disjoint sectors of the face are given by connecting their end points to the centroid of all intersection points. These sectors were already used to resolve ambiguities in the definition of the polygon edges (see subsection 5.4.2 on page 92) and are shown in figure 5.24 (a).

For each sector the corresponding edge is refined as for a face with a single polygon edge. That means a linear search space is defined by its midpoint p_{mid} and the density gradient in p_{mid} projected on the face. The linear search space is restricted to the sector. If its end points have different density classes an intersection point with the 0.5 iso-surface exists. The search space is shortened according to the density class of p_{mid} and its 0.5 iso-surface intersection p_{new} is determined by a root

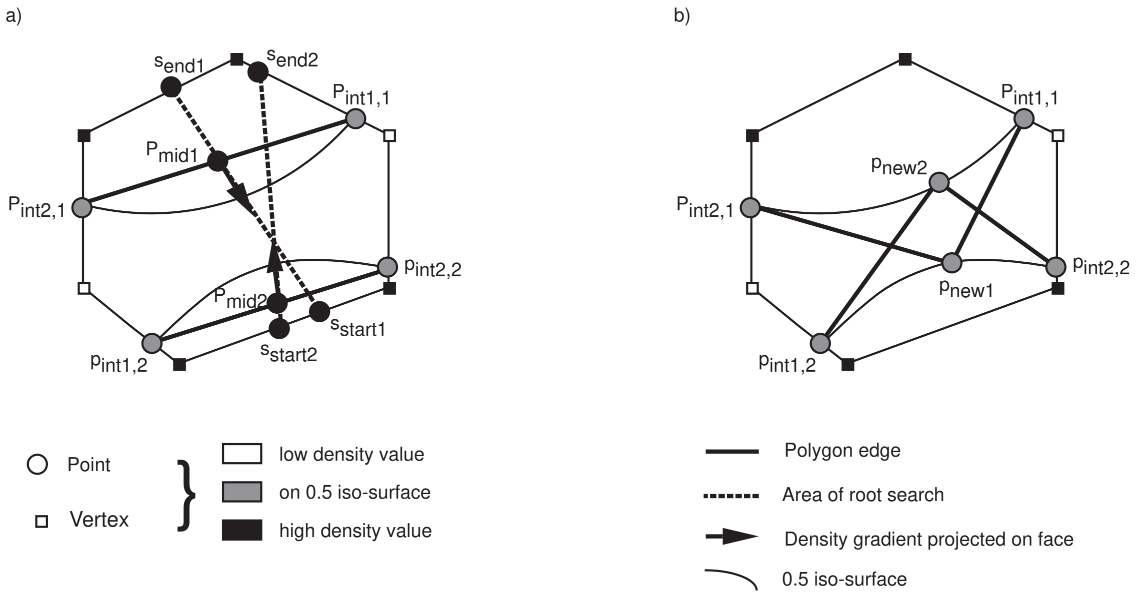

Figure 5.23. *Refinement of two polygon edges on a face. Without modification no refinement point is found (a) or wrong refinement points are computed (b).*

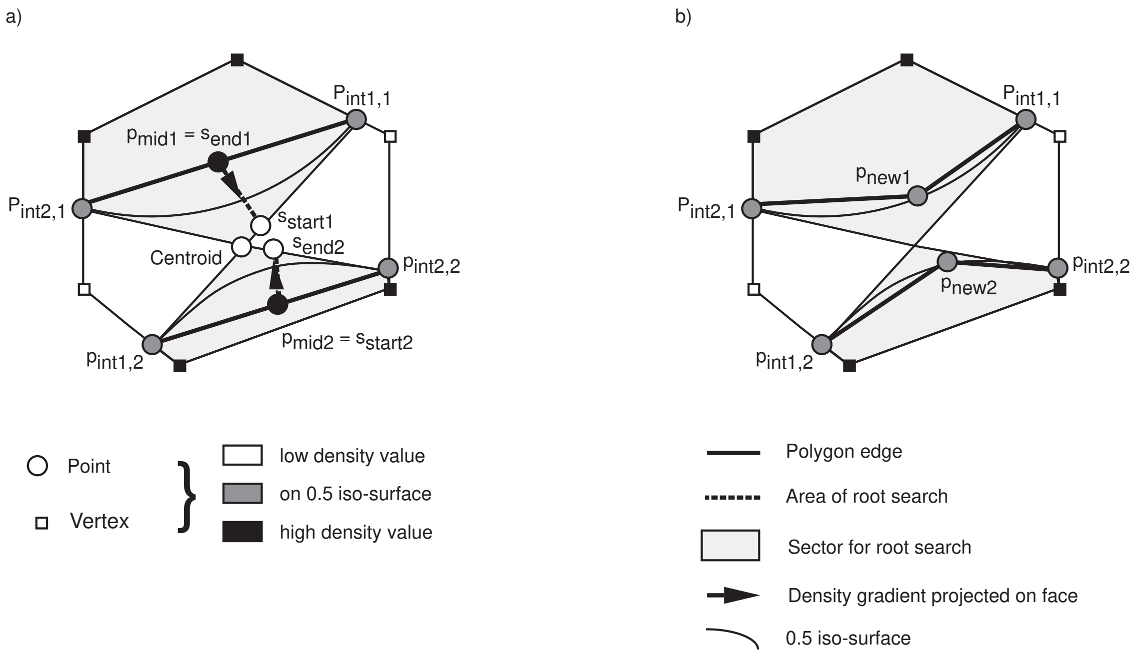

Figure 5.24. *Refinement of two polygon edges on a face. The refinement process is improved by restricting the linear search spaces to disjoint sectors (a). The refined polygon edges do not intersect (b).*

search. The polygon edge is then refined with p_{new}. Figure 5.24 (b) shows that all refined polygon edges lie in disjoint areas of the face and do not intersect.

NOTE 1. Restricting the refined edges to disjoint sectors does not guarantee that a polygon edge can in fact be refined.

NOTE 2. Since a face is convex the intersection points form a convex object as well. The centroid lies inside this convex object; therefore all sectors are disjoint. The center of the face can lie outside the convex hull of all intersection points and therefore can not be used to define disjoint sectors.

Algorithm 5.5 summarizes the algorithm to refine polygon edges.

5.6 Local Density Fields

The subspace polygonization and the refinement of polygon edges described in the previous sections require a frequent evaluation of the density field inside an unclassified cell. A fast evaluation of the density field is hence desireable. The next section shows that a better description of the density field inside a cell also enables an improved subspace polygonization. This section derives a reduced local density field for each cell, which enables a faster density computation and gives information about the topology of the 0.5 iso-surface.

As explained previously the density field ρ^r_{Obj} of an object quasi-convolutionally smoothed with a spherical filter of radius r, is a continuous function

$$\rho^r_{Obj} : \mathbb{R}^3 \to \mathbb{R}$$

defined as an arithmetic tree. The leaves of the arithmetic tree are density fields of convolutionally smoothed half-spaces. The leaf density fields are constant zero (one) at a distance greater (less) than the smoothing radius r to the half-space plane. For a small region in $\mathbb{R}^3$ it can therefore be expected that most branches of an arithmetic tree are constant zero or one. This motivates the introduction of a reduced local density field ρ^r_C for each unclassified cell of a polyhedral subdivision.

Observe that the local density fields for the regions of a rounded half-space H (see figure 5.1) classified as zero, low, high, and one are given by definition 2.5 as

$$\begin{aligned}
\rho^r_{zero}(x) &= 0 \\
\rho^r_{low}(x) &= (1-\alpha)^2(2+\alpha)/4 \\
\rho^r_{high}(x) &= (1-\alpha)^2(2+\alpha)/4 \\
\rho^r_{one}(x) &= 1
\end{aligned}$$

where $\alpha = d/r$, $d =< x, \vec{n}_H >$ is the distance of point x to the plane of the half-space H, and $\vec{n}_H$ is the outward normal of H.

Algorithm 5.5 (Refine Polygon Edges)

INPUT: n polygon edges $(p_{int_1,i}, p_{int_2,i})$, $i = 1, \ldots, n$
   Face F with local[a] density field ρ
OUTPUT: Set of n polylines P_i , $i = 1, \ldots, n$ (as point lists)

$p_{centroid}$:= $\frac{1}{2n}\sum_{i=1}^{n}(p_{int_1,i} + p_{int_2,i})$
for i := 1 to n do
begin
   p_{mid} := $\frac{p_{int_1,i}+p_{int_2,i}}{2}$
   v_{search} := $\nabla_{proj_F}\rho(p_{mid,i})$
   S_{search} := $\{x \mid x$:= $p_{mid,i} + tv_{search}, t \in \mathbb{R}\}$
         {define linear search space}
   S_{search} := $S_{search} \cap F$ {restrict search space to F}
   if $n > 1$ then   {more than one polygon edge}
      begin
      Define sector F_{sector} from face F by connecting
         the centroid $centroid$ to $p_{int_1,i}$ and $p_{int_2,i}$
      S_{search} := $S_{search} \cap F_{sector}$ {restrict search space to F_{sector}}
      end
   let s_{start} and s_{end} be the end points of S_{search}
   if $\neg$ equal_density_class(s_{start},s_{end}) then
      begin   {iso-surface intersection exists}
         {shorten search space}
      if equal_density_class(s_{start},p_{mid}) then s_{start} := p_{mid}
      if equal_density_class(s_{end},p_{mid}) then s_{end} := p_{mid}
      Compute the 0.5 iso-surface intersection p_{new} with S_{search}
      P_i := $[p_{int_1,i}, p_{new}, p_{int_2,i}]$ {refine edge with new point}
      end
   else P_i := $[p_{int_1,i}, p_{int_2,i}]$ {do not refine edge}
end {for loop}

---

[a] For the algorithm it is sufficient to know the density field in all points of the face. Section 5.6 introduces a reduced local density field for a cell. The density field in a face is given by one or the other of the two local density fields of the cells sharing that face. For faster evaluation the density field defined by the smaller arithmetic tree is chosen.

During the polyhedral subdivision of the density field, the density field ρ^r_C local to a cell C, changes only if the cell takes part in a set operation. Tables 5.5 – 5.7 show the local density field $\rho^r_{C_1 \oplus C_2}$ of a cell obtained with a set operation $\oplus^\star$ ($\oplus \in \{\cup, \cap, \setminus\}$) between two cells C_1 and C_2 with local density fields $\rho^r_{C_1}$ and $\rho^r_{C_2}$, respectively. The result is dependent on the density classes of the cells C_1 (rows) and C_2 (columns).

$\cap$	zero	low	unclassified	high	one
zero	0	0	0	0	0
low	0	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_2}$
unclassified	0	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_2}$
high	0	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_1} * \rho^r_{C_2}$	$\rho^r_{C_2}$
one	0	$\rho^r_{C_1}$	$\rho^r_{C_1}$	$\rho^r_{C_1}$	1

Table 5.5. *Density field of the intersection of two cells C_1 and C_2. The top row and left column give the density classes of C_1 and C_2, respectively.*

$\cup$	zero	low	unclassified	high	one
zero	0	$\rho^r_{C_2}$	$\rho^r_{C_2}$	$\rho^r_{C_2}$	1
low	$\rho^r_{C_1}$	$\rho^r_{C_1} + \rho^r_{C_2}$	$\rho^r_{C_1} + \rho^r_{C_2}$	$\rho^r_{C_1} + \rho^r_{C_2}$	1
unclassified	$\rho^r_{C_1}$	$\rho^r_{C_1} + \rho^r_{C_2}$	$\rho^r_{C_1} + \rho^r_{C_2}$	$\rho^r_{C_1} + \rho^r_{C_2}$	1
high	$\rho^r_{C_1}$	$\rho^r_{C_1} + \rho^r_{C_2}$	$\rho^r_{C_1} + \rho^r_{C_2}$	$\rho^r_{C_1} + \rho^r_{C_2}$	-
one	1	1	1	-	-

Table 5.6. *Density field of the union of two cells C_1 and C_2. The top row and left column give the density classes of C_1 and C_2, respectively.*

$\setminus$	zero	low	unclassified	high	one
zero	0	0	0	-	-
low	$\rho^r_{C_1}$	$\rho^r_{C_1} - \rho^r_{C_2}$	$\rho^r_{C_1} - \rho^r_{C_2}$	-	-
unclassified	$\rho^r_{C_1}$	$\rho^r_{C_1} - \rho^r_{C_2}$	$\rho^r_{C_1} - \rho^r_{C_2}$	$\rho^r_{C_1} - \rho^r_{C_2}$	0
high	$\rho^r_{C_1}$	$\rho^r_{C_1} - \rho^r_{C_2}$	$\rho^r_{C_1} - \rho^r_{C_2}$	$\rho^r_{C_1} - \rho^r_{C_2}$	0
one	1	$1 - \rho^r_{C_2}$	$1 - \rho^r_{C_2}$	$1 - \rho^r_{C_2}$	0

Table 5.7. *Density field of the set difference of two cells C_1 and C_2. The top row and left column give the density classes of C_1 and C_2, respectively.*

Example 5.6 As example consider the intersection of two cells C_1 and C_2 with density fields $\rho^r_{C_1}$ and $\rho^r_{C_2}$, respectively. The density field of the intersection of C_1

and C_2 is given as the product of $\rho^r_{C_1}$ and $\rho^r_{C_2}$. If the density class of C_1 is zero then the density field $\rho^r_{C_1}$ is constant zero. Hence the product of the density fields is constant zero. Similar if $\rho^r_{C_1}$ is constant one the product of the density fields is $\rho^r_{C_2}$.

The computation of a local density field is performed during the polyhedral subdivison. Each table for the computation of a density class (tables 5.2 – 5.4) can be combined with the corresponding table for the computation of a local density field (tables 5.5 – 5.7). The local density field is stored in the leaves of the DBSP tree yielding the new data structure:

```
::  DBSPTree = DBSPNode Plane DBSPTree DBSPTree
               | DBSPLeaf PolygonEdges DensityClass DensityField
```

Triage Polygonization uses the local density fields for all computations of density values. Note that the density fields of two cells that meet along a common face are identical on the face. The next section shows that the local density field of a cell gives also topological information about the 0.5 iso-surface. The information is used to obtain an improved subspace polygonization.

5.7 Intersection of Two Half-spaces

Inspecting the polyhedral subdivision produced by Triage Polygonization reveals that the density field inside of many cells is extremly simple. This is due to the fact that the density field of a smoothed half-space is constant zero (one) at a distance greater (smaller) than the smoothing radius to the half-space plane. Section 7.4.5 will show that for the scenes tested the local density field inside of about 30% of the unclassified cells represents a quasi-convolutionally smoothed intersection of two half-spaces. Theorem A.13 states that the 0.5 iso-surface of the density field of a quasi-convolutionally smoothed intersection of two half-spaces is convex. The above results motivate the development of an improved subspace polygonization for cells containing such a simple density field.

A desireable result for our improved subspace polygonization is to obtain less polygons and to approximate the 0.5 iso-surface better than with the triangulation algorithm presented in section 5.4.4

Example 5.7 Figure 5.25 (a) shows the 0.5 iso-surface of the quasi-convolutionally smoothed intersection of two half-spaces. The marked intersection points and polylines are the iso-surface intersections of the cell's edges and faces derived by precomputing the face intersections. The intersection points inside a face result from edge refinement.

The previously introduced subspace polygonization computes the centroid of all intersection points and, if possible, adjusts it such that it lies on the 0.5 iso-surface.

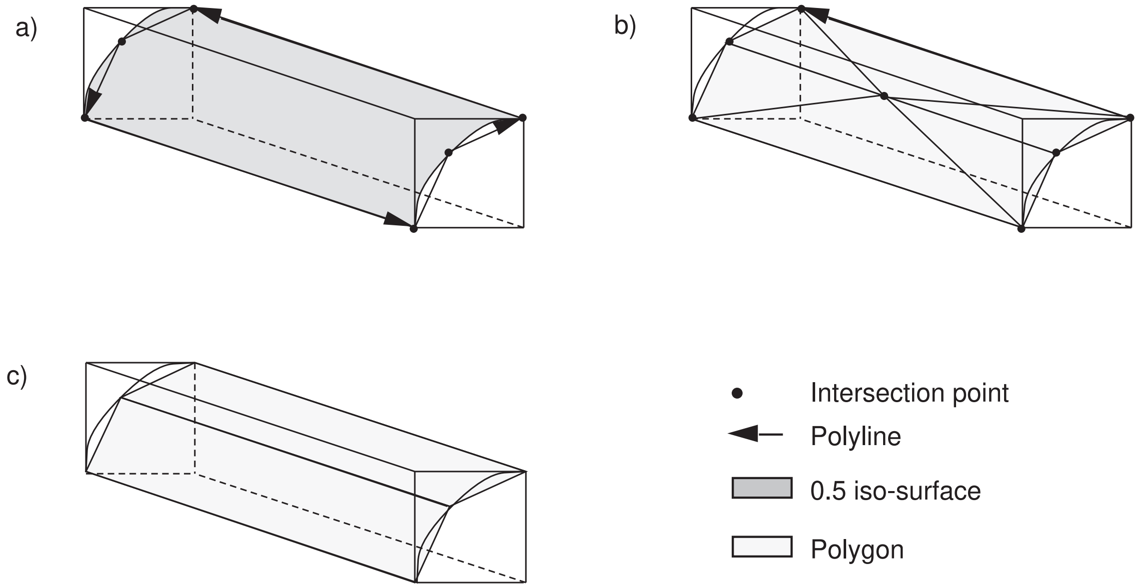

Figure 5.25. *(a) Cell containing a quasi-convolutionally smoothed intersection of two half-spaces. (b) The subspace polygonization leads to fragmentation. (c) Desired result of the subspace polygonization.*

Then all intersection points are connected to the centroid forming a set of triangles. Figure 5.25 (b) gives an example where this procedure yields six pointed triangles. A more desireable result in above situation are two long rectangles as shown in (c).

Definition 2.9 yields that the density field of a quasi-convolutionally smoothed intersection of two half-spaces is constant in points of constant distance to the corresponding half-space planes. The 0.5 iso-surface is hence a convex swept surface. Figure 5.25 (c) suggested to approximate the iso-surface by long rectangles with their major edges given by the line defining the swept surface. This, however, is only possible for cuboidal cells. Figure 5.26 (a) shows a cell with a more complicated geometry (e.g., as resulting from a splitting operation during the polyhedral subdivison).

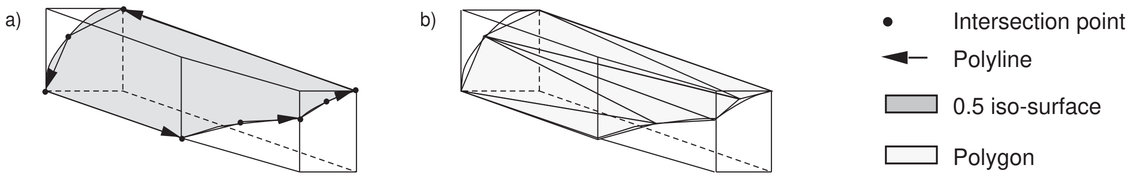

Figure 5.26. *The subspace polygonization for a quasi-convolutionally smoothed intersection of two half-spaces is part of the convex hull of the intersection points.*

Since the iso-surface is convex an optimal polygonization must be convex as well. Also, in order to achieve continuity, the polygonization must intersect the cell's faces in the precomputed polygon edges, i.e., it is a tessellation of the topological polygons formed by connecting the precomputed polygon edges. The improved subspace polygonization is therefore defined as a convex tesselation of the topological polygons.

Recall that a polygon is given as a list of its vertex points in anti-clockwise order. A convex tesselation of a topological polygon is given as its convex hull with all polygons removed, which have an opposite orientation to the topological polygon. Figure 5.26 (a) shows an iso-surface and (b) the resulting subspace polygonization.

We present here an alternative algorithm with expected running time $O(n \log n)$, which produces the tesselation directly.

5.7.1 Convex Tessellation of a Topological Polygon

As in the previous section let their be given a convex topological polygon with all its vertices on a convex 0.5 iso-surface. Note that the centroid of a set of points always lies inside the convex hull of the corresponding point set. Also, since the 0.5 iso-surface is a convex swept surface, the convex tesselation of the topological polygon has no inside polygons, i.e., all polygons of the tesselation share an edge with the topological polygon. An algorithm for a convex tessellation is obtained by detecting for each edge of the topological polygon a polygon of the convex tesselation.

Let T be a topological polygon with n vertices $p_1, \dots, p_n$. Take an arbitrary edge, say (p_1, p_2), and compute the vectors $\vec{e}_1 = p_2 - p_1$ and $\vec{c} = centroid - p_1$. For all vertices $q \in \{p_3, \dots, p_n\}$ of the topological polygon T compute the vector $\vec{e}_2(q) = q - p_1$ and the normal $\vec{n}(q) = \vec{e}_1 \times \vec{e}_2(q)$ of the triangle $[p_1, p_2, q]$. Figure 5.27 depicts the situation.

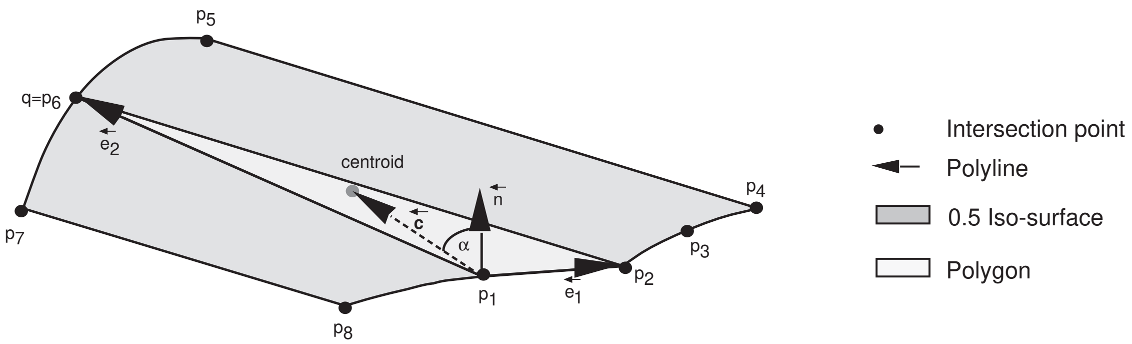

Figure 5.27. *The subspace polygonization for a quasi-convolutionally smoothed intersection of two half-spaces is given by a convex tessellation.*

The triangle $P = [p_1, p_2, q]$ is part of the convex tessellation if the angle $\alpha(q) = \angle(\vec{n}(q), \vec{c})$ is maximal with respect to all points q. If there are several points $q_1, \dots, q_m$ with maximal angle $\alpha(q_i)$, $i = 1, \dots, n$, then they form together a polygon $P = [p_1, p_2, q_1, \dots, q_m]$ of the convex tessellation.

Removing the points of the polygon P from the topological polygon T subdivides T into (at most two) smaller topological polygons for which the algorithm is called recursively. The subdivision is performed by taking all points enclosed by two lists of consecutive points of P including the two points next to the enclosed list of points. The following example illustrates the process.

Example 5.8 Figure 5.27 shows a topological polygon $T = [p_1, \ldots, p_8]$. For the edge (p_1, p_2) there is only one point $q = p_6$ with maximum angle $\alpha(q)$. The lists of points of the polygon $P = [p_1, p_2, q]$ consecutive in T are $[p_1, p_2]$ and $[q]$. The lists of points enclosed by these lists of consecutive points are $[p_3, p_4, p_5]$ and $[p_7, p_8]$. Removing the polygon P from the topological polygon leaves two new topological polygons $[p_2, p_3, p_4, p_5, p_6(= q\ )]$ and $[p_6(= q\ ), p_7, p_8, p_1]$. Applying the tessellation algorithm recursively to these topological polygons yields a polygonization as shown in figure 5.26 (b).

The complete algorithm for a convex tessellation is given in figure 5.28. Note that we assume that no three point of the topological polygon are collinear.

The function `ConvexTessellation` takes as input a (convex) topological polygon and returns a list of polygons forming a convex tessellation. If the topological polygon is a triangle no tessellation is necessary. If the topological polygon has less than three vertices it is a degenerate polygon (i.e., a line or a point) and the tessellation is empty. Otherwise the algorithm takes the edge given by the first two vertices p_1 and p_2 of the polygon and computes the angle $\alpha(q)$ for all vertices $q \neq p_1, p_2$. The maximum angle $\alpha(q)$ is called `maxAngle`. The function `ExtractPolygonPoints` takes the list of vertices of the topological polygon with their angle $\alpha(q)$ and returns all vertices q with maximum angle $\alpha(q)$ and a list of topological polygons lying between any two lists of consecutive points with maximum angle. For computational reasons we move the first point p_1 of the topological polygon to its end. The algorithm `ConvexTessellation` is applied recursively to the newly created topological polygons. The vertices with maximum angle form together a new polygon of the convex tessellation.

The function `ExtractPolygonPoints` takes as input an angle `maxAngle` and a list of vertices q of the topological polygon with angle $\alpha(q)$. The first point p_i of the list must fulfill $\alpha(p_i) =$ `maxAngle`. The list is then divided into a list of maximal length of points with maximum angle (`polygonPoints`), a list of maximal length of points with angle smaller than `maxAngle` (`smallerPoints`), and a list of remaining points starting again with a point with maximum angle (`restPoints`). If the list of remaining points is empty only the vertices with maximum angle are returned (they are part of the new polygon of the tessellation). Otherwise the list `smallerPoints` is enclosed by two lists of consecutive points with maximum angle and forms together with the end point of the first list and the start point of the second list a new topological polygon. Since in this case the list of remaining points is not empty the function is called recursively to find more points with maximum angle. Note that the first point of the list of remaining points (`restPoints`) is guaranteed to have the maximum angle `maxAngle`.

```
ConvexTessellation ::  Polygon -> [Polygon]
ConvexTessellation polygon=:[p1,p2:points]
   | #polygon < 4 = if (#polygon < 3) [] [polygon]
   = [newPolygon:restPolygons]
where
   centroid          = ArithmeticMiddle polygon
   pointsWithAngle   = [(q,Alpha q) \\ q <- points]
   maxAngle          = Maximum (>) (map snd pointsWithAngle)
   (polyPoints,convexTPolys) = unzip (ExtractPolygonPoints maxAngle
                           ([(p2,maxAngle):pointsWithAngle] ++ (p1,maxAngle)))
   newPolygon        = flatten polyPoints
   restPolygons      = flatten (map ConvexTessellation convexTPolys)
   Alpha q = Angle ( (q - p1) Cross (p2 - p1) , centroid - p1 )


ExtractPolygonPoints ::  Real [(Point,Real)] -> [([Point],Polygon)]
ExtractPolygonPoints maxAngle pointsWithAngle
   | isEmpty restPoints = [(polygonPoints,[])]
   = [(polygonPoints,newTPolygon) :  ExtractPolygonPoints maxAngle restPoints]
where
   (polygonPoints`,restPoints`)  = DecomposeWhile equalToMaxAngle pointsWithAngle
   (smallerPoints`,restPoints)   = DecomposeWhile smallerThanMaxAngle restPoints`
   polygonPoints  = map fst polygonPoints`
   smallerPoints  = map fst smallerPoints`
   newTPolygon    = [(fst (hd restPoints)),(last polygonPoints):smallerPoints]
   smallerThanMaxAngle (_,d)  = d < maxAngle
   equalToMaxAngle (_,d)      = d == maxAngle
```

Figure 5.28. *Convex tessellation of a topological polygon.*

The algorithm employs a divide and conquer tactic resulting in a worst case running time of $O(n^2)$ and an expected running time $O(n \log n)$. The solution in figure 5.26 (c) was constructed with this algorithm. Note that the algorithm yields polygons with maximum size. That means, e.g., a rectangle on the convex hull is never replaced by two triangles.

5.8 Summary

This chapter introduced Triage Polygonization, a polygonization scheme designed for quasi-convolutionally smoothed objects. Triage Polygonization is performed in three steps: polyhedral subdivision, computation of tree polygons, and a subspace polygonization. All algorithms required to achieve these tasks were presented in this chapter. Where appropriate the algorithms were given directly in executable functional code. To conclude this chapter we summarize the algorithm for Triage Polygonization. Figure 5.29 shows the result.

```
TriagePolygonization ::  Real CSGObject -> [Polygon]
TriagePolygonization radius csgObject = treePolygons ++ subspacePolygons
where
   dbspTree    = (DCSG2DBSP o (CSG2DCSG radius)) csgObject
                                 // DBSP tree with local density fields
   boundingBox = BoundingBox csgObject
   treePolygons      = TreePolygons dbspTree boundingBox
   subspacePolygons  = SubspacePolygonization dbspTree boundingBox
```

Figure 5.29. *Triage Polygonization.*

The composition of the functions `CSG2DCSG` and `DCSG2DBSP` subdivides the density field defining a quasi-convolutionally smoothed CSG object in a BSP-like manner into low, high, and unclassified polyhedral cells. All points in a low and high cell lie inside and outside the 0.5 iso-surface, respectively. The points in an unclassified cell may lie on either side of the 0.5 iso-surface or on it. The 0.5 iso-surface lies therefore either between low and high cells or inside of unclassified cells. The parts of the 0.5 iso-surface separating low from high cells are called tree polygons and are extracted by the function `TreePolygons`. The function `SubspacePolygonization` yields the subspace polygons approximating the iso-surface inside of unclassified cells. The tree polygons and the subspace polygons form the polygonization of a quasi-convolutionally smoothed object.

The quality of the polygonization is improved by refining the precomputed polygon edges and the subspace polygons. Section 5.5 introduced an algorithm for the refinement of polygon edges, which we used in the implementation of the function `PrecomputePolygonEdges`. We indicated that our refinement scheme is sufficient for most quasi-convolutionally smoothed scenes but mentioned that also a more advanced adaptive refinement scheme can be employed.

Efficiency is improved by computing local density fields for the unclassified cells of the polyhedral subdivision. A positive side-effect is that the complexity of the 0.5 iso-surface inside a cell can be determined. Unclassified cells containing a quasi-convolutionally smoothed intersection of two half-spaces are identified and polygonized with an improved subspace polygonization scheme. The improved method uses the fact that a quasi-convolutionally smoothed intersection of two half-spaces is convex.

CHAPTER 6

Implementation Details

The previous chapter introduced Triage Polygonization, a new polygonization method developed for quasi-convolutionally smoothed objects. Most of the algorithm was given directly in executable functional code. This chapter explains some implementation details, deals with numerical problems, and adapts Triage Polygonization to an extended object model.

We first present some implementation details for the polyhedral subdivision and the subspace polygonization of Triage Polygonization. A description of the implementation of two important subtasks of Triage Polygonization follows: the root search used to find the 0.5 iso-surface intersection with a line and the computation of the gradient of a density field used in the definition of a linear search space.

An interesting question is whether the polygonized surface is continuous. We show under which circumstances discontinuities can arise and that in most cases they can be recognized during polygonization.

We then extend the object model with clipping planes and allow different rounding radii for a quasi-convolutionally smoothed object. The resulting changes for Triage Polygonization are only minor and are explained briefly.

The next section deals with the problem of numerical stability. To increase numerical stability we test for iso-surface intersections against a displaced iso-surface. Problems due to the use of logical alternatives are reduced by using ϵ-intervals.

The chapter concludes with a description of the polygonization of a complete scene. A scene is usually composed of several smoothed and unsmoothed objects. Triage Polygonization polygonizes only quasi-convolutionally smoothed objects. A scene is polygonized by using a b-rep algorithm for unsmoothed CSG objects and Triage Polygonization for quasi-convolutionally smoothed CSG objects.

6.1 Implementation of the Polyhedral Subdivision

The first step of Triage Polygonization is a polyhedral subdivision of a density field in a BSP-like manner. The density field is defined by transforming a quasi-convolutionally smoothed CSG object with a given smoothing radius into an arithmetic tree. The result of the polyhedral subdivision is a DBSP tree which partitions the density field into zero, low, high, one, and unclassified cells. The DBSP tree is built by replacing each primitive object of the CSG object with polyhedral cells defining regions of non-zero density in the density field. The resulting object is called a DCSG object and is transformed into a DBSP tree with the algorithm `DCSG2DBSP` described in detail in section 5.3.

Fragmentation of the Density Field

When reviewing the polyhedral subdivision algorithm it becomes apparent that the DBSP tree for a DCSG object heavily subdivides the corresponding density field. This means that a convex region with a constant density class, e.g., a low region, is often fragmented into several cells. The reason for this is that every face of a cell of the DCSG object defines a partitioning plane in the DBSP tree. Many partitioning planes are unbounded and lead to fragmentation of the density field. A small number of unbounded partitioning planes is hence desirable. The number of unbounded partitioning planes depends strongly on the order in which the faces of a cell are chosen if defining partitioning planes for a DBSP tree.

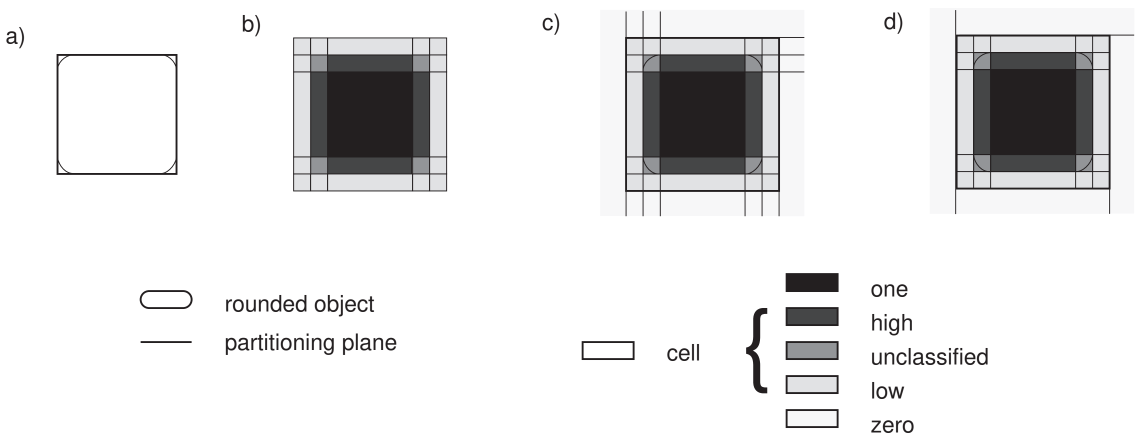

Figure 6.1. *A quasi-convolutionally smoothed primitive object (a) and the corresponding DCSG object (b). Two possible DBSP trees for the DCSG object are given in (c) and (d). The number of unbounded partitioning planes in the DBSP tree can vary strongly. For the quasi-convolutionally smoothed square in (a) the worst case are* 12 *unbounded partitioning planes (c) and the best case are* 4 *unbounded partitioning planes (d).*

Example 6.1 Figure 6.1 shows a quasi-convolutionally smoothed square (a) and the corresponding DCSG object (b). For better illustration, we have used a 2-dimensional object. The following results are similar for a cube (the corresponding 3-dimensional object) and are given in brackets. The DCSG object consists of $5^2 = 25$ ($5^3 = 125$ for a cube) non-zero cells. Transforming the DCSG object into a DBSP tree produces, in the worst case, a partitioning of the density field with 12 (15) unbounded partitioning planes, shown in (c), and in the best case a partitioning with 4 (5) unbounded partitioning planes, shown in (d).

Bounding a DCSG Object

The solution to the fragmentation problem is found by recognizing that the non-zero regions of the density field of a quasi-convolutionally smoothed primitive object are bounded by the r-ohs-planes of the object. Hence the density field is first partitioned with the r-ohs-planes and only the region inside all r-ohs-planes is partitioned according to the DCSG object. Since a primitive object is convex, its r-ohs-planes form a linear DBSP tree. Figure 6.1 (d) shows the result obtained by forming a linear DBSP tree from the r-ohs-planes (the planes separating regions with zero and non-zero density values) of a DCSG object before inserting the DCSG object itself.

In many cases a quasi-convolutionally smoothed object consists of several primitive objects. With above discussion the corresponding DCSG object must have bounding planes for each of its primitive objects. During DBSP tree construction a DCSG object is inserted into an already existing DBSP tree. This means the DCSG object is split on partitioning planes. To retain the r-ohs-planes of a quasi-convolutionally smoothed primitive object as bounds for the corresponding DCSG object we extend the data structure for a DCSG object by a set of bounding faces. The bounding faces must bound the non-zero density field of a quasi-convolutionally smoothed primitive object. Since a primitive CSG object is convex the bounding faces are given as the faces of the polyhedral region, which lies inside all r-ohs-planes of the primitive object. Figure 6.2 gives the new data structure for a DCSG object.

```
::  DCSGObject = DUnion DCSGObject DCSGObject
               | DIntersection DCSGObject DCSGObject
               | DSetDifference DCSGObject DCSGObject
               | DPrimitive Polyhedron DensityClass
               | DAssembly [Face] [DCSGObject]
```

Figure 6.2. *New data type of a DCSG object.*

Here a polyhedral primitive of a quasi-convolutionally smoothed CSG object is given as a "DCSG assembly". This is a list of DCSG objects (representing the non-zero cells in the density field of the quasi-convolutionally smoothed CSG object) and the list of bounding faces of the non-zero density field. The function `CSG2DCSG` from figure 5.6 must be adapted to the new data type. The only change required is the transformation of a quasi-convolutionally smoothed primitive CSG object into a DCSG assembly. To do this compute additionally to the classified cells the faces bounding the non-zero density field of the quasi-convolutionally smoothed primitive CSG object. Figure 6.3 gives the extended function `CSG2DCSG`. The function `ClassifiedCell` is the same as defined in figure 5.6.

```
CSG2DCSG ::  Real CSGObject -> DCSGObject

CSG2DCSG r (Union obj1 obj2)
   = DUnion (CSG2DCSG r obj1) (CSG2DCSG r obj2)
CSG2DCSG r (SetDifference obj1 obj2)
   = DSetDifference (CSG2DCSG r obj1) (CSG2DCSG r obj2)
CSG2DCSG r (Intersection obj1 obj2)
   = DIntersection (CSG2DCSG r obj1) (CSG2DCSG r obj2)
CSG2DCSG r (Primitive (Intersection hs_Planes))
   = DAssembly boundingFaces classifiedCells
where
   r_ohs_Planes      = map (Translation r) hs_Planes
   r_ihs_Planes      = map (Translation (-r)) hs_Planes
   outerPolyhedron   = MakePolyhedron r_ohs_Planes
   polyhedra         = PartitionWith outerPolyhedron (hs_planes ++ r_ihs_Planes)
   boundingFaces     = FacesOf outerPolyhedron
   classifiedCells   = map (ClassifiedCell r hs_Planes) polyhedra
```

Figure 6.3. *Transforming a quasi-convolutionally smoothed CSG object into a DCSG object.*

Next we have to adapt the function `DCSG2DBSP` from figure 5.8 to the extended data structure of a DCSG object. To form a DBSP tree a DCSG object is inserted into an initially empty DBSP tree. Recall that a DCSG object is split or transformed into a DBSP tree by splitting or transforming, respectively, its child objects. A DCSG assembly is split on a partitioning plane by splitting its (primitive) DCSG objects and the bounding faces. The DCSG assembly is transformed into a DBSP tree by building a linear tree from its bounding faces and then inserting its (primitive) DCSG objects with a union operation. The new definition of the function `DCSG2DBSP` is given in figure 6.4. The function `LinearDBSPTree` has already been defined in figure 5.8.

6.2 Implementation of the Subspace Polygonization

The subspace polygonization forms the third and last step of Triage Polygonization and polygonizes the 0.5 iso-surface inside all unclassified cells of the polyhedral

```
DCSG2DBSP ::  DCSGObject -> DBSPTree

DCSG2DBSP (DUnion obj1 obj2) = UnionDBSP_DCSG (DCSG2DBSP obj1) obj2
DCSG2DBSP (DIntersection obj1 obj2) = IntDBSP_DCSG (DCSG2DBSP obj1) obj2
DCSG2DBSP (DSetDifference obj1 obj2) = SetDifDBSP_DCSG (DCSG2DBSP obj1) obj2
DCSG2DBSP (DAssembly bounds primitiveDCSG_objects)
   = foldl UnionDBSP_DCSG (LinearDBSPTree Zero bounds) primitiveDCSG_objects
DCSG2DBSP (DPrimitive (Polyhedron faces) densityClass)
   = LinearDBSPTree densityClass faces
```

Figure 6.4. *Transforming a DCSG object into a DBSP tree.*

subdivision. Section 5.4 has presented the corresponding algorithm. We only make one remark with regard to the precomputation of polygon edges.

Recall subsection 5.4.2 where a polygon edge is computed for an unclassified face by computing density values in the density field. Section 5.6 introduced for each cell a local density field. Since the global density field is continuous, the local density fields of two adjacent cells are identical on the face common to those two cells. Therefore the computation of density values on a face can be made with either of the density fields of the cells sharing the face. For efficiency reasons the density field with a simpler arithmetic tree is taken. The simplicity of a arithmetic tree is given by the number of its density half-spaces.

6.3 Root Search

An important part of Triage Polygonization is the computation of the 0.5 iso-surface intersection with a linear search space. Such a computation is necessary for the computation of intersection points on a face (subsection 5.4.2), for the refinement of topological polygons (subsection 5.4.3), and for the refinement of polygon edges (section 5.5).

We compute in Triage Polygonization the 0.5 iso-surface intersection with a linear search space with a root search. A precondition for the root search is that the endpoints of the linear search space lie on different sides of the 0.5 iso-surface. A (bounded) linear search space is a line segment $p(t)$ with a start point p_{start} and an end point p_{end} and is parameterized as

$$p(t) = p_{start} + t(p_{end} - p_{start}),\ t \in [0,1] \tag{6.1}$$

The density values on the linear search space are translated by the 0.5 iso-surface threshold to give a parameterized density field

$$\begin{aligned} &\rho_{search} : [0,1] \to \mathbb{R} \\ &\rho_{search}(t) = \rho(p(t)) - 0.5 \end{aligned} \tag{6.2}$$

The root of ρ_{search} is then the parameter for the 0.5 iso-surface intersection with the linear search space. As root search we choose a *regula falsi* method [PVTF92]. To accelerate worst case convergence we interleave it with a binary search. Figure 6.5 gives the function `IsosurfaceIntersection` to find the 0.5 iso-surface intersection with a linear search space. Inputs are the start and the end point of the linear search space and a density field.

```
IsosurfaceIntersection ::  Point Point DensityField -> Point
IsosurfaceIntersection start_point end_point densityField = intersectionPoint
where
   searchDirection   = end_point - start_point
   RayAt parameter   = start_point + (parameter * searchDirection)
   DensityOnRayAt t  = (Density (RayAt t) densityField) - 0.5
   intersectionParameter = SecantRoot False tolerance DensityOnRayAt
                        (0.0, DensityOnRayAt 0.0) (1.0, DensityOnRayAt 1.0)
   tolerance         = GeometricTolerance * 0.5 / (length searchDirection)
   intersectionPoint = RayAt intersectionParameter

SecantRoot ::  Bool Real (Real -> Real) (Real,Real) (Real,Real) -> Real
SecantRoot flag tolerance f left=:(t_left, f_left) right=:(t_right, f_right)
   | lengthParameterInterval <= tolerance = t_new
            {if search interval small enough, return new parameter}
   | (abs f_left) <= RootTolerance = t_left
            {if density in left point is small enough, return its parameter}
   | (abs f_right) <= RootTolerance = t_right
            {if density in right point is small enough, return its parameter}
   | (f_left * f_new) < 0.0 = SecantRoot (not flag) tolerance f left new
   = SecantRoot (not flag) tolerance f new right
where
   t_new = t_left + (lengthParameterInterval * ratio)
   f_new = f t_new
   new = (t_new, f_new)
   ratio = if flag (0.5) (f_left / (f_left - f_right))
            {binary subdivision every second iteration}
   lengthParameterInterval = t_right - t_left
```

Figure 6.5. *Root finder for a linear search space.*

The function `RayAt` implements equation 6.1 and `DensityOnRayAt` gives the parameterized density field ρ_{search} for the linear search space. The function `Secant Root` finds the root of a function f in an interval $[t_{left}, t_{right}]$. In our case f is the parameterized density field ρ_{search} and the interval $[t_{left}, t_{right}]$ gives the parameters of a subsegment of the linear search space $\overline{p_{start}p_{end}}$. Initially $t_{left} = 0$ and $p(t_{left}) = p_{start}$

and similarly $t_{right} = 1$ and $p(t_{right}) = p_{end}$. By assumption the parameterized density field ρ_{search} has different signs in the end points of the parameter interval. Since the density field is continuous a root exists.

A root is found, if the length of the linear search space is not bigger than half a predefined constant $c_{geometric\_tolerance}$, i.e.,

$$\|p(t_{right}) - p(t_{left})\| \leq \frac{c_{geometric\_tolerance}}{2} \tag{6.3}$$

which is implemented in `SecantRoot` as

$$|t_{right} - t_{left}| \leq \frac{c_{geometric\_tolerance}}{2\|p_{start} - p_{end}\|}$$

As result the parameter computed with the next iteration of `SecantRoot` is returned.

Similarly a root is found if the value of the parameterized density field in the end points of the search interval is not larger than a predefined constant `RootTolerance`, i.e.,

$$|\rho_{search}(t_{left})| \leq c_{root\_tolerance} \vee |\rho_{search}(t_{right})| \leq c_{root\_tolerance} \tag{6.4}$$

The corresponding parameter (t_{left} or t_{right}) is returned as the parameter of the 0.5 iso-surface intersection of the linear search space. The next subsections offer an explanation for these stopping conditions and suggests values for the constants $c_{geometric\_tolerance}$ and $c_{root\_tolerance}$.

If no root is found `SecantRoot` computes the zero crossing

$$t_{new} = t_{left} + \frac{t_{right} - t_{left}}{f(t_{left}) - f(t_{right})} f(t_{left}) \tag{6.5}$$

of the secant through $f(t_{left})$ and $f(t_{right})$. The point t_{new} subdivides the interval $[t_{left}, t_{right}]$ and dependent on the sign of $f(t_{new})$ the root search is recursively performed on the interval $[t_{left}, t_{new}]$ or $[t_{new}, t_{right}]$.

To guarantee convergence at every other iteration of `SecantRoot` a binary search is performed. This is achieved by choosing t_{new} as the mid point between t_{left} and t_{right}.

Geometric Tolerance

An important aspect of the root search is the quality of the approximation to the 0.5 iso-surface intersection. This becomes important in the subspace polygonization of an unclassified cell. If the intersection points are not computed exactly the precomputed polygon edges for a cell can not be connected to form a closed topological polygon.

Figure 6.6 shows as an example two neighbored faces F_1 and F_2 with collinear but not identical edges. Such a case can result from face fragmentation during DBSP tree construction. Though the 0.5 iso-surface intersections are identical for

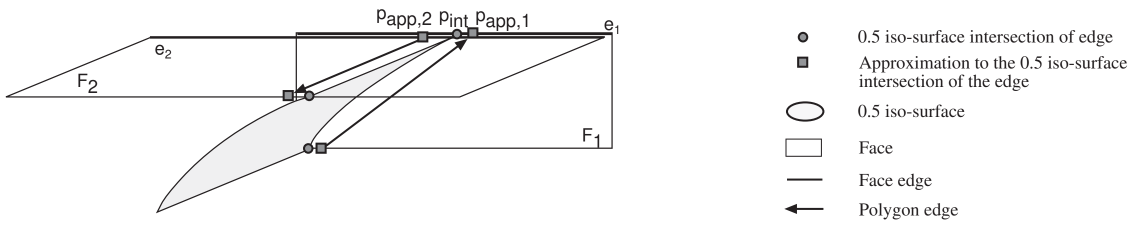

Figure 6.6. *Face fragmentation can lead to faces F_1 and F_2 with collinear but not identical edges e_1 and e_2. Though the iso-surface intersections of e_1 and e_2 are identical the root finder yields different results $p_{app,1}$ and $p_{app,2}$, respectively.*

the edges e_1 and e_2 the root finder yields different approximations $p_{app,1}$ and $p_{app,2}$, respectively.

Both points are part of a polygon edge on the corresponding face. For the subspace polygonization the polygon edges are connected to form a topological polygon. The points $p_{app,1}$ and $p_{app,2}$ are only recognized as equal if their distance is less than a predefined tolerance $c_{geometric}$:

$$||p_{app,2} - p_{app,1}|| \leq c_{geometric} \tag{6.6}$$

which under idealized conditions is fulfilled if

$$||p_{app,i} - p_{int}|| \leq \frac{c_{geometric}}{2}, \quad i \in \{1, 2\}$$

The latter condition motivates equation 6.3.

Note that the problem of different approximations to identical intersection points is avoided by computing the 0.5 iso-surface intersection only once for an edge shared by two faces. Wyvill, McPheeters, and Wyvill [WMW86b] introduce an efficient data structure for this scheme based on hash tables. Unfortunately we could not implement this data structure because the current implementation of CLEAN provides neither a working array data type nor pointers. Note though, that our approach simplifies Triage Polygonization and we can reuse selected existing libraries.

Root Tolerance

If the derivative of the parameterized density field ρ_{search} is close to zero, the denominator of equation 6.5 becomes very small (theorem A.10). Since we assume that the density values in the end points of the parameterized linear search space have different signs, the denominator of equation 6.5 is only zero if the density values are themselves zero.

To prevent division by zero in the root search it is sufficient to test whether the density values in the end points of the search interval are within a so-called root tolerance t_{root} of zero (see equation 6.4). In this case the iteration of the root search is stopped and the corresponding point is returned as root of the parameterized density field. Note that the error in the position of the root can be arbitrarily large if the gradient in the vicinity of the root goes to zero.

Results

We tested Triage Polygonization with the example scenes introduced in section 2.5. In our implementation we chose:

$$\begin{aligned} c_{geometric} &= 10^{-6} \\ c_{root} &= 10^{-9} \end{aligned}$$

The root search was applied to linear search spaces between 0.0001 and 10 units long. Dependent on the complexity of the quasi-convolutionally smoothed object we measured 5–12 iterations on average. In the best case one iteration was sufficient to find the root and in the worst case 21 iterations were necessary.

In most cases the root search terminated because equation 6.4 was fulfilled rather than equation 6.3. However equation 6.6 always proved valid for our test scenes.

Remarks

Note that the rate of change of the density field and its gradient is limited by two *Lipschitz constants* L and G, respectively. The 0.5 iso-surface of a quasi-convolutionally smoothed object is therefore a so-called "LG-implicit surface".

Kalra and Barr [KB89] give an algorithm guaranteed to numerically find all intersections of an LG-implicit surface with a ray, which may be unbounded.

6.4 Gradient of a Density Field

The gradient of a density field is used to define a linear search space for the edge refinement (section 5.5) and for the refinement of a topological polygon (subsection 5.4.4). Additionally section 6.8 will show that the gradient of a density field is computed at each vertex of a polygon to enable Gouraud shading of a polygonized scene. We obtain the gradient of a density field by differentiating the arithmetic tree defining the density field (see definition 2.5).

Corollary 6.1 (Gradient of a density field) *Let be given the density field of a quasi-convolutionally smoothed object Obj as defined in definition 2.5. Then the gradient of the density field* ρ^r_{Obj} *at a point* x *is defined as*

$$\nabla \rho_H^r(x) = \begin{cases} \vec{0} & \alpha \geq 1 \\ \vec{0} & \alpha \leq -1 \\ -\frac{3}{4r}(1-\alpha^2)\vec{n}_H & otherwise \end{cases} \tag{6.7}$$

$$\nabla \rho_{A\cup B}^r(x) = \nabla \rho_A^r(x) + \nabla \rho_B^r(x) \tag{6.8}$$

$$\nabla \rho_{A\cap B}^r(x) = \rho_A^r(x) \nabla \rho_B^r(x) + \rho_B^r(x) \nabla \rho_A^r(x) \tag{6.9}$$

$$\nabla \rho_{A\setminus B}^r(x) = \nabla \rho_A^r(x) - \nabla \rho_B^r(x) \tag{6.10}$$

where $\alpha = d/r$, $d =< x, \vec{n}_H >$ *is the distance of point* x *to the plane of the half-space* H, $\vec{n}_H$ *is* H*'s surface normal,* $\vec{0}$ *is the zero vector,* $< .,. >$ *is the dot product (inner product) of two vectors, and* A *and* B *are CSG objects.*

6.5 Continuity

This section examines whether continuity can be achieved in the polygonization produced by Triage Polygonization. In section 4.3.3 we encountered three conditions sufficient to ensure continuity of a polygonized surface:

1. Cells that meet at a common cell edge share a common intersection point (*edge continuity*).

2. Cells that meet along a common face share common polygon edges (*face continuity*).

3. The subspace polygonization inside a cell is continuous, i.e., every polygon edge inside a cell is shared by a neighbored polygon.

Since the subspace polygonization performs only a triangulation or convex tessellation of a topological polygon formed from the polygon edges, the third condition is always fulfilled.

6.5.1 Face Continuity

Face continuity is given if cells that meet along a common face share common polygon edges. Three cases for cells sharing a common face occur. If two unclassified cells share a common face then face continuity is automatically fulfilled since polygon edges are precomputed and shared by the cells sharing the face.

If two classified (i.e., low, high, zero, or one) cells share a common face, then either the common face is a tree polygon or no polygon edges are computed. In both cases face continuity is given.

The third case occurs for a face shared by an unclassified cell and a classified cell (a simple unclassified face). A classified cell is not intersected by the 0.5 iso-surface, but may have a tree polygon lying on its boundary. Hence all polygon edges lie on edges of the shared face. The algorithm for the 0.5 iso-surface intersection with a simple unclassified face tests all edges whether they lie on the 0.5 iso-surface. Therefore the correct polygon edges are found and face continuity exists.

6.5.2 Edge Continuity

Edge continuity is given if two cells sharing the same edge have the same intersection point with the 0.5 iso-surface. This might not be the case if an edge is subdivided during the polyhedral subdivision. Then, since edges are not shared by neighbored cells, different intersection points with the 0.5 iso-surface can be computed for the edge separating two adjacent faces. The consequence is a discontinuity in the polygonized surface. An example is shown in figure 6.7.

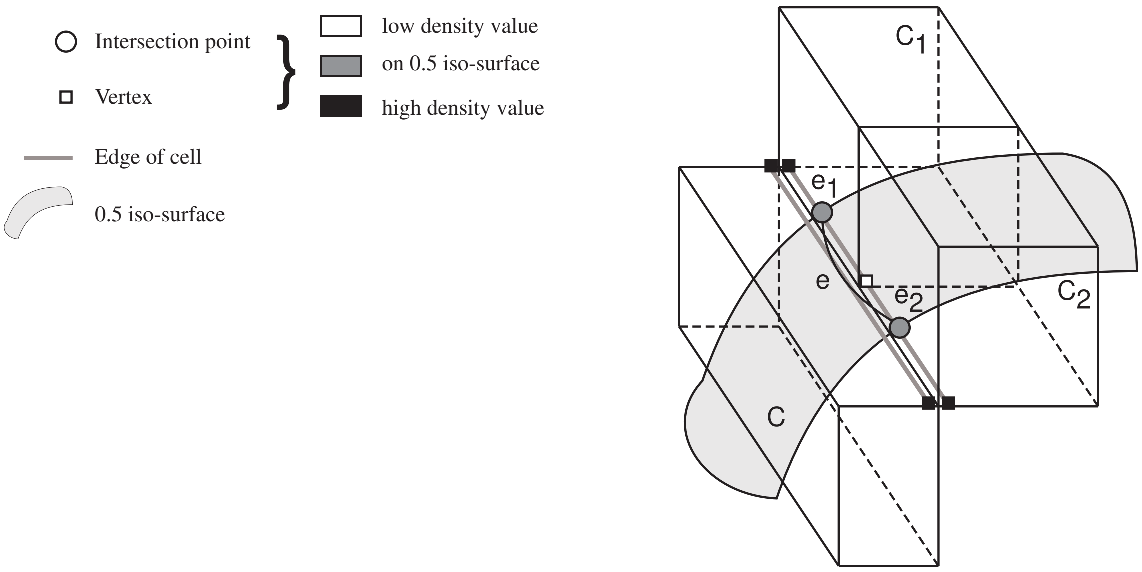

Figure 6.7. *The edges e_1 and e_2 are not shared by the cell C, which has instead the edge e. Both the edge e_1 and the edge e_2 have an iso-surface intersection, but for the edge e no intersection point is found, because both end points of e have a high density value. As a result the polygonized iso-surface is not continuous.*

Example 6.2 In figure 6.7 the edges e_1, e_2, and e belong to the cells C_1, C_2, and C, respectively. Since the edges e_1 and e_2 are not shared by the cell C the edges may have intersection points with the 0.5 iso-surface which are not found for the edge e. As a result the polygonized iso-surface has a discontinuity along the edge e.

6.5.3 Discontinuities

The previous subsection has shown that edge continuity can not be guaranteed. The described discontinuities are due to edge fragmentation and can only be detected by visually checking the polygonized surface. Note, though, that these discontinuities can only occur if a cell's edge has two or more iso-surface intersections. We think this is very unlikely for our polyhedral subdivision.

Since neighbored faces do not share a common edge, different intersection points may be computed for the edges where two faces meet. This might result in an unclosed topological polygon in the subspace polygonization (see the discussion of the geometric tolerance on page 121). In the current implementation Triage Polygonization stops if a topological polygon can not be closed (compare figure 5.18).

In all scenes tested by us the polygonized surface was always continuous and we think discontinuities are extremely rare.

6.5.4 Guaranteeing Continuity

Above problems can be avoided by sharing edges between adjacent faces. Wyvill et al. (see subsection 4.2.2) use a hash table for all edges where the rational edge vertex coordinates are the keys. Edge intersections are stored for each edge. Note however, that implementing above method destroys the simplicity of the BSP partition, and for our polyhedral subdivision we can expect many edges (e.g., edges shared with tree polygons) to lie exactly on the 0.5 iso-surface. Additionally we would have had implementation problems[1].

6.6 Model Extensions

The object model described in chapter 2 is a quasi-convolutionally smoothed CSG object with polyhedral primitives. The CSG object is smoothed with a spherical filter of constant radius. Though this object model is sufficient to represent many natural scenes a more general approach is desirable.

A useful feature of a CSG modeler is support for clipping planes. With the help of clipping planes it is easy to produce a smoothed object with some sharp edges. An example is given in figure D.5. Here we have clipped a rounded cube in its middle and subtracted the result from an unrounded cube. The resulting shape is shown in detail in the top enlargement of the figure.

Another desirable feature is to define different rounding radii for the edges of an object. Though this effect is not possible for quasi-convolutionally smoothed objects, a similar effect is obtained by defining different smoothing radii for the half-spaces which form a polyhedral primitive. This technique was used to produce

[1]The current version of the Clean language provides neither an array data type nor pointers.

the cylindrical metal pins with smoothly flattened ends in figures D.2 (see also the enlargements).

6.6.1 Clipping Planes

Support for clipping planes is a useful feature of a CSG modeler. They provide the possibility to cut off (possibly rounded) edges and corners of an object or to cut an object open to provide an inside view. Therefore clipping planes simplify and enrich modeling capabilities.

For our object model clipping planes can be straightforwardly applied as a post-processing step by simply clipping all polygons of the polygonized object. An example for this approach is seen in figure 6.8 (b).

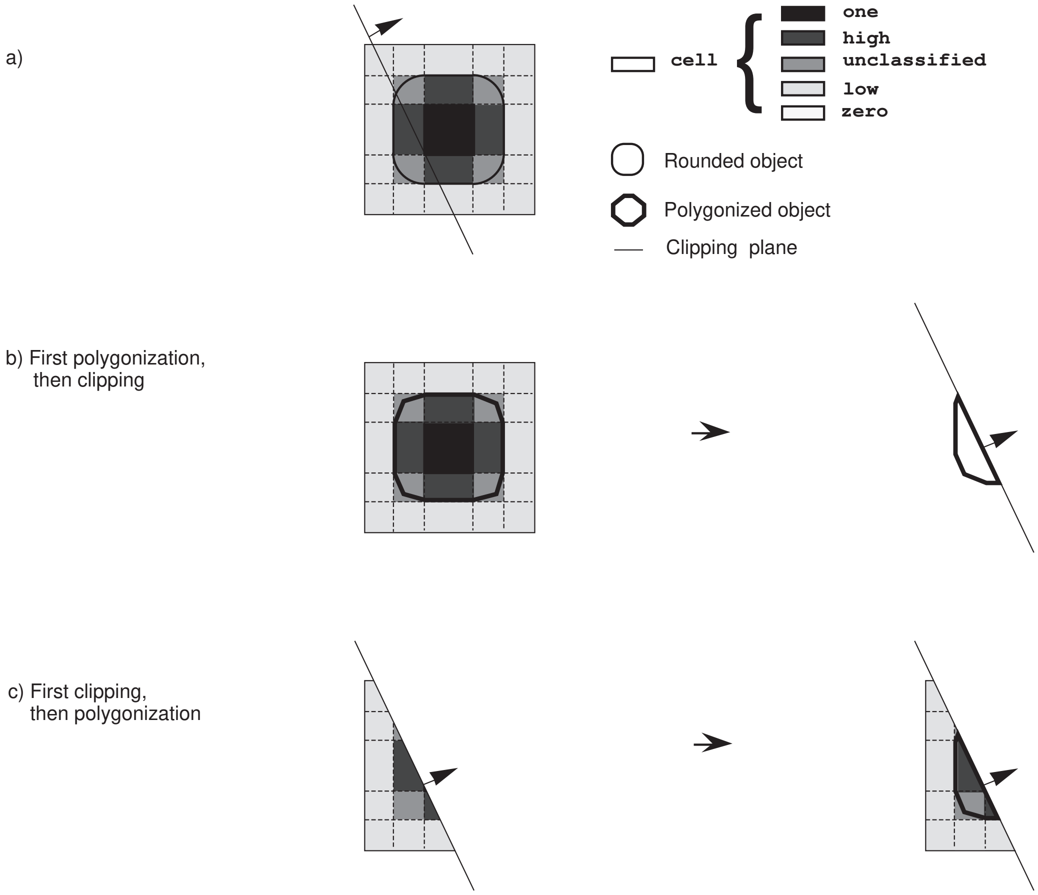

Figure 6.8. *A rounded object with a clipping plane (a). The straight forward method is to clip the polygonized object (b). A better solution is to clip the density field of the quasi-convolutionally smoothed object before polygonization (c).*

Clipping the Density Field

Though clipping the polygonized object is a straightforward approach it has several disadvantages. Firstly note that many smoothed edges and corners are polygonized though they are clipped off in the post-processing step. Secondly note that the clipping plane forms a face where it intersects the object. This face must be generated in an additional post-processing step. An approach coherent with Triage Polygonization is more desirable.

The solution is to clip the quasi-convolutionally smoothed object before polygonizing it. This is achieved by clipping the corresponding DCSG object. The clipped DCSG object is then transformed into a DBSP tree, which partitions the density field defining the clipped quasi-convolutionally smoothed object. The region outside the clipping plane has the density class zero. The subspace polygonization is only required for the remaining unclassified cells.

An example for this approach is shown in figure 6.8 (c). Note that clipping the object before polygonizing it, halves the number of unclassified cells. The polygonization of a clipped object is different from the original polygonization if an unclassified cell is clipped. In this case, since subspace polygonization and edge refinement is independent of the size of an unclassified cell, a more accurate polygonization results.

Implementation

To implement the clipping algorithm observe that clipping the DCSG object produces a discontinuity of the corresponding density field along the clipping plane. Since the subspace polygonization is applied only to cells inside the clipping plane, the discontinuity affects only the computation of tree polygons and not the subspace polygonization.

Recall that a tree polygon is defined as a polygon which separates a cell outside the 0.5 iso-surface from a cell inside the 0.5 iso-surface. In a clipped DBSP tree a high or one cell can be adjacent to a zero or low cell[2]. To take this into account, the function `SingleSideTreePolygons` defined in figure 5.10 is changed by calling the function `InsertInDCells` first with a low and zero density class in its first argument and then with a high and one density class in its first argument (compare figure 5.11). The function `InsertInDCells` must be extended by allowing a list of density classes in its first argument. The second and third argument do not change, i.e., they remain a DBSP tree and a list of faces. The specification of the extended function `InsertInDCells` is then that all faces are inserted in the DBSP tree and the face fragments reaching a cell having any of the given density classes are returned.

Note that part of the 0.5 iso-surface might lie on a clipping plane without being a tree polygon. This is the case if clipping an unclassified cell as is shown in figure 6.8 (c). The polygons on the clipping plane must be produced during the

[2] A one cell is never adjacent to a low cell, but allowing this case makes the implementation easier.

subspace polygonization. The solution is to extend the function `TraverseTree` (see figure 5.19), which traverses a DBSP tree and performs a subspace polygonization for all unclassified cells. The new version of `TraverseTree` is shown in figure 6.9.

```
TraverseTree ::  DBSPTree Plane Polyhedron -> [Polygon]
TraverseTree (DBSPNode plane inDBSPTree outDBSPTree) clippingPlane boundingBox
   = inSubspacePolygons ++ outSubspacePolygons
where
   (inBox,outBox)          = SplitPolyhedron plane boundingBox
   inSubspacePolygons      = TraverseTree inDBSPTree clippingPlane inBox
   outSubspacePolygons     = TraverseTree outDBSPTree clippingPlane outBox
TraverseTree dbspLeaf=:(DBSPLeaf _ densityClass) clippingPlane boundingBox
   | densityClass != Unclassified = []
   = polygonsInClippedCell ++ polygonsOnClippingFace
where
   clippingFace            = Intersection boundingBox clippingPlane
   polygonsInClippedCell   = SubspacePolygonsOfCell dbspLeaf
   polygonsOnClippingFace  = if (IsVoid clippingFace) [] (InsertInCells IN
                                (Faces2BSPTree polygonsInClippedCell) clippingFace)
```

Figure 6.9. *A plane clipping an unclassified cell might contain parts of the object surface. The subspace polygonization is extended to find the corresponding polygons.*

The function `TraverseTree` is extended by computing for every intersection of the clipping plane with an unclassified cell the polygons on the clipping plane inside the cell. Since we assumed that the DBSP tree is built from the clipped DCSG object, the clipping plane can only lie on the boundary of an unclassified cell. In that case the intersection of the clipping plane with the unclassified cell in non-void and is given by a boundary face (`clippingFace`) of the cell. The subspace polygons on the clipping plane are obtained by clipping the face `clippingFace` with all subspace polygons of the cell. This is most easily achieved by forming a BSP tree from the subspace polygons[3] and inserting the face into the tree. Only fragments of the clipping face that lie in IN cells are retained. The insertion is done with the function `InsertInCells` from figure 3.13.

6.6.2 Rounding Radius of a Half-space

As mentioned in the introduction to this section it is not possible to define different rounding radii for the edges of a quasi-convolutionally smoothed object. The reason

[3] If the list of subspace polygons is empty the BSP tree is either an IN cell or an OUT cell depending on the density class of the center of the cell.

lies in the definition of a quasi-convolutionally smoothed object which is a CSG object with convex polyhedra as primitives. The polyhedral primitives are modeled as intersection of half-spaces. The smallest primitive in the description of a quasi-convolutionally smoothed object is hence a half-space. Edges are not explicitly described.

However, interesting effects are obtained by smoothing the half-spaces of a polyhedral primitive with different rounding radii. The enlargements in figures **??** and **??** show as an example two metal pins of a hole punch. A metal pin is modeled as a cuboid with a diameter of one unit. Smoothing it with a spherical filter of radius 0.5 produces a cylinder. The flat right end of a metal pin is obtained by using a spherical filter of radius 0.1 for the corresponding half-space. The metal pin is clipped on the left end to fit it smoothly to the hinge of the hole punch. Figure 6.10 gives the scene description of a metal pin.

```
metalPin = Clipped
   (Rounded 0.5 (Intersection
      [Rounded 0.1 leftPlane, rightPlane,topPlane,
       bottomPlane, frontPlane, backPlane]))
   (Translate (-0.5) rightPlane)
```

Figure 6.10. *Scene description of a metal pin.*

To allow different rounding radii for the half-spaces of an object the object model is extended by allowing rounding attributes at every position of a CSG object. The interpretation is that a rounding radius defined nearer to a primitive object overwrites a previously defined rounding radius. Hence every half-space of a rounded object is associated with the rounding attribute defined nearest to it in the tree structure of a CSG object.

The only change necessary in the implementation of Triage Polygonization is to define for every half-space of a primitive object the associated rounding attribute and to use it for all computations involving a half-space (e.g., computation of a density value, polyhedral subdivision). The algorithm itself is not changed. Note that we introduced the idea of different rounding radii for two half-spaces already in figure 5.2.

6.7 Numerical Stability

The implementation of Triage Polygonization on a machine with final precision arithmetic leads to the occurrence of numerical errors. Though numerical errors in general can not be prevented it is often possible to limit the consequences of numerical errors. That means, a small error in the computation should lead only to a small error in the result.

The critical parts in Triage Polygonization are the using of logical alternatives. We identify two cases: The first case occurs when classifying the vertices of a face as outside, inside, or on the 0.5 iso-surface. The second case occurs during the construction of a DBSP tree when classifying an object as inside, outside, or on a plane. The following subsections describe these problems in more detail and present our solutions.

Precomputation of Polygon Edges

Initial results suggested that the error prone part of Triage Polygonization is the classification of a point in a density field into inside, outside, or on the iso-surface. The "on" case is needed to compute the face intersection with a simple unclassified face of the polyhedral subdivision (compare section 5.4 on page 91). The definition of the polyhedral subdivision implies that many vertices of unclassified cells lie exactly on the 0.5 iso-surface (also compare figure 5.2). The fact that face vertices are not shared only increases this problem.

The solution is to classify vertices against a displaced iso-surface with a threshold value of $0.5 + \epsilon$. We chose $\epsilon = 0.001$ to polygonize the example scenes given in section 2.5 and never detected a vertex lying on the $0.5 + \epsilon$ iso-surface. A continuity argument yields that if an edge intersects the $0.5 + \epsilon$ iso-surface the 0.5 iso-surface intersection is near. The exact 0.5 iso-surface intersection is computed by a root search.

For simple unclassified faces the vertex densities are either all greater than or equal to 0.5 (high cell on opposite side) or all smaller than or equal to 0.5 (low cell on opposite side). For the latter case the iso-surface does not intersect the face (since a low cell lies completely outside the $0.5 + \epsilon$ iso-surface). In the first case all vertices classified as low are considered as intersection points with the 0.5 iso-surface. We show that their density values differ less than ϵ from 0.5.

> **Proof:** A simple unclassified face bounds a high cell. Its vertices hence have density values greater or equal to 0.5. On the other hand the low vertices lie outside the $0.5 + \epsilon$ iso-surface and hence have a density value smaller then $0.5 + \epsilon$. $\square$

The polygon edges on a simple unclassified face are produced by connecting the intersection points with the same method as for a fully unclassified face (see page 92).

A potential pitfall in using the $0.5 + \epsilon$ iso-surface is that the displaced iso-surface can also intersect high cells. This is, for example, the case for a concave three plane corner as shown in the bottom enlargements of figure D.5 (a) and (b). Since the subspace polygonization is only performed for unclassified cells, regions of the iso-surface inside a high cell are not polygonized. This can result in holes in the polygonization. However, the goal of Triage Polygonization is to approximate the 0.5 iso-surface, which lies completely outside the high regions. Hence the missing polygons are exactly the parts of the simple unclassified face, which lie outside the

$0.5 + \epsilon$ iso-surface[4].

The missing polygons are defined by the intersection points of the face edges with the 0.5 iso-surface, and the low vertices of the face. The orientation of such a polygon is given by recognizing that the high cell must lie on its inside.

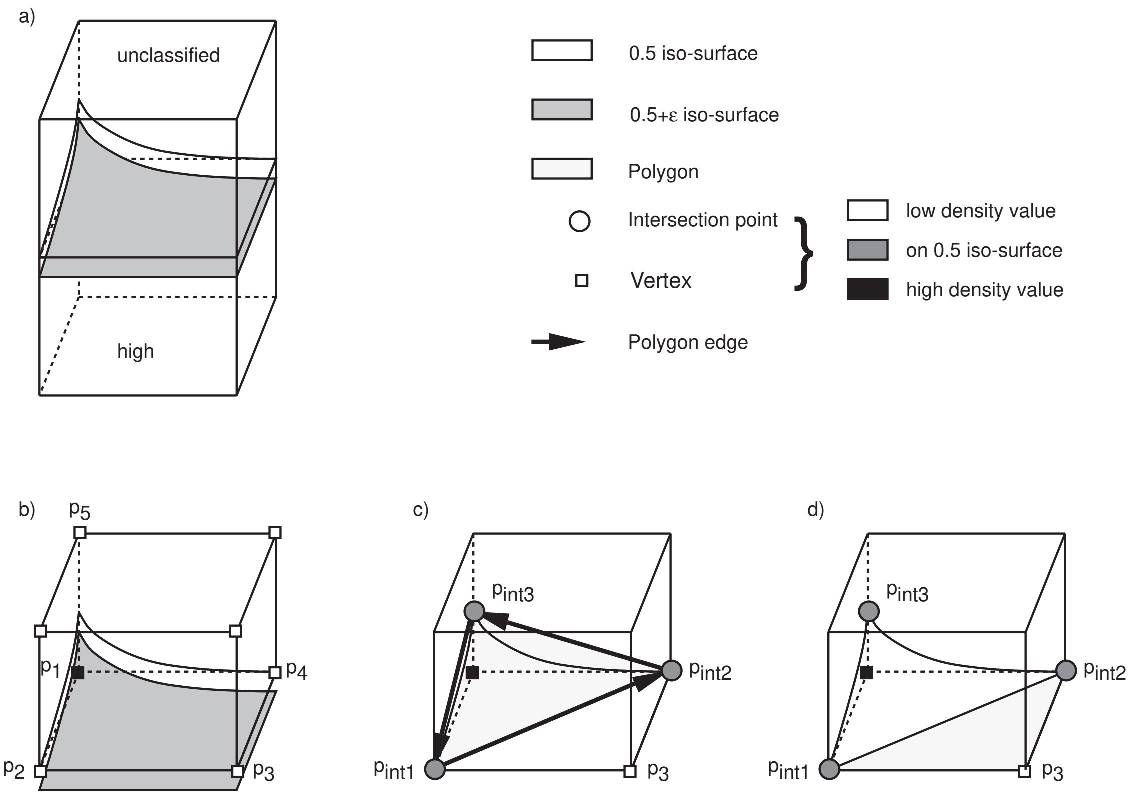

Figure 6.11. *Using the* $0.5+\epsilon$ *iso-surface for vertex classification (b) might produce holes during the subspace polygonization (c). The missing polygons are produced in an additional step as part of a simple unclassified face separating a high cell and an unclassified cell (d).*

Example 6.3 Figure 6.11 (a) shows an unclassified cell and a high cell separated by a simple unclassified face. The 0.5 iso-surface is depicted transparent, the $0.5 + \epsilon$ iso-surface shaded. The subspace polygonization is restricted to the unclassified cell shown in pictures (b) - (d). Picture (b) shows the classes of the cell's vertices. The result of our subspace polygonization is shown in (c). The produced polygon represents only part of the 0.5 iso-surface inside the cell. In picture (d) we apply the method introduced above and define an additional polygon from the intersection points p_{int1} and p_{int2} and the low vertex p_3. Both polygons together provide the desired approximation to the 0.5 iso-surface.

[4] In rare cases the exact 0.5 iso-surface might actually not intersect the simple unclassified face. However, the error is smaller than ϵ and our solution guarantees continuity.

6.7.1 DBSP Trees

The numerical problems occurring during DBSP tree construction are identical to the problems occurring for a simple BSP tree. We mention two problems. The first problem occurs if classifying an object as inside, outside, or on a plane. Here small errors in the definition of the object might lead to different classifications and hence completely different results. We use ϵ-intervals to discriminate whether a real value x is smaller, bigger, or equal to another value y, e.g., $x = y$ if and only if $|x - y| \leq \epsilon$. Other boolean expressions are computed similarly.

A related problem arises if splitting an object with a partitioning plane. Numerical errors or bad input data can lead to extremely small bits as result of a splitting operation, which makes subsequent BSP operations unstable. We avoid this problem by ignoring results of a splitting operation with a size below a given threshold.

A more detailed discussion of the numerical robustness of BSP tree algorithms is given by Naylor et al. [NAT90].

6.8 Polygonizing a Scene

A scene is defined as a CSG object with quasi-convolutionally smoothed and unsmoothed polyhedral objects as primitives. Additionally the definition of clipping planes is allowed. The scene is polygonized by first pushing the clipping planes down the tree structure of the CSG object until they either reach a quasi-convolutionally smoothed object or a convex polyhedral primitive of an unsmoothed object. A clipping plane is pushed down the tree structure of a CSG object by recalling that a clipping operation is modeled as an intersection with a half-space H and by using the set equivalence

$$(A \oplus^\star B) \cap^\star H = (A \oplus^\star H) \cap^\star (B \oplus^\star H), \qquad (\oplus^\star \in \{\cup^\star, \cap^\star, \setminus^\star\})$$

where A and B represent CSG objects. The clipping of a quasi-convolutionally smoothed object is part of Triage Polygonization (subsection 6.6.1). A convex polyhedral primitive is clipped by adding the half-space defined by the clipping plane to the list of half-spaces whose intersection defines the polyhedron.

The scene is then polygonized by polygonizing its quasi-convolutionally smoothed and unsmoothed polyhedral primitives. A quasi-convolutionally smoothed polyhedron is polygonized with Triage Polygonization. A polygonization for an unsmoothed polyhedron is given by its b-rep (see chapter 3). Set operations between unsmoothed and smoothed objects are performed by building BSP trees from the polygonized objects and merging them as described in section 3.4. Note that the b-rep of an unsmoothed object is already represented as a BSP tree. A BSP tree for a quasi-convolutionally smoothed object can be obtained either from the DBSP tree for its polyhedral subdivision or by inserting the polygons obtained with Triage Polygonization into an initially empty BSP tree.

To gain efficiency, only disjoint unions are allowed in the scene description. Then two objects forming a union are polygonized independently and no merging operation for the polygonized objects is necessary.

Gouraud Shading

A polygonized scene is naturally everything but smooth. To obtain a pleasant image of a polygonized scene we render it using Gouraud shading. To do this the vertex normals of each polygon must be known.

The vertex normals of the polygons of unsmoothed objects and of tree polygons are given by the surface normal of the corresponding polygon (recall that tree polygons describe planar areas of the object surface). The vertex normal $\vec{n}_{p_i}$ of a vertex p_i of a subspace polygon is given by the gradient of the density field in the vertex

$$\vec{n}_{p_i} = \frac{\nabla \rho(p_i)}{\|\nabla \rho(p_i)\|}$$

Again the local density field of the corresponding cell can be used to compute the gradient. Section 6.4 describes the computation of the gradient of a density field.

CHAPTER 7

Results

7.1 Introduction

In this chapter we present a number of findings obtained from analyzing Triage Polygonization. We start this chapter with a look at a few color images showing the polygonization process and the achieved results.

The next two sections give a performance analysis of Triage Polygonization. We perform a complexity analysis and present statistical results obtained for the example scenes from section 2.5. The statistical results are discussed and compared with the more theoretical results from the complexity analysis.

The chapter concludes with a comparison of Triage Polygonization and the general polygonization methods presented in chapter 4. We implemented the Marching Cubes algorithm and the obtained statistics are compared with the corresponding results achieved with Triage Polygonization. It will be seen that for the complex scenes Triage Polygonization is considerably faster and produces far fewer polygons than the Marching Cubes algorithm.

7.2 Images of Polygonized Scenes

This section explains the color images given in appendix D. We first try to visualize the three steps of Triage Polygonization: polyhedral subdivision, extraction of tree polygons, and subspace polygonization. We then present images of the example scenes presented in section 2.5. All images were produced by polygonizing the example scenes with Triage Polygonization (or the Marching Cubes algorithm where noted) and rendering the resulting set of polygons with the program SPIN 1.0 using QUICKDRAW3D on a POWER MACINTOSH 9500/120.

Visualization of the Polygonization Process

Figure D.1 shows the result of visualizing the three steps of Triage Polygonization. The figure shows a simple scene constructed by quasi-convolutionally smoothing the set difference of a big and a small cube. The resulting object is shown in part (h) of the figure.

The first step of Triage Polygonization is a polyhedral subdivision of the density field into zero, low, high, one, and unclassified cells. Figure D.1 (a) – (c) give the low, high, and unclassified cells, respectively, of the resulting subdivision. Note the small size of the unclassified cells in part (c) of the figure as compared to the size of the object. Only for these cells must a subspace polygonization be performed.

The second step of Triage Polygonization is the extraction of tree polygons, which separate low from high cells. The low and high cells are shown in part (a) and (b) of the figure and the resulting tree polygons separating the cells are given in (d). Back facing polygons are illuminated only with an ambient light source and are hence shaded in reddish black.

The third and last step of Triage Polygonization is the subspace polygonization of the density field inside unclassified cells. The resulting subspace polygons are shown in (e). Note that for almost all unclassified cells in (c) subspace polygons are found, which suggests that our polyhedral subdivision correctly identifies regions of space containing a curved iso-surface.

The complete polygonization output by Triage Polygonization is given in (f) flat shaded, in (g) as a wire-frame representation and in (h) Gouraud shaded. For better understanding of the images in this and all following wire-frame representations the back-faces are removed.

Polygonization Quality

The figure D.2 shows a quasi-convolutionally smoothed and polygonized hole punch Gouraud shaded and in a wire-frame representation, respectively. The base of the hole punch is rounded with a smoothing radius considerably smaller than the base itself. As a result Triage Polygonization extracts most of the object's surface as large rectangles. A smoothed edge and corner is represented with two long rectangles and 6 triangles, respectively. The Gouraud shaded picture in figure D.2 (b) shows that the produced polygons are sufficient to achieve the visual impression of a smoothed surface.

The enlargements of figure D.2 depict the punch, part of the hinges, and some metal pins in detail. Note that the punch is clipped on the top and that Triage Polygonization correctly polygonizes the clipped surface. The clipping algorithm, introduced in subsection 6.6.1, needs only four quadrilaterals to represents the surface formed by clipping. Four quadrilaterals are produced because the clipping plane intersects four unclassified cells of the polyhedral subdivision of the density field defining the quasi-convolutionally smoothed punch. The clipping algorithm clips all four unclassified cells, for each of which the cells' surface resulting from clipping is given by one quadrilateral.

The two metal pins at the bottom right corner of the enlargement are constructed from half-spaces with different rounding radii (see subsection 6.6.2). This gives the impression of a cylinder with a smoothly flattened end. Observe that the cylindrical part of a smoothed metal pin is approximated with rectangles whereas the more complicated end of a pin is represented by triangles.

Figure D.3 shows a quasi-convolutionally smoothed and polygonized stapler in a wire-frame representation (a) and Gouraud shaded (b). Note that very thin objects such as the side plates of the hinge are polygonized without problems. The front left side of the stapler is polygonized by long triangles. Inspecting the corresponding DBSP tree reveals that in this area the local density field is given as the intersection of two smoothed half-spaces, but that the corresponding cells in the polyhedral subdivision are not cuboids. The polygons in this area form a convex tessellation of a topological polygon as introduced in subsection 5.7.1.

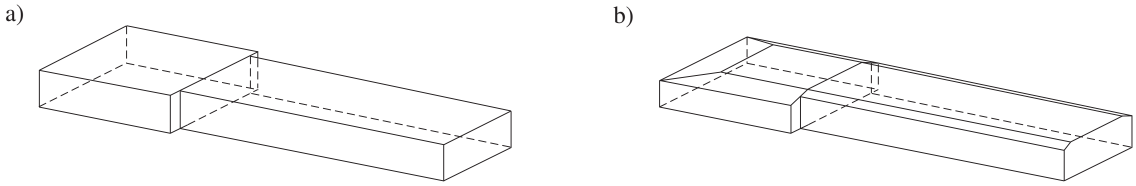

Figure 7.1. *The base of a stapler is a union of two cuboids (a) with the top edges clipped off at the front and the sides (b).*

The base part of the stapler is modeled as a union of two cuboids with the top edges clipped off at the front and the sides of the union. Figure 7.1 gives the resulting object before it is quasi-convolutionally smoothed. The magnified part of figure D.3 depicts the region where both objects forming the base of the stapler touch. It can be seen that our polygonization method also handles such a complex region without problems.

Figure D.4 shows 27 blended cubes modeled as a quasi-convolutionally smoothed union of 27 cubes. Observe that the resulting polygonization is sufficiently fine to give the impression of smoothly blended cubes. The polygonization is coarsest at the outer corners of the object, such as the top back corner, depicted in the enlargement of the figure. The reason for the coarse polygonization is that in the corners of the object the corresponding density field is given as the intersection of only three rounded half-spaces. The corresponding polyhedral subdivision has larger cells in these regions, which leads to a coarser polygonization.

Figure D.5 shows the polygonized "CSG Example" scene. The scene is modeled as a union of six objects each derived by applying various combinations of rounding operations and a set operation to a cube and a small cuboid. Two interesting cases are shown as enlargements. The top enlargements of both parts of the figure depicts a clipped quasi-convolutionally smoothed small cube subtracted from a bigger unsmoothed cube. The object is polygonized by computing the b-rep of the unrounded object and polygonizing the clipped quasi-convolutionally smoothed object with Triage Polygonization. The resulting polygonized objects are transformed into BSP trees (the b-rep is already given as a BSP tree) and merged with a set difference operation. The merging of BSP trees according to a set operation is achieved with the merged BSP tree algorithm from section 3.4.

The bottom enlargements of figure D.5 (a) and (b) give an example of a concave three plane corner. The corner results from a quasi-convolutionally smoothed union of a big cube and a small cuboid. It can be seen that the corner is nicely polygonized with only 26 triangles. This corner is a case where the displaced $0.5 + \epsilon$ iso-surface lies inside a high cell (see section 6.7 on page 131 for an explanation of the resulting problem and a solution for it).

Figure D.6 gives as a final example the "Variable Radius" scene, which shows an object constructed as a set difference of a cube and a small cuboid smoothed with varying rounding radii. The object in part (a) was polygonized with Triage Polygonization whereas for (b) the Marching Cubes algorithm was used. Note that our algorithm achieves a good polygonization for all objects, independent of the rounding radius. The polygonization for the objects with small rounding radius can be considered as optimal. For the objects rounded with a rather large smoothing radius some "bands" are visible where the object is polygonized more finely. This is due to small cells in the polyhedral subdivision of the density fields defining the quasi-convolutionally smoothed objects.

A comparison of the Triage Polygonization results with the Marching Cubes results is deferred to section 7.5.

7.3 Complexity Analysis

In this section we examine the time and space complexity of Triage Polygonization. The space complexity gives the number of polygons of the resulting polygonization.

For all results obtained it is important to differentiate between quasi-convolutionally smoothed scenes and quasi-convolutionally smoothed objects. A quasi-convolutionally smoothed scene is built from quasi-convolutionally smoothed objects and unsmoothed objects. In most cases the scene objects are disjoint, but they may also be obtained by applying set operations to more primitive scene objects. The object model and the example scenes are described in chapter 2.

In the following discussions we assume a scene is built predominantly from disjoint quasi-convolutionally smoothed objects of relatively low complexity. We concentrate on the polygonization of a quasi-convolutionally smoothed object and then remark on the polygonization of a quasi-convolutionally smoothed scene.

7.3.1 Triage Polygonization

Triage Polygonization builds a DBSP tree from a CSG object, extracts tree polygons, and performs a subspace polygonization on the unclassified cells of the polyhedral subdivision. We show in the next paragraphs that this process is analog to the lazy b-rep algorithm introduced in section 3.3 and therefore has a similar asymptotic time complexity as it.

Time Complexity

The polyhedral subdivision builds a DBSP tree from a CSG object. The operation is similar to the lazy BSP tree algorithm (see subsection 3.3.1) except that cells are differently labeled and local density fields are produced. The input size to the polyhedral subdivision is three times the input size to the lazy b-rep algorithm. The reason for this is that Triage Polygonization uses for the polyhedral subdivision, not only the half-space planes of a scene, but also the half-space planes translated in positive and negative normal direction by the rounding radius of the scene. Since the lazy b-rep algorithm has a polynomial time complexity and the input size of the polyhedral subdivision increases only by a constant factor the polyhedral subdivision has a similar time complexity to the lazy BSP tree algorithm.

The extraction of tree polygons corresponds to the boundary extraction for a BSP tree and therefore can be expected to have a similar time complexity.

The subspace polygonization is performed by precomputing polygon edges for all unclassified faces and then performing a constant time operation on all unclassified cells. The number of unclassified cells is bounded by the size of the polyhedral subdivision. The precomputation of polygon edges takes constant time for an unclassified face. The extraction of unclassified faces, in turn, is an operation equivalent to the extraction of boundary faces. Therefore we expect the subspace polygonization to have a similar time complexity to the lazy b-rep algorithm.

Above analysis suggests that overall Triage Polygonization has a similar time complexity to the lazy b-rep algorithm.

In section 3.6 we suggested that the lazy b-rep algorithm has a best case time complexity of $\Theta(n \log^2 n)$, an average case time complexity of $\Theta(n^{\log_2 \gamma + 1})$, and a lower bound for the worst case time complexity of $\Omega(n^3)$. The factor γ gives the average size increase of a face split on a partitioning plane.

Since the polyhedral subdivision uses for each half-space plane two additional orthogonally translated duplicates (the r-ihs-planes and r-ohs-planes), we expect a higher face fragmentation than for the lazy b-rep algorithm, but not a significant change of the complexity class.

We suggest therefore for Triage Polygonization the same best case and worst case time complexity as for the lazy b-rep algorithm, and in the average case a still sub-quadratic time complexity but with a worse asymptotic behavior than the $\Theta(n^{1.1})$ measured for the lazy b-rep algorithm.

Space Complexity

For a rounded object most of the polygons produced with Triage Polygonization can be expected to be subspace polygons. Since the cells of a polyhedral subdivision usually have a bounded number of faces, the number of subspace polygons produced for each unclassified cell can be assumed as constant. Then the size of the polygonization of a quasi-convolutionally smoothed object is proportional to the number of unclassified cells and the space complexity of Triage Polygonization is similar to its time complexity.

7.3.2 Polygonization of a Scene

We assume that a quasi-convolutionally smoothed scene is built predominantly from disjoint quasi-convolutionally smoothed objects of low complexity. Therefore the objects in the scene can be polygonized independently from each other and only in rare cases must set operations be performed between polygonized objects. Hence we expect that the time complexity of Triage Polygonization is nearly linear in the size of the scene. A similar argument suggests that the number of produced polygons increases nearly linearly with the size of a scene.

7.4 Statistical Results

7.4.1 Introduction

We have implemented Triage Polygonization and the Marching Cubes algorithm in CLEAN 1.0. The following statistical results were achieved on an APPLE MACINTOSH QUADRA 700 with 9 MByte heap space and 1 MByte stack space.

The results are given in separate graphs for the complex scenes and the n^3 blended cubes. The complex scenes represent examples of quasi-convolutionally smoothed scenes. We mentioned previously that complex scenes are built predominantly from disjoint quasi-convolutionally smoothed objects of relatively low complexity.

The n^3 blended cubes represent an example of a quasi-convolutionally smoothed object with increasing complexity. This object is atypical for a quasi-convolutionally smoothed object, since it is totally rounded, i.e., it has no planar surfaces. However, the object illustrates that Triage Polygonization also performs well on complicated fully smoothed objects.

Note that the execution time, where given, does not include garbage collection. For scenes where the polygonization method uses a large DBSP tree (in the case of Triage Polygonization) or a large grid (in the case of the Marching Cubes algorithm) the garbage collection time can exceed the execution time. However, we have recognized that the garbage collection time is strongly influenced by the evaluation order used by Clean and on the memory space available. Furthermore the garbage collection time may be regarded as an artifact of a functional language implementation. It would have no counterpart in an optimal imperative language implementation.

7.4.2 Triage Polygonization

The main performance measures for Triage Polygonization are its execution time and the number of output polygons. Figure 7.2 gives these values as obtained for the complex scenes and the n^3 blended cubes.

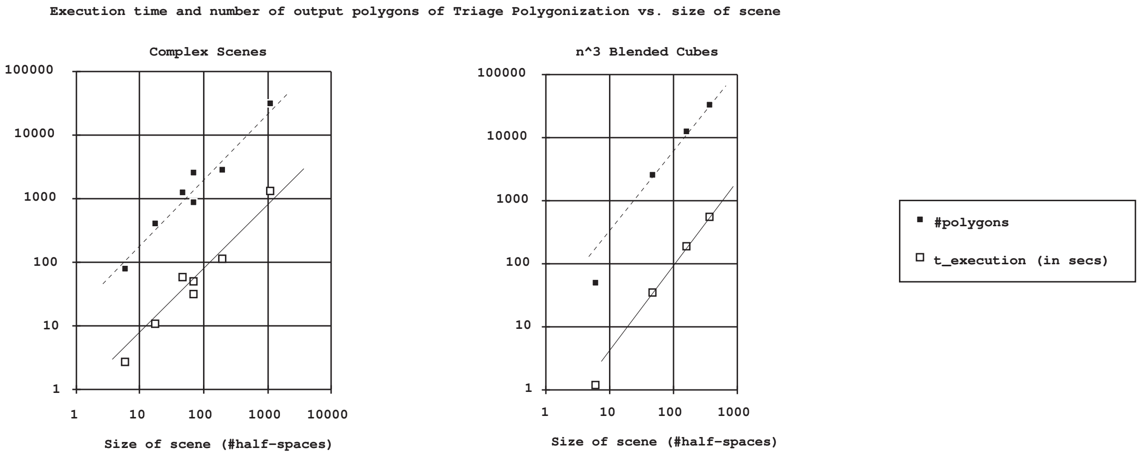

Figure 7.2. *The number of output polygons and the execution time of Triage Polygonization.*

The left graph, given with a double logarithmic scale, suggests that the number of output polygons and the execution time of Triage Polygonization is approximately linear in the size of a scene. Though in some of the scenes set operations are performed with polygonized objects, this does not seem to influence the asymptotic behavior. This indicates that Triage Polygonization produces a tessellation such that set operations involving polygonized objects are not more expensive than the polygonization itself.

For the n^3 blended cubes, which represent a quasi-convolutionally smoothed object, the number of output polygons and the execution time of Triage Polygonization is clearly increasing at a more than linear rate but less than quadratic in the size of the object. The graph suggests for Triage Polygonization a space complexity (i.e., number of output polygons) of about $\Theta(n^{1.2})$ and a time complexity of about $\Theta(n^{1.3})$.

Table 7.1 gives the total number of polygons for the polygonized example scenes. For some complex scenes the total number of polygons is larger than the sum of its tree and subspace polygons, given in the following subsections. We identify two reasons for this size increase: the "Cube In Cube" scene and the "CSG Example"

Scene	#polygons	Number of vertices				
		3	4	5	6	≥ 7
Cube	78	61.54	38.46	0	0	0
Cube in Cube	396	90.63	8.85	0	0	0.52
Stapler	1254	87.78	11.90	0.16	0.16	0
CSG Example	876	80.10	19.37	0.53	0	0
Variable Radius	2521	93.38	5.99	0.48	0	0.16
Hole Punch	2951	89.72	9.97	0.21	0.10	0
Many Staplers	30096	87.78	11.90	0.16	0.16	0
1 blended Cube	48	100.00	0	0	0	0
8 blended Cubes	2496	100.00	0	0	0	0
27 blended Cubes	11952	100.00	0	0	0	0
64 blended Cubes	33024	100.00	0	0	0	0

Table 7.1. *Percentage of polygons by number of vertices.*

scene also contain unrounded objects, which are polygonized with the lazy b-rep algorithm; the “Hole Punch” scene and again the “CSG Example” scene use the merging BSP tree algorithm to perform set operations between polygonized objects. The merging of BSP trees leads to fragmentation of polygons and hence usually increases the size of a polygonization.

The table shows that the majority of the output polygons are triangles. However, a significant number of polygons have more than three vertices. We show in the following subsections that most polygons with more then three vertices are tree polygons.

An interesting question is what proportion of the total execution time is spent on the three steps of Triage Polygonization. Figure 7.3 indicates that for complex scenes the polyhedral subdivision is the most complex task. The subspace polygonization still takes on average about a quarter of the computation time and the extraction of tree polygons about 5–10% of the time.

For the n^3 blended cubes the polyhedral subdivision and the subspace polygonization both seem to take almost half of the total execution time, independent of the size of the object. We omit here the results obtained for the case where n equals one because one blended cube represents a primitive object. The precomputation of the face intersections takes only a minor amount of the computation time, but its percentage seems to be fairly constant at about 5%.

The results for both complex scenes and the n^3 blended cubes indicate that the three steps of Triage Polygonization have similar asymptotic time complexities. This observation corresponds to results of the complexity analysis in the previous section.

In the following subsections we examine the three steps of Triage Polygonization in more detail.

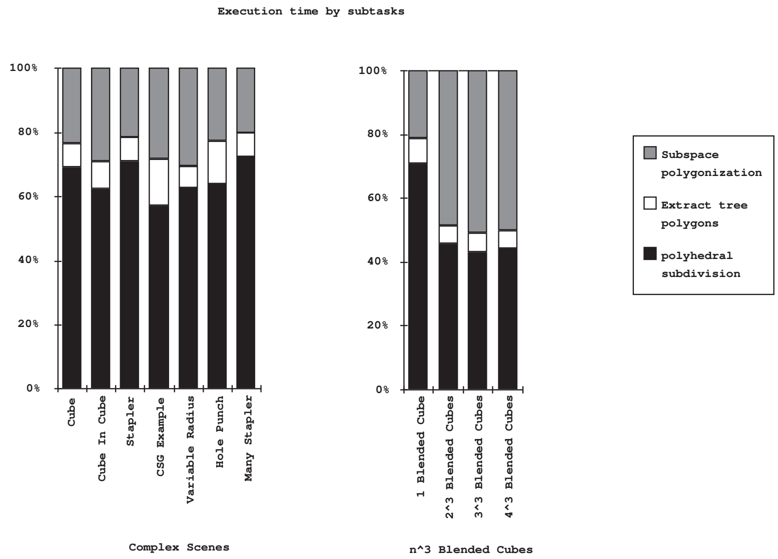

Figure 7.3. *Distribution of execution time by subtasks of Triage Polygonization.*

7.4.3 Polyhedral Subdivision

Size and Execution Time

The polyhedral subdivision partitions a density field in a BSP-like manner into zero, low, high, one, and unclassified cells. For each unclassified cell a reduced arithmetic tree is computed defining a local density field, which is identical to the original density field, inside the cell.

The size of the polyhedral subdivision of a quasi-convolutionally smoothed object is defined as the number of partitioning planes in its DBSP tree. The size of the polyhedral subdivision of a scene is given as the total size of the polyhedral subdivisions of all quasi-convolutionally smoothed objects in the scene. We define the size of a scene as the number of half-spaces defining primitive objects.

Figure 7.4 gives the size of the polyhedral subdivisions of the example scenes and the time necessary to compute them.

Complex scenes are constructed predominantly as disjoint unions of quasi-convolutionally smoothed objects of relatively low complexity. As suggested in the complexity analysis, the size of the polyhedral subdivision (i.e., the number of partitioning planes in the corresponding DBSP tree) for a complex scene seems to depend linearly on the size of the scene. The graph suggests that the subdivision consists of about 20 times more partitioning planes than the corresponding CSG object has primitive half-spaces.

More interesting is the size of the polyhedral subdivision for a quasi-convolutionally smoothed object. For the n^3 blended cubes the size of the polyhedral subdivision seems to increase more than linearly. Interpolating the values in figure 7.4 suggests

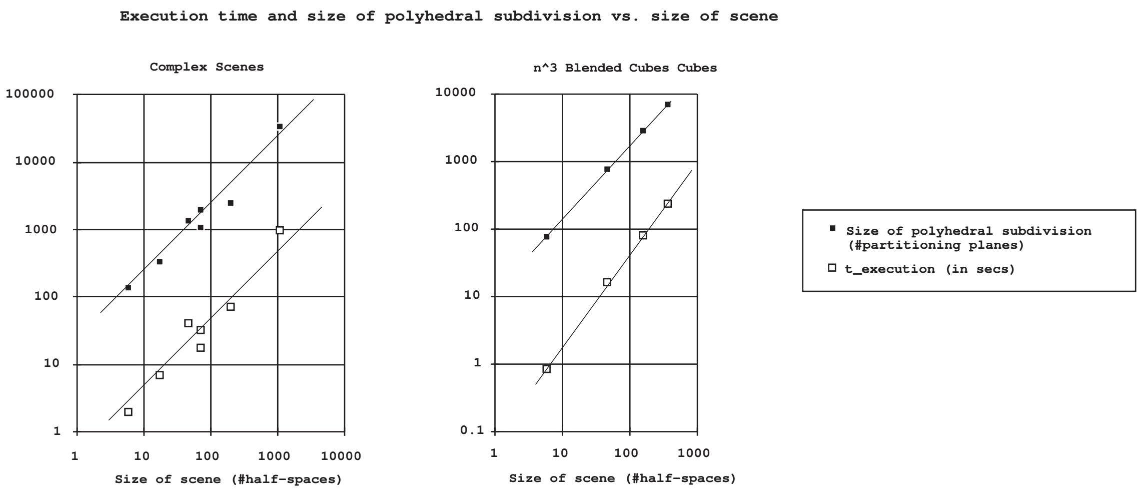

Figure 7.4. *Size and execution time of the polyhedral subdivision vs. size of the scene.*

a space complexity of the polyhedral subdivision of about $\Theta(n^{1.1})$. For the time complexity of the polyhedral subdivision the figure suggests a value of about $\Theta(n^{1.3})$.

Density Classes

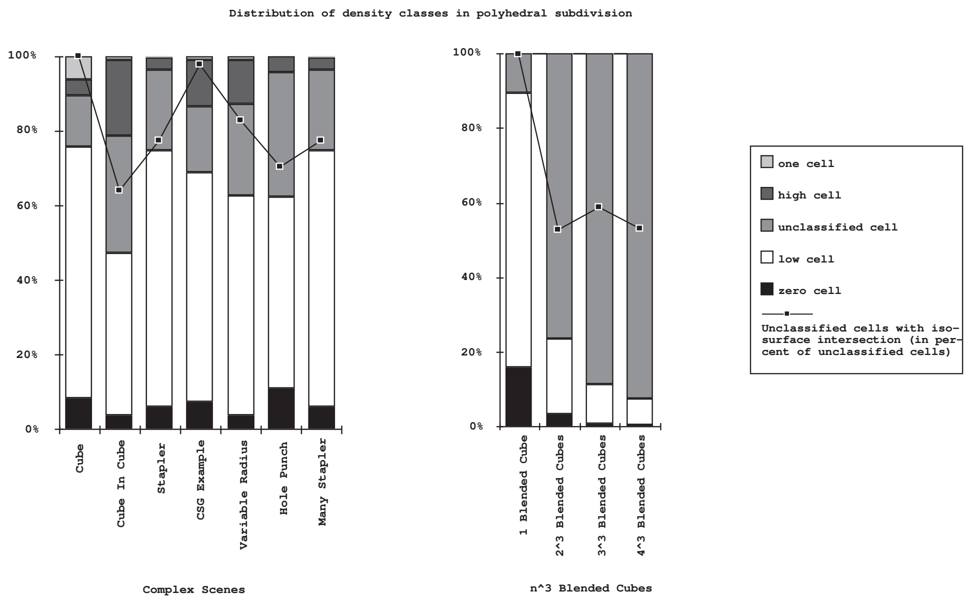

Figure 7.5. *Distribution of density classes in the polyhedral subdivision and percentage of unclassified cells with 0.5 iso-surface intersection.*

An important aspect for the efficiency of Triage Polygonization is the number of low, high, and unclassified cells in the polyhedral subdivision. Faces separating low and high cells are immediately extracted as part of the 0.5 iso-surface, whereas for unclassified cells a comparatively expensive subspace polygonization is performed.

Figure 7.5 shows that for complex scenes only about 25% of all cells are unclassified. The complexity of the polyhedral subdivision of the n^3 blended cubes increases with n and therefore the proportion of unclassified cells increases, too. Recall that the n^3 blended cubes are completely rounded (i.e., the rounding radius of the object is half the diameter of a cube), which is atypical for a quasi-convolutionally smoothed object. As a consequence the polyhedral subdivision for the n^3 blended cubes has no high or one cells.

The overlay graph in figure 7.5 gives the actual percentage of unclassified cells intersected by the 0.5 iso-surface. For most complex scenes, on average 75% of all unclassified objects are intersected by the 0.5 iso-surface. Note that for the n^3 blended cubes the subspace polygonization is still successful in at least half of the cases.

Summarizing, we demonstrated that for a quasi-convolutionally smoothed scene only a small percentage of the cells of a polyhedral subdivision is unclassified. The majority of the unclassified cells are indeed intersected by the 0.5 iso-surface. Even for an atypical quasi-convolutionally smoothed object, such as the n^3 blended cubes, more than half of its unclassified cells are intersected by the 0.5 iso-surface. These results suggest that the polyhedral subdivision is very effective.

Density Fields

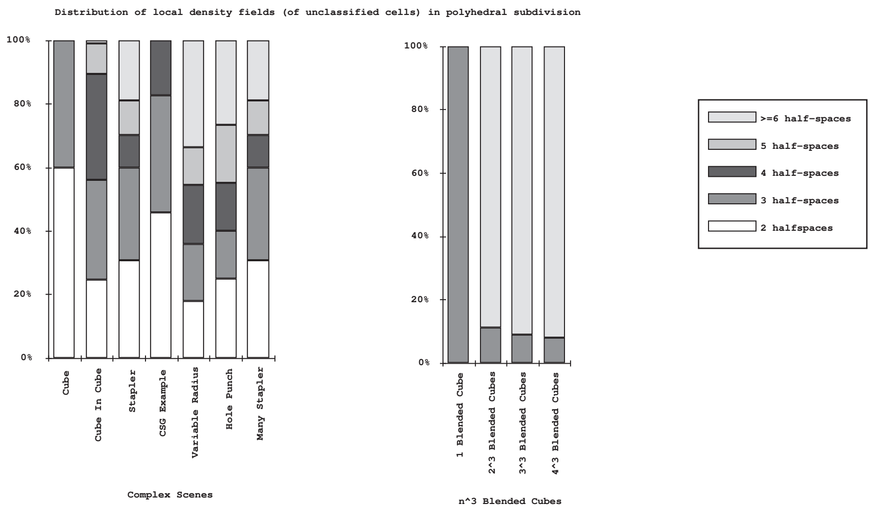

Figure 7.6. *Size distribution of the local density fields for unclassified cells.*

The polyhedral subdivision computes a local density field for each cell simultaneously with its density class. The complexity of a local density field is important for two reasons: firstly, for a local density field defined as the product of two half-spaces the 0.5 iso-surface is convex. This allows us to apply a specialized, more efficient polygonization method. Secondly, the density field is frequently evaluated during the subspace polygonization (for the root search). The evaluation time for the local density field is proportional to the size of the arithmetic tree defining it. Since the density computation for a smoothed half-space is the most complex operation in an arithmetic tree, we quantify the size of a local density field by the number of density half-spaces in the corresponding arithmetic tree.

Figure 7.6 shows the size distribution of the local density fields for all unclassified cells of a polyhedral subdivision. For the complex scenes hardly any cell has an arithmetic tree with a size greater than five. It can be said that a local arithmetic tree is of constant size. About 30% of all unclassified cells have a local density field that is the product of two density half-spaces. This means that in about 30% of all cases the 0.5 iso-surface inside an unclassified cell is polygonized with the algorithm for a convex tessellation (section 5.7). Additionally we found that the size of these cells is generally bigger than the of size of those cells with a more complex density object. For example, consider a simple smoothed cube. The unclassified cells containing the smoothed edges are bigger than the cells containing the smoothed corners. These results suggest that was worthwhile to find an optimal polygonization for cells with an intersection of two half-spaces.

The graph for the n^3 blended cubes suggests that the complexity of a local arithmetic tree increases with the size of a quasi-convolutionally smoothed object. This result is not surprisingly, since with the increasing size of a quasi-convolutionally smoothed object, a region of space is likely to be in the vicinity of more half-spaces defining the object.

Summarizing we have seen that the density field inside an unclassified cell is usually extremely simple, and that in fact we can expect to evaluate the corresponding arithmetic tree in constant time. About 30% of all unclassified cells contain a simple convex surface, which justifies the development of a more efficient subspace polygonization method for these cells.

7.4.4 Extraction of Tree Polygons

The second step of Triage Polygonization extracts tree polygons, which are faces separating low and high cells in a polyhedral subdivision. Table 7.2 shows the number and size distribution of the extracted tree polygons. The majority of the tree polygons are quadrilaterals, and when examined most of them prove to be rectangles. This should be expected since most polyhedral primitives of quasi-convolutionally smoothed objects are cuboids.

Though the tree polygons make up only 1–2% of the output polygons we found that they represent in average about 80% of the area of a polygonized surface.

A small proportion of the tree polygons has five or even six vertices This indicates that some cells in the polyhedral subdivision have a complicated non-regular geometry.

Scene	#tree polygons	Number of vertices				
		3	4	5	6	≥ 7
Cube	6	0	6	0	0	0
Cube in Cube	12	0	12	0	0	0
Stapler	36	2	30	2	2	0
CSG Example	46	0	46	0	0	0
Variable Radius	49	0	49	0	0	0
Hole Punch	50	0	45	4	1	0
Many Staplers	864	48	720	48	48	0
n^3 blended Cubes	0	0	0	0	0	0

Table 7.2. *Number of tree polygons by number of vertices.*

The only example scene containing tree polygons on a clipping plane is the "CSG Example" scene, which has five tree polygons on a clipping plane. These polygons result from clipping a rounded cube with a plane parallel to its top face, which forms a zero cell adjacent to four high cells and one one cell.

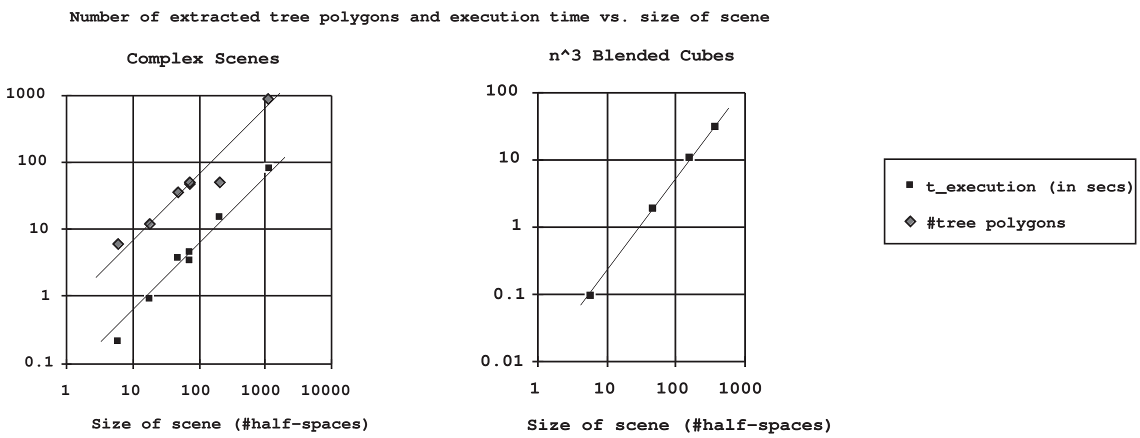

Figure 7.7. *The number of tree polygons and the execution time of the algorithm for the extraction of tree polygons vs. size of the scene.*

Figure 7.7 shows that the number of tree polygons and the execution time of the algorithm for the extraction of tree polygons is linear in the size of the complex scenes. For the n^3 blended cubes no tree polygons exist because the applied rounding radius is half the diameter of a primitive cube. The graph suggests a time complexity of about $\Theta(n^{1.3})$, which is the same time complexity as for the polyhedral subdivision.

7.4.5 Subspace Polygonization

The final step of Triage Polygonization is a subspace polygonization for all unclassified cells. The subspace polygonization first approximates all 0.5 iso-surface intersections with the faces of unclassified cells. The approximations form polygon edges for the subspace polygons. The polygon edges are connected to form topological polygons, which are then subdivided into planar polygons. In some cases the 0.5 iso-surface inside an unclassified cell is convex and is polygonized with a convex hull algorithm.

Precomputation of Face Intersections

In all cases it was possible to refine the precomputed polygon edges. This suggests that all polygon edges are good approximations to the corresponding 0.5 iso-surface intersection with the face.

Topological Polygons

Triage Polygonization connects the precomputed polygon edges to form topological polygons. Figure 7.8 shows the distribution of the resulting topological polygons by the number of vertices.

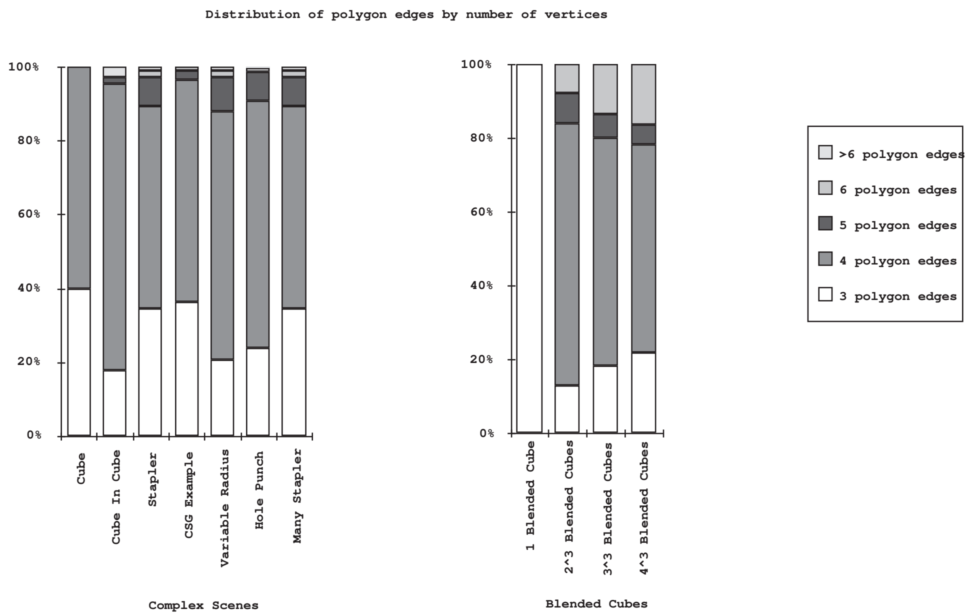

Figure 7.8. *Distribution of topological polygons by number of vertices.*

In the majority of cases a topological polygon has four vertices. The reason for this is that the polyhedral subdivision produces generally cuboidal cells which usually have four edge intersections with an arbitrary surface. In some cases topological polygons with considerably more vertices are formed. Examining these cases reveals that faces of a cell are fragmented during the precomputation of face intersections. If, for example, a cell shares a face with three different cells, as the cell on the left in figure 7.9, then the face is split into three faces and for each face a polygon edge is formed. The topological polygon for the cell has accordingly more vertices.

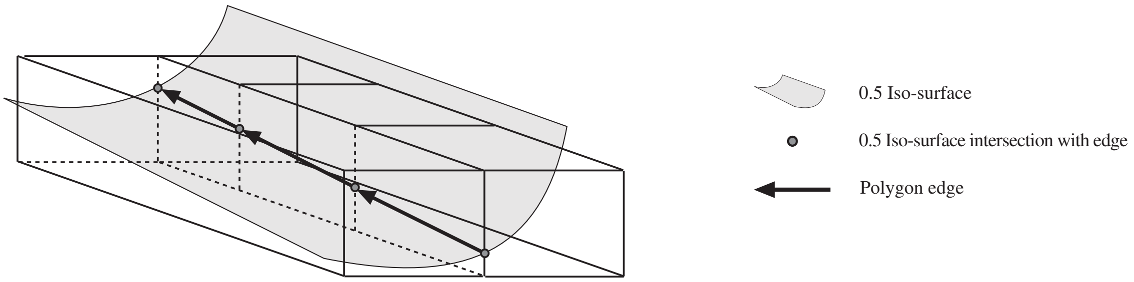

Figure 7.9. *A face shared with several cells is split during the precomputation of polygon edges. For each face fragment a polygon edge is computed.*

Closed topological polygons were formed for all subspace polygonizations and no discontinuities were detected. When tessellating a topological polygons in order to form planar polygons we never discovered two topological polygons in one cell. This indicates that the complicated procedure to avoid intersecting subspace polygons is unnecessary.

When refining a topological polygon, the centroid of the topological polygons could be moved to the iso-surface in all but two scenes: in the "Variable radius" scene, one out of 325 topological polygons could not be refined with a centroid on the 0.5 iso-surface. In the "Hole Punch" scene this worsened to 32 out of 355. All cases occurred for very small and narrow topological polygons.

Summarized, these results suggest that the subspace polygonization usually correctly approximates the 0.5 iso-surface intersection with a cell.

Subspace Polygons

Subspace polygons are obtained either by Tessellating a refined topological polygons or by applying the convex tessellation algorithm. Table 7.3 gives the size distribution of the subspace polygons for our example scenes by the number of vertices.

The majority of polygons are triangles and their percentage seems to increase with the complexity of a quasi-convolutionally smoothed object, as can be seen for the n^3 blended cubes. All subspace polygons with more than three vertices are produced with the algorithm for a convex tessellation of a topological polygon (subsection 5.7.1). Polygons with more than four edges are in general due to face fragmentation as described in figure 7.9.

Scene	#subspace polygons	Number of vertices				
		3	4	5	6	≥ 7
Cube	72	66.67	33.33	0	0	0
Cube in Cube	378	90.63	8.85	0	0	0.52
Stapler	1218	90.80	9.20	0	0	0
CSG Example	764	80.10	19.37	0.53	0	0
Variable Radius	2472	95.31	4.05	0.49	0	0.16
Hole Punch	2781	91.58	8.28	0.07	0.07	0
Many Staplers	29232	90.80	9.20	0	0	0
1 blended Cube	48	100.00	0	0	0	0
8 blended Cubes	2496	100.00	0	0	0	0
27 blended Cubes	11952	100.00	0	0	0	0
64 blended Cubes	33024	100.00	0	0	0	0

Table 7.3. *Percentage of subspace polygons by number of vertices.*

We also examined the number of subspace polygons lying on a clipping plane. The "CSG Example" and "Hole Punch" scenes have respectively 4 and 96 subspace polygons produced by clipping.

The number of subspace polygons in an unclassified cell is not affected by the size of a scene. We recognized, however, that with the increasing size of a quasi-convolutionally smoothed object, its polygonization seems to produce more subspace polygons in each cell. An example is given by the n^3 blended cubes for which figure 7.10 gives the average number of subspace polygons for a cell intersected by the 0.5 iso-surface. The cause for the increased number of subspace polygons is given by an the increased face fragmentation for larger quasi-convolutionally smoothed objects, which results in topological polygons with more vertices.

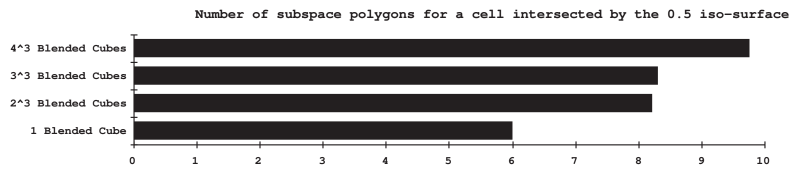

Figure 7.10. *Number of subspace polygons for a cell intersected by the 0.5 iso-surface.*

Figure 7.11 suggests that for complex scenes the number of subspace polygons and the execution time of the subspace polygonization increases linearly in the size of the scene.

For the n^3 blended cubes the execution time and the number of produced subspace polygons increases clearly more than linearly in the size of the object. We suggest a time complexity of $\Theta(n^{1.3})$ and a space complexity of $\Theta(n^{1.2})$. These are the same values we have suggested for the complete Triage Polygonization algorithm.

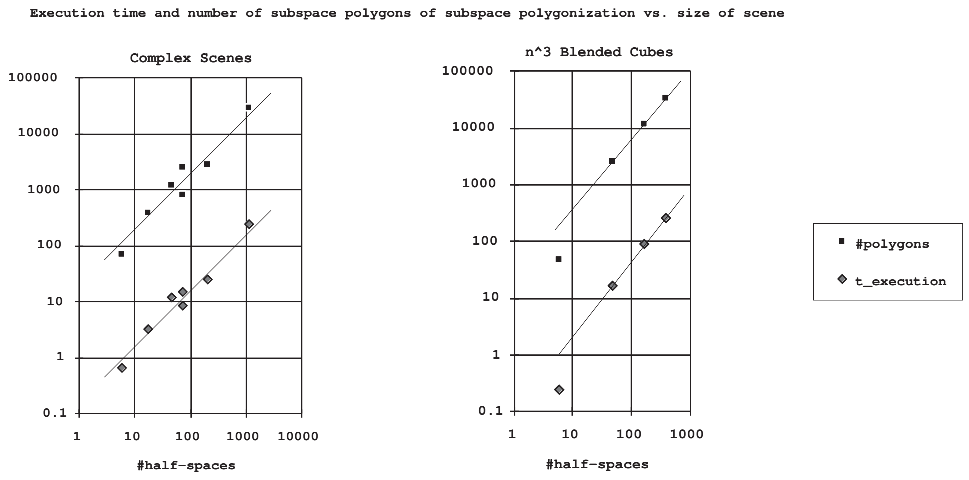

Figure 7.11. *Number of subspace polygons and execution time of subspace polygonization.*

7.4.6 Summary

The statistical results suggest that the time and space complexity of Triage Polygonization is linear in the size of a scene and sub-quadratic in the size of a quasi-convolutionally smoothed object. We found evidence that the three steps of Triage Polygonization have approximately the same asymptotic complexity, as suggested in the complexity analysis.

The results suggest that the polyhedral subdivision of Triage Polygonization is very effective. Most cells are correctly classified into inside or outside the 0.5 iso-surface. The subspace polygonization is only necessary for a small percentage of the cells of the subdivision, most of which are indeed intersected by the 0.5 iso-surface.

For the subspace polygonization we have indications that it approximates the 0.5 iso-surface inside a cell correctly. In all cases the topological polygons could be closed and no discontinuities were detected. Also polygon edges could always be refined, and in most cases the same was true for topological polygons. Subspace polygons are usually small triangles whereas tree polygons are usually big quadrilaterals.

7.5 Triage Polygonization vs. Common Polygonization Methods

In the previous sections we analyzed Triage Polygonization both from a theoretical and practical point of view. It remains to compare Triage Polygonization with already existing polygonization methods for implicit surfaces. We restrict ourselves here to the four polygonization algorithms introduced in chapter 4. One of them, the Marching Cubes algorithm, has been implemented by us. We polygonize the example scenes, introduced in section 2.5, with the Marching Cubes algorithm and compare the results with those obtained with Triage Polygonization.

7.5.1 Triage Polygonization vs. Marching Cubes

The Marching Cubes algorithm and Triage Polygonization both polygonize a density field by subdividing it and polygonizing the density field inside the resulting subspaces. The Marching Cubes algorithm uses a fixed size array of cubes. To gain a similar resolution as Triage Polygonization the grid size should not be larger than half the rounding radius of a quasi-convolutionally smoothed object.

Example

Figure D.6 shows the "Variable Radius" scene polygonized with Triage Polygonization (a) and the Marching Cubes algorithm (b). The left image of each part shows a wire-frame representation with removed back-faces and the right image shows the polygonization Gouraud shaded. The left object in the back of part (a) of the figure could not be polygonized with the Marching Cubes algorithm because of memory problems[1].

It can be clearly seen that for the three objects in the back of the image Triage Polygonization produces considerably fewer polygons than the Marching Cubes algorithm. For the three front objects Triage Polygonization produces about the same number of polygons. Despite formed from fewer polygons the Gouraud shaded picture obtained with Triage Polygonization has at least the same quality as that yielded with the Marching Cubes algorithm. Note how Triage Polygonization represents the planar area of the object's surface with large polygons and similarly rounded edges with long narrow rectangles. The Marching Cubes algorithm, on the other hand, always produces about equal size polygons independent of the curvature of the surface.

Statistical Results

We are interested in the execution time and the number of output polygons of Triage Polygonization and the Marching Cubes algorithm. Figure 7.12 compares the number of polygons of the polygonizations obtained with Triage Polygonization and the Marching Cubes algorithm.

[1]For efficiency we computed the cubic array subdividing a density field slice by slice, as recommended in the original paper of Lorenson and Cline [LC87]. Because we wanted to achieve a similar resolution as for Triage Polygonization we chose the grid size to be half the rounding radius of the represented object. The resulting grid did not fit into the memory of our machine.

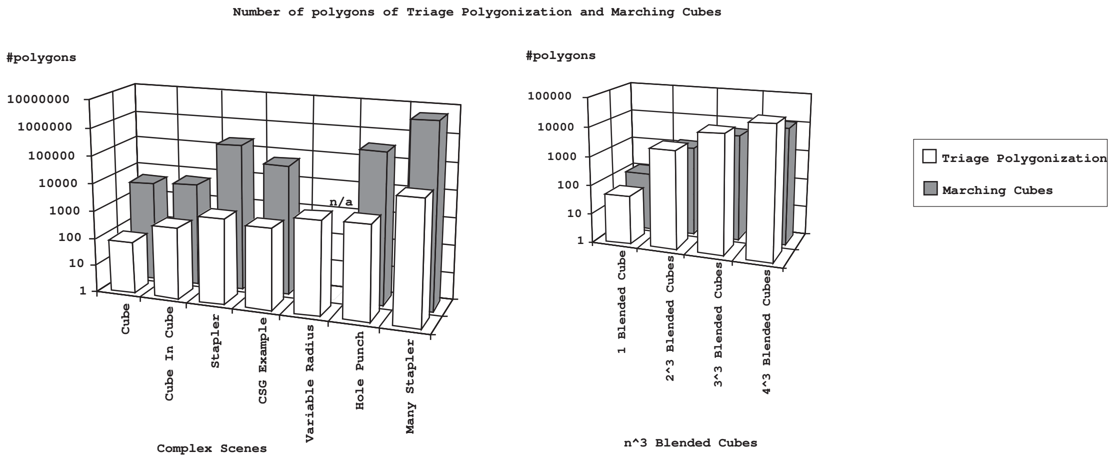

Figure 7.12. *Number of polygons obtained with Triage Polygonization and the Marching Cubes algorithm.*

Note that we use a logarithmic scale, i.e., one unit means a difference of factor ten. It can be clearly seen that the polygonization obtained with Triage Polygonization has considerably fewer polygons for the complex scenes. However, the polygonization with the Marching Cubes algorithm produces slightly fewer polygons than Triage Polygonization for the n^3 blended cubes. This is not surprising since the n^3 blended cubes are totally rounded objects, without any planar surfaces. Therefore they are atypical for quasi-convolutionally smoothed objects and are not subject to the special properties of Triage Polygonization.

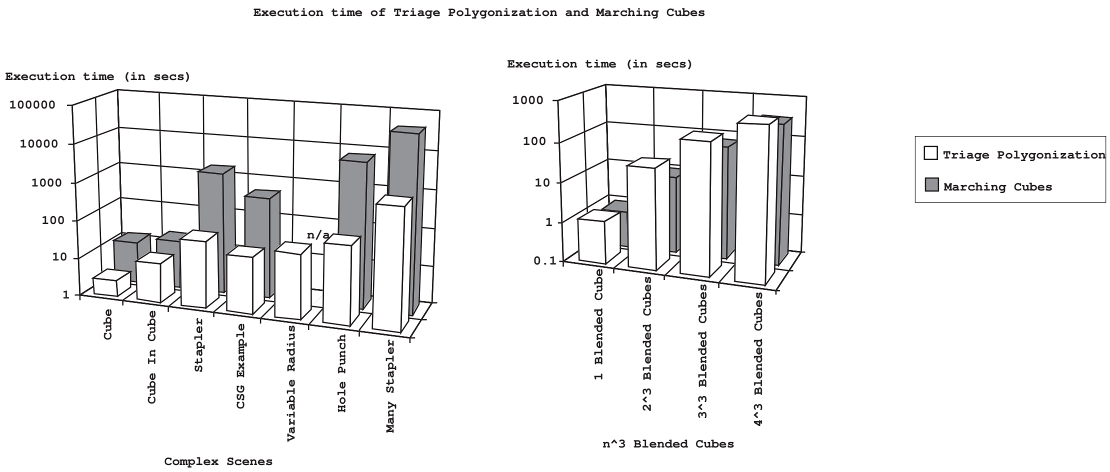

Figure 7.13. *Execution Time of Triage Polygonization and the Marching Cubes algorithm.*

Figure 7.13 gives the execution time of Triage Polygonization and the Marching

Cubes algorithm. It can be seen that Triage Polygonization is considerably faster for the complex scenes and only slightly slower for the n^3 blended cubes.

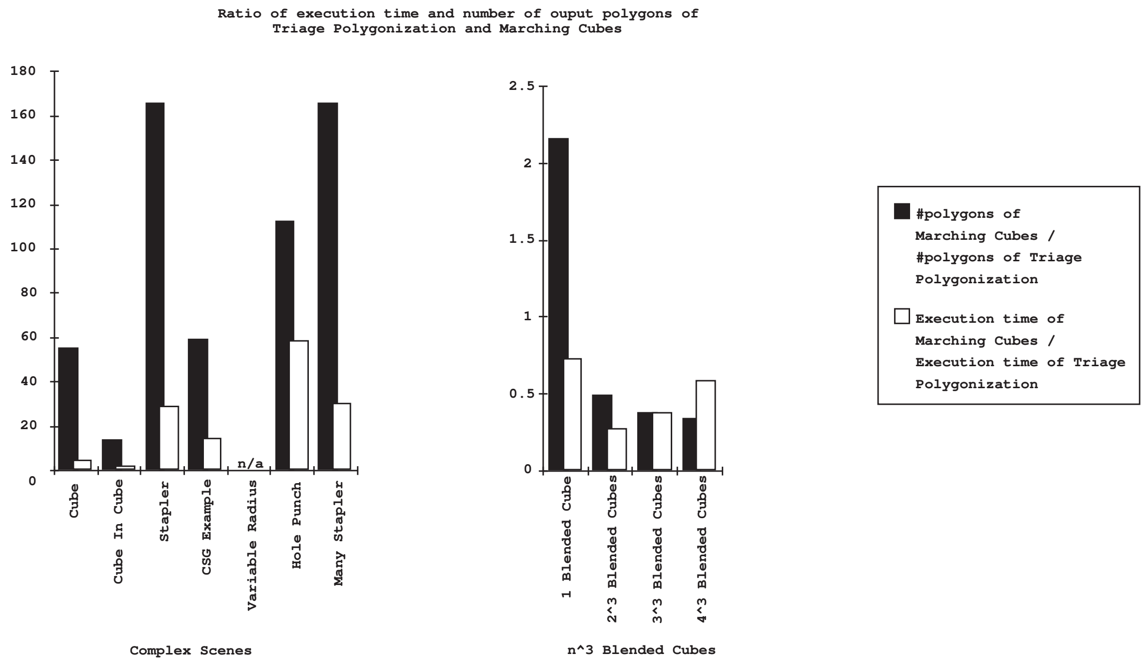

Figure 7.14. *The ratio of the number of output polygons and the ratio of the execution time of Triage Polygonization and the Marching Cubes algorithm.*

The differences between the algorithms become clear in figure 7.14. In average Triage Polygonization is about 20–30 times faster than the Marching Cubes algorithm and outputs only a fraction ($\approx$ 1–2%) of its number of polygons. Note the large speed advantage of Triage Polygonization for the polygonization of the "Hole Punch" scene. In this scene the fact that Triage Polygonization produces fewer polygons becomes especially important since the object model for the scene defines several intersection and set difference operations on quasi-convolutionally smoothed objects. Since these set operations are performed by merging BSP trees their cost depends on the size of the polygonization of the corresponding quasi-convolutionally smoothed object. The Marching Cubes algorithm produces more polygons for a polygonized object, which leads to a more expensive set operation involving the object.

For the n^3 blended cubes Triage Polygonization is slightly slower than the Marching Cubes algorithm, but the difference seems to decrease with increasing size of the quasi-convolutionally smoothed object. We offer as an explanation that the Marching Cubes algorithm always has to evaluate the complete density field if computing the density value for a point. In contrast, Triage Polygonization constructs local density fields during subdivision. For the subspace polygonization the density values are computed with the local density field of the corresponding cell. The number of output polygons is higher for Triage Polygonization because with increasing object size the polyhedral subdivision becomes more fragmented. However, as a result the polygonization approximates the quasi-convolutionally smoothed object better.

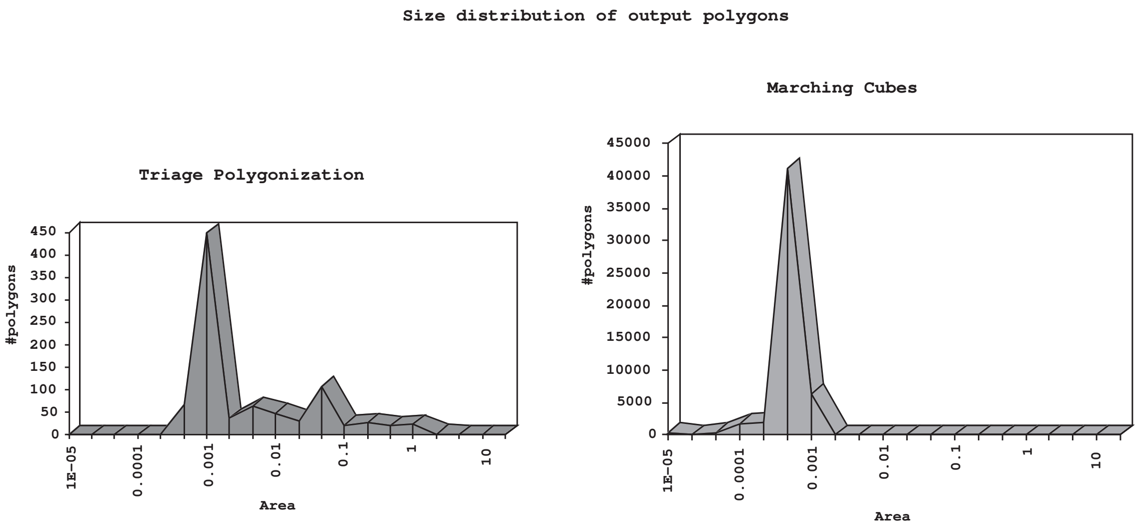

Figure 7.15. *Size distribution of polygons for the polygonization of the "CSG Example" scene.*

Figure 7.15 shows the size distribution of polygons obtained with the implemented polygonization methods. It can be seen that the Marching Cubes algorithm produces many small approximately equal size polygons. Triage Polygonization produces far fewer polygons which spread over a size range greater than 10^4. The result indicates that Triage Polygonization approximates the 0.5 iso-surface with large polygons if possible and with small polygons if necessary.

Figure 7.16 examines the effect on the implemented polygonization methods, of varying the rounding radius of an object. As an example we consider the "Cube In Cube" scene, which is depicted[2] in figure D.1 (f) – (h) quasi-convolutionally smoothed with a rounding radius of 0.08 units and polygonized with Triage Polygonization. To compare the Marching Cubes algorithm with Triage Polygonization we want to obtain polygonizations of equal quality, which is achieved by increasing the grid resolution of the Marching Cubes algorithm proportional to the rounding radius.

It can be seen that with decreasing rounding radius the number of output polygons and the execution time of Triage Polygonization changes only by a factor of around three, whereas for the Marching Cubes algorithm the execution time grows as the cube of the grid resolution and the number of output polygons increases quadratically. The results indicate that Triage Polygonization is much less sensitive to a varying rounding radius than the Marching Cubes algorithm.

For a rounding radius of around 0.25 units the Marching Cubes algorithm starts

[2]The original scene has a small unrounded object inside the hole in the big cube. Since the object is unrounded it is not interesting for us and we omit it in the picture and the following discussion.

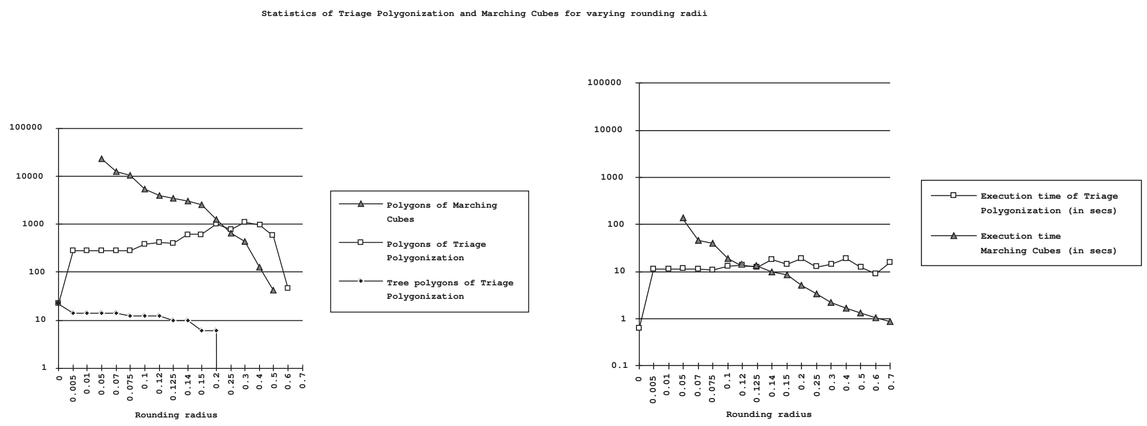

Figure 7.16. *Varying the rounding radius for the "Cube In Cube" scene. The graphs show the execution time and the number of output polygons of Triage Polygonization and the Marching Cubes algorithm.*

to produce less polygons than Triage Polygonization. This is also the threshold value from where on the object is completely rounded, i.e., tree polygons do not exist anymore. However, for normal quasi-convolutionally smoothed objects (i.e., with predominantly planar surfaces) Triage Polygonization outputs always fewer polygons than the Marching Cubes algorithm. In the graph for the execution time the crossover point, where the Marching Cubes algorithm becomes better than Triage Polygonization, is a little bit earlier, namely for a rounding radius of about 0.125 units. However, with this smoothing radius the "Cube In Cube" scene has already predominantly rounded surfaces. For a smoothing radius below the 0.125 value the execution time of Triage Polygonization is less than that of the Marching Cubes algorithm, indicating that for quasi-convolutionally smoothed objects with predominantly planar surfaces Triage Polygonization is indeed faster than the Marching Cubes algorithm. It can also be seen that for very small rounding radii it is not feasible to obtain with the Marching Cubes algorithm a polygonization of similar quality to Triage Polygonization.

Note that the number of tree polygons for Triage Polygonization decreases monotonically with increasing rounding radius and that Triage Polygonization yields a b-rep for an object quasi-convolutionally smoothed with a rounding radius of zero.

Summary

Table 7.4 summarizes some of the differences between Triage Polygonization and the Marching Cubes algorithm. Additional results are given in the next subsection.

The main difference between Triage Polygonization and the Marching Cubes algorithm is that Triage Polygonization uses information about a quasi-convolutionally smoothed object. Therefore Triage Polygonization emphasizes the polyhedral subdivision step of a polygonization. The resulting subdivision is hence considerably more efficient. Large parts of the object's surface are directly extracted from the subdivision and the number of cells in the subdivision is reduced. The subspace polygonization is only necessary for a small percentage of the cells of the subdivision and in general an iso-surface intersection is found.

	Triage Polygonization	Marching Cubes
Execution time spent on polyhedral subdivision	$\approx$65%	<10%
Successful subspace polygonizations (iso-surface intersection found)	$\approx$75%	$\approx$20%
#Subspace polygons per unclassified cell	$\approx$7	$\approx$2.4
Percentage of area of subspace polygons from total polygonized surface	$\approx$25%	$\approx$95%[a]

[a] The result is not 100%, as might be expected, because the total polygonized surface of a scene includes the surfaces of unsmoothed objects, which are extracted with a b-rep algorithm.

Table 7.4. *Summary of the polygonization statistics of Triage Polygonization and the Marching Cubes algorithm for the complex scenes.*

7.5.2 Comparison with Common Polygonization Methods

In this subsection we compare Triage Polygonization briefly with the four general polygonization methods for implicit surfaces reviewed in chapter 4. We restrict the comparison to the properties and quality criteria found for the polygonization methods and give some additional comments.

Properties

Table 4.2 on page 69 identified several properties of the four polygonization methods reviewed in chapter 4. We repeat the properties here and compare the results with those for Triage Polygonization.

The first three properties concern the space subdivision:

1.1 Type of cells.
1.2 Honeycomb property fulfilled?
1.3 Adaptive subdivision used?

The next items describe the subspace polygonization:

2.1 Type of polygons.

2.2 Does the polygonization have ambiguities?
2.3 Is the polygonization continuous?
2.4 Computation of intersection points.

Finally we are interested in the measures used to ensure continuity:

3.1 Continuity at shared edges.
3.2 Continuity at shared faces.
3.3 Disambiguation (for connection of intersection points).

	Lorenson & Cline (subsection 4.2.1)	Wyvill et al. (subsection 4.2.2)	Hall & Warren (subsection 4.2.3)	Bloomenthal (subsection 4.2.4)	Triage Polygonization (chapter 5)
1.1	Cubes	Cubes	Tetrahedra	Cubes	Convex polyhedra
1.2	Yes	Yes	Yes	No	Yes
1.3	No	No	Yes	Yes	No
2.1	Triangles	Triangles	Triangles	Triangles	Convex polygons
2.2	Yes	No	No	No	No
2.3	No	Yes	Yes	Yes	No
2.4	linear interpolation	linear interpolation	root search (regula falsi)	root search	root search (regula falsi)
3.1	Interpolates edge intersections linearly	Computes edge intersections only once	Compute edge intersections only once	Compute edge intersections only once	(no continuity)
3.2	(no continuity)	Has honeycomb and resolves ambiguities	Has tetrahedral honeycomb	Computes face intersections only once	Computes face intersections only once
3.3	(not resolved)	facial average	(no ambiguities)	central average	central average

Table 7.5. *Comparison of the reviewed polygonization algorithms with Triage Polygonization.*

Table 7.5 compares Triage Polygonization with the general polygonization methods for implicit surfaces introduced in chapter 4. The table demonstrates that Triage Polygonization is the only polygonization method using a polyhedral subdivision consisting of arbitrarily complex convex polyhedral cells. The subdivision trivially fulfills a honeycomb property (see figure 4.5) and does not use adaptive subdivision.

In contrast to the reviewed polygonization methods, which generate only triangles, Triage Polygonization outputs arbitrarily complex convex polygons. As with most of the other reviewed polygonization methods, Triage Polygonization has no ambiguities. A continuous surface can not be guaranteed with the current implementation but is very likely (also see the remarks in section 6.5). The computation of intersection points with the 0.5 iso-surface is performed with a regula falsi root search, the same method as used by Hall and Warren.

An intended alternative implementation of Triage Polygonization (section 6.5) allows us to use linear interpolation, which would bring Triage Polygonization more in line with the "Soft object" method from Wyvill et al. (see subsection 4.2.2).

Our method is the only one that does not guarantee continuity at shared edges. Section 6.5 gave reasons for this and suggested an alternative imperative language implementation. Note, though, that with our definition of the polyhedral subdivision and the assumption that a quasi-convolutionally smoothed object is intrinsically smooth, discontinuities are unlikely, and we in fact never encountered any.

We achieve continuity at shared faces by precomputing the intersection of a face with an iso-surface. A similar idea is used by Bloomenthal to achieve continuity over subdivided faces (see subsection 4.2.4). Our method to disambiguate the connection of intersection points with the central average can be understood as a generalization of Bloomenthal's method, which only deals with square faces.

7.5.3 Quality Criteria

Section 4.4 introduced a set of desirable features of a general-purpose polygonization method suggested by van Gelder and Wilhelms [vGW94]:

1. The algorithm should yield a continuous surface. Each polygon edge should be shared by exactly two polygons or lie in an external face of the entire volume.

2. The iso-surface should be a continuous function of the input data. A small change in the threshold value or some data value should produce a small change in the iso-surface.

3. The iso-surface should be topologically correct when the underlying function is "smooth enough".

4. The iso-surface produced should be neutral with respect to positive and negative sample data values (relative to threshold). Multiplying the samples (and threshold) by -1 should not alter the surface.

5. The algorithm should not create artifacts not implied by the data, such as bums and holes.

6. The algorithm should be fast.

Though in our application execution speed and a small number of output polygons are the most important features, it is interesting to categorize Triage Polygonization with above criteria. Table 7.6 compares Triage Polygonization with the algorithms reviewed in chapter 4.

It can be seen that Triage Polygonization is not guaranteed to produce a continuous and artifact free polygonization. However, we never had problems and section 6.5 gives some comments on how continuity might be achieved. For the remaining quality criteria Triage Polygonization is in line with the reviewed algorithms.

Note that for quasi-convolutionally smoothed scenes Triage Polygonization is considerably faster than the Marching Cubes algorithm. Since the Marching Cubes algorithm is known as one of the fastest polygonization algorithms available we have reason to believe that Triage Polygonization is also faster than the other reviewed algorithms.

	Lorenson & Cline (subsection 4.2.1)	Wyvill et al. (subsection 4.2.2)	Hall & Warren (subsection 4.2.3)	Bloomenthal (subsection 4.2.4)	Triage Polygonization (chapter 5)
1.	No	Yes	Yes	Yes	No
2.	No	No[a]	(unknown)	No	No
3.	Yes	Yes	Yes	Yes	Yes
4.	No[b]	Yes	(unknown)[c]		Yes
5.	No	Yes	Yes	Yes	No
6.	Van Gelder and Wilhelms [vGW94] report similar speed		(unknown)	(unknown)	20–30 times faster[d] than Lorenson & Cline

[a] Take an ambiguous case and change facial average value continuously from *high* to *low*.
[b] Case 12 in figure 4.1.
[c] With some extra effort this property can be achieved.
[d] For typically quasi-convolutionally smoothed scenes.

Table 7.6. *Quality criteria of the reviewed polygonization algorithms and Triage Polygonization.*

Finally note that Triage Polygonization is invariant under an affine linear transformation, a property which is not true or which is implementation dependent for the other reviewed algorithms. For example, our implementation of the Marching Cubes algorithm uses an axis aligned cube array for sampling and so is not invariant under affine linear transformation.

7.6 Conclusion

7.6.1 Triage Polygonization

It was shown both by images and statistical results that Triage Polygonization produces a fast and effective polygonization of quasi-convolutionally smoothed objects. The algorithm for Triage Polygonization divides into three subtasks: polyhedral subdivision, extraction of boundary faces, and subspace polygonization, all of which have about the same asymptotic time complexities.

The polyhedral subdivision is the most expensive subtask, but it is also very effective. Planar (unsmoothed) regions of a quasi-convolutionally smoothed object are in general identified and extracted without any subspace polygonization. The statistical results show that about 75% of an object's surface is extracted directly from the polyhedral subdivision.

The subspace polygonization is reduced to only a small fraction of the original object volume, hence reducing computation time considerably. Most of the cells identified for the subspace polygonization are indeed intersected by the 0.5 isosurface, which confirms that that the polyhedral subdivision is very effective.

Computation time is further reduced by computing local density fields, which allow a faster evaluation of density values and are used to identify convex swept surfaces.

Triage Polygonization also performs well for the extreme cases occurring for a quasi-convolutionally smoothed scene: for an object rounded with a spherical filter of zero radius Triage Polygonization performs optimally and yields the b-rep. A quasi-convolutionally smoothed object, which is completely rounded, i.e., it has no planar surface patches, is polygonized nicely with a computation time similar to that of the Marching Cubes algorithm.

We gave arguments that Triage Polygonization has on average a sub-quadratic space and time complexity. Several example scenes suggested a space complexity of $\Theta(n^{1.2})$ and a time complexity of $\Theta(n^{1.3})$. A theoretical complexity analysis yielded additionally a best case time complexity of $\Theta(n \log^2 n)$ and a lower bound for the worst case time complexity of $\Omega(n^3)$.

We can not guarantee that Triage Polygonization produces a closed continuous polygonized surface. However, no discontinuities occured with our example scenes.

7.6.2 Triage Polygonization vs. Marching Cubes

We implemented the Marching Cubes algorithm and compared it with Triage Polygonization. The statistical results obtained suggest that for general quasi-convolutionally smoothed scenes Triage Polygonization is about 20–30 times faster and outputs only 1%–2% of the polygons of the Marching Cubes algorithm.

The execution time and the number of output polygons of Triage Polygonization proves to be almost independent of the rounding radius of a quasi-convolutionally smoothed object. In the limit Triage Polygonization can calculate the b-rep of an unsmoothed object. Triage Polygonization is a considerable improvement over the Marching Cubes algorithm which has a time complexity increasing as the cube of the reciprocal rounding radius and a space complexity increasing quadratically. The Marching Cubes algorithm becomes infeasible for very small rounding radii.

The output polygons of Triage Polygonization can have arbitrarily many vertices and a vastly varying size, suggesting that indeed a surface is approximated by large polygons where possible and by small polygons where necessary. The Marching Cubes algorithm, on the other hand, outputs a large number of very small triangles, regardless of the shape of the polygonized surface.

The number of output polygons is especially important if complex objects are constructed by applying intersection and set difference operations to quasi-convolutionally smoothed objects. Since a set operation is performed by merging BSP trees the cost of the set operation increases more than linearly in the size of the polygonization. For this kind of scenes Triage Polygonization can be expected to perform even better in comparison to the Marching Cubes algorithm.

CHAPTER 8

Conclusion

In the previous chapters we presented Triage Polygonization, a new polygonization method for quasi-convolutionally smoothed objects. In this chapter we summarize this thesis. The main results are repeated and improvements and directions for future research are suggested. We conclude this thesis with a summary of its achievements.

8.1 Thesis Overview

This thesis began with an introduction to the concept of CSG objects and quasi-convolutionally smoothed polyhedra. The corresponding data structures were introduced and some example scenes were presented, which we also used in various subsequent chapters to test our algorithms.

Chapter 3 introduced BSP trees and two b-rep algorithms. We computed a b-rep by representing a CSG object with a BSP tree, which we augmented with a superset of the boundary of the CSG object. The boundary was extracted in a postprocessing step. A second algorithm computed a b-rep by merging BSP trees. The chapter concluded with a complexity analysis and statistical results, which suggested a sub-quadratic time and space complexity of the b-rep algorithms.

Chapter 4 gave an overview of existing polygonization methods. We presented four methods in detail and extracted three common aspects: space subdivision, subspace polygonization, and a continuity constraint. These aspects formed the motivation for Triage Polygonization. We successfully implemented the Marching Cubes algorithm and employed it in chapter 7 as a benchmark program for Triage Polygonization. The chapter concluded with a listing of quality criteria for polygonization methods.

Chapter 5 presented Triage Polygonization, a new fast polygonization method for quasi-convolutionally smoothed polyhedra. The polygonization method was developed by combining concepts of CSG objects, BSP tress, and general polygonization methods for implicit surfaces.

Chapter 6 gave implementation details for Triage Polygonization. We described numerical problems in the point classification and successfully solved the problems by testing for intersection points against a displaced iso-surface. The aspect of continuity of the polygonized surface was dealt with briefly. Triage Polygonization was successfully adapted to an object model extended by introducing clipping planes and different rounding radii for the half-spaces of an object.

Chapter 7 gave statistical results and analyzed the performance of Triage Polygonization. We suggested an approximately linear time and space complexity for quasi-convolutionally smoothed scenes and a sub-quadratic time and space complexity for quasi-convolutionally smoothed objects. In comparison to the Marching Cubes algorithm, Triage Polygonization was considerably faster and resulted far fewer polygons.

8.2 Results

We examined the performance of Triage Polygonization for quasi-convolutionally smoothed scenes and quasi-convolutionally smoothed objects.

8.2.1 Performance Analysis

Triage Polygonization, applied to quasi-convolutionally smoothed objects, proved to be in the same complexity class as a b-rep algorithm. We suggested therefore that Triage Polygonization has a sub-quadratic time and space complexity in the average case. The best case time complexity was given as $\Theta(n \log^2 n)$ and for the worst case a lower bound of $\Omega(n^3)$ was obtained.

The statistical results confirmed these results suggesting an average time complexity of about $\Theta(n^{1.3})$ and an average space complexity of about $\Theta(n^{1.2})$.

We assumed that quasi-convolutionally smoothed scenes are usually assembled from disjoint objects and gave evidence that Triage Polygonization is approximately linear in the size of a scene.

Influence of the Rounding Radius

Triage Polygonization performs best for quasi-convolutionally smoothed objects smoothed with a rounding radius small in comparison to their size. Such objects have predominantly planar surfaces with only edges and corners rounded. Triage Polygonization extracts planar surfaces with minimum fragmentation and successfully approximates rounded edges and corners with a minimum number of polygons. We believe that for the above case Triage Polygonization is superior to all general polygonization methods for implicit surfaces known to us.

Triage Polygonization also performs well for strongly rounded objects and in such cases its performance is similar to general polygonization methods for implicit surfaces.

Distribution of Execution Time

For most quasi-convolutionally smoothed scenes and objects all three steps of Triage Polygonization seemed to have similar asymptotic complexities. This observation was supported on a theoretical basis by an analogy to a b-rep algorithm. In absolute terms the execution time of the polyhedral subdivision and subspace polygonization took about 60% and 30%, respectively, of the total execution time for quasi-convolutionally smoothed scenes and was more closely balanced for more complicated quasi-convolutionally smoothed objects. The extraction of tree polygons in all cases took only about 10% of the total execution time of Triage Polygonization.

8.2.2 Comparison with the Marching Cubes Algorithm

We compared statistical results for Triage Polygonization with those obtained for the Marching Cubes algorithm. For typical quasi-convolutionally smoothed scenes we found out that Triage Polygonization is about 20–30 times faster and creates only about 1–2% of the number of polygons produced by the Marching Cubes algorithm. Whereas the Marching Cubes algorithm produces many small polygons of approximately the same size, Triage Polygonization tries to adjust the polygon size to the surface geometry. Planar areas of the object's surface are represented with very large polygons and rounded edges with long thin rectangles or triangles.

As a result of the comparison we identified different "philosophies" of the algorithms. Summarized, Triage Polygonization is more intelligent than the Marching Cubes algorithm and uses information about a quasi-convolutionally smoothed object. This results in a more efficient subdivision. Large parts of the object's surface are directly extracted from the subdivision, the number of cells in the subdivision is reduced, and the subspace polygonization, if necessary, is usually successful.

8.2.3 Properties of Triage Polygonization

Independent of the Rounding Radius

The quality of the polygonization obtained with Triage Polygonization is largely independent from the rounding radius. We noticed that varying the rounding radius by a factor of 1000 caused variations in the execution time and the number of produced polygons by only a factor of three. Triage Polygonization gives correct results in the extreme cases of quasi-convolutional smoothing: the polygonization of a quasi-convolutionally smoothed object rounded with a spherical filter of radius zero is its b-rep, and if the object is rounded with a sufficiently large rounding radius, the polygonization is empty.

Invariant under Affine Linear Transformation

Since the spatial search structure of a BSP trees is intrinsic to the object it represents and transforms with it, Triage Polygonization is invariant under affine linear transformations. Hence, for example, the polygonization of a rotated object is identical to the rotated polygonization of the original object.

Continuity Aspect

Continuity of the polygonized surface can not be guaranteed. However, in all our example scenes we obtained a continuous polygonization. We suggested a different implementation in an imperative language to reduce the possibility of a discontinuous polygonization even further.

8.3 Future Work

8.3.1 Application Improvements

Execution Time

We implemented Triage Polygonization in non-optimized CLEAN code. Preliminary results suggest that an efficient implementation in C/C++ can be expected to be 3–10 times faster.

Using a hash table for the computation of edge and face intersections eliminates the tree traversal for the precomputation of face intersections and avoids repeated computations of intersection points. Since an edge is usually shared by at least two unclassified cells we expect this scheme to halve the execution time of the subspace polygonization.

Furthermore sharing edge intersections allows us to use linear interpolation to compute the iso-surface intersection of an edge. A considerable improvement in the execution time, compared to the root search, can be expected.

Naylor, Amatides, and Thibault [NAT90] suggest a b-rep algorithm which directly merges BSP trees and they report that their method is more efficient than the conventional approach to insert a CSG object into a BSP tree. Constructing a DBSP tree with a similar method as Naylor et al. use for BSP trees is likely to improve the asymptotic complexity of our algorithm.

The algorithm suggested by Naylor et al. is also suitable for the polygonization of a scene when set operations on polygonized objects must be performed.

If the scene viewpoint is fixed the algorithm can be improved by pruning parts of the DBSP tree. The subspace polygonization and the extraction of tree polygons is not necessary for cells hidden by an already detected surface.

Quality of the Polygonization

In the current implementation we always approximate a face intersection with two edges and refine a topological polygon with a point near its center. Clearly it is more appropriate to make the refinement process dependent on the curvature of the surface. An adaptive refinement process similar to that suggested by Hall and Warren [HW90] or Bloomenthal [Blo88, BW90] can be employed.

Refinement takes place if the surface curvature over a polygon varies strongly. Estimations of the surface curvature could be obtained from the density gradients in the polygon vertices. Other refinement criteria are suggested by von Herzen and Barr [vHB87].

The quality of the subspace polygonization could also be improved with techniques common to triangulation methods (e.g., [Bow81, vHB87]). For example it might be desired to minimize the aspect ratio of triangles.

Figure D.6 shows that the polygonization of heavily rounded objects might have a "banded structure". This is due to very small and thin cells in the polyhedral subdivision. It would be desirable to eliminate these cells. However, it is not clear how this would compromise the classification of cells of the polyhedral subdivision.

8.3.2 Extended Applications

We have designed Triage Polygonization to polygonize quasi-convolutionally smoothed objects. Currently research is under way to adapt Triage Polygonization to a truly convolutionally smoothed scene. Initial results are promising and suggest that the principles used in Triage Polygonization might prove useful in many other applications.

8.3.3 Parallelization

The subspace polygonization is limited to cells of a DBSP tree. In the implementation presented in this thesis face intersections are precomputed and therefore the subspace polygonizations for different cells are independent of each other and can be parallelized. An implementation with shared edges and face intersections demands a distributed system with message passing. A common hash table for edge and face intersections would be required and would be accessed to by processes polygonizing different subspaces.

The parallelization of the polyhedral subdivision and tree extraction steps is more difficult. Note, however, that the two subtrees of a BSP node cover disjoint areas of space and therefore operations on them can be performed by different processes.

8.4 Summary

The starting point of the research was the work done by Dr. Richard Lobb on quasi-convolutionally smoothed polyhedra. We developed, implemented, and analyzed Triage Polygonization, a new polygonization method designed for quasi-convolutionally smoothed polyhedra.

Triage Polygonization proved to be very successful. It identified large planar surfaces of quasi-convolutionally smoothed objects correctly and approximated rounded edges and corners well. In comparison to the Marching Cubes algorithm, a popular polygonization method implemented by us as a benchmark program, Triage Polygonization was 20–30 times faster and produced only 1–2% of the polygons.

Two b-rep algorithms were implemented and a complexity analysis on them was developed. We investigated conventional polygonization methods and extracted a common framework to act as a basis for comparison.

All the goals for this thesis were achieved and although there is much room for improvement, the results are more than satisfactory considering the limited time frame available. Hopefully the work has established Triage Polygonization as a new polygonization method in the computer graphics community, which can be further improved in the future.

APPENDIX A

Theorems

A.1 Series and Sequences

Theorem A.1 *Let* $s_k = \sum_{i=0}^{n} q^i i$*. Then*

$$s_k = \begin{cases} \Theta(nq^n) & q > 1 \\ \Theta(n^2) & q = 1 \\ \Theta(1) & q < 1 \end{cases}$$

Proof:
For $q = 1$

$$\sum_{i=0}^{n} q^i i = \sum_{i=0}^{n} i = \Theta(n^2)$$

For $q \neq 1$

$$\begin{aligned} qs_n - s_n &= \sum_{i=0}^{n} q^{i+1} i - \sum_{i=0}^{n} q^i i \\ &= q^{n+1} n + \sum_{i=1}^{n} (q^i (i-1) - q^i i) - q^0 0 \\ &= q^{n+1} n - \sum_{i=1}^{n} q^i \\ &= q^{n+1} n - (\sum_{i=0}^{n} q^i - 1) \\ &= q^{n+1} n - \left(\frac{q^{n+1} - 1}{q - 1} - 1 \right) \end{aligned}$$

$$= q^{n+1}n - \frac{q^{n+1}-q}{q-1}$$

hence

$$\begin{aligned} s_k &= \frac{qs_k - q}{q-1} \\ &= \frac{q^{n+1}n - \frac{q^{n+1}-q}{q-1}}{q-1} \end{aligned}$$

□

A.2 Recurrence Relations

The following recurrence relations are only given for the special case that $n = b^m$ for some $m \in \mathbb{N}$. It is possible to formulate them for general $n \in \mathbb{N}$. As an example we do this for theorem A.2 and show that the asymptotic complexity does not change. Similar arguments apply for the other recurrence relations in this section, though we won't proof it.

Theorem A.2 *Let*

$$\begin{aligned} &T(n) = aT(\frac{n}{b}) + n^k \\ &T(1) = c \\ &\textit{where } a, b, c > 1 \in I\!N \end{aligned}$$

Then

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^k) & a < b^k \end{cases}$$

Proof: [Man95] □

Remark A.1 The above recurrence relation is in [Man95] only defined for n being a power of b. In most applications (e.g., complexity analysis) recurrence relations of this type are formulated for arbitrary $n \in \mathbb{N}$ by choosing the next highest integer for a fraction $\frac{n}{b}$. The next theorem shows that this does not change the asymptotic complexity.

Theorem A.3 *Let*

$$T_1(n) = aT_1(\frac{n}{b}) + n^k$$
$$T_1(1) = c$$
$$\textit{where } a, b, c > 1 \in I\!N$$

and

$$T_2(n) = aT_2(\lceil \frac{n}{b} \rceil) + n^k$$
$$T_2(1) = c$$
$$\textit{where } a, b, c > 1 \in I\!N$$

Then

$$T_2 = \Theta(T_1) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^k) & a < b^k \end{cases}$$

Proof: For n define n^* as

$$\begin{aligned} n^* &= b^{\lceil \log_b n \rceil} \\ &= \frac{nb^{\lceil \log_b n \rceil}}{b^{\log_b n}} \\ &= nb^{\lceil \log_b n \rceil - \log_b n} \\ &= nb^{\gamma} && \text{where } 0 \le \gamma < 1 \\ &= nc && \text{for some constant } c > 1 \end{aligned}$$

i.e., n^* is the first natural number bigger than or equal to n being a power of b. We can prove by strong induction on n that $T_2(n) \le T_1(n^*)$.

Induction bases: $n = 1$

If $n = 1$ then $n^* = 1$ and

$$T_2(1) = T_1(1)$$

Induction hypotheses:

For all $k < n$

$$T_2(k) \le T_1(k^*)$$

Induction step: $n > 1$

$$\begin{array}{lcll}
T_2(n) & = & aT_2(\lceil \frac{n}{b} \rceil) + n^k & [\text{Definition of } T_2] \\
 & \leq & aT_2(\lceil \frac{n}{b} \rceil) + n^{*k} & [n^* > n] \\
 & \leq & aT_1(\lceil \frac{n}{b} \rceil^*) + n^{*k} & [\text{IH., possible since } b \geq 2 \text{ and hence } \lceil \frac{n}{b} \rceil \leq n\ ] \\
 & = & aT_1(\frac{n^*}{b}) + n^{*k} & [\lceil \frac{n}{b} \rceil^* = \frac{n^*}{b}, \text{ see below}] \\
 & = & T_1(n^*) &
\end{array}$$

It remains to show that $\frac{n^*}{b} = \lceil \frac{n}{b} \rceil^*$,i.e., $\frac{n^*}{b}$ is next highest power of b with respect to $\lceil \frac{n}{b} \rceil$.

$$\begin{array}{lcll}
\lceil \frac{n}{b} \rceil^* & = & b^{\lceil \log_b \lceil \frac{n}{b} \rceil \rceil} & [\text{Definition of } ^*] \\
 & = & b^{\lceil \log_b \frac{n}{b} \rceil} & \left[\begin{array}{l} \lceil \log_b x \rceil = \lceil \log_b y \rceil = m \text{ for some } m \in N \\ \Longleftrightarrow b^{m-1} < x, y \leq b^m \end{array} \right] \\
 & = & b^{\lceil (\log_b n) - 1 \rceil} & \\
 & = & b^{\lceil (\log_b n) \rceil - 1} & \\
 & = & \frac{n^*}{b} & [\text{Definition of } n^*]
\end{array}$$

From theorem A.2 we know that

$$T_1(n^*) = \begin{cases} \Theta(n^{*\log_b a}) & a > b^k \\ \Theta(n^{*k} \log n^*) & a = b^k \\ \Theta(n^{*k}) & a < b^k \end{cases}$$

Since $n^* = nc$ for some constant $c > 1$ we get

$$T_1(n^*) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^k) & a < b^k \end{cases}$$

and because $T_2(n) \leq T_1(n^*)$ we get

$$T_2(n) = \begin{cases} O(n^{\log_b n}) & a > b^k \\ O(n^k \log n) & a = b^k \\ O(n^k) & a < b^k \end{cases} \tag{A.1}$$

Analogously we can define n^{**} as

$$\begin{array}{lcll}
n^{**} & = & b^{\lfloor \log_b n \rfloor} & \\
 & = & \frac{nb^{\lfloor \log_b n \rfloor}}{b^{\log_b n}} & \\
 & = & nb^{\lfloor \log_b n \rfloor - log_b n} & \\
 & = & nb^{\mu} & \text{where } -1 < \mu \leq 0 \\
 & = & nc & \text{for some constant } 0 < c \leq 1
\end{array}$$

i.e., n^{**} is the first natural number smaller than or equal to n being a power of b. Similarly as above we can prove by induction on n that $T_2(n) \geq T_1(n^{**})$, and hence we get

$$T_2(n) = \begin{cases} \Omega(n^{\log_b a}) & a > b^k \\ \Omega(n^k \log n) & a = b^k \\ \Omega(n^k) & a < b^k \end{cases} \tag{A.2}$$

Equation A.1 and A.2 together yield

$$T_2(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^k) & a < b^k \end{cases} \tag{A.3}$$

as claimed.

□

Theorem A.4 *Let* $T(n) = aT(\frac{n}{b}) + f(n)$, $T(1) = c$. *Then*

$$\begin{aligned} & T(n) = n^{\log_b a}(\sum_{i=1}^{m} g(b^i) + c) \\ & where \\ & m = \log_b n \ , \ g(n) = \frac{f(n)}{n^{\log_b a}} \ , \\ & a, b, c \in N \ \ and \ a, b > 1 \end{aligned}$$

Proof: The proof is made by induction on n. We only give it for n being a power of b. To extend theorem A.4 for all $n \in \mathbb{N}$ see remark A.1.

Induction bases: $n = 1$

$$T(1) = c = 1^{\log_b a}(\sum_{i=1}^{0} g(b^i) + c)$$

Induction hypotheses: For all $l < nb$

$$T(l) = l^{\log_b a}(\sum_{i=1}^{\log_b l} g(b^i) + c)$$

Induction step:

$$\begin{array}{rcll}
T(nb) & = & aT(n) + f(nb) & [\text{Definition of } T(n)] \\
& = & a(n^{\log_b a}(\sum_{i=1}^{m} g(b^i) + c)) + f(nb) & [\text{IH.}] \\
& = & a(a^m(\sum_{i=1}^{m} g(b^i) + c)) + f(nb) & [n^{\log_b a} = a^{\log_b n}] \\
& = & a^{m+1}((\sum_{i=1}^{m} g(b^i) + c) + \frac{f(nb)}{a^{m+1}}) & \\
& = & (nb)^{\log_b a}((\sum_{i=1}^{m} g(b^i) + c) + \frac{f(nb)}{(nb)^{\log_b a}}) & \left[\begin{array}{l} a^{m+1} = (b^{\log_b a})^{m+1} \\ = (b^{m+1})^{\log_b a} = (nb)^{\log_b a} \end{array}\right] \\
& = & (nb)^{\log_b a}((\sum_{i=1}^{m} g(b^i) + c) + \frac{f(b^{m+1})}{(b^{m+1})^{\log_b a}}) & [nb = b^m b = b^{m+1}] \\
& = & (nb)^{\log_b a}(\sum_{i=1}^{m+1} g(b^i) + c) &
\end{array}$$

□

Theorem A.5 *Let $n, m \in I\!N$ and*

$$T(m,n) = aT(\frac{m}{b_1}, \lceil \frac{n}{b_2} \rceil) + n^k \ , \ T(1,n) = f(n)$$

where

$$a, b_1 \in I\!N,\ b_2, k \in I\!R^+ \text{ and } b_1 > 1$$

Then

$$T(m,n) = \begin{cases} \Theta(f(\frac{n}{m^{\log_{b_1} b_2}})\ m^{\log_{b_1} a} + n^k m^{\log_{b_1}(\frac{a}{b_2^k})}) & a > b_2^k \\ \Theta(f(\frac{n}{m^{\log_{b_1} b_2}})\ m^{\log_{b_1} a} + n^k \log_{b_1} m) & a = b_2^k \\ \Theta(f(\frac{n}{m^{\log_{b_1} b_2}})\ m^{\log_{b_1} a} + n^k) & a < b_2^k \end{cases}$$

Proof: The proof is made by induction on m. We only give it for m being a power of b_1. Furthermore, with an argument similar to theorem A.3, it is sufficient to compute

$$T(m,n) = aT(\frac{m}{b_1}, \frac{n}{b_2}) + n^k \ , \ T(1,n) = f(n)$$

where

$$a, b_1 \in \mathrm{I\!N},\ b_2, k \in \mathrm{I\!R}^+ \text{ and } b_1 > 1$$

We show

$$T(m,n) = f(\frac{n}{m^{\log_{b_1} b_2}})\ a^{\log_{b_1} m} + n^k \sum_{i=0}^{\log_{b_1} m - 1} \left(\frac{a}{b_2^k}\right)^i$$

To extend the proof and the theorem to all $m \in \mathbb{N}$ see the remark A.1.

Induction bases: $m = 1$

$$T(1,n) = f(n) = f(\frac{n}{1^{\log_{b_1} b_2}})\ a^{\log_{b_1} 1} + n^k \sum_{i=0}^{\log_{b_1} 1 - 1} \left(\frac{a}{b_2^k}\right)^i$$

Induction hypotheses: For all $l < mb$

$$T(l,n) = f(\frac{n}{l^{\log_{b_1} b_2}})\ a^{\log_{b_1} l} + n^k \sum_{i=0}^{\log_{b_1} l - 1} \left(\frac{a}{b_2^k}\right)^i$$

Induction step: $m > 1$

$$\begin{aligned} T(m,n) &= aT(\frac{m}{b_1}, \frac{n}{b_2}) + n^k \\ &= a\ (f(\frac{n}{b_2(\frac{m}{b_1})^{\log_{b_1} b_2}})\ a^{\log_{b_1}(\frac{m}{b_1})} + \left(\frac{n}{b_2}\right)^k + \sum_{i=0}^{log_{b_1}(\frac{m}{b_1})-1} \left(\frac{a}{b_2^k}\right)^i) + n^k \\ &= f(\frac{n}{m^{\log_{b_1} b_2}})\ a^{\log_{b_1} m} + n^k \sum_{i=0}^{\log_{b_1} m - 1} \left(\frac{a}{b_2^k}\right)^i \end{aligned}$$

□

Theorem A.6 *Let $T(n) = aT(\frac{n}{b}) + n^k \log n$, $T(1) = c$ where $a, b, c \in \mathbb{N}$, $a, b > 1$.*

then

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log^2 n) & a = b^k \\ \Theta(n^k \log n) & a < b^k \end{cases}$$

Proof: First note that

$$f_1 \in \Theta(g_1), f_2 \in \Theta(g_2) \Rightarrow f_1 f_2 \in \Theta(g_1 g_2) \tag{A.4}$$

With theorem A.4 we can rewrite the recurrence relation as

$$\begin{aligned} T(n) &= n^{\log_b a}(\sum_{i=1}^{m} \frac{(b^i)^k \log b^i}{a^i} + c) \\ &= n^{\log_b a}(\log b \sum_{i=1}^{m} (\frac{b^k}{a})^i i + c) \end{aligned} \tag{A.5}$$

Define

$$q := \frac{b^k}{a}$$

and note

$$\begin{array}{lll} q < 1 & \text{iff} & a > b^k \\ q = 1 & \text{iff} & a = b^k \\ q > 1 & \text{iff} & a < b^k \end{array}$$

and

$$\begin{aligned} a^{\log_b n} &= n^{\log_b a} \\ (b^k)^{\log_b n} &= n^k \end{aligned}$$

then theorem A.1 yields

$$\begin{aligned} \sum_{i=0}^{m} q^i i &= \begin{cases} \Theta(mq^m) & q > 1 \\ \Theta(m^2) & q = 1 \\ \Theta(1) & q < 1 \end{cases} \\ &= \begin{cases} \Theta(\log_b n \frac{n^k}{n^{\log_b a}}) & a < b^k \\ \Theta(\log_b^2 n) & a = b^k \\ \Theta(1) & a > b^k \end{cases} \end{aligned} \tag{A.6}$$

Since b and c are constants, we can use equation A.4 to combine equation A.5 and A.6 to

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \\ \Theta(n^k \log^2 n) & a = b^k \\ \Theta(n^k \log n) & a < b^k \end{cases}$$

as was claimed. □

A.3 Set Operations

Definition A.1 *Let A be a set. Then its interior $int(A)$, its closure $\overline{A}$ and its boundary ∂A with respect to a domain Ω are defined as*

$$\begin{aligned} int(A) &= \{x \in \Omega | \exists \epsilon > 0 : B_\epsilon(x) \subseteq A\} \\ \overline{A} &= int(A^c)^c \\ \partial A &= \{x \in \Omega | \forall \epsilon > 0 : (B_\epsilon(x) \cap A) \neq \emptyset \wedge (B_\epsilon \cap A^c) \neq \emptyset\} \end{aligned}$$

where $B_\epsilon(x)$ is the ϵ-neighborhood of x.

Theorem A.7 *Let A and B be arbitrary sets.*

Then

$$\begin{aligned}\partial(A \cap^\star B) &= (\partial A \cap int(B)) \cup (\partial B \cap int(A)) \cup \\ &\left\{ x \in \Omega | \forall \epsilon > 0 : \begin{array}{l} (B_\epsilon \cap int(A) \cap int(B)) \neq \emptyset \wedge \\ (B_\epsilon \cap int(A)^c \cap int(B)^c) \neq \emptyset \end{array} \right\}\end{aligned}$$

Proof:

$$\begin{aligned}
& \partial(A \cap^\star B) \\
= \; & \partial(\overline{int(A \cap B)}) \\
= \; & \partial(int(A \cap B)) \\
= \; & \partial(int(A) \cap int(B)) \\
= \; & \left\{ x \in \Omega | \forall \epsilon > 0 : \begin{array}{l} B_\epsilon(x) \cap (int(A) \cap int(B)) \neq \emptyset \wedge \\ B_\epsilon(x) \cap (int(A) \cap int(B))^c \neq \emptyset \end{array} \right\} \\
= \; & \left\{ x \in \Omega | \forall \epsilon > 0 : \begin{array}{l} B_\epsilon(x) \cap (int(A) \cap int(B)) \neq \emptyset \wedge \\ B_\epsilon(x) \cap \left(\begin{array}{l} (int(A)^c \cap int(B)) \cup (int(A) \cap int(B)^c) \\ \cup (int(A)^c \cap int(B)^c) \end{array} \right) \neq \emptyset \end{array} \right\} \\
= \; & \left\{ x \in \Omega | \forall \epsilon > 0 : \begin{array}{l} B_\epsilon(x) \cap (int(A) \cap int(B)) \neq \emptyset \wedge \\ \left(\begin{array}{l} B_\epsilon(x) \cap int(A)^c \cap int(B) \neq \emptyset \vee \\ B_\epsilon(x) \cap int(A) \cap int(B)^c \neq \emptyset \vee \\ B_\epsilon(x) \cap int(A)^c \cap int(B)^c \neq \emptyset \end{array} \right) \end{array} \right\} \\
= \; & \begin{array}{l} \{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(A) \cap int(B) \neq \emptyset \wedge B_\epsilon \cap int(A)^c \cap int(B) \neq \emptyset\} \cup \\ \{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(A) \cap int(B) \neq \emptyset \wedge B_\epsilon \cap int(A) \cap int(B)^c \neq \emptyset\} \cup \\ \{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(A) \cap int(B) \neq \emptyset \wedge B_\epsilon \cap int(A)^c \cap int(B)^c \neq \emptyset\} \end{array} \\
= \; & \begin{array}{l} (\{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(A) \neq \emptyset \wedge B_\epsilon \cap int(A)^c \neq \emptyset\} \cap int(B)) \cup \\ (\{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(B) \neq \emptyset \wedge B_\epsilon \cap int(B)^c \neq \emptyset\} \cap int(A)) \cup \\ \{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(A) \cap int(B) \neq \emptyset \wedge B_\epsilon \cap int(A)^c \cap int(B)^c \neq \emptyset\} \end{array} \\
= \; & \begin{array}{l} (\partial A \cap int(B)) \cup (\partial B \cap int(A)) \cup \\ \{x \in \Omega | \forall \epsilon > 0 : B_\epsilon \cap int(A) \cap int(B) \neq \emptyset \wedge B_\epsilon \cap int(A)^c \cap int(B)^c \neq \emptyset\} \end{array}
\end{aligned}$$

□

Theorem A.8 *Let P be a polyhedron and h a partitioning plane with outward normal $\vec{n}_h$ and inside half-space H_{in}. Denote with $f_1, \ldots, f_n$ the faces of the polyhedron P and with $\vec{n}_{f_1}, \ldots, \vec{n}_{f_n}$ their outward normals.*

Then (except for a zero set[1]*)*

$$\begin{aligned}\partial(P \cap^\star H_{in}) &= (\partial P \cap H_{in}) \cup (h \cap int(P)) \cup \\ &\quad \{x \in f_i | i = 1, \ldots, n, f_i \subseteq h, \vec{n}_{f_i} = \vec{n}_h\}\end{aligned}$$

This means that the boundary of the regularized intersection of the polyhedron P and the half-space H_{in} is the boundary of P inside of H_{in}, the part of the plane h inside of P, and the faces of the polyhedron P, which lie on the partitioning plane h and have the same orientation as it.

Proof:

With theorem A.7 we only need to show $S_1 = S_2$ (except for a zero set) where

$$\begin{aligned}S_1 &= \{x \in \Omega | \forall \epsilon > 0 : B_\epsilon(x) \cap int(P) \cap H_{in} \neq \emptyset \wedge B_\epsilon(x) \cap int(P)^c \cap H_{in}^c \neq \emptyset\} \\ S_2 &= \{x \in f_i | i = 1, \ldots, n, f_i \subseteq h, \vec{n}_{f_i} = \vec{n}_h\}\end{aligned}$$

Every element x of S_1 lies both on the boundary of P and h, and hence on some face f_i of the polyhedron P. We can partition S_1 into

$$S_1 = \bigcup_{i=1,\ldots,n} S_{1,i}$$

where $S_{1,i}$ is the intersection of S_1 with face f_i, i.e.,

$$S_{1,i} = \left\{ x \in (f_i \cap h) | \forall \epsilon > 0 : \begin{array}{l} B_\epsilon(x) \cap int(P) \cap H_{in} \neq \emptyset \wedge \\ B_\epsilon(x) \cap int(P)^c \cap H_{in}^c \neq \emptyset \end{array} \right\}$$

If face f_i does not lie on h the intersection of f_i with h is at most an edge and hence $S_{1,i}$ is a zero set. If f_i lies on h but has different orientation, the intersection of the interior of P and H_{in} is empty, and hence $S_{1,i}$ is the empty set. Finally if f_i lies on h and has the same orientation, then $S_{1,i} \subseteq S_2$ and hence $S1 \subseteq S2$ (except for a zero set).

Vive versa partition S_2 into

$$S_2 = \bigcup_{i=1,\ldots,n} S_{2,i}$$

[1]This a zero set with respect to the two-dimensional boundary ∂P. Formally define a piecewise continuous bijective mapping from ∂P into $\mathbb{R}^2$. A set is a zero set in ∂P if its bijective mapping is a zero set in $\mathbb{R}^2$ (see also appendix C).

$$S_{2,i} = \{x \in f_i | f_i \subseteq h, \vec{n}_{f_i} = \vec{n}_h\}$$

If f_i does not lie on h or has a different orientation, then $S_{2,i}$ is the empty set. Otherwise we can find for each x in each ϵ-neighborhood of x a point, which lies inside both P and H_{in}, and also a point which lies outside both P and H_{in}, i.e., $S_{2,i} \subseteq S_1$ and hence $S_2 \subseteq S_1$. Therefore $S_2 = S_1$ which concludes the proof. □

A.4 Analysis

Theorem A.9 *Let G be open in X and $f : G \rightarrow Y$ be a continuously differentiable map. Let $p_0, p_1 \in G$ be such that the line segment S between p_0 and p_1 is contained in G. Then*

$$f(p_1) - f(p_0) = \int_0^1 f'(p_0 + t(p_1 - p_0))dt \ (p_1 - p_0)$$

and it follows the inequality

$$\|f(p_1) - f(p_0)\| \leq \max_{p \in S} \|f'(p)\| \ \|p_1 - p_0\| \tag{A.7}$$

Proof: [Heu81, page 340] □

Theorem A.10 *Let G be open in $I\!R^3$, $p : [0,1] \rightarrow G$ be a parameterized line segment and $\rho : G \rightarrow I\!R$ be a continuous density field. Then*

$$|\rho(p(t_1)) - \rho(p(t_0))| \leq \max_{t \in [t_0, t_1]} \|\nabla \rho(p(t))\| \ \|p(t_1) - p(t_0)\| \tag{A.8}$$

Proof: EquationA.7 from theorem A.9 yields

$$|\rho(p(t_1)) - \rho(p(t_0))| \leq \max_{t \in [t_0, t_1]} \|\partial \frac{\rho(p(t))}{\partial t}\| \ |t_1 - t_0|$$

This expression is simplified by using the chain rule

$$\partial \frac{\rho(p(t))}{\partial t} = \langle \nabla \rho(p(t), p'(t)) \rangle$$

and the Cauchy-Schwarzsche inequality

$$\|\partial\frac{\rho(p(t))}{\partial t}\| \leq \|\nabla\rho(p(t))\| \; \|p'(t)\|$$

to yield equation A.8 $\square$

Definition A.2 *A function $f \in C(I\!R, I\!R)$ is convex in $[a,b]$ if for any choice of x_1 and x_2, $a \leq x_1 < x_2 \leq b$, and any α with $0 < \alpha < 1$*

$$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha)f(x_2)$$

f is concave in $[a,b]$ if $-f$ is convex.

Theorem A.11 *If $f''(x)$ exists and keeps a constant sign in (a,b), then $f'(x)$ is monotone and $f(x)$ or $-f(x)$ is convex according as $f''(x) > 0$ or < 0 in (a,b).*

Proof: [Hil65, page 463] $\square$

Theorem A.12 *Let $e_1, e_2 \in C^2(I\!R, I\!R)$ be two plane spread functions with e_1 and e_2 being positive, monotonous, and concave, and $e_1^{'}, e_2^{'}$ having the same sign. Apply the plane spread functions e_1 and e_2 to the planes $Plane_1$ and $Plane_2$, respectively, and define a density field $\rho \in C^2(I\!R^3, I\!R)$ as*

$$\rho(x) = e_1(dist(Plane_1, x))e_2(dist(Plane_2, x))$$

Then each isosurface in the density field is convex.

Proof: Show that for an arbitrary constant $c \in \mathbb{R}$ the set $A := \{x \in \mathbb{R}^3 \mid \rho(x) \geq c\}$ is convex. With definition A.2 and theorem A.11 it is sufficient to show

$$\forall p_1, p_2 \in \mathbb{R}^3\!: \; \rho(p_1) = \rho(p_2) = c \text{ and } t \in (0,1) \subseteq \mathbb{R}$$
$$\frac{\partial^2}{\partial^2 t}\rho(p_1 + t(p_2 - p_1)) \leq 0$$

Take any p_1, p_2 with $\rho(p_1) = \rho(p_2) = c$. Define $d_1(x) := dist(Plane_1, x)$, $d_2(x) := dist(Plane_2, x)$ and let $p := p_1 + t(p_2 - p_1)$ for any $t \in (0,1)$.

First note that

$$d_1(p_1 + t(p_2 - p_1)) = d_1(p_1) + t(d_1(p_2) - d_1(p_1))$$

and

$$d_2(p_1 + t(p_2 - p_1)) = d_2(p_1) + t(d_2(p_2) - d_2(p_1))$$

and hence

$$\begin{aligned} \frac{\partial}{\partial t} e_1(d_1(p)) &= \Delta d_1 \; e_1^{'}(d_1(p)) \\ \frac{\partial}{\partial t} e_2(d_2(p)) &= \Delta d_2 \; e_2^{'}(d_2(p)) \end{aligned}$$

where $\Delta d_1 := d_1(p_2) - d_1(p_1)$ and $\Delta d_2 := d_2(p_2) - d_2(p_1)$.

We obtain for the derivative of ρ

$$\frac{\partial}{\partial t}\rho(p) = \Delta d_1 \; e_1^{'}(d_1(p)) \; e_2(d_2(p)) + \Delta d_2 \; e_2^{'}(d_2(p)) \; e_1(d_1(p))$$

Assume the products $\Delta d_1 \; e_1^{'}(d_1(p))$ and $\Delta d_2 \; e_2^{'}(d_2(p))$ are both positive. Then, since e_1 and e_2 are positive and monotonous, $\frac{\partial}{\partial t}\rho > 0$ and hence $\rho(p_1) < \rho(p_2)$ in contradiction to $\rho(p_1) = \rho(p_2) = c$.

With a similar argument $\Delta d_1 \; e_1^{'}(d_1(p))$ and $\Delta d_2 \; e_2^{'}(d_2(p))$ are not both negative and hence have opposite sign.

We obtain

$$\begin{aligned} \frac{\partial^2}{\partial^2 t}\rho(p) &= (\Delta d_1)^2 \underbrace{e_1^{''}(d_1(p)) \; e_2(d_2(p))}_{\leq 0} + 2 \underbrace{\Delta d_1 \; e_1^{'}(d_1(p)) \; \Delta d_2 \; e_2^{'}(d_2(p))}_{\leq 0} \\ &\quad + (\Delta d_2)^2 \underbrace{e_1(d_1(p)) \; e_2^{''}(d_2(p))}_{\leq 0} \\ &\leq 0 \end{aligned}$$

This proofs the claim. □

Theorem A.13 *A quasi-convolutionally smoothed intersection of two half-spaces $H_1 \cap H_2$ is convex.*

Proof: The quasi-convolutionally smoothed intersection of two half-spaces H_1 and H_2 is defined as $H_1 \cap H_2 = \{p \in \mathbb{R}^3 \mid \rho(p) \geq 0.5\}$, where $\rho = \rho_{H_1}\rho_{H_2}$, and ρ_{H_1} and ρ_{H_2} are the density fields obtained by convolutionally smoothing the half-spaces H_1 and H_2 with spherical filters of radius r_1 and r_2, respectively.

In section 5.3 we explained that the curved surface of the quasi-convolutionally smoothed object $H_1 \cap H_2$ is restricted to the region bounded by the planes of the half-spaces, h_1 and h_2, and the planes displaced by the negative rounding radii, $h_{1,-r_1}$ and $h_{2,-r_2}$. (see figure 5.2).

Recall section 2.2 and note that the density fields ρ_{H_1} and ρ_{H_2} are defined by two plane spread functions e_1 and e_2, respectively, where

$$e_1(d) = \begin{cases} 0 & \alpha \geq 1 \\ 1 & \alpha \leq -1 \\ (1-\alpha)^2 * (2+\alpha)/4 & \text{otherwise} \end{cases}$$

$\alpha = \frac{d}{r_1}$, and r_1 is the radius of the spherical smoothing filter. The plane spread function e_2 is defined similarly.

It is easy to check that e_1 and e_2 are positive , monotone decreasing, and concave on the intervals $(-r_1, 0)$ and $(-r_2, 0)$, respectively. The claim follows with theorem A.12 $\square$

APPENDIX B

Data Types & Library Functions

B.1 Data Types

This section summarizes all data types presented in this thesis. For each data type a reference is given to the page where the data type is first used. A motivation and explanation of the data type is usually found on the corresponding page.

```
::  BSPTree // (page 25)
      = BSPNode Plane [Face] BSPTree BSPTree
      | BSPLeaf LeafClass

::  CSGObject // (page 16)
      = Union CSGObject CSGObject
      | Intersection CSGObject CSGObject
      | SetDifference CSGObject CSGObject
      | Rounded Radius CSGObject
      | Primitive PolyhedralPrimitive

::  DBSPTree // (page 81)
      = DBSPNode Plane DBSPTree DBSPTree
      | DBSPLeaf DensityClass

::  DBSPTree // extended with polygon edges (page 91)
      = DBSPNode Plane DBSPTree DBSPTree
      | DBSPLeaf PolygonEdges DensityClass

::  DBSPTree // extended with local density field (page 107)
      = DBSPNode Plane DBSPTree DBSPTree
      | DBSPLeaf PolygonEdges DensityClass DensityField
```

```
:: DCSGObject // (page 81)
      = DUnion DCSGObject DCSGObject
      | DIntersection DCSGObject DCSGObject
      | DSetDifference DCSGObject DCSGObject
      | DPrimitive Polyhedron DensityClass

:: DCSGObject // extended with density assembly (page 117)
      = DUnion DCSGObject DCSGObject
      | DIntersection DCSGObject DCSGObject
      | DSetDifference DCSGObject DCSGObject
      | DPrimitive Polyhedron DensityClass
      | DAssembly [Face] [DCSGObject]

:: DensityClass = Zero | Low | Unclassified | High | One // (page 81)

:: DensityField // (page 17)
   = Sum DensityField DensityField
   | Product DensityField DensityField
   | Difference DensityField DensityField
   | DensityFieldOfHalfSpace Radius HalfSpace

:: Edge :== (Point,Point) // (page 95)

:: Face :== Polygon // (page 17)

:: HalfSpace :== Plane // (page 16)

:: LeafClass = IN | OUT // (page 25)

:: Plane :== (Vector,REAL) // (page 16)

:: Polygon :== [Point] // (page 17)

:: PolygonEdges = Edges [Edge] | NoEdges // (page 91)

:: PolyhedralPrimitive = Intersection [HalfSpace]
           // External (model) representation (page 16)

:: PolyhedralPrimitive = Polyhedron [Face]
           // Internal representation (page 17)

:: Radius :== REAL // (page 16)
```

```
::  RPolyhedron = RPolyh [Face] [Face] // (page 42)

::  Scene :== CSGObject // (page 16)
```

B.2 Library Functions

This section provides a list of library functions, which are used in this thesis and which in our opinion have no equivalent in an imperative language implementation. It is our hope that this section provides a useful reference for the reader inexperienced with functional languages. Functions starting with a lower case letter are taken from standard CLEAN libraries, whereas functions starting with an upper case letter are defined by us.

```
// Takes a list and a predicate on the list elements and splits the list in
// front of the first element, which does not fulfill the given predicate.
//
DecomposeWhile ::  (a -> Bool) [] -> ([a],[a])
DecomposeWhile predicate list=:[head:tail]
   | predicate head = ([head:tail_allTrue],tail_firstFalse)
   = ([],list)
where
   (tail_allTrue,tail_firstFalse) = DecomposeWhile predicate tail
DecomposeWhile predicate [] = ([],[])

// Takes a list of sublists and concatenates the elements of the sublists
// to a single list.
//
flatten ::  [[a]] -> [a]
flatten [head:tail] = head ++ flatten tail
flatten [] = []

// Takes as input a binary operator, an element, and a list and yields:
// foldl ⊗ a [b1, b2, ..., bn] = (...((a ⊗ b1) ⊗ b2)... ⊗ bn).
//
foldl ::  (a -> (b -> a)) a [b] -> a
foldl op a [b:bs] = foldl op (op a b) bs
foldl op a [] = a

// Takes as input a unary function and a list of elements and applies the
// function to each element of the list.
```

```
//
map :: (a -> b) [a] -> [b]
map f [a:x] = [f a : map f x]
map f [] = []

// Takes as input a predicate and a list and splits the list into two sublists
// the first of which contains all elements fulfilling the predicate and
// the second of which contains all elements not fulfilling the predicate.
//
SplitListWith :: (a -> Bool) [a] -> ([a],[a])
SplitListWith predicate list=:[head:tail]
   | predicate head = ([head:tail_allTrue],tail_allFalse)
   = (tail_allTrue,[head:tail_allFalse])
where
   (tail_allTrue,tail_allFalse) = SplitListWith predicate tail
SplitListWith predicate [] = ([], [])

// Transforms a list of tuples into a tuple of lists. The first and the
// second list contain the first and second elements, respectively, of
// the tuples of the original list.
//
unzip :: [(a,b)] -> ([a],[b])
unzip [(a,b):ab_tuples] = ([a:as],[b:bs])
where
   (as,bs) = unzip ab_tuples
unzip [] = []

// Takes as input a function returning a tuple and a list. The function
// is applied to each element of the list resulting in a list of tuples,
// which is transformed into a tuple of two lists containing the first and
// second result, respectively, of each function application.
//
UnZipWith :: (a -> (b,c)) [a] -> ([b],[c])
UnZipWith f = unzip o (map f)
```

APPENDIX C

Glossary

analytic set A set is analytic, if it can be expressed as $\{p|F(p) > 0\}$, where F is an analytic function.

analytic function A function is analytic, if it can be expanded into a convergent power series at every point in its domain.

aspect ratio The aspect ratio of a triangle (simplex) is the ratio of the radius of a circumscribed circle (sphere) to the radius of an inscribed circle (sphere).

augmented BSP tree A common means of augmenting the generic BSP tree is to include other sets within the BSP structure. In particular, leaves can each include a collection of sets (objects) contained completely within the corresponding cell [SBGS69], and similarly, internal nodes can include sets lying in the corresponding sub-hyperplane (e.g., [FKB80]).

average squared prediction error Let $y_1, \ldots, y_n$ be a data set of n points and

$$\hat{y} = \alpha + \beta f(x)$$

be a curve with intercept α and slope factor β.

Then the average squared prediction error e_{sq} is defined as

$$e_{sq} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

characteristic function The characteristic function of a set A, λ_A , is one for each point inside the set and zero for each point outside the set.

$$\lambda_A(x) = \begin{cases} 1 & x \in A \\ 0 & otherwise \end{cases}$$

closed regular A set X is closed regular, if it is equal to the closure of its interior, i.e., $X = \overline{int(X)}$.

homeomorphic Let $S, T \in \mathbb{R}^n$ and $\phi : S \to T$ be a mapping. If the map $\phi : S \to T$ is a bijection, and both ϕ and ϕ^{-1} are continuous, ϕ is called homeomorphism and S and T are said to be homeomorphic.

honeycomb A honeycomb is the 3D analog of a tessellation. It is a polyhedral partition of space in which the face of each polyhedron is adjacent to only one other polyhedral face.

least squares Let $y_1, \ldots, y_n$ be a data set of n points and

$$\hat{y} = \alpha + \beta f(x)$$

be a curve with intercept α and slope factor β.

Under the least squares criterion the best fitting curve is the one with the smallest *average squared prediction error* e_{sq}.

Lebesgue measure The Lebesgue measure for $\mathbb{R}^n$, Λ^n on $\mathcal{L}^n$, is the measure induced by the Lebesgue integral [KS88]. In this thesis we only deal with 'well-behaved' sets and hence may say that the Lebesgue measure of a set is equal to the Riemann-Stieltjes integral of its characteristic function.

Lipschitz constant A (positive) real number $\mathcal{L}$ is called a Lipschitz constant on a function $F(x)$ in a region $\mathcal{R}$, if given any two points x_1 and x_2 in $\mathcal{R}$, the following condition holds:

$$\|f(x_1) - f(x_2)\| < \mathcal{L}\|x_1 - x_2\|$$

where $\|.\|$ is a vector norm.

manifold A set M is a n-manifold, if for every point $x \in M$ there exists an open neighborhood $B(x)$ such that $B(x) \cap M$ is homeomorphic to an open subset of $\mathbb{R}^n$.

r-set A bounded, closed regular, semi-analytic set.

semi-analytic set A set is semi-analytic, if it can be expressed as a finite Boolean combination of analytic sets.

tessellation A tessellation is a tiling of a surface, in which polygons intersect each other edge to edge or vertex to vertex.

zero set A set of Lebesgue measure zero.

APPENDIX D

Color Images

The following pages contain color images of the example scenes introduced in section 2.5. Figure **??** was produced by a RasterOps CorrectPrint 330 dye-sublimation printer and the other images were obtained with a Tektronix Phaser 550 color laser printer.

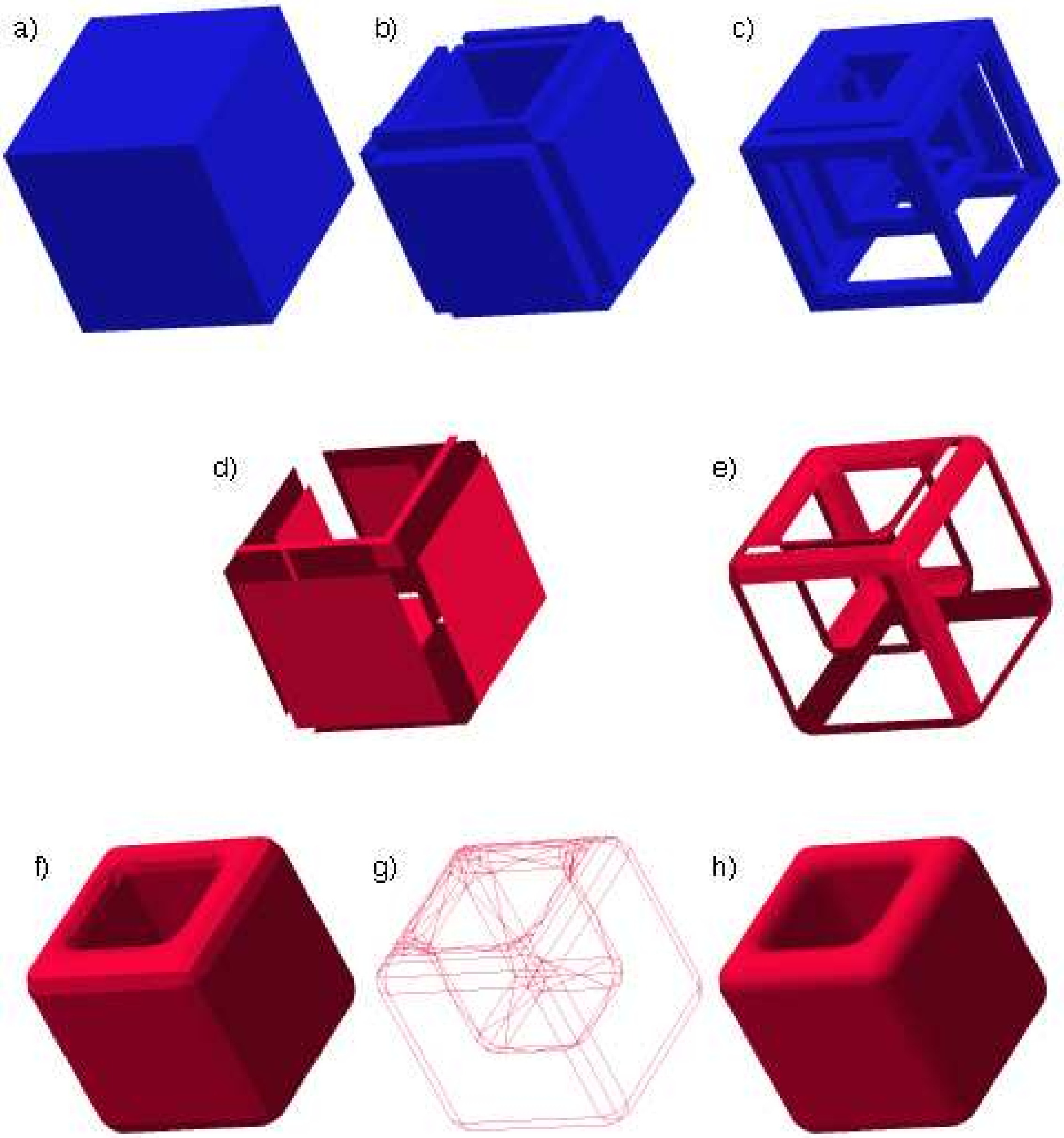

Figure D.1. *The three steps of Triage Polygonization: first the polyhedral subdivision partitions a density field into low cells (a), high cells (b), and unclassified cells (c). The second step extracts tree polygons (d), which separate low cells and high cells. Finally subspace polygons are obtained by applying a subspace polygonization to the unclassified cells in (c). The final polygonization is shown flat shaded in (f), as a wire-frame representation with removed back-faces in (g), and Gouraud shaded in (h).*

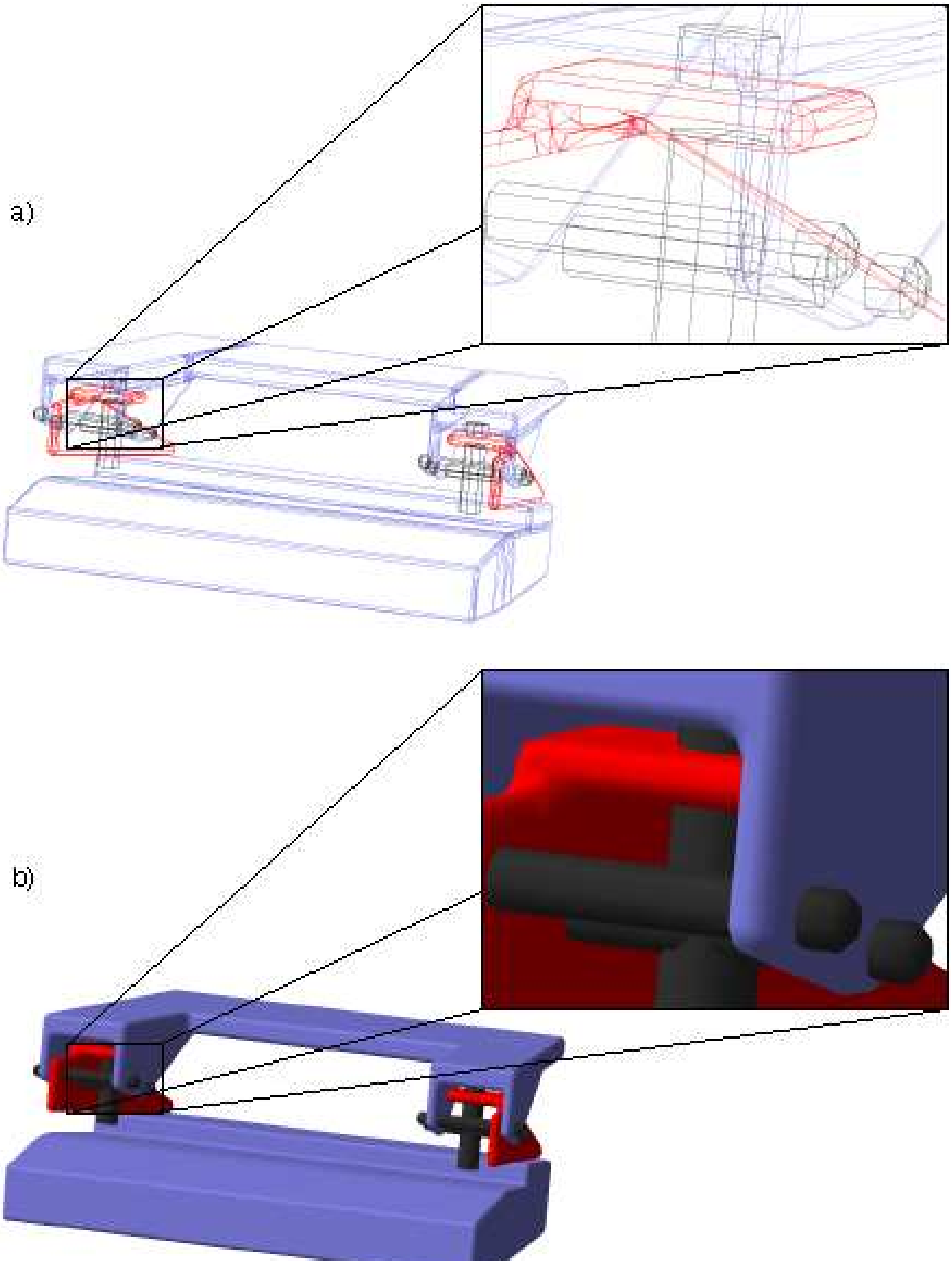

Figure D.2. *The "Hole Punch" scene polygonized with Triage Polygonization. The result is shown as a wire-frame representation (a) and as Gouraud shaded polygons (b).*

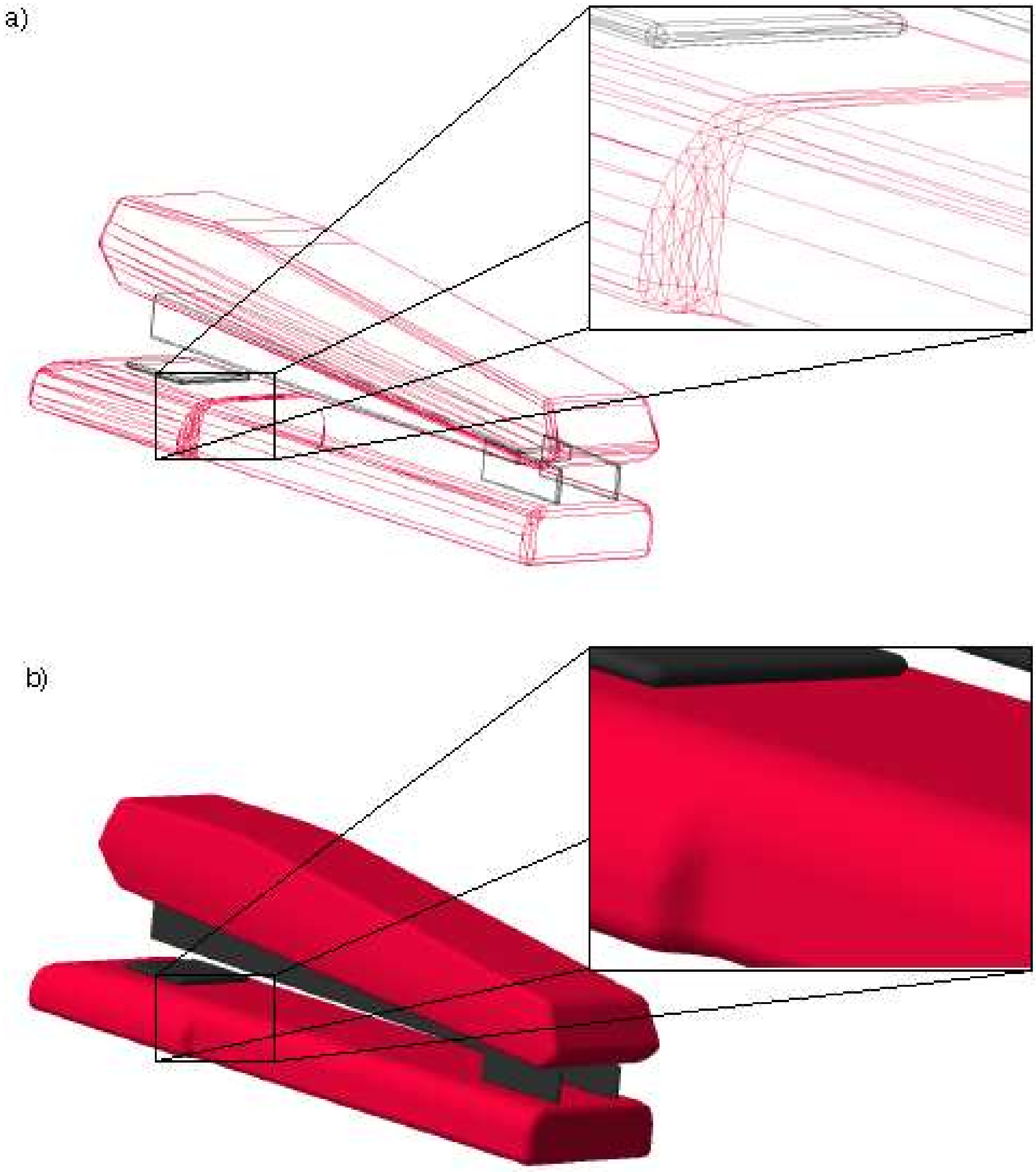

Figure D.3. *The "Stapler" scene polygonized with Triage Polygonization. The result is shown as a wire-frame representation (a) and as Gouraud shaded polygons (b).*

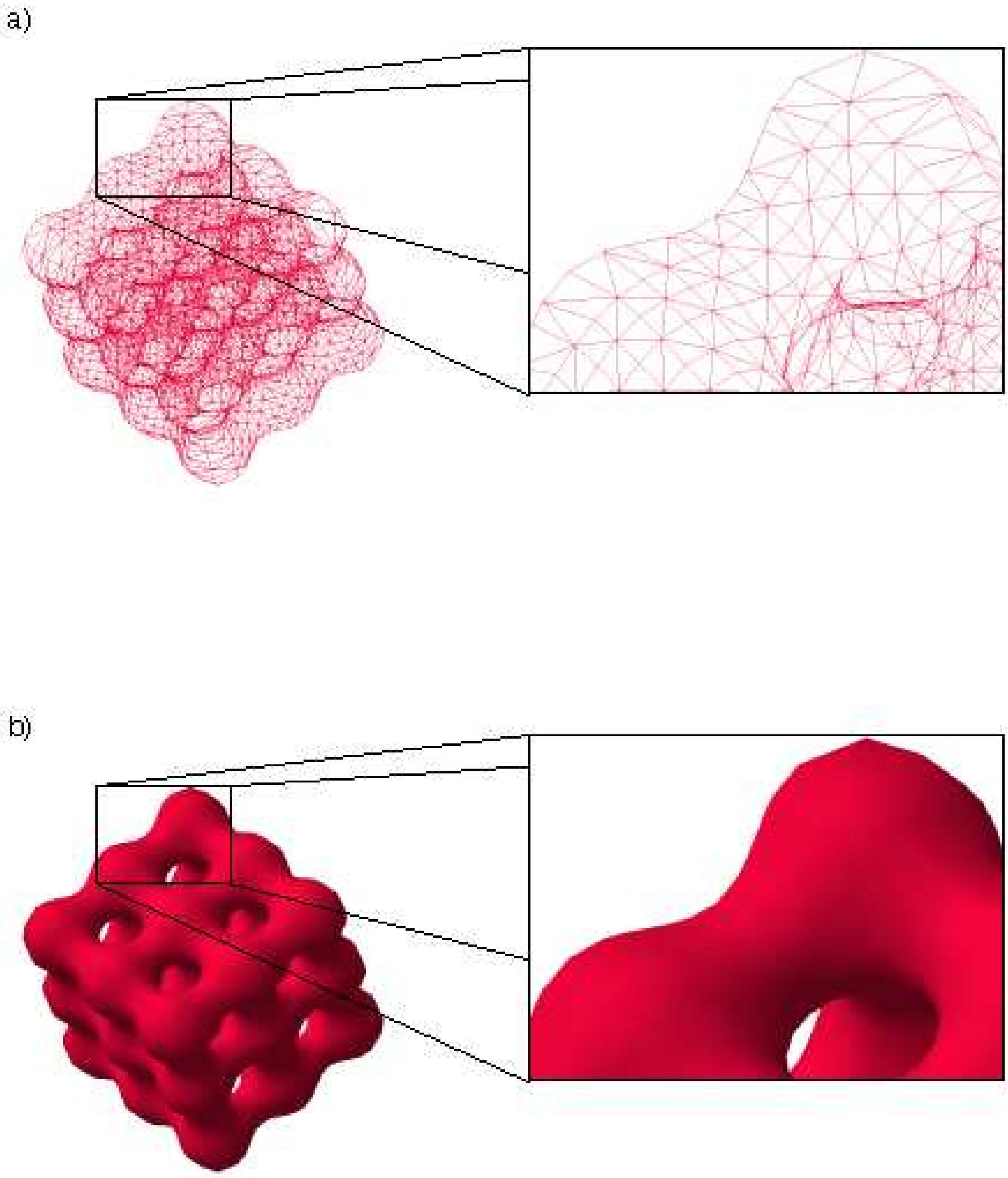

Figure D.4. *The "27 Blended Cubes" scene polygonized with Triage Polygonization. The result is shown as a wire-frame representation (a) and as Gouraud shaded polygons (b).*

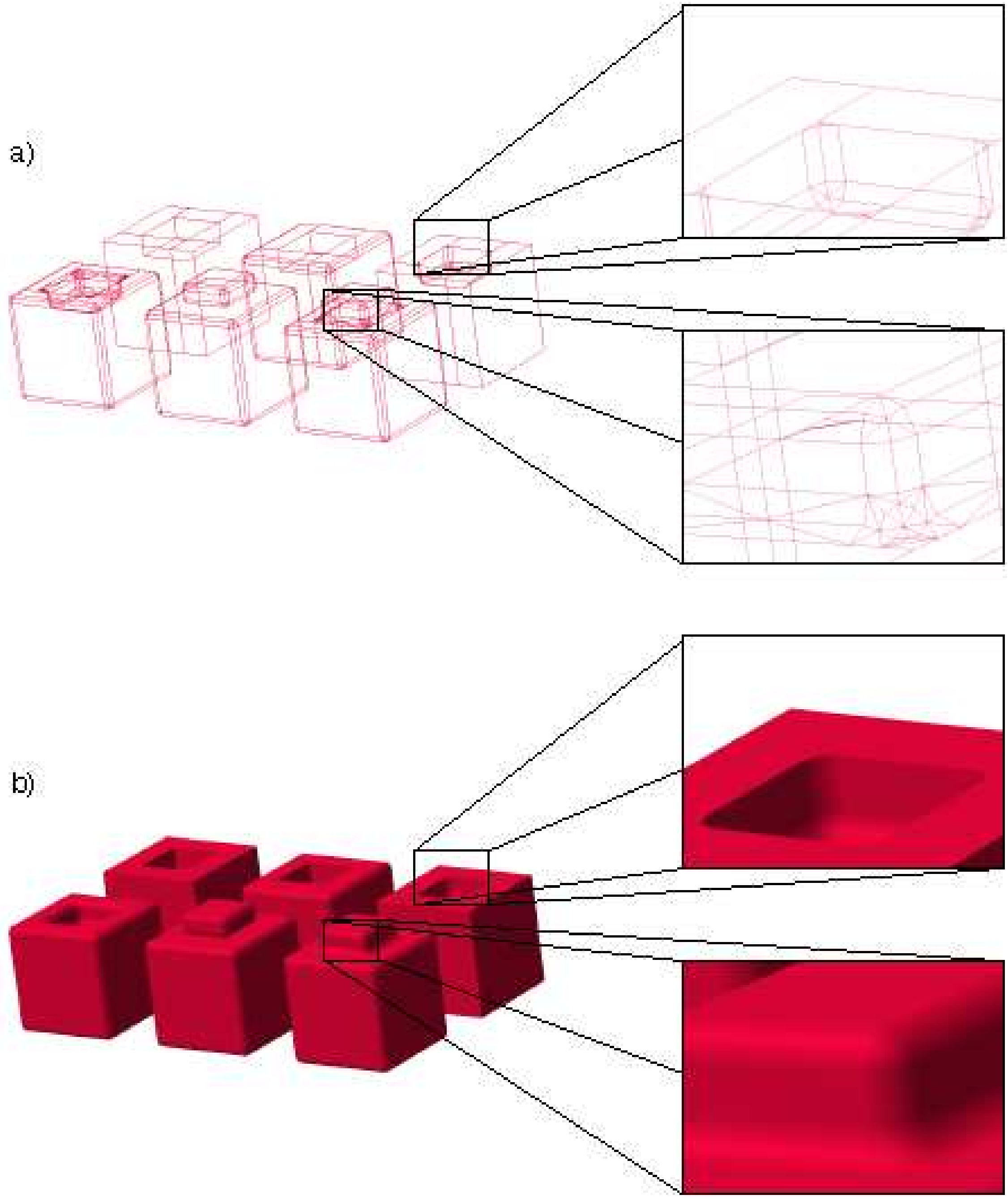

Figure D.5. *The "CSG Example" scene polygonized with Triage Polygonization. The result is shown as a wire-frame representation (a) and as Gouraud shaded polygons (b).*

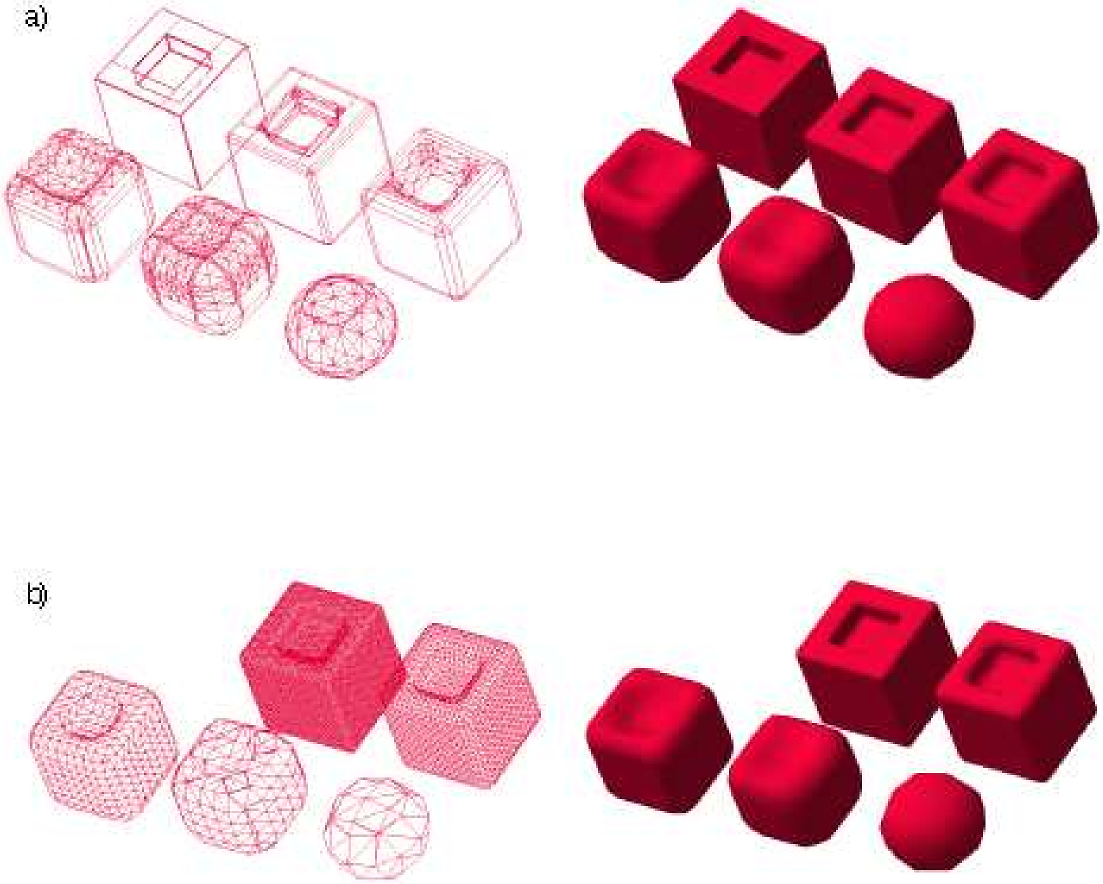

Figure D.6. *Triage Polygonization and the Marching Cubes algorithm applied to the "Variable Radius" scene. Part (a) shows the result of Triage Polygonization as a wire-frame representation (left) and as Gouraud shaded polygons (right). Part (b) shows the same representations for the result of the Marching Cubes algorithm.*

Bibliography

[AG87] Eugene L. Allgower and Stefan Gnutzmann. An algorithm for piecewise linear approximation of implicitly defined two-dimensional surfaces. *SIAM Journal of Numerical Analysis*, 24(2):452 – 469, April 1987.

[All90] R. E. Allen, editor. *The concise Oxford dictionary of current English*. Clarendon Press, Oxford, 8 edition, 1990.

[Aur91] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345 – 405, September 1991.

[Bak89] H. Harlyn Baker. Building surfaces of evolution: The weaving wall. *International Journal of Computer Vision*, 3(1):51 – 71, May 1989.

[BBK78] R. E. Barnhill, J. H. Brown, and I. M. Klucewicz. A new twist in CAGD. *Computer Graphics and Image Processing*, 8:78 – 79, 1978.

[Bee86] Etienne Beeker. Smoothing of shapes designed with free-form surfaces. *Computer-Aided Design*, 18(4):224 – 231, May 1986.

[Bli82] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235 – 256, July 1982.

[Blo88] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer-Aided Geometric Design*, 5(4):341 – 355, November 1988.

[Bow81] A. Bowyer. Computing dirichlet tesselations. *The Computer Journal*, 24(2):162 – 166, May 1981.

[BS91] Jules Bloomenthal and Ken Shoemaker. Convolution surfaces. *Computer Graphics*, 25(4):251 – 256, July 1991.

[BW90] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. *Computer Graphics*, 24(2):109 – 116, March 1990. Special Issue on 1990 Symposium on Interactive 3D Graphics.

[CC78] Edwin Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350 – 355, November 1978.

[Chi87] Hiroaki Chiyokura. An extended rounding operation for modeling solids with free-form surfaces. *IEEE Computer Graphic and Applications*, 7(12):27 – 36, December 1987.

[Col90] Steve Colburn. Solid modeling with global blending for machining dies and patterns. SAE technical paper series, SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A., April 1990. 41st Annual Earthmoving Industry Conference.

[CTKH91] Hiroaki Chiyokura, Teiji Takamura, Koichi Konno, and Tsuyoshi Harada. G^1 surface interpolation over irregular meshes with rational curves. In Gerald Farin, editor, *NURBS for Curve and Surface Design*, chapter 2, pages 15 – 34. SIAM Activity Group on Geometric Design, 1991.

[DK91] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans. Commun. Elec. Inf. Syst.*, E-74(1):214 – 224, January 1991.

[Duf92] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics*, 26(2):131 – 138, July 1992.

[Düu88] Martin J. Düurst. Additional reference to "marching cubes". *Computer Graphics*, 22(2):72, April 1988. Letter.

[Fjä86] Per-Olof Fjällström. Smoothing of polyhedral models. In *Proceedings of the ACM 2nd Symposium on Computational Geometry, Yorktown Heights*, pages 226 – 235, 1986.

[FKB80] H. Fuchs, Z. Kedem, and B.Naylor. On visible surface generation by a priority tree structure. *Computer Graphics*, 14(3):124–268, June 1980.

[Heu81] Harro Heuser. *Lehrbuch der Analysis*, volume 2. B.G. Teubner, 1981.

[HH87] Christoph Hoffmann and John Hopcroft. The potential method for blending surfaces and corners. In Gerald E. Farin, editor, *Geometric Modelling: Algorithms and New Trends*, pages 347 – 365, Philadelphia, 1987. Society for Industrial and Applied Mathematics.

[HH91] Josef Hoschek and Erich Hartmann. G^{n-1}-functional splines for modeling. In Hans Hagen and D. Roller, editors, *Geometric Modeling - Methods and Applications*, pages 185 – 211. Springer Verlag, 1991. Based on lectures presented at an international workshop held in Böblingen, Germany in June 1990.

[Hil65] Einar Hille. *Analysis*, volume 2. Blaisdale Publishing Company, second edition, 1965.

[HJP+92] P. Hudak, S. Peyton Jones, P. Wadler P, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, and W. Partain J. Peterson. Report on the programming language Haskall. *ACM SigPlan Notices*, 27(5):1–164, 1992.

[HW90] Mark Hall and Joe Warren. Adaptive polygonization of implicitly defined surfaces. *IEEE Computer Graphic and Applications*, 10(5):33 – 42, November 1990.

[Kal91] A. D. Kalvin. *Segmentation and surface-based modeling of objects in three-dimensional biomedical images.* PhD thesis, New York University, New York, 1991.

[KB89] Devendra Kalra and Alan H. Barr. Guaranteed ray intersections with implicit surfaces. *Computer Graphics*, 23(3):297 – 306, July 1989. Proceedings of SIGGRAPH '89, Boston.

[KDK86] A. Koide, A. Doi, and K. Kajioka. Polyhedral approximation approach to molecular orbit graphics. *J. Molec. Graph.*, 4:149 – 156, 1986.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3 - Sorting and Searching. Addison-Wesley Publication Company Inc, 1973.

[Kop91a] Pramod Koparkar. Designing parametric blends: surface model and geometric correspondence. *Visual Computer*, 7:39 – 58, 1991.

[Kop91b] Pramod Koparkar. Parametric blending using fanout surfaces. In *Proc. Symp. on Solid Modelling Foundations and CAD/CAM Applications (Austin, Texas, June 5-7 1991)*, 1991.

[Kos91] Menno Kosters. An extension of the potential method to higher-order blendings. In *Proc. Symp. on Solid Modelling Foundations and CAD/CAM Applications (Austin, Texas, June 5-7 1991)*, 1991.

[KS88] J. L. Kelly and T. P. Srinivasan. *Measure and Integral.* Graduate Texts in Mathematics 116. Springer Verlag, 1988.

[LC87] W. Lorenson and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163 – 169, July 1987. Proceedings of SIGGRAPH.

[Lob95] Richard Lobb. Quasi-convolutional smoothing of polyhedra. Technical report, University of Auckland, 1995.

[LVG80] S. Lobregt, P. W. Verbeek, and F. C. A. Groen. Three-dimensional skeletonization: Principle and algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(1):75 – 77, January 1980.

[Man95] Ubi Manber. *Introduction to Algorithm - A creative Approach.* Addison-Wesley Publication Company Inc, 1995.

[MS85] Alan E. Middleditch and Kenneth H. Sears. Blend surfaces for set theoretic volume modelling systems. *Computer Graphics*, 19(3):161 – 170, July 1985. Proceedings of SIGGRAPH '85, San Francisco.

[Mur91] Shigeru Muraki. Volumetric shape description of range data using "blobby model". *Computer Graphics*, 25(4):227 –235, July 1991. Proceedings of SIGGRAPH '91.

[NAT90] Bruce F. Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. *Computer Graphics*, 24(4):115 – 124, August 1990.

[Nay81] Bruce F. Naylor. *A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes.* PhD thesis, University of Texas, Dallas, Texas, May 1981.

[PPW87] Carl S. Petersen, Bruce R. Piper, and Andrew J. Worsey. Adaptive contouring of a trivariate interpolant. In Gerald E. Farin, editor, *Geometric Modelling: Algorithms and New Trends*, pages 385 – 395, Philadelphia, 1987. Society for Industrial and Applied Mathematics.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley Publication Company Inc, 1993.

[PvE95] R. Plasmeijer and M. van Eekelen. *Concurrent Clean Language Report, Version 1.0.* University of Nijmegen, April 1995.

[PVTF92] William H. Press, William T. Vetterling, Saul A. Teukolsky, and Brian P. Flannery. *Numerical Recipes in C - The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[Req80] Aristides A. G. Requicha. Representation for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437 – 464, December 1980.

[RO87] Alyn P. Rockwood and John C. Owen. Blending surfaces in solid modeling. In Gerald E. Farin, editor, *Geometric Modelling: Algorithms and New Trends*, pages 367 – 382, Philadelphia, 1987. Society for Industrial and Applied Mathematics.

[RR84] Jaroslaw R. Rossignac and Aristides A. G. Requicha. Constant-radius blending in solid modeling. *Computers in Mechanical Engineering*, 3(1):65 – 73, July 1984.

[RV85] Aristides A. G. Requicha and Herbert B. Voelcker. Boolean operations in solid modelling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1):30 – 44, January 1985.

[SBGS69] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, U.S.Air Force Human Resource Laboratory, 1969.

[Sed85] T. Sederberg. Piecewise algebraic surface patches. *Computer-Aided Geometric Design*, 2(1):53 – 59, 1985.

[Sny92] John M. Snyder. Interval analysis for computer graphics. *Computer Graphics*, 26(2):121 – 130, 1992.

[SSS74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden surface algorithms. *ACM Computing Surveys*, 6(1), 1974.

[TN87] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, 21(4):153 – 162, July 1987. SIGGRAPH /32487 Proceedings.

[TR80] R. B. Tilove and Aristides A. G. Requicha. Closure of boolean operations on geometric entities. *Computer-Aided Design*, 12(5):219 – 220, September 1980.

[Tur85] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Proceedings Conference on Functional Programming Languages and Computer Architecture (Nancy, France) LCNS 201*, pages 1 – 16, Berlin, 1985. Springer-Verlag.

[vGW94] Allen van Gelder and Jane Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337 – 375, October 1994.

[vHB87] B. von Herzen and A. Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics*, 21(4):103 – 110, July 1987. Proceedings of SIGGRAPH.

[War89] Joe Warren. Blending algebraic surfaces. *ACM Transactions on Graphics*, 8(4):263 – 278, October 1989.

[WMW86a] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Animating soft objects. *Visual Computer*, 2(4):235 – 242, August 1986.

[WMW86b] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *Visual Computer*, 2(4):227 – 234, August 1986.

[Woo87] J. R. Woodwark. Blends in geometric modelling. In R. R . Martin, editor, *The Mathematics of Surfaces 2*, The Institute of Mathematics and its Applications Conference Series, pages 255 – 297. Clarendon Press, Oxford, 1987.

[WWM87] Geoff Wyvill, Brian Wyvill, and Craig McPheeters. Solid texturing of soft objects. *IEEE Computer Graphic and Applications*, 7(12):20 – 26, December 1987.