Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Department of Informatics

Bachelor Thesis

An Edge-Based Approach to Changeable 2D Landscapes for Computer Games

Tobias Zwick

2011-03-23

Supervisors
Dr. Werner Hansmann
Prof. Dr. Leonie Dreschler-Fischer

Preface

On the 23rd of May 2009, the source code of the *Clonk* game engine was officially released under the ISC license (a permissive open source license) after fifteen years of being distributed as shareware. A group of hobby programmers and enthusiasts in which I was part of founded the *OpenClonk*[1] project soon after the game engine was made open source. The project was dedicated to the further development of the game. That the game engine was made open source made it possible to completely overhaul the game, and try out new technologies.

Inspired by various physics sandboxes like *Phun* or *Crayon Physics*, the idea to make use of an open source physics engine for the game came up at one time. However, we quickly had to find out that it is very difficult to get this to work because the landscape of *Clonk* is represented as a two-dimensional array, like a raster graphic. Physics engines however, work only with geometric shapes like lines, circles, rectangles and polygons. Additionally, as the landscape of *Clonk* is basically a raster graphic, it looks all pixelated when zoomed in. Now, there is a very good reason for Clonk to have this kind of landscape representation: half of the gameplay revolves around digging through the landscape and mining for resources. It is essential for the gameplay that the landscape is changeable and specifically, destructible. A pixel-based landscape enables an easy and fast modification and destruction of the landscape through simple modification of the single pixels in the array, like in a graphics painting program.

Game landscapes based on polygons had been done before. What hadn't been done before was a landscape representation based on polygons that was also creatable and changeable like a raster graphic: paint and erase the polygon-based landscape with a brush tool, not just create and change the landscape by (re)defining the vertices of the polygons. So the initial motivation to investigate whether or not an implementation of a changeable landscape based on polygons is feasible or not came from my involvement in the development of this game.

However, after some tests, it quickly turned out that the implementation of a landscape with these properties was anything but trivial. Also, I noticed that much work has been done on the field of polygon clipping (the operation with which polygons are added or subtracted from each other) and related areas in science already. Since the subject of reliable clipping operations for continuous use (which is required for games with changeable landscapes) has not been covered yet, I decided to discuss this subject properly in a thesis.

At this point I want to thank the people who supported me by pointing out mistakes, missing or ambiguous explanations in the thesis or helping me to debug the implementation and find errors: Dr. Werner Hansmann, one of my supervisors, my fellow student and flatmate Niels Beuck, my mother (for trying) and also the people from the OpenClonk project, especially Armin Burgmeier, Peter Wortmann, Charles Spurril, Günther Brammer, Sven Eberhardt and Matthias Rottländer.

Except for the screenshots in figures 1-4 which are taken from various games, all figures in this work are created by myself (in *Inkscape)*.

1 The project's website can be found at http://www.openclonk.org/

Abstract

There are two main approaches for the representation of a landscape in two-dimensional computer games: *Interior-based* landscapes and *edge-based* landscapes. Modern games with an edge-based landscape enable, amongst other things, the use of advanced physics engines. However, no game is known to the author that features an edge-based landscape which is at the same time completely *changeable* (*destructible*).

This thesis offers an approach for an implementation of a two-dimensional landscape based on polygons which is dynamically changeable in real-time. This includes a description of algorithms that are utilized for typical operations in an edge-based game landscape like collision detection or containment checks. The main focus of this thesis, however, lies on the proposal of a *polygon-clipping* algorithm that is eligible for continuous usage in the game and enables a polygon-based landscape to be completely changeable. Finally, efficient spacial data structures are discussed that can be used to store this kind of landscape and make queries on the landscape perform in reasonable time.

Table of contents

1 Introduction

There are different approaches for how the *game area* or *landscape* of computer games that take place in a two dimensional game world is represented. Subsequently, two basic approaches are described. Interior-based approaches define a (solid) area on the basis of its interior, while edge-based approaches define that area only by a border which encloses the area.

The usual approach for the popular genre of platformers (also called jump 'n' runs) and other side-scrollers[2] is to use a tile-based one. Some of the first popular video games that used this technique were Super Mario Bros. (1985), Commander Keen (1990) and Sonic the Hedgehog (1991).

The landscape of many modern 2D platformers is still based on tiles, sometimes however only to reuse graphics and make level design easier. A landscape that is represented in tiles (*tile-landscape*) consists of a two-dimensional array in which each entry references a predefined tile type. Each tile type defines a graphic and a solid area which is usually a simple geometric shape like a box or a slope, but is not limited to that. Collision checks between moving objects and the landscape in these games are simple and fast because normally only the one tile the object is in needs to be looked at.

Figure illustrates how tiles work.

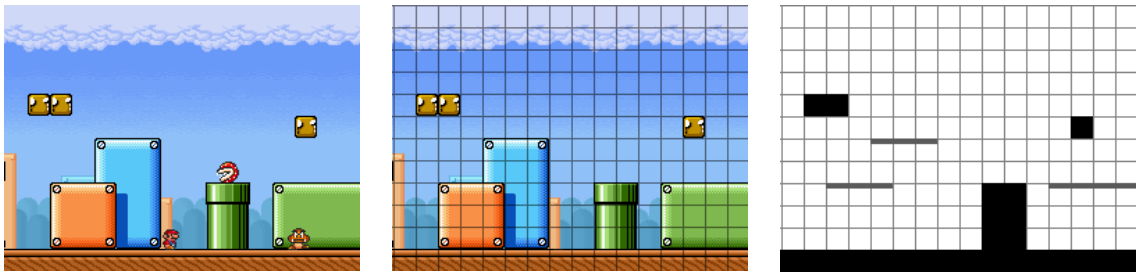

Figure 1: This figure illustrates how tiles work on the basis of a screenshot from Super Mario Bros. 3.

The left image shows a screenshot from the game. If the landscape is overlayed with a grid and the characters removed (center image), one can see that the landscape is actually composed of square tiles of equal size with the same graphics for the same tile type. The right image shows the solid areas in the tile-based landscape. Some tile types in this game are solid (depicted as black blocks), others are platforms on which it is possible to jump from below (depicted as thick lines). For each tile type any number of unique properties such as geometric shape, animations, damage when touched etc. can be defined but each tile can only have one property – its tile type.

The first computer games that featured fully destructible 2D landscapes were the (usually turn-based) *artillery* games in which two or more players would shoot at each other in trajectories. Popular games in this minor genre included *Scorched Earth* (1991), the *Worms* series (1995-present) plus derivatives and *GunBound* (2005). Other games not based on the *artillery* genre that take advantage of a fully destructible (and changeable) 2D landscape include *Lemmings* (1991) and the *Clonk* series (1994-present). Figure 2 shows two examples for games with pixel-based landscapes.

2 Side-scroller is the term originally used for video games that are seen from the side and whose landscape is too extensive to fit on one screen so that the camera needs to follow the gameplay action. The most popular genre that uses this format is that of platformers but it is also used in other genres such as *beat em' ups* or (2D) *shooters*.

Figure 2: On the left: A phone port of Worms (2009), on the right: Clonk Rage (2008)

This kind of game landscape has too been implemented as a two-dimensional array in any of the mentioned games. Basically, a *pixel-landscape* is a *tile-landscape* with a tile size of 1x1 pixels where the solid pixels do not define any other geometric shape than a box. On the other hand, the landscape as a whole can have any shape and is not limited to the predefined tile shapes. Each pixel in that landscape can be changed in the game like in a raster graphic which is the prime advantage of this representation. Of course, this also means that this approach takes up much more memory than the standard tile-based approach. For example, a pixel-landscape that covers a game area of the same size as that of a tile-landscape with tiles that have a size of 20x20 pixels would need to store 400 times more tiles than the tile-landscape.

In this approach, every pixel in the landscape references a certain tile type too, in this context however it should be called *material*.

The access to single areas of the landscape for simple collision checks is slower because of the increased resolution but still reasonably simple as it is also based on a two-dimensional array.

A completely different approach to the representation of the landscape is the *edge-based* approach. In this approach, the solid landscape is represented by line segments or polygons. Simple things like walls and platforms which are essential for platformers can be represented by lines. More complex shapes that could otherwise only be represented in a pixel-landscape can be represented as polygons (a *polygon-landscape*).

Edge-based landscapes only emerged recently and are used mostly in indie[3] or open source games where physics play an all important role in the gameplay. Examples of 2D games with advanced physics simulations are *X-Moto* (2005), *World of Goo* (2008), *LittleBigPlanet* (2008), *Trine* (2009), *PixelJunk Shooter* (2009) and *Limbo* (2010).

So, a prime reason for the use of a polygon-landscape is that freely available 2D physics engines like *ODE* (2001), *Bullet* (2003) or *Box2D* (2007) require the landscape to consist of only edge-based geometric shapes. Advanced physics simulations turned out to be very popular in games and nowadays, computers are powerful enough to perform this kind of simulation in real-time. Generally, collision checks and other simple physics are less trivial to implement and perform in reasonable time in polygon based landscapes than in tile based landscapes, however, when it comes down to more advanced physics calculations, the surface and shape of the involved parts of the landscape must be known. Of course it is possible to use (local) edge detection algorithms on an interior based landscape but this is where it gets really cumbersome and slow.

As only the border of a solid area is stored, an edge-based approach tends to be more memory

3 Games that are developed in small teams or companies with no financial support by publishers are commonly referred to as *independent games* or *indie games*. They are usually only distributed online. The games themselves are low budget and focus more on innovation rather than established game mechanics and graphics.

efficient than an interior-based approach. However, its memory usage and performance is not dependent of its actual size (in pixels) but its number of edges (its detail).

Figure 3 shows two examples for games whole landscape is based on polygons.

Figure 3: X-Moto (2005) to the left and Limbo (2010) to the right

Additionally, a tile-based landscape imposes many limitations on the design of a level. This limit is quickly reached in graphically more elaborate games. That is why many modern so-called *2.5D platformers*, 3D platform games with a two-dimensional game area, use at least[4] an edge-based approach for the solid parts of a landscape while rendering the graphics in 3D. Examples include *Klonoa: Door to Phantomile* (1998), *Viewtiful Joe* (2003), *Shadow Complex* (2009) and *Donkey Kong Country Returns* (2010). See also figure 4.

Figure 4: Left: Donkey Kong Country Returns (2010), right: Klonoa: Door to Phantomile (1998)

Currently there is no game known to the author that features a landscape which is both edge-based to enable the use of advanced physics simulation and is at the same time fully destructible. This might be the case because the implementation of a destructible (or generally: changeable) landscape on the basis of polygons is not trivial to implement, but can also be explained by considering that the development in the sector of 2D computer games did not get quite as much attention in the past decade by the big game development studios as the sector of 3D games. With that reasoning, it is no wonder that the recent innovations in 2D gameplay have been coming from independent games rather than from the big game studios.

However, it should be noted that there are two most probably edge-based games which feature at least the destruction of parts of the game area: The 2D physics sandbox *Algodoo* (2009)

4 It is difficult to find out what technique a game uses exactly. One can only guess from looking at how it works and what would be expedient in their case. In the case of 2.5D games, they could of course also technically work like a fully fleshed 3D game but limit the controls to a 2D plane.

enables to draw and erase free shapes in the game area. The game *PixelJunk Shooter* (2009) features certain destructible regions in the landscape.

So, the main goal of this thesis is to offer an approach how to implement a 2D landscape based on polygons which is also completely changeable during the course of the game and in real-time. This also includes a description of the basic operations on polygons like collision detection (intersection with a line), containment checks etc.

Changeable in this context means to be able to change the region enclosed by the polygons, similar to what is done in a pixel-based landscape: Replace portions of the landscape with a different material, remove them or add new portions of a material somewhere in the landscape. In a landscape based on polygons, these kind of operations are done with *clipping operations*. For the implementation of a clipping operation and the landscape consisting of polygons in general, some special requirements should be met:

First of all, as the operations on the landscape ought to run in real-time, the **performance** is important. Checks and clipping operations on the landscape must be bounded in their execution time, so the same operation should not for example take ten times as long on one side of the landscape as on the other side of the landscape. Since the field of application is a very specific one in this case, the operations should be **optimized** for the typical operations in a game world: It must be considered that the clipping operations are not just performed once on a set of input polygons but repeatedly or even continuously throughout the course of the game at many different locations in the landscape. Additionally, the data structure must still allow often-used checks like collision detection to be fast.

Because one must be prepared for continuous changes on the landscape, the clipping algorithm must be **robust**: The algorithm must always output polygons that it also accepts as input polygons (which is not given for all implementations). Specifically, the algorithm needs to handle the limited precision of numbers in computers without outputting invalid polygons at some point.

Another issue in the context of games is **network synchronization**. Because over the years, different floating point representations have been used in different system architectures, its usage might become a problem when synchronized across computers with different system architectures.

That is why the implementation should also be able to use only fixed point numbers for calculations. At the same time, this allows to define an **upper limit for the detail** of the polygons in the landscape. This is important because with every clipping operation on the landscape, it tends to become more and more detailed which conversely makes its access slower. As it is possible to set an upper limit for the detail of a tile-based landscape by defining the size of a tile in pixels, the same should be able to be done with a landscape based on polygons.

The rest of this thesis is structured as follows:

In chapter 2, the primitive operations on polygons which are essential for basic operations on a landscape that consists of polygons will be described in detail. However, the main focus of this thesis lies on the implementation of how to make changes on a polygon-based landscape, specifically, how to perform clipping operations on single polygons. This is described in chapter 3.

Methods and data structures to make operations on the whole landscape rather than single polygons perform in reasonable time are roughly described chapter 4. However, detailed descriptions about the implementation of this would go beyond the scope of this work.

Finally, a conclusion is drawn in chapter 5 in which the pixel-based landscape is compared with the solution presented in this work.

2 Polygon basics

2.1 Definition

A *polygon* is typically defined as a sequence of *vertices* which are connected by straight line segments, the *edges* of the polygon. All edges taken together form the *boundary* of a polygon. The *faces* of a polygon are defined by the areas which are enclosed by the boundary.

Figure 5 shows an example of a *simple polygon*. Simple polygons are polygons which are not *self-intersecting*. That is, polygons whose boundary does not intersect. Simple polygons have a clearly defined inside and outside because their boundary only defines one face.

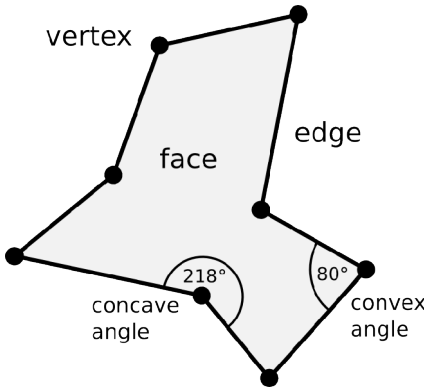

Figure 5: An example of a simple polygon

A simple polygon can be either convex or concave. If the internal angle between each two successive line segments of a polygon is less or equal than 180 degrees (is a *convex angle*), it is a *convex polygon*. Otherwise it is a *concave polygon*.

Simple polygons whose edges may intersect but not cross each other can be called *self-touching* or *weakly simple polygons*.[5] Figure 6 depicts the difference between a non-simple and a weakly simple polygon. In all other respects, weakly simple polygons share the same properties as simple polygons.

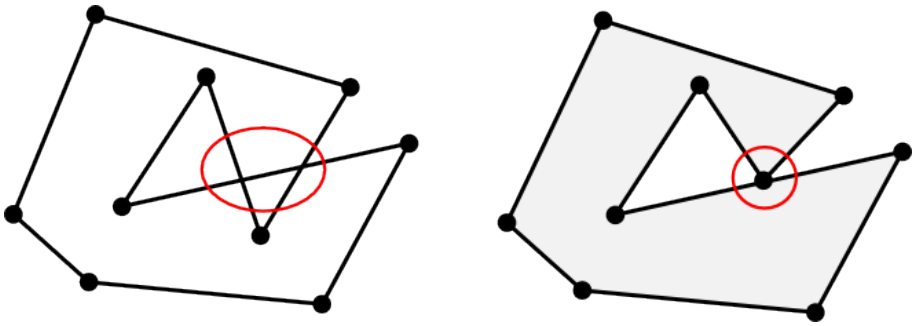

Figure 6: Both depicted polygons show self-intersections but only the edges of the left polygon cross each other. The right polygon is a weakly simple or self-touching polygon.

The algorithms presented in this work are able deal with arbitrary simple polygons (*not* weakly simple polygons). For convenience, simple polygons will just be called polygons here.

5 There are a couple of different definitions for “weakly simple polygons” in literature. That is because the term is often used to describe a type of polygon that is almost but not quite a “simple polygon”. In this work, the definition provided by D. W. Krumme et al. [K2] is used.

2.2 Size and circumference

The area of a polygon is calculated by the Gaussian trapezoid formula:

$$2A = \left| \sum_{i=0}^{n-1} \left(y_i + y_{(i+1) \bmod n} \right) \cdot \left(x_i - x_{(i+1) \bmod n} \right) \right|$$

A is the area of the polygon. (x_i, y_i) are the coordinates of the vertex at index i. The first vertex is (x_0, y_0). If the sum in above formula is not reduced to an absolute value, the result of this formula will be positive if the polygon is defined counter-clockwise and negative if the polygon is defined clockwise. So, the Gaussian trapezoid formula is an easy way to find out in which direction the polygon is defined as well.

The circumference is calculated by adding the distance between all successive vertices of the polygon.

2.3 Determining convexity

All interior angles of a convex polygon are less or equal 180°. So, given that the polygon is defined counter-clockwise, each vertex v_i of the polygon has to be left of the previous line spanned by v_{i-2} and v_{i-1}. If any of the vertices do not fulfill this condition, the algorithm can terminate and return false.

This check runs in O(n). n is the number of edges of the polygon. The pseudo-code in Figure 7 returns true if the polygon is convex.

```
for each vertex Vi in the polygon
  if Vi is right of the line spanned by Vi-2 and Vi-1
    return false
  endif
endfor
return true
```

Figure 7: Pseudo-code to determine convexity

2.4 Intersection with a line

The intersection of a line and a polygon is a set of points and lines. The set may be empty (there is no intersection) or contain any number of points and lines.

As depicted in Figure 8, the intersection with the boundary and the intersection with the face must be distinguished since it is something completely different: the points where the line *crosses* the polygon boundary mark the start- end endpoints of the intersections with the polygon face. For most operations, only the intersections of the boundary are of any interest; the intersection with the polygon face can be calculated from the polygon boundary intersection.

In the application of polygon-based landscapes, intersections with lines are performed for collision checks with objects. The line then stands for the path a moving object covered in a single time step. If this line intersects with a polygon, the objet collides with it at the intersection point.

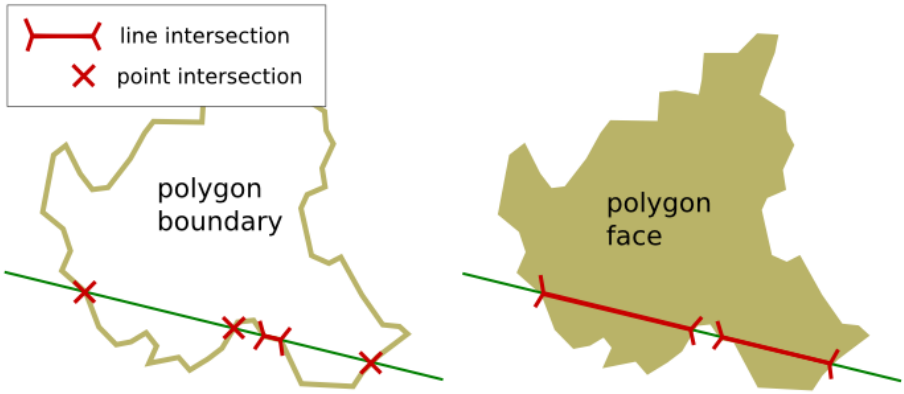

Figure 8: The difference between intersections of a line with the boundary of a polygon and the face of a polygon

The polygon boundary consists of a number of line segments. Every single one of those line segments needs to be checked for an intersection with the given line in order to determine all intersections. To avoid that the same two intersections are found twice, the line segments need to be treated as open to one end (that is, one end point is not included in it).

For a single polygon, this check runs in linear time O(n) while n is the number of edges of the polygon and thus intersection checks. See Figure 9 for the pseudo-code.

```
for each edge in the polygon
  if edge and line intersect
    if intersection is not at start-point of edge
      memorize intersection
    endif
  endif
endfor
return memorized intersections
```

Figure 9: Pseudo-code for the intersection check for any polygon

2.4.1 Optimization: Line intersects a convex polygon

The procedure is the same for all polygons. However, it is known that any line drawn through a *convex* polygon can only share two intersections with the boundary of that polygon: one intersection where the line enters the polygon face and one where the line exits the polygon face.

If the line does not cross the boundary of the polygon but only touches it, there is only one intersection. Note that if the intersection of the line and the polygon is a line itself, several successive line intersections might be found by the algorithm. These however all lie on the same line. See Figure 10 for an illustration.

So if the polygon is a convex polygon, the algorithm can already terminate after either two point intersections or one set of successive line intersections have been found. This saves about fifty percent of all intersection checks that need to be made for lines *that actually intersect* with the polygon.

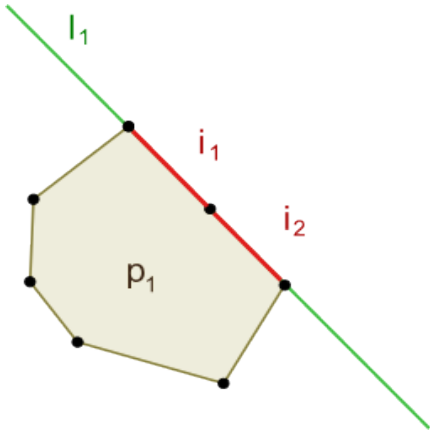

Figure 10: The line l_1 intersects the convex polygon p_1 in two lines i_1 and i_2. Both lie on the same line.

2.4.2 Optimization: Intersection with a ray

In many applications such as collision detection, only the first intersection of a ray and a polygon is of any interest. If it is known whether the starting point of the ray is inside or outside the polygon, the number of edges to be checked for an intersection with the ray can be reduced to about the half:

Back-face culling is a technique used in 3D computer graphics which excludes all polygons from the rendering process whose faces do not face towards the camera. This is done by a simple comparison between the viewing direction vector V of the camera and the normal vector N of the face:

$$V \cdot N > 0$$

If the scalar product of V and N is greater than zero, the *counter-clockwise* defined polygon faces away from the camera and thus will not be rendered (For polygons that are defined clockwise, when the scalar product is smaller than zero). The same technique can be used in this application: if the starting point of the ray is known to be outside the polygon, the *back-facing* edges can be left out of the intersection checks. However, if the starting point is inside the polygon, the *front-facing* edges can be left out.

Figure Error: Reference source not found shows an example of this technique:

The intersection between the ray and the edge e_1 is not checked because the scalar product of $\vec{n}_1$ and $\vec{r}$ is greater than zero, which means the edge is back-facing. All green edges in the image are back-facing and thus are not checked for intersections with the ray.

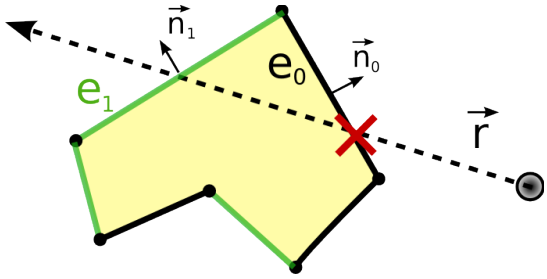

Figure 11: Example of the back-face culling technique: $\vec{r}$ is the direction vector of the ray. $\vec{n}_0$ And $\vec{n}_1$ are the right-hand normal vectors of the edges e_0 and e_1.

When using the back-face culling technique on convex polygons, the algorithm can already terminate when one intersection point has been found because the other intersection with the ray would be found on a back-facing edge.

It should be noted that to determine whether the starting point of the ray is inside or outside of the polygon is about as expensive as the line and polygon intersection check itself. That is why it is not expedient to do the point in polygon test here just to be able to discard about half of the intersection checks by using back-face culling. See the next section for details on the point in polygon check.

The function in pseudo code shown in Figure 12 returns true if the given edge may be culled:

```
culling_check(edge,ray):
  scalar = direction vector of the edge * right-hand normal vector of the ray
  if starting point of the ray is outside the polygon
    if scalar > 0
      return true
    endif
  else
    if scalar < 0
      return true
    endif
  endif
  return false
end
```

Figure 12: Pseudo-code for a back-/front-face culling check

2.5 Point in polygon check

For polygon-based landscapes, point-in-polygon checks are used for a couple of things. This includes checking whether an object is stuck in a solid polygon or finding out what material can be found at a certain location.

There are two common methods to test whether a point lies inside or outside of a simple polygon. Both the winding-number algorithm and the ray-casting algorithm were already roughly described as early as 1974 by Sutherland et al. [S1]. Only the ray-casting algorithm and its special cases will now be described in detail.

In this algorithm, a ray is shot from the point into an arbitrary direction. Whether the point lies inside or outside of the polygon is determined by how many times the ray *crosses* the boundary of the polygon. If the number of crossings is odd, the point lies inside of the polygon. If it is even, the point lies outside. The method is similar to the intersection check with lines, but here only the number of crossings is of interest, not the intersections themselves.

The ray can be shot in any direction. However, it is easier to compute intersections with lines that are parallel to one of the axes.

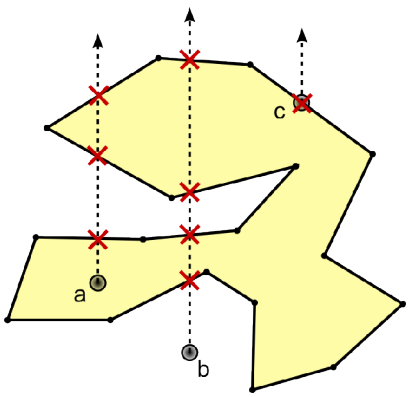

Figure 13: Point p is inside the polygon, q is outside and r is on the border

Three cases should be distinguished (see figure 13):

(a) The number of crossings is odd which means the point is inside the polygon.

(b) The number of crossings is even. The point is outside the polygon.

(c) The intersection of the ray and a polygon edge is the point itself. The point is on the border of the polygon

The border of the polygon is defined as still inside the polygon so case three is not required because it is covered by case one. However, in case three the algorithm can terminate without any further (intersection) checks.

See Figure 14 for the pseudo-code.

```
for each edge in the polygon
  if edge and ray intersect
    if intersection is at start-point of edge
      continue with next edge
    endif
    if the intersection point is the starting point of the ray
      return true
    endif
    if the intersection is a line or at a polygon vertex
      if crosses_boundary(ray,intersection)
        increment crossingCount
      endif
    else
      increment crossingCount
    endif
  endif
endfor
return true if crossingCount is odd, otherwise false
```

Figure 14: pseudo-code for the ray-casting algorithm

2.5.1 Special case: Ray hits vertex or intersects in a line

In this algorithm, a clear distinction has to be made between *cross* and *intersect*. Only if the ray crosses the boundary of the polygon, the intersection may be counted:

Only in the special case that the ray intersects with a vertex of the polygon *or* the intersection consists of a line, it may happen that the ray does not cross the boundary. Otherwise, an intersection is always a crossing.

Figure 15 depicts the possible cases:

The ray of point a_0 intersects with a vertex of the polygon but does not cross its boundary. The intersection is disregarded. The same can be observed for a_1. However, a_1 is correctly identified as inside the polygon because the second intersection is a crossing.

Like a_0, the ray of point b intersects with a vertex. But here, it *crosses* the polygon boundary.

The first intersection of the ray of point c is a line segment. Because the ray enters into the polygon face and thus crosses the boundary at the end of the intersection line, the intersection counts as a crossing.

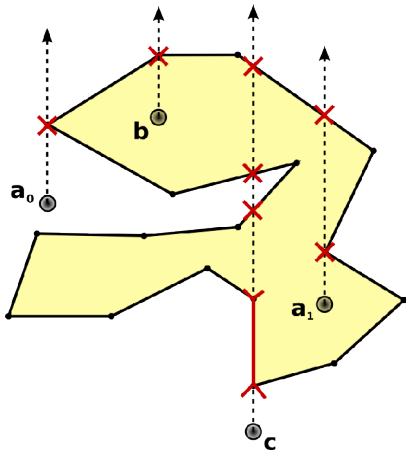

Figure 15: Intersections with the vertices of a polygon

To determine whether a line crosses a boundary in one intersection or not, the following easy check has to be made:

A line splits a plane into two sides. If both the vertex of the polygon before and the vertex after the intersection lie on the same side of the line, the line does not cross the boundary. Otherwise, it crosses the boundary.

Note that this line intersection doesn't need to consist of only one line. There could be several successive line intersections before the ray actually crosses or leaves the boundary of the polygon. Generally, successive line intersections which lie on the same line should be treated as one intersection in this case. Figure 16 shows the pseudo-code for this check.

```
crosses_boundary(ray,intersection):
  A = previous vertex of the polygon which is not part of the intersection
  B = next vertex of the polygon which is not part of the intersection
  if A is left of the ray and B is right of the ray
    return true
  endif
  if B is left of the ray and A is right of the ray
    return true
  endif
  return false
end
```

Figure 16: pseudo-code for the ray-casting algorithm

2.5.2 Optimization: Convex polygons

The point-in-polygon check for convex polygons is much easier and faster: if the vertices of a convex polygon are defined in a counter-clockwise order, the point must be on the left side of all polygon edges to be inside the polygon. If the point is on the right side of any edge, the algorithm can terminate and it is outside of the polygon.

3 Clipping operations on polygons

Clipping operations (also called *Boolean operations*) are binary operations on two polygons. These include *union*, *difference*, *intersection* and *exclusion*. The polygon on which the operation is performed is the *subject polygon*, the other is the *clip polygon*. See figure 17.

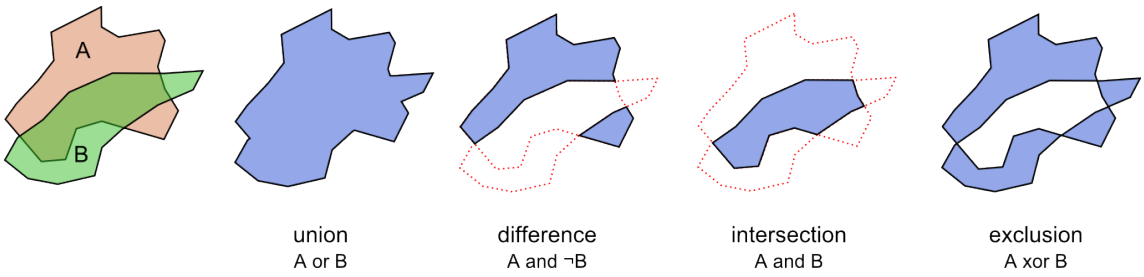

Figure 17: The four clipping operations for the subject polygon A and clip polygon B.

In this context, clipping operations are used to change the landscape. If for example a round hole should be blasted into a specific spot in the landscape, a polygon in the shape of that hole is subtracted from the landscape at that spot. If a chunk of another material is inserted into the landscape, first the polygon which represents this chunk is subtracted from the landscape (to make a hole for itself to fit in) and then inserted into it.

3.1 Previous work

The clipping of polygons has been a basic problem in areas such as 3D computer graphics and computational geometry. Early algorithms were tailored to clip polygon faces with rectangular areas like the viewport boundaries or single lines. The clipping of polygons with the display area in computer graphics is a standard and fundamental task that is carried out in any application that uses 3D graphics. The wide range of applications of 3D graphics explains the large amount of scientific papers that deal with this problem [L1, S2, F1, R1].

Later, various algorithms were developed that clip arbitrary planar polygons like concave polygons, polygons with holes or non-simple polygons. The basic procedure of polygon clipping is similar for most of these algorithms:

First, the intersections of all edges of the two polygons are calculated and subdivided at the intersection points. Then, it is determined which edges are inside or outside the other polygon and the resulting polygon is constructed accordingly. Some algorithms do this in separate steps while others already construct the resulting polygons in the same loop as the intersections are calculated.

Algorithms based on a mathematical model of *simplicial chains* (*simplex theory*) were developed by Rivero and Feito [R2] and enhanced by Peng et al. [P1]. They accept simple polygons without or with any number of holes.

Algorithms based on a *scanbeam* (also: *sweep-line*) were presented by Sechrest and Greenberg [S3], later by Vatti who developed an algorithm that worked for arbitrary polygons that may self-intersect [V1]. The algorithm of Martínez et al. [M1] is also based on the scanbeam paradigm but is faster than Vatti's, especially for polygons with a lot of edges. The scanbeam is used to find the intersections between the two polygons more efficiently. A more thorough description of this method is given in section 3.3.2 (Optimization: Using a scanbeam) as this method is compatible with the algorithm presented in this thesis and other *edge labeling*

approaches. The algorithms based solely on a scanbeam derive the data needed to construct and return the resulting polygons directly during that operation.

An *Edge labeling* approach has been introduced by Greiner and Hormann [G1], Schutte [S4] and Desmera [D1] independently.[6] In this approach, the algorithm is split up into three steps. After the intersections between the two polygons have been found and inserted into both polygons as vertices, the edges of both polygons are labeled according to their orientation with respect to the other polygon. In the third step, the resulting polygons are constructed on the basis of this data. So, the part of finding the intersections between the two polygons is separated from the logic of putting together the resulting polygons.

Subsequently, these algorithms will be analyzed in more detail because the algorithm for polygon clipping presented in this thesis is also based on the edge labeling approach:

Greiner and Hormann's algorithm can handle self-intersecting polygons because it included a clear definition of which points are inside and which are outside the polygon. Their general concept of the labeling process is the following: The algorithm starts by determining if one vertex of the subject polygon is inside or outside the clip polygon. The following edges are labeled the same as their respective predecessors until an edge of the other polygon is crossed. Whenever this happens, the labeling of the following edges is switched: if the previous edges were labeled as outside, the next edges are labeled as inside and the other way round. When the starting vertex is reached, the labeling is complete. The same is done with the clip polygon.

However, this method assumes that no vertex of one polygon lies exactly on the boundary of the other polygon. This can hardly be assumed for the general case of arbitrary polygons. The solution Greiner and Hormann offered for this is to displace the affected vertex by a tiny distance to circumvent this special case. It is explained that if the chosen distance is smaller than one pixel on the screen, the difference will be too small for the eye to see. But this solution is not satisfactory because it can't be assumed that there is no case where the displacement of one vertex does not lead either to the same situation with another edge, or a new intersection of the surrounding edges with edges of the other polygon. The problem becomes more obvious when working with integers instead of floating point numbers.

Furthermore, Greiner and Hormann's algorithm doesn't account for possible holes in resulting polygons, nor the possibility that the edges of the subject and clip polygon may intersect in lines.

The edge labeling part of Schutte's algorithm is similar to the one presented in this work. However, only the operations *union* and *difference* are described in their paper. Their algorithm only handles simple polygons but can be extended to accept self-intersecting polygons and polygons with holes as input. Polygons with holes are first split up into several polygons by subtracting the hole from the polygon. For self-intersecting polygons a similar technique is used.

6 It is to say that Greiner and Hormann didn't use the term *edge labeling* in their paper but ultimately their approach is an edge labeling approach according to the explanation in the text.

3.2 General concept

Subsequently, another algorithm based on the *edge labeling* approach will be described. It is able to handle all simple polygons and covers all four clipping operations: *union*, *difference*, *exclusion* and *intersection*.

To match the requirements for an implementation mentioned in the introduction, the algorithm is able to completely work in integer space. This both makes it possible to define an upper limit for the detail of the polygons as well as makes network synchronization easier. However, in integer space some special cases become more obvious and must be covered for mathematical robustness.
The algorithm only outputs polygons it itself accepts as input polygons because the algorithm is supposed to work repeatedly on the same set of polygons.

Roughly, the algorithm can be split up into 3 main steps:

1. Find and insert all intersections into both polygons. As a result of this step, all edges intersect only at their endpoints. Thus, all edges are now clearly situated either inside or outside of the other polygon.
2. Label each of the polygon edges of both polygons as either *inside* (inside the other polygon), *outside* (outside the other polygon), *shared* or *reverse-shared*. *Shared* means that both polygons share this edge. *Reverse-shared* edges are shared edges with opposite orientation.
3. Connect the edges according to their labels and the designated clipping operation. For difference (see figure 18): connect all outside and reverse-shared edges of the subject polygon with all inside edges of the clip polygon. For union, connect all outside and shared edges of the subject polygon with all outside edges of the clip polygon.

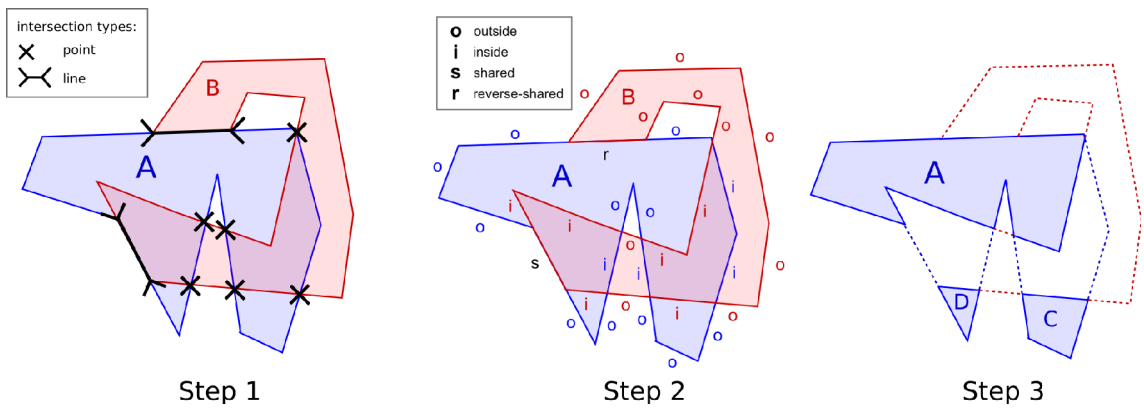

Figure 18: The three steps of the clipping operation "difference"

3.3 First step: Intersections

In the first step, all intersections are found by checking each edge of the subject polygon and each of the clip polygon for intersections. The calculated intersections are then inserted as extra vertices in both polygons (if they are not already in the polygon at this position) and memorized for the next step as shown in Figure 19. Because new vertices can be inserted at random positions in the polygon, the sequence of vertices should be available as a linked list rather than an array list (*vector*) during the clipping operation. The complexity to insert a new vertex at a given iterator is O(1) for linked lists as opposed to O(n) for array lists. More importantly, iterators to array lists are invalidated when a new element is inserted. As the intersections need to be memorized for the second and third step of the algorithm, an array list is not eligible for this algorithm.

```
for each edge Aᵢ of subject polygon A
  for each edge Bᵢ of clip polygon B
    if Aᵢ and Bᵢ intersect
      insert(intersection,Aᵢ)
      insert(intersection,Bᵢ)
      memorize all vertices in intersection and its positions in A and B
    endif
  endfor
endfor
```

Figure 19: First step of the polygon clipping algorithm

The result of this step is that no two edges of the polygons may intersect at any other point than at its vertices. In other words: each edge of both polygons must be clearly classifiable as either outside, inside, or on the boundary of the other polygon. That is why the intersection points are inserted in both polygons.

For line intersections, both the start and end points have to be inserted into the polygons. The order in which the points are inserted is of importance here to not mess up the shape of the polygon: if the intersection line goes into the same direction as intersecting polygon edge, the start point is inserted before the end point. Otherwise, if the line goes into the opposite direction, the end point is inserted before the start point. The scalar product of two vectors that go into the same direction is always positive. So, all that needs to be done to check this is to multiply the direction vector of the intersection line and the edge. The pseudo code can be seen in Figure 20.

```
insert(intersection, edge):

  if intersection is a point
    if point is neither start nor end point of the edge
      insert point into polygon at edge
      check for new intersections and self-intersections at neighboring edges
    endif
  elseif intersection is a line with start point P and end point Q
    if line direction vector * direction vector of edge ≥ 0
      insert(P,edge)
      insert(Q,edge)
    else
      insert(Q,edge)
      insert(P,edge)
    endif
  endif

end
```

Figure 20: Insert an intersection into an edge

In case no intersections have been found in this step, the polygons are either not intersecting at all or one polygon completely encloses the other (all points of polygon A is in polygon B or the other way round).

If the second applies, all edges of the smaller polygon can be labeled as inside and all edges of the bigger polygon as outside. The second step of this algorithm can be skipped in this case.

The time complexity for this step is $O(n^2)$. This is the most expensive operation in this algorithm. In section 3.3.2, it is explained how to exclude most edges from the intersection check using *scanbeams* in order to make this step more efficient. Other than that, there is no way to reduce this complexity without using spatial data structures which make the exclusion of edges that do not intersect faster. A solution with a geometric data structure is presented in chapter 4 (Spatial data structures and algorithms).

3.3.1 Special case: Additional intersections are created

The shape of the polygon is theoretically not altered by the insertion of the intersections since these points are already part of the boundary. In reality, numbers in computers have limited precision. Thus, any insertion of a new vertex in a polygon may alter its shape.

A minimal displacement of the intersection points due to rounding imprecision leads to a slightly different shape of the polygon. This means, that at the edges around the newly inserted vertex, there might occur new intersections between the two polygons.

The data displayed in figure 21a and 21b is taken from an actual test of the implementation.

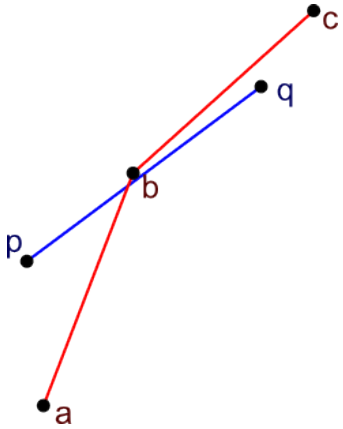

Figure 21a: $\overrightarrow{ab}$ intersects with $\overrightarrow{pq}$ at a point close to b. The intersection point must be inserted in both polygons between p,q and a,b.

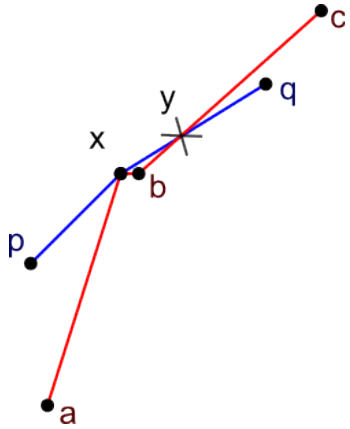

Figure 21b: The intersection point is inserted. The shape of both polygons change slightly and now the newly created edge $\overrightarrow{xq}$ intersects with an edge of the other polygon.

Note that this special case can happen always when the intersection is not exactly located at one point.

Unfortunately, to be sure that all intersections are found, the four neighboring edges of the inserted point (of both polygons) must each be checked for new intersections with the other polygon. If there are new intersections, these have to be inserted and checked for new intersections, too.

This issue is the prime reason why the finding of the intersections need to be separated from the labeling and constructing of the resulting polygons.

Additionally, it must be noted that through the insertion of new vertices, the shape of the polygons can be altered in a way that (one of) the polygons do not meet the constraints to fall into the category of simple polygons anymore. The data displayed in figure 22a and 22b is also taken from actual test data. After the intersection points have been inserted into both polygons, one edge of the extremely narrow section of polygon *A* slipped down past its own boundary which means that polygon A *self-intersects*, it has a section which is defined clockwise.

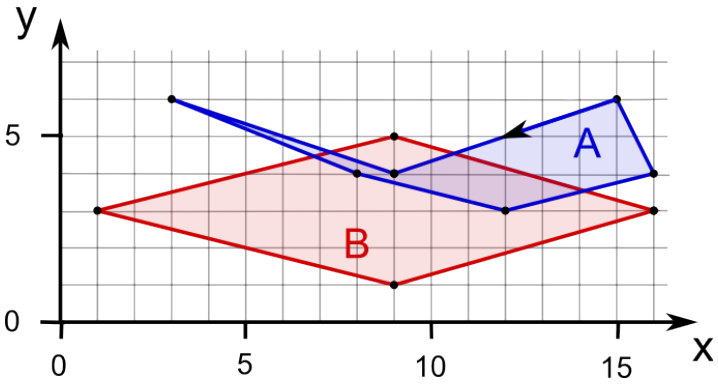

Figure 22a: Two polygons are clipped. This figure shows the two original polygons, the arrow shows the orientation of the edges in polygon A.

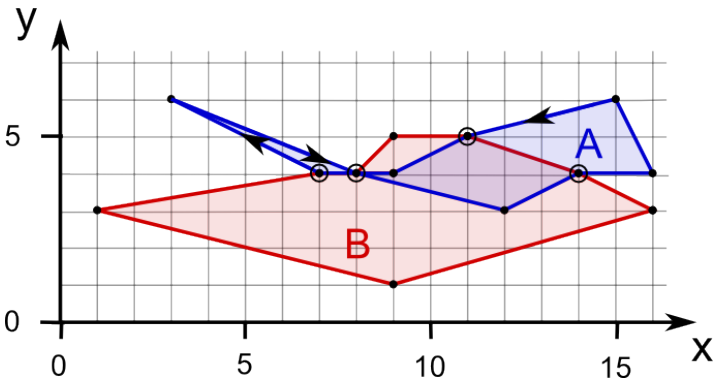

Figure 22b: The two polygons from figure 22a after the intersections have been added to both polygons. The encircled vertices are the newly inserted vertices, the arrows show the orientation of the edges in polygon A.

It is not possible to prevent a polygon from deforming into a non-simple polygon at this point, but it is possible to detect if self-intersections occurred and split the original polygon up at these points. These have to be clipped separately in the rest of the clipping algorithm. In the above example, polygon *A* must be split up into two polygons at position (8,4).

The self-intersections are found the same way as the intersections between two polygons are found. Also, the same rules apply: the intersection points need to be inserted in the polygon if they do not exist yet and memorized thereafter in order to easily split them up after the first step of the clipping algorithm completed. If new points are inserted, again all neighboring edges need to be checked for new intersections.

See figure 23 for the pseudo-code to split up the polygon at its self-intersections.

So, the actual (worst-case) cost of the first step of the clipping algorithm adds up to $n^2+8 \cdot c \cdot n$ intersection checks with n being the number of edges of the polygons and c the number of intersections between the two. The factor is 8 because for each inserted intersection point that alters the shape of the polygons, all 4 neighboring edges need to be checked for new intersections as well as each the 2 neighboring edges of both polygons for self-intersections.

```
separate(polygon P):

  Start = any vertex of the self-intersections
  L = list of separated polygons
  S = stack of start vertices
  S.push(Start)

  while Start is not reached again, walk along the boundary of P
    if another self-intersection I is found

      Begin = the vertex at the top of stack S
      Current = the current vertex

      if I is at the same point as P
        add the vertices from Begin to Current as a new polygon to L
        S.pop
        if S is empty now
          Start = Current
          S.push(Start)
        endif
      else
        S.push(Current)
      endif

    endif
  endwhile

  return L
end
```

Figure 23: Split up the polygon into several polygons at the point(s) at which it self-intersects

At the time of this writing, it is unknown whether there is a method that somehow circumvents this problem as this issue has not even been mentioned in past works. A reason for this might be that past works were based on floating point numbers. So when an intersection had been missed, the resulting error in the output usually happened at some place after the decimal point and thus was usually not visible (on the screen). Another reason might be that past clipping algorithms were not tested to work with polygons that were defined at the highest precision available, e.g. a narrow and long polygon with a width of exactly one unit or even less (as shown in figure 22).

With random test data, the insertion of new vertices and the resulting change of the polygon's shape rarely leads to new intersections to occur between the two polygons. However, it happens quite often if the boundaries of the polygons are represented near the maximum level of detail available. Because in each clipping operation, the intersections are inserted in both polygons as new vertices, the polygons tend to become more and more detailed with every clipping operation performed on them. As polygons can be clipped continuously in a game with a changeable landscape, it must be assumed that many polygons might be represented near the maximum level of detail available.

This does not change the fact that through such errors introduced in this step, a polygon can result that is not a simple polygon anymore or the algorithm fails completely due to erroneous intermediate results.

It is not trivial to keep track of the intersections that have to be memorized in this case because when the shape of the polygon is changed, not only new intersections can arise, others might also be transformed into line intersections or previously found intersections are found again. Because of this, memorized vertices that are part of an intersection should be stored in a data structure that stores only unique values like a set or a map.

3.3.2 Optimization: Using a scanbeam

With this algorithm, the intersection checks are only made between edges that are in an active stripe: the scanbeam. The scanbeam can either be vertical or horizontal and moves from one side of the polygon to the other; for example from the top (the topmost vertex of the polygons) to the bottom (the bottommost vertex of the polygons). During the process, a list of active edges for each polygon is kept. Intersection checks are only made against edges in this list which limits the number of necessary intersection checks substantially. For a horizontal scanbeam that goes from the top to the bottom, the algorithm can be roughly described like this:

Before the algorithm starts, all vertices of both polygons must be sorted by their vertical position. After that, each vertex of both polygons is processed in the order they appear in the sorted list: If the current vertex is the upper endpoint of an edge, that edge is put into the list of active edges. On insertion, it is checked whether the newly inserted edge intersects with any edges of the other polygon already in the list. If it does, the intersection point is calculated and inserted into both polygons. The sorted list of vertices needs to be updated to include the newly inserted vertices. Since the two active edges that intersected do not exist in the two polygons anymore, they need to be altered to have their lower endpoint on the newly inserted vertex. If the current vertex is the lower endpoint of an edge, that edge is removed from the list of active edges.

Since vertices in polygons are adjacent to two other vertices, vertices can be the upper endpoint of one edge and the lower endpoint of another, the upper endpoint of two edges (like the topmost vertex) or the lower endpoint of two edges (like the bottommost vertex).

To efficiently implement this, the vertices of the polygons and the active edges need to be able to link to each other so that it is easy to identify the associated active edges of one vertex.

Figure 24 shows the generic scanbeam algorithm in action with two example polygons. As illustrated by the example, the algorithm can make the intersection check between the two polygons much faster because some edges do not even have to be checked for any intersections; others only check against a few edges of the other polygon. However, this optional optimization requires some additional implementation effort as well as a change to the data structure of the polygons. Also, the special case described in section 3.3.1 has to be accounted for in the scanbeam algorithm, too.

For the average case, the time complexity of a polygon intersection using this technique is the time complexity of the sort operation before using the scanbeam algorithm. To sort a list with *quicksort* has an average time complexity of O(n log n), with *n* being the number of vertices.

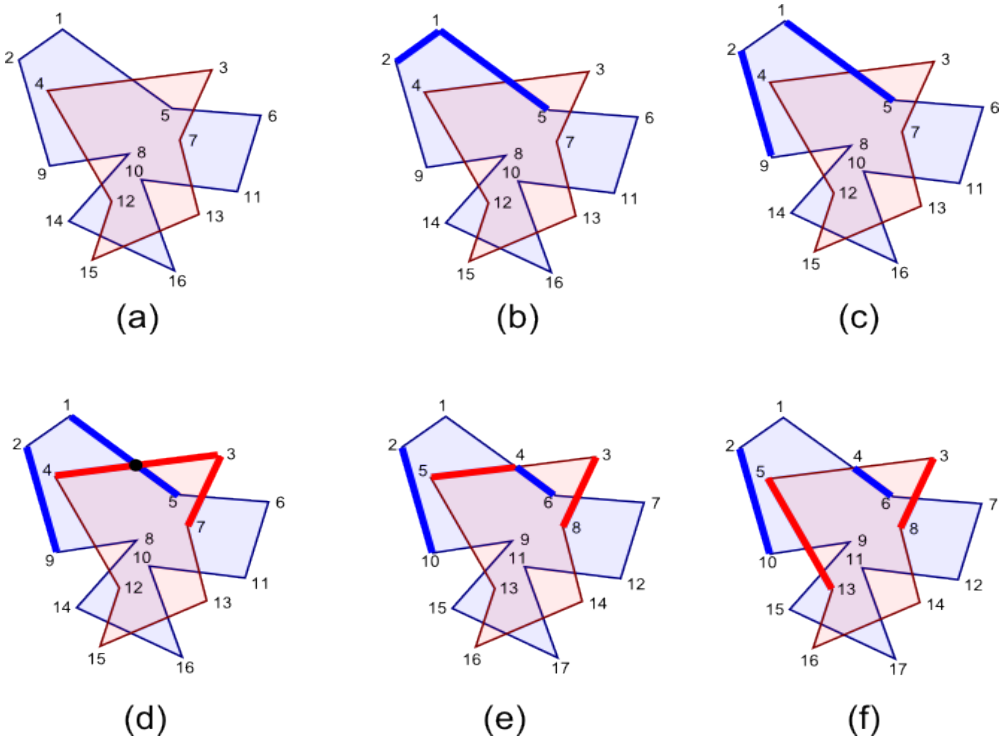

Figure 24: The first few steps of the scanbeam algorithm

The bold lines in the figure 24 mark the active edges. The numbers at the vertices show their order in the list sorted by their vertical position. The first vertex in this list will be called p_1, the second p_2 and so forth.

(a) Before the algorithm can begin, all vertices of both polygons must be sorted by their vertical position in a list.

(b) The algorithm starts at the first vertex. Its associated edges are added to a list of active edges. Since there are no active edges of the other polygon, no intersection check is made.

(c) The algorithm continues with the second vertex which is the lower endpoint of the edge $\overline{p_1 p_2}$. The edge is removed from the list of active edges. The second vertex is also the upper endpoint of $\overline{p_2 p_9}$ which is then added to the list of active edges.

(d) The third vertex is reached. It is the upper endpoint of two edges, both are added to the list of active edges one after another. Both added edges are checked with all active edges of the other polygon for intersections. $\overline{p_3 p_4}$ intersects with $\overline{p_1 p_5}$.

(e) The vertices at the intersection point are inserted normally into both polygons and then inserted into the sorted list of vertices at the adequate position. After that, the old edges are altered to have the newly inserted vertex as their endpoint. The vertices at the fourth point are reached. The just altered edges are removed and the other associated edges are added as active edges and each checked for intersections with the other polygon.

(f) The fifth vertex is reached. It is the lower endpoint of $\overline{p_4 p_5}$ and the upper endpoint of $\overline{p_5 p_{13}}$. $\overline{p_4 p_5}$ is removed and $\overline{p_5 p_{13}}$ is added as an active vertex. The algorithm continues with the sixth vertex...

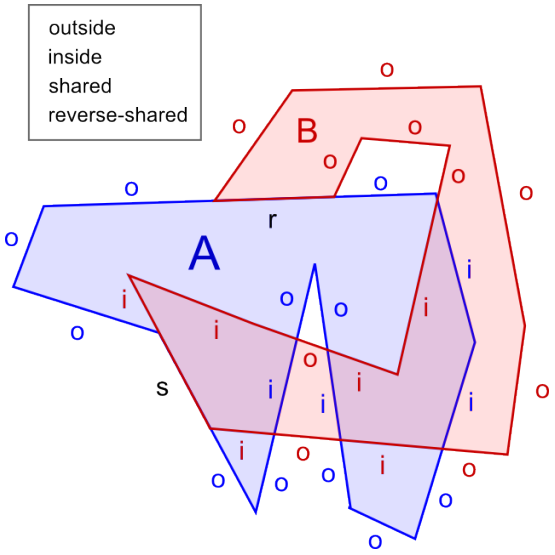

Figure 25: Two labeled polygons

3.4 Second step: Labeling

There are four possibilities how a polygon edge can be positioned with respect to the other polygon: it can be an *inside*, an *outside*, a *reverse-shared* or a *shared* edge. All possibilities are depicted in Figure 25.

Shared and reverse-shared polygons are special cases which normally appear very rarely with arbitrary polygons when two polygon edges intersect in a line.

Schutte [S4, p.4] mentioned that the obvious (naive) method to determine how each edge is located is to check whether the midpoint of the edge is inside or outside the other polygon.

However, as previous works already pointed out, one point-in-polygon check for concave polygons results in n intersection checks (n is the number of edges) with the ray-casting algorithm. Thus, labeling all edges of both polygons would require $2n^2$ intersection checks.

Apart from the performance issue, the method returns incorrect results if any edge is too short: while it is possible to find a midpoint that is located between any two points in $\mathbb{R}^2$ in mathematics, the numbers in computers have a limited precision. This is particularly obvious when using integers. In integer space, it is impossible to get the midpoint of a line with the length of one unit:

Figure 26a and 26b show an example for these erroneous results with two polygons in integer space. The same problem exists of course for floating point numbers but only after the decimal point.

The edge at (4,2),(3,3) of polygon B should be labeled as outside as well as the neighboring edges of polygon A marked in bold. However, in the example the edges are labeled as inside the other polygon. This is because when the midpoint of this edge is calculated, the result is rounded and then happens to be one of its endpoints. Both endpoints touch polygon A, thus are identified as inside polygon A by the point-in-polygon algorithm. This is correct for the endpoints but not for the midpoint.

In some cases this can lead to an invalid output (the resulting edges do not form a closed polygon), in other cases the output is just erroneous.

The errors produced by incorrect labeling are usually not much bigger than the rounding imprecision itself as shown in figure 26b. However, it can't be excluded that this error results in a completely scrambled output: for example, if the erroneous section was the entry point to a larger gap in the polygon.

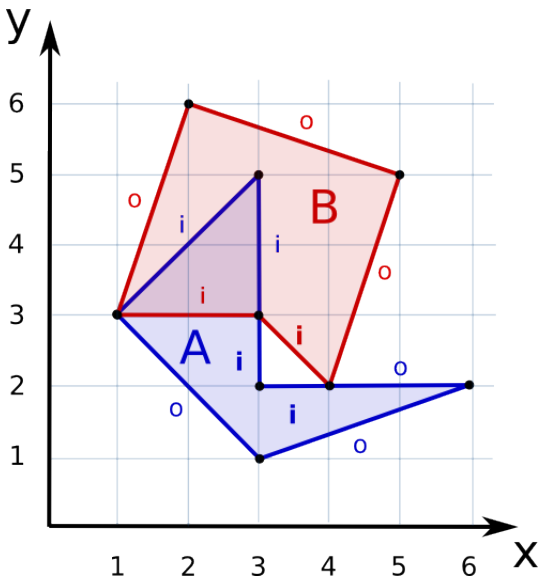

Figure 26a: Erroneous labeling of two polygons in integer space. The labels around the hole between polygon A and B are wrong.

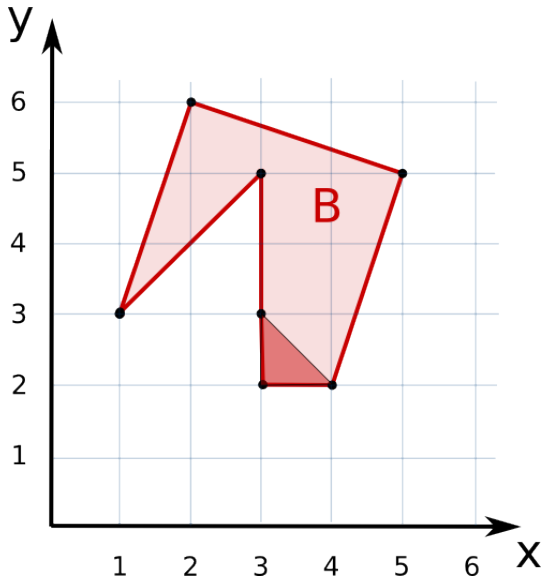

Figure 26b: The difference of B and A using the erroneous labels. The red section to the bottom is the part which shouldn't be part of the polygon.

As Schutte pointed out himself, for these reasons, the point-in-polygon check is rather unfit to label the edges of the polygons.The algorithm that is presented in the following section works without any point-in-polygon checks. Like Schutte's algorithm it works in integer space, but is marginally simpler than his solution and works with arbitrary simple polygons. It can be understood as a combination of Greiner and Hormann's chalk cart metaphor [G1, p. 4] and a simplified version of Schutte's *label_angle* algorithm [S4, p. 5] which determines if the edges at an intersection cross or just touch each other at the intersection point. The time complexity is O(n). *n* is the number of edges of both polygons.

3.4.1 Labeling process

The labeling process is started at one intersection point *p* of polygon *A* and *B*. For its successive edge e_A it is determined whether it is inside or outside the other polygon and labeled accordingly. After this has been done, the following edges of polygon *A* are labeled the same as e_A until the next intersection *q* is reached.

For *q* and its following intersection points, the process is repeated until the first intersection *p* is reached again. All edges of A are labeled at this point; figure 27 demonstrates this process. The same is done for polygon B.

While it suggests itself to just switch the labeling around after each intersection point (label the successive edges as *outside* if the ones before were labeled as *inside* and the other way round) like it has been described in Greiner and Hormann's approach, this will produce incorrect results whenever the edges of the two polygons do not cross but only touch each other at the intersection. This is why a new check is required at every intersection.

Also note that this method assumes that the positions at which the two polygons intersect are known. Because of this, the positions have to be memorized in step one.

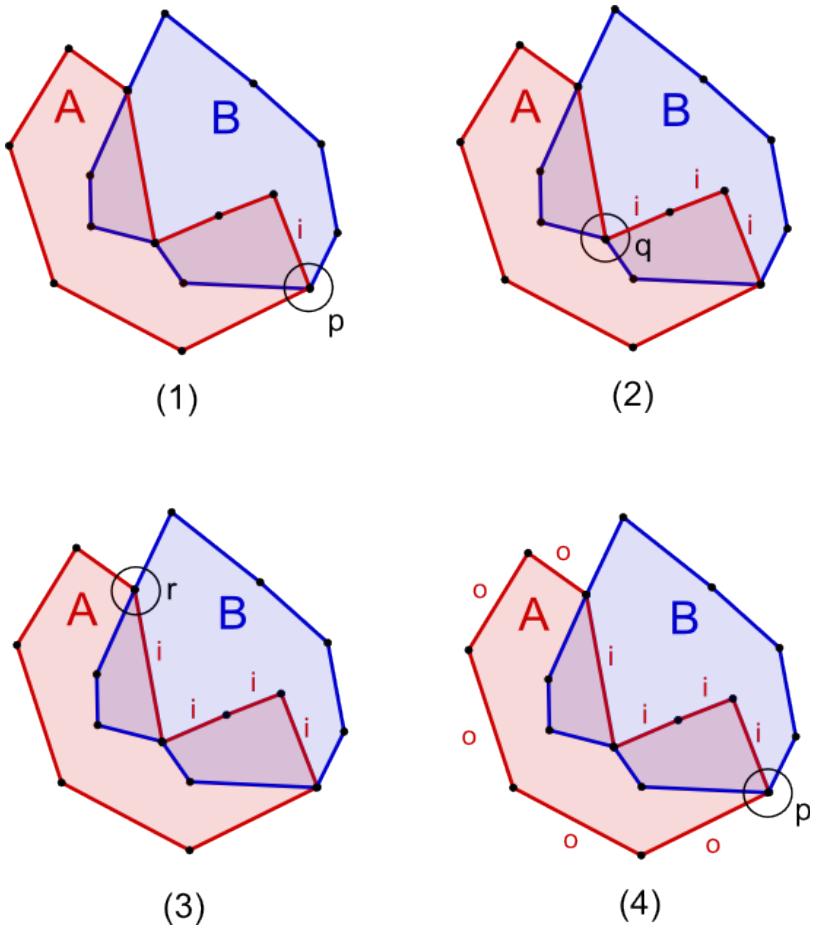

Figure 27: The labeling process for polygon A

(1) The process starts at a random intersection *p*. First of all, the orientation of the successive edge of polygon *A* is determined and labeled accordingly.

(2) Then, all successive edges of polygon A are labeled the same until the next intersection *q* is reached.

(3) The process begins anew: the successive edge of *q* is labeled according to its orientation and the successive edges are labeled the same (which are none in this case) until a next intersection *r* is reached.

(4) The same is done for intersection *r*. The labeling is done when the first intersection *p* has been reached again.

3.4.2 Determine status of edges at intersections

To determine the status (*inside* or *outside* the other polygon) of an edge at an intersection point, it simply has to be checked on which side of the other polygon's edge it is located.

After the first step, it is guaranteed that the edges of the two polygons can intersect each other only at its start- or endpoints. The edges can be clearly labeled as inside, outside or on the border of the other polygon. This also means that whether an edge adjacent to an intersection is inside or outside the other polygon is only dependent on the two edges of the other polygon that are also adjacent to the same point. Since the polygons are defined counter-clockwise, the left side of these two edges is defined as inside the polygon while the other side is outside of the polygon.

Figure 28a illustrates this. It also illustrates that the *convex* and the *concave* case must be distinguished. For b_0 or b_1 (and thus e_{b0} and e_{b1}) to be inside the polygon A, it must be ***either*** on the left side of e_{a0} ***or*** on the left side of e_{a1} because those two edges are *concave*. If the two edges are *convex*, like in polygon *B*, the point for which its containment in that polygon is checked, must be on the left side of e_{b0} ***and*** e_{b1}.

The check on which side of a line a point p_x is situated is trivial: if the scalar product of the line's normal vector and the vector (p_x minus the line's start point) is positive, p_x is located on the right side of the line. If it is negative, it is on the left side and if it is zero, it is on the line.

Line intersections are not treated any different than point intersections: For determining how e_{a0} is situated, only the adjacent edges are being looked at, namely e_{b1} and e. See Figure 28b for reference. This is why it is enough to memorize only the vertices that are part of intersections in the first step of the algorithm, not the (line) intersections themselves. The two endpoints of the intersection line e in figure 28b were memorized separately.

To for example find out if the edge e_{a1} of figure 28a is on the border of polygon B, as it is the case for line intersections, the following simple check can be made: If a_1 is on the same spot as the first point of e_{b0}, then the edge is *reverse-shared*. If a_1 is on the same spot as the second point of eb1, it is a *shared* edge.

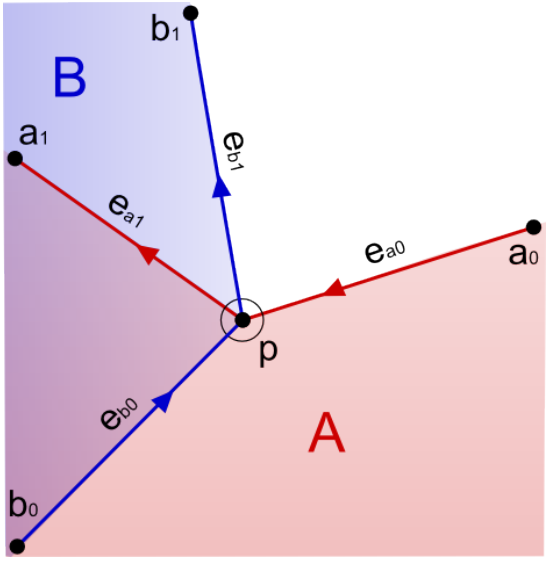

Figure 28a*: The left side of the edges around the intersection point p is inside their polygon. The arrows on the lines show the orientation of the edges.*

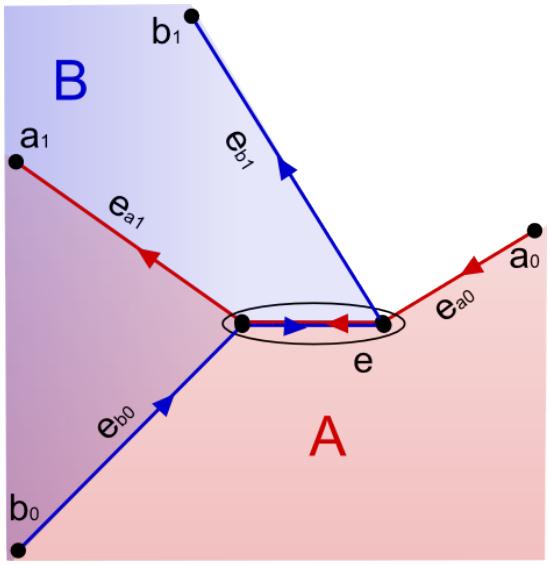

Figure 28b: A line intersection

3.4.3 Implementation and data structure

In step one, it was mentioned that the intersections need to be memorized for step two. They should be memorized in a way that it is fast and easy to:

(1) access the edges and points around each intersection point / line

(2) access the corresponding intersection of the other polygon

(3) access the next intersection in one polygon

The simple data structure *poly_vertex_link* can be used to meet these requirements. It links to two vertices of the subject and clip polygon that are at the same spot (see figure 29). Because of the special case described in section 3.3.1, the *poly_vertex_link*s should be kept in a set or map. Additionally, each vertex which is part of an intersection with the other polygon links back to the corresponding *poly_vertex_link*.

```
struct poly_vertex_link:
  iterator to the vertex at the intersection point in polygon A
  iterator to the vertex at the intersection point in polygon B
end
```

Figure 29: poly_vertex_link data structure

Figure 30 shows the algorithm for the second step in pseudo-code. Figure 31 shows the method for determining the status at an intersection.

```
label(polygon A, polygon B, set of poly_vertex_links Links):
  for each poly_vertex_link I in Links
    edge_label = successive_edge_status(I,A,B)
    for each edge e of polygon A between I and I+1
      label e like edge_label
    endfor
  endfor
end
```

Figure 30: Labeling process for the polygon A. The process is the same for the other polygon.

```
successive_edge_status(poly_vertex_link I, polygon A, polygon B):
  e = edge after I in A
  a = first vertex of e (= point of intersection)
  b = second vertex of e

  e1 = edge before I in B
  e2 = edge after I in B

  label = “outside”

  if a == first vertex of e1
    label = “reverse-shared”
  else if a == second vertex of e2
    label = “shared”
  else if e1 and e2 are concave
    if b is left of e1 or e2
      label = “inside”
    endif
  else
    if b is left of e1 and e2
      label = “inside”
    endif
  endif

  return label
end
```

Figure 31: Function for determining the status of the successive edge at an intersection

3.5 Third step: Connect edges

In the third step, the tagged edges are connected to form the new polygon(s). Which edges are used and which are discarded is dependent on the clipping operation that is performed. Used are:

for ***union***	*outside* edges of both polygons, *shared* edges of the subject polygon
for ***difference***	*outside* and *reverse-shared* edges of the subject polygon, *inside* edges of the clip polygon in reverse direction
for ***intersection***	*inside* edges of both polygons, *shared* edges of the subject polygon
for ***exclusion***	*outside* edges of both polygons, *inside* edges of both polygons in reverse direction

The time complexity of this step is O(n). *n* is the number of edges of both polygons.

3.5.1 Basic procedure

Starting at any intersection, the tags of the four neighboring edges are looked at. Dependent on which clipping operation is performed, an adequate edge is selected and added to the resulting polygon. If there are several edges at the intersection which are tagged adequately, the algorithm selects the one which takes the sharpest turn left (see next section). The successive edges are then followed and also added to the resulting polygon until the next intersection is reached. At that intersection, the algorithm again determines which path to follow. This is repeated until the edge with which the algorithm started is reached. At this point, the resulting polygon is completed.

However, there can be several resulting polygons, so if there are intersections left that have not been visited yet, another resulting polygon has to be started and the whole process is repeated until that polygon is completed, too. If there are still intersections left which have not been reached, the process is repeated.

The result of this final step can be any number of simple polygons.

In the below example (figure 32a-32d), the clip polygon B is subtracted from the subject polygon A (*difference* operation). Each on the left side of the figures, the subject and clip polygons are shown. On the right side, the resulting polygons are shown. Since it is obvious in this case, the edges are not specially marked as *inside* or *outside* the other polygon.

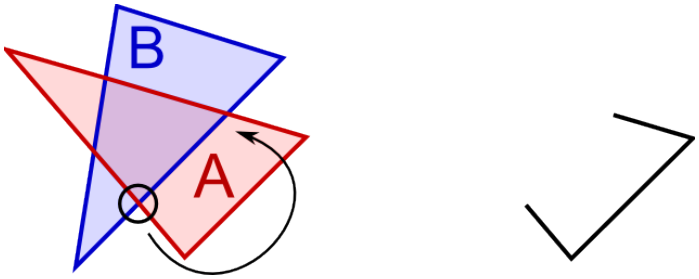

Figure 32a: The procedure starts at an intersection and follows the polygon border of A which is outside of the polygon B to the next intersection point. The edges on the way to that point are added to the resulting polygon (right side).

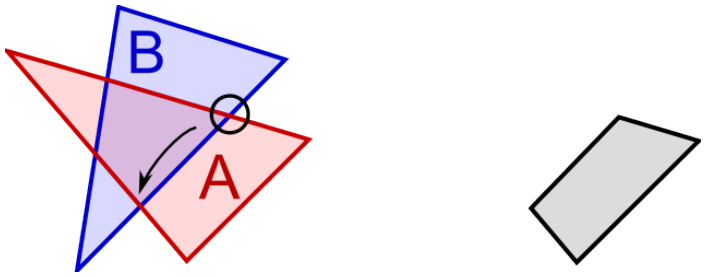

Figure 32b: On the next intersection, the border of polygon B is followed in reverse (clockwise) direction because it is inside polygon A and added to the resulting polygon. The next intersection is reached and one resulting polygon is complete.

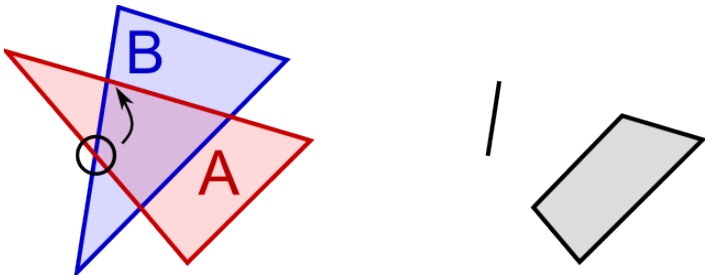

Figure 32c: Two intersections have not been reached yet. That is why the procedure starts anew at another intersection. A new resulting polygon is created and an edge is added to it.

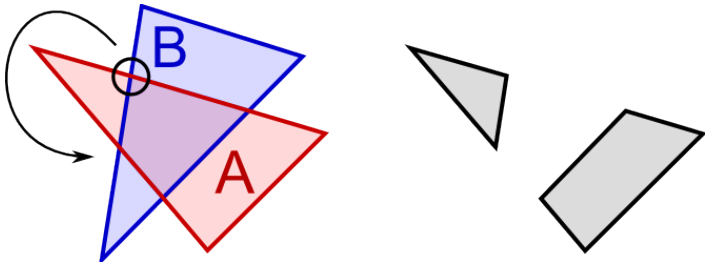

Figure 32d: Another intersection is reached which is the intersection with which the second resulting polygon has been started. It is completed and no intersections are left that have not been reached. The process is done.

3.5.2 Left hand rule

Normally, at one intersection, there is only exactly one path to progress along the polygon border – the other surrounding edges are out of question because they are tagged differently. However, as also mentioned by Schutte [S4, p.6], there are exceptions. The best example is any *exclusion* operation (XOR). In an exclusion, both *inside* and *outside* edges of both polygons are used to construct the resulting polygons. At every intersection, there are two paths for the procedure to choose. Figure 33 illustrates this:

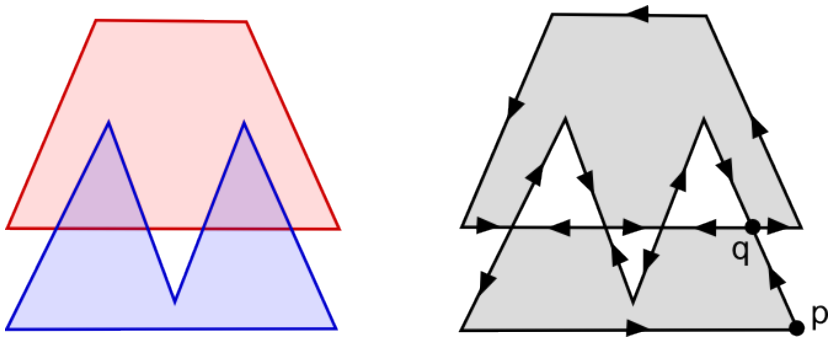

Figure 33: Exclude operation. Left side: subject and clip polygon, right side: resulting polygons with arrows to mark in which direction the edges are defined.

When coming from edge $\overrightarrow{pq}$, the resulting polygon could either be continued to be constructed by choosing to proceed with the left edge or with the right edge that face away from q. The rule here is to always choose the edge which takes the sharpest turn left (seen from the edge $\overrightarrow{pq}$, in this case) and thus make the resulting polygon as small as possible. If the direction is chosen arbitrarily, the resulting polygon could intersect itself at one of its vertices and thus would not be a simple polygon anymore. In some cases, it could even lead to erroneous results.

Note that there are intersections which can be gone through twice (all in the above example). So if a resulting polygon is completed, that intersection can still be used to construct the next resulting polygon. So, an intersection should not be marked as visited until all possible paths away from the intersection have been walked through by the construction algorithm.

3.6 Dealing with holes in polygons

During all clipping operations but *intersection*, there can arise holes in the resulting polygon(s). If the clipping algorithm outputs polygons that it cannot handle as input, the set of polygons it accepts would not be *closed* under the clipping operation. This is fatal if the clipping operation must be designed to be used over and over again on the same set of polygons – as it is the case in the application described in the introduction.

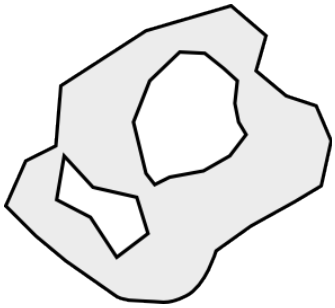

Figure 34: A polygon with two holes.

A hole for example arises in *difference* or *exclusion* clipping operations when a small polygon is subtracted from a big polygon that completely surrounds the small polygon without intersecting its border. This is already detected during the first step of the algorithm. If the two polygons in figure 33 were to be clipped with *union*, it would also result in a polygon with a hole.

Clearly, a polygon with a hole is not a simple polygon anymore, let alone *one* polygon. So either, the definition of what kind of polygons are accepted and outputted by the clipping algorithm must be changed, or the polygon with a hole must somehow be transformed into a simple polygon.

Whether a polygon is defined clockwise and thus is a hole can be detected by calculating the size of the polygon. If it is negative, it is a hole. Note that during one clipping operation, several holes can arise.

There are three approaches to deal with holes in polygons:

3.6.1 Negative sub-polygons

The first is to let every polygon define any number of holes inside itself. These holes would be represented by "negative" polygons which are defined in reverse (clockwise) order. See figure 35 for an example. This seems to be a swift solution because clockwise defined polygons are what the clipping operation outputs as holes. This approach is used by Peng et al. [P1]: they distinguish between the counter-clockwise defined outer contour and the clockwise defined inner contours of one polygon.

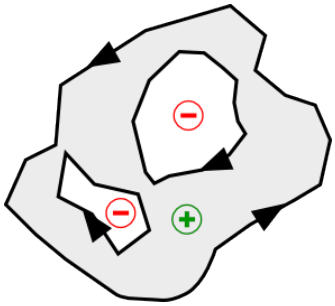

Figure 35: The arrows depict in which direction the polygon is defined. Clockwise defined polygons have a negative size.

However, this approach has the big disadvantage that these holes need to be accounted for not just in the clipping operation but in every single algorithm regarding the polygon because the holes also define the polygon's shape. While this doesn't necessarily make things slower, it just adds a lot of unnecessary complexity to the implementation (of any polygon-related algorithm).

Additionally, from the practical point of view, it is difficult to depict the polygon face (not the border, though) on the screen. So, the polygons would probably end up being split into simple polygons for the display operation anyway.

3.6.2 Weakly simple polygons with “pipes”

The second approach is the one that was originally conceived for that problem in this work: if all algorithms are designed to work not only with simple polygons but with weakly simple polygons, holes can be connected to the outer border of the polygon by a “pipe”: two edges on the same line with opposite directions (see figure 36). The depicted polygon does not qualify as a simple polygon anymore because its boundary touches itself. It then is a weakly simple polygon (see section 2.1).

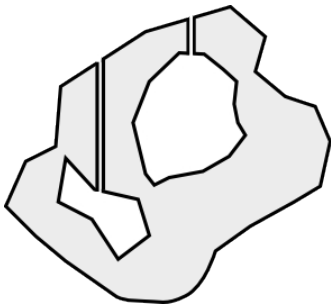

Figure 36: The holes are connected to the outer border through pipes. The width of the pipes in this illustration are magnified to depict their functionality. In reality, there is no space between the two edges that form the pipe.

A similar solution was also used by Sechrest and Greenberg [S3, p. 22] in their algorithm.

All algorithms presented in this work except for the clipping operations work with weakly simple polygons without any modifications. However, as it turned out, it added a lot of unnecessary complexity to the clipping algorithm for being able to claim it can handle *any* weakly simple polygon: any point of a weakly simple polygon can be part of any number of intersections with itself (*touching*, not *crossing*). To account for this in every step of the algorithm makes the algorithm a lot more complicated.

On the other hand, the trick with the pipe is not such a swift solution after all because it has to be determined first where to put it. The topology of the polygon does not always allow to just put two lines between the first vertex of the outer polygon border and the first vertex of the hole. Also, in most cases, new vertices have to be inserted into the polygon which could cause the polygon to change its shape in a way that self-intersections may arise (see section 3.3.1).

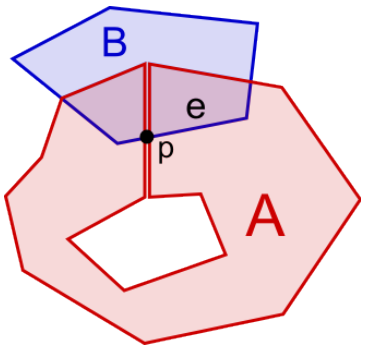

Figure 37: Two polygons intersect. One of them has a hole that is connected to its outer border through a pipe.

However, the most significant problem with this representation is that the method to determine the status of an edge at an intersection during the clipping operation presented in section 3.4.2 can fail for polygons with a pipe. See figure 37 for an example:

Polygon *B* intersects polygon *A* which has a hole that is connected to the outer border through a pipe. At the intersection point *p*, it is not possible to determine the status of the successive edge *e* in polygon *B* because the edge is both left of two edges (thus, inside) and right of two edges (thus, outside) around the intersection point of polygon *A*. To simply tag the successive edge *e* the same as the previous edge in polygon *B* works in this example but fails if all intersections between polygon *B* and polygon *A* are on the pipe. This case can not be excluded.

Because any weakly simple polygon can also be represented by several simple polygons, this approach turned out to be too complex while offering no advantages over the third approach:

3.6.3 Splitting into simple polygons

In this approach, the polygon is split into two (or more) polygons for each hole there is. In which direction the cuts are made, does not matter. Also the positions of the cuts do not matter as long as they go anywhere through their polygon face. See figure 38 for an example. This approach is also used in Schutte's [S4, p.8] algorithm.

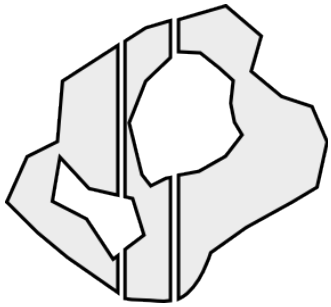

Figure 38: Again, the spaces between the three polygons are magnified to show that there are three polygons now. In reality, there is no space between them.

The process is the following: for every hole there is, the enclosing polygon (without the holes) is split into two. The cuts are made where the holes are located. Then, all holes are redefined to be defined counter-clockwise. The interim result is a set of adjacent polygons without the holes and a set of (positive) holes. After that, **each** hole is subtracted from **each** of the polygons (*difference* clipping operation).

Splitting a polygon into two is similar to a clipping operation but with the split line treated as a simple path (a simple polygon where the first and last vertex is not connected): The intersections of the polygon and the line are first calculated and memorized. Then, the new polygons are constructed along the outside edges of the polygon and the inside segments of the line. However, the split line needs to be handled differently in the second step of the clipping algorithm (edge labeling) since it doesn't define the area that is on its left side to be inside. Also, during the third step its segments may be used in both directions to create the resulting polygons as opposed to the edges of the polygon.

An easier but slower implementation is to create two split polygons that completely enclose the subject polygon from the split line. These two split polygons are then clipped with the subject polygon, using the *intersection* operation. See figure 39 for an illustration.

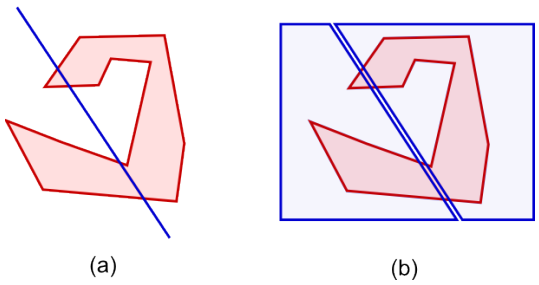

Figure 39: (a) The line along which the subject polygon should be split. (b) Two polygons that enclose the subject polygon run along the split line.

Splitting a polygon into two has a time complexity of O(n): For a normal clipping operation without the optimization using a scanbeam, $n \cdot m$ intersection checks have to be made (which is the most expensive operation in the algorithm). n is the number of edges in the subject polygon, m is the number of edges in the clip polygon. Since the number of edges of the clip polygon(s) is constant in both implementations mentioned, the time complexity only depends on the subject polygon.

4 Spatial data structures and algorithms

In the previous two chapters it was shown that the performance of any operation or check on a polygon is solely dependent on the number of edges – neither its length nor its overall size. Also, the polygon clipping is the most expensive operation as it runs in quadratic time (without the optimization using a scanbeam) while all other checks run in linear time.

Now, for the application of a polygon-based landscape, not single polygons or a set of two polygons are being dealt with but with a landscape that consists of a large number of polygons. For example, if a point-in-polygon check is performed on the landscape, one does not want to know whether the point is in one specific polygon; one wants to know in which of the possibly thousands of polygons it is in (if it is in any). However, simply performing the point-in-polygon check for every polygon there is, is highly inefficient. The same applies to intersection checks with rays and for clipping operations.

So the goal of the spatial data structure in which the polygons are stored must be to efficiently exclude polygons that are out of range of the point, line or polygon for which a check or operation is made. Also, since the performance of an operation is solely dependent on the number of edges, single polygons should be kept small (in number of edges, not necessarily in size) and compact. It is also desirable that operations and checks on polygons take about the same time anywhere on the landscape in order to maintain an even processor load during runtime.

Algorithms for the polygon clipping presented in the previous section have been optimized to be used one-time on a set of input polygons. The output polygons were normally not used as input for further clipping operations. Specifically, the time t_{init} needed to process the data before the algorithm and the time t_{algo} to actually perform operations on the polygon(s) have not been separated because it has been assumed that the algorithms are only applied once on the same polygons anyway. However, for the application in games, the initial loading time is far less important than the performance at runtime. So, if any calculations can be done initially (before the actual operations take place), this will result in a speed gain every time an operation on the landscape is performed.

One example for work that can be done at loading time is to sort the vertices of each polygon if the *scanbeam* algorithm is used. This is otherwise done directly before the clipping operation. If the vertices of each polygon are sorted initially, the sorted lists of the subject and clip polygon just need to be merged before the actual clip operation. Merging two such sorted lists only takes linear time. If a sorted list of vertices is already available, the scanbeam technique can also be used in the point-in-polygon check, given that the ray cast (see section 3.3.2) is chosen to be orthogonal to the scanbeam.

Much research has been done on the field of spatial data structures and algorithms as they proved indispensable for the management of large amounts of data in fields such as database management systems, computer graphics, computational geometry and geographic information systems.

Due to the magnitude of this field of research, only an overview over the data structures that are relevant for this application can be given in this work. Specifically, the implementation and its affiliated issues and optimizations will not be discussed in detail.

A more in-depth description and comparison over the different spatial data structures with a focus on hierarchical representations has been given by Samet [S5, S6] as well as Gaede and Günther [G2].

4.1 Bounding box

A simple and common way to quickly determine if a given point, polygon or line is outside another polygon is to use a *bounding box*. Sutherland et al. have described this method as the *minimax test* [S1, p. 12-13] which quickly rejects polygon faces that do not overlap each other.

The bounding box of a given polygon is the smallest rectangle parallel to the axes that can be drawn around it. See figure 40 for an example. The rectangle can be constructed by choosing the x-coordinate of the leftmost vertex as the left border, the x-coordinate of the rightmost vertex as the right border, the y-coordinate of the topmost vertex as the upper border and y-coordinate of the bottommost vertex as the lower border of the rectangle.

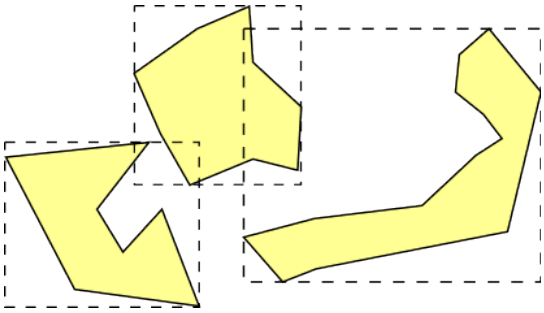

Figure 40: A few polygons with their bounding boxes (in dashed lines)

Since a rectangle has a much simpler shape than a (simple) polygon, it is faster to first check if the given point is inside the bounding box (for a point-in-polygon check), the given line intersects the bounding box (for an intersection with a line) or the bounding box of the given polygon overlaps with this bounding box (for a clipping operation) before proceeding with the algorithm.

The more box-shaped a polygon, the more time is saved by using this technique. So if a polygon is spread out very far, the bounding box is less effective. However, it is possible to actively modify a polygon:

4.2 Splitting up a polygon

A polygon is a boundary-based representation of a region. If the boundary of a polygon is only used to define the region it encloses, this region can also be represented by any number of smaller polygons, as depicted in figure 41.

Thus, if a given polygon spreads out too far or grows too big (as in number of edges), it can always be split up into several smaller polygons. This does not reduce the total number of edges but still makes operations on that region faster: the bounding boxes more closely resemble the actual shape of the region enclosed by the polygons, therefore more space can be disregarded as outside the region by looking at the bounding boxes. Additionally, operations that only affect parts of the region enclosed by the polygon have to work with fewer edges.

An example: To figure out whether the points p and q in figure 41 (a) lie inside or outside the polygon A, a full point-in-polygon check has to be made for both. In 41 (b), the point p can be discarded (as outside the region) after it has been determined that it is not in any of the bounding boxes of the polygons in B. For point q, an actual point-in-polygon check (after checking all bounding boxes of B) is only made for polygon f which consists of much fewer edges than polygon A.

Since the bounding box is an axis-aligned rectangle, it makes sense to make the cuts which split the polygon *A* into multiple polygons along the axes. Apart from that, it can also make sense to systematically split up the simple polygons into convex polygons. As described in the sections 2.4.1 and 2.5.2, both the point-in-polygon check and the intersection check with a line is faster with convex polygons.

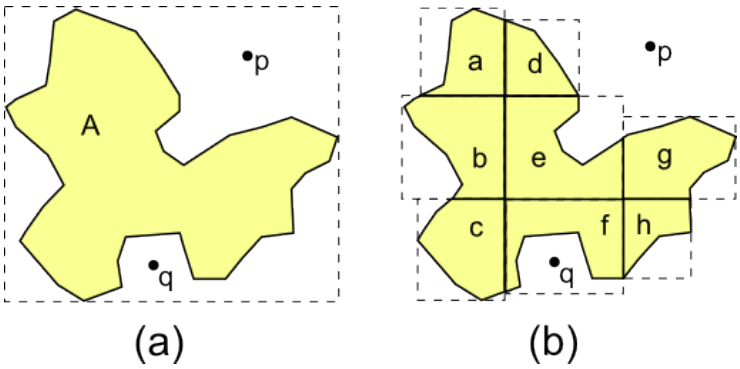

Figure 41: Polygon A and the set of polygons B= {a,b,c,d,e,f,g,h} are both representations of the same region. The bounding boxes are shown as dashed lines. Also depicted are the points p and q.

The process of splitting up polygons is the same as described in section 3.6.3.

Of course, the smaller the pieces into which the polygons are cut, the more polygons are created. If there are many small polygons instead of few big polygons in the landscape, the main goal in order to increase performance shifts from optimizing operations on single polygons to optimizing the traversal through the bounding boxes of polygons.

4.3 Scanbeam algorithm

As described by Hsiao and Tsai [H1], the *scanbeam* (or: *plane-sweep*) paradigm can be used on the level of sets of polygons, too. The algorithm works similar as described in section 3.3.2, only that instead of the edges of a polygon, the bounding boxes of a set of polygons are traversed: Given that the scanbeam moves vertically from top to bottom, the upper and lower edges of all bounding boxes must be sorted by their y-coordinates first. During the algorithm a bounding box is added to the list of active rectangles when its upper edge is reached and removed again when its lower edge is reached. If new polygons appear during the operation (for example if a polygon has been clipped), the list of sorted edges needs to be updated.

Hsiao and Tsai added that for most practical problems the plane-sweep algorithm alone is not efficient enough and should be used in conjunction with a spatial data structure. The reason is that the algorithm needs O(n log n) time to complete, where *n* is the number of polygons. Even if the upper and lower edges of all bounding boxes were already assumed to be pre-sorted, the algorithm would still have a time complexity that is linear to *n*, the number of polygons, since all polygons need to be traversed for any operation. This does not scale well.

Additionally, it must be assumed that the number of polygons in the landscape changes constantly during runtime: new polygons are added, some others are clipped and others removed. Any data structure that uses the scanbeam algorithm on this level needs to re-sort the list of bounding boxes on any of these events.

4.4 Grid

A simple and efficient solution is to arrange the polygons in a uniform grid. Each cell in the grid contains a set of links to polygons that intersect with it. This set can also be empty if there is no polygon in the grid cell. If a polygon changes its shape due to a clipping operation or is removed altogether the links to that polygon need to be updated in the data structure. To efficiently implement this, each polygon also needs to link back to any grid cells it is linked from.

If the grid is implemented as an array, the access to one cell has an unbeatable time complexity of O(1). So, to perform an operation on a region only depends on the number of polygons in the grid cells that intersect with that region, but not on the overall number of polygons.

On initialization the density of the grid (the size of the cells) has to be defined. The smaller a cell, the less polygons it intersects. This makes accessing polygons at random positions more efficient. However, the smaller a cell, the more cells there are and the more cells link to one and the same polygon which in turn needs to link back to the cells (see above). This does not only lead to a higher memory usage but also slows down any operation that alters the landscape because the references between the altered polygons and the grid need to be updated. The more cells and polygons that need to be updated, the longer the process takes: if a polygon that spans over 10 x 10 cells is deleted, 100 references to the polygon need to be deleted from the grid cells (as well as 100 references to the grid cells in the polygon).

Determining an optimal grid density is a difficult task because the polygons can be very different in size and in number of edges at different areas of the landscape. Regardless of which cell size has been defined, some polygons might span over dozens of cells while others might share one cell with dozens of other polygons. So in areas with big polygons the cells should be larger, whereas in areas where many small (and detailed) polygons are crammed together, the cells should be smaller. Figure 42 shows an example of this arrangement:

Polygon *A* spans over four grid cells while for example the upper right cell (of the ones *A* is in) only intersects with two polygons. However the cell in which polygon *B* is contained in links to eleven polygons, most of which have a lot of edges compared to their size. Thus, one problem of this data structure is that the cells in the grid have a static size. It is not possible to define different sizes for the cells in different areas of the landscape.

Also, once the density of the grid has been defined, it can't be changed anymore without completely rebuilding the data structure. This can become a problem because it can lead to notable differences in performance on the access on different areas of the landscape. This is undesirable for an application that should run in real-time, especially if one keeps in mind that during runtime, the landscape can change continuously: If the landscape is clipped in the same area with different polygons repeatedly, the total number of edges in that area tends to increase because the boundaries of the polygons get more detailed (as in short but numerous edges).

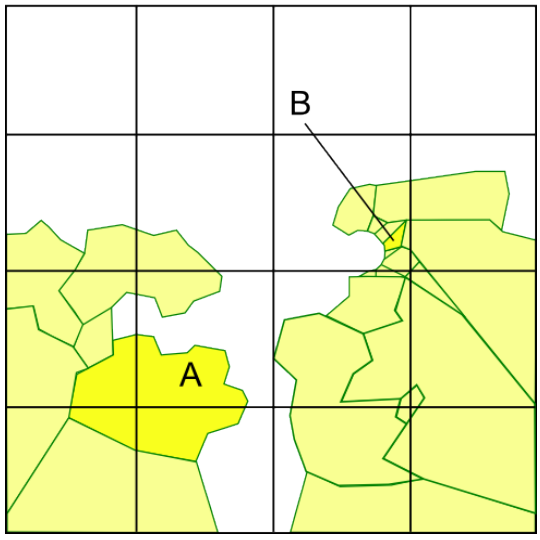

Figure 42: A two dimensional landscape shown with a uniform grid. Two polygons (A and B) are highlighted.

Since any polygon could be referenced by several grid cells, it must be ensured that the same polygon is not included twice in operations that deal with the polygons of more than one cell (like an intersection check with a line or a clipping operation).

For this reason, it makes sense to cut the polygons along the grid as shown in figure 43. After doing this, no two cells in the grid link to the same polygon anymore. Thus, one cell does not link to all polygons that *intersect* with it but it links to all polygons that are *contained* in it which effectively makes a grid cell a bounding box of the (set of) polygons it contains.

However, this does not solve the problem of the static grid size.

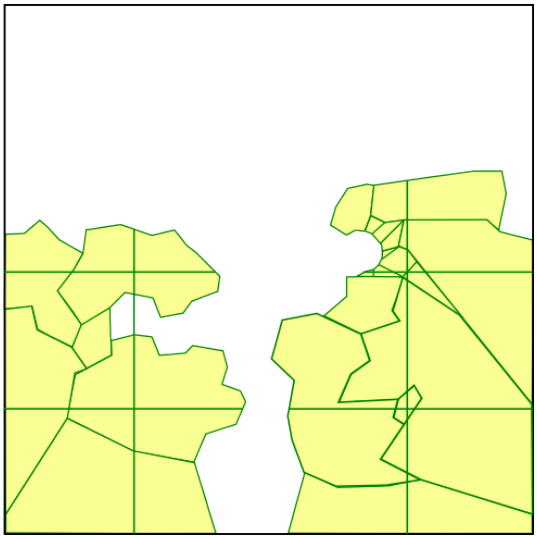

Figure 43:The polygon landscape split up on the grid

4.5 Trees

A tree is a hierarchical data structure which can solve this problem. Specifically, trees that partition a two dimensional space recursively into smaller regions overcome the limit of the static cell size in a grid.

4.5.1 Storing point data

A *quadtree* is a tree data structure in which each node that is no leaf has four children. A leaf node has no children and represents a cell in this context, thus holds the spatial data. When a previously defined maximum capacity of a leaf has been reached, its underlying space is split up into four new cells - one to the upper left (also referred as northwest), upper right (northeast), lower left (southwest) and lower right (southeast). The former leaf node now has four children by himself. If data inside the cells is deleted, this can of course also be done in reverse so that the tree always adapts to changes in the landscape.

Because of this property, quadtrees are eligible to represent cells that are variable in size and thus different densities throughout the landscape. However, the average time complexity to access a specific cell in a quadtree is usually O(log n) (where n is the number of elements stored in it) as opposed to the constant time complexity in a grid.

Originally, Finkel and Bentley introduced quadtrees as a data structure that stores two dimensional point data, *point quadtrees* [B1]. In their approach, each node itself stores one point which at the same time defines the center of the space underlying that node. The vertical and horizontal division lines are drawn from that center. A quadtree which always decomposes its underlying space into cells of equal size and thus only stores the point data in its leafs has been described as a *PR quadtree* by Hamet [S5] (P stands for point, R for region). The difference between these two quadtrees is shown in figures 44 and 45.

As the structure of a PR quadtree is not dependent on the location of the points inside of it, it is more eligible to work with data that changes continuously even though it tends to have a higher depth than the point quadtree with the same maximum capacity.

The *kD tree* which holds point data (*2D tree* for two dimensional data) has been introduced by Bentley [B2]. Like the quadtree, it is also based on a recursive partition of space. However, for each level in the tree, the space is only subdivided into two sides. So, per level, each one dimension is cycled through: For example, in a *2D tree* the plane is split into two along the x-axis on the first level, along the y-axis on the second level, again along the x-axis on the third level etc.

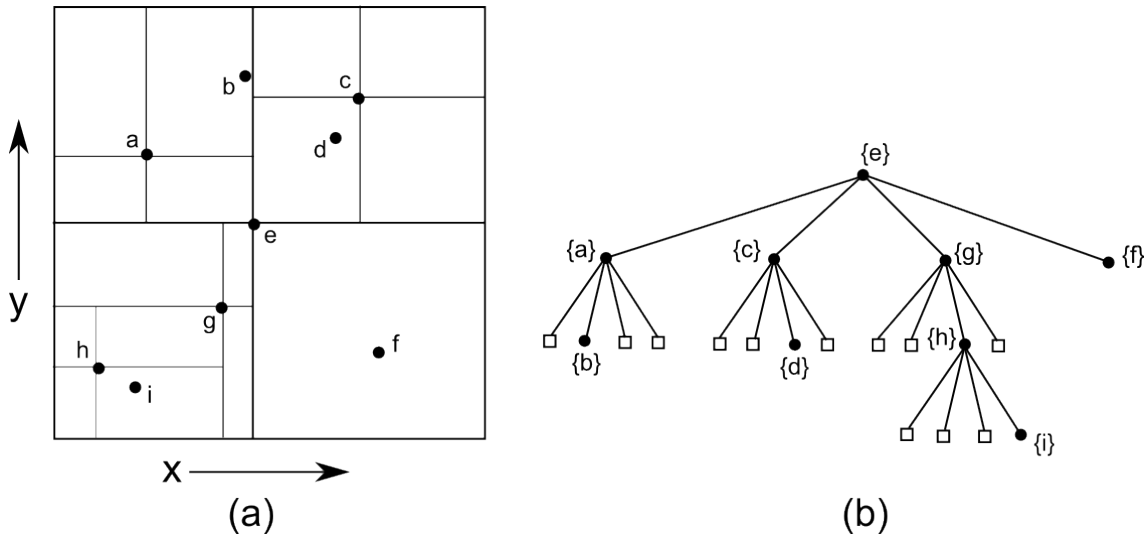

Figure 44: A point quadtree with some point data. The maximum capacity of a cell in this example is 1. (a) shows the partition of space and (b) the representation of the points in the tree. The bordered squares stand for empty leaf nodes.

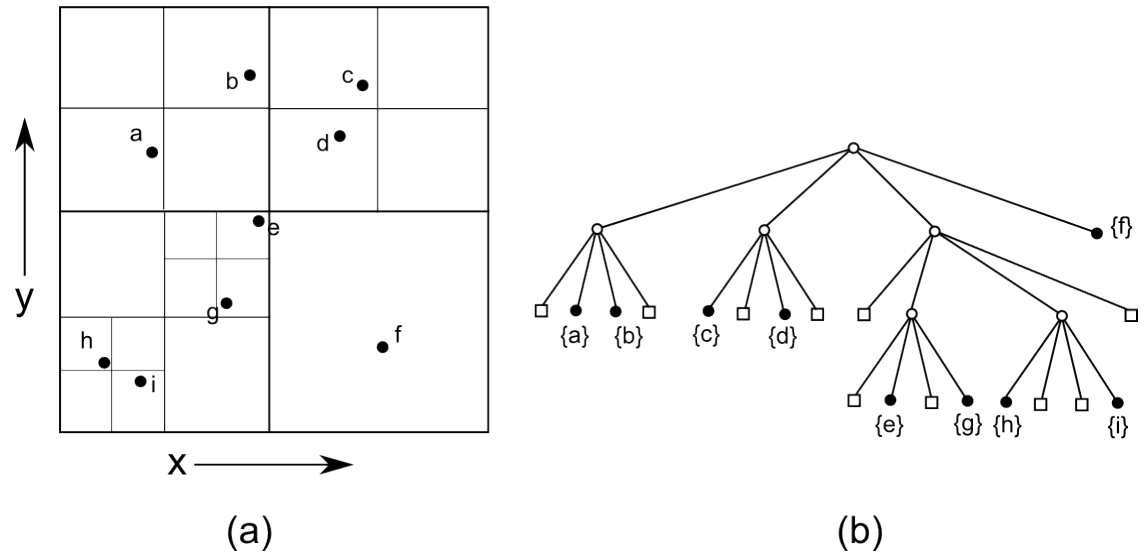

Figure 45: A PR quadtree with the same point data. The maximum capacity of a cell in this example is 1. (a) shows the partition of space and (b) the representation of the points in the tree. The bordered squares and circles stand for empty (leaf) nodes.

A *kD* tree is less wide but usually has a higher depth than a quadtree. Rosenberg compared the memory usage and overall performance of a *kD* tree with that of a quadtree [R3]. He concluded that it is more expensive in terms of memory (but only by a constant factor) and, more importantly, outperforms the quadtree at region searching and particularly in point searches (less nodes are visited). The average time complexity of a kD tree is the same as for a quadtree.

Like the point quadtree, the *kD tree* has been designed to store a point in each node and perform the subdivision at that point. So, a *PR kD tree* is a *kD tree* in which each node splits the space into two evenly sized sides as shown in figure 46.

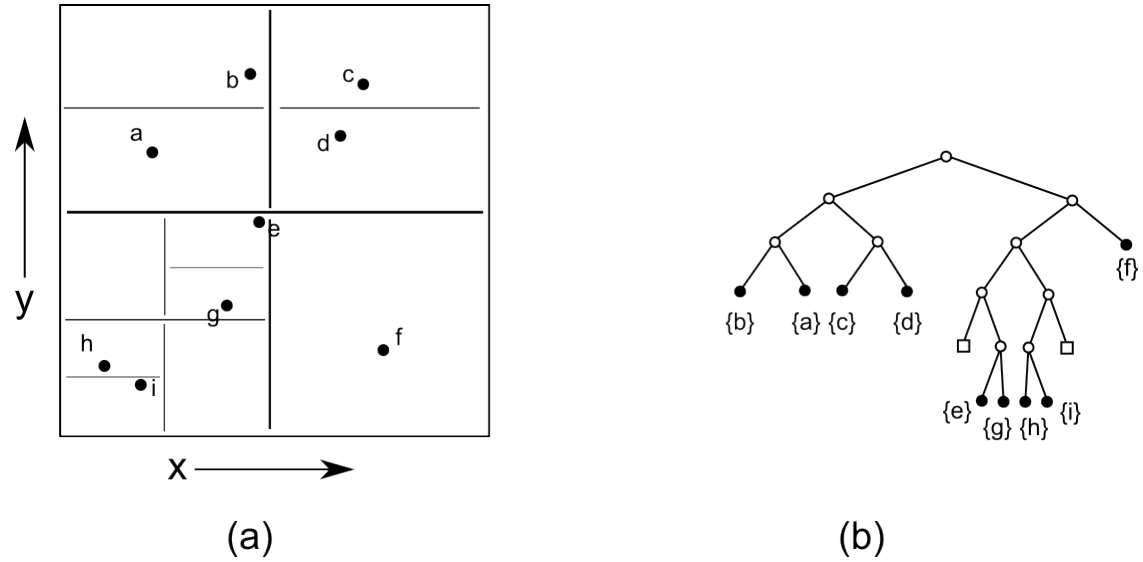

Figure 46: A PR 2-d tree with again the same point data and a maximum capacity of a cell of 1. (a) shows the partition of space and (b) the representation in the tree. The bordered squares and circles stand for empty (leaf) nodes.

4.5.2 Storing rectangle data

So far, only tree data structures that store point data, not rectangle data, have been described. However, both presented PR trees can be generalized to also be able to store rectangle data: Each node of a PR tree that stores rectangles contains a set of rectangles it *fully contains*. Figure 47 shows an example for a PR quadtree that stores rectangles.

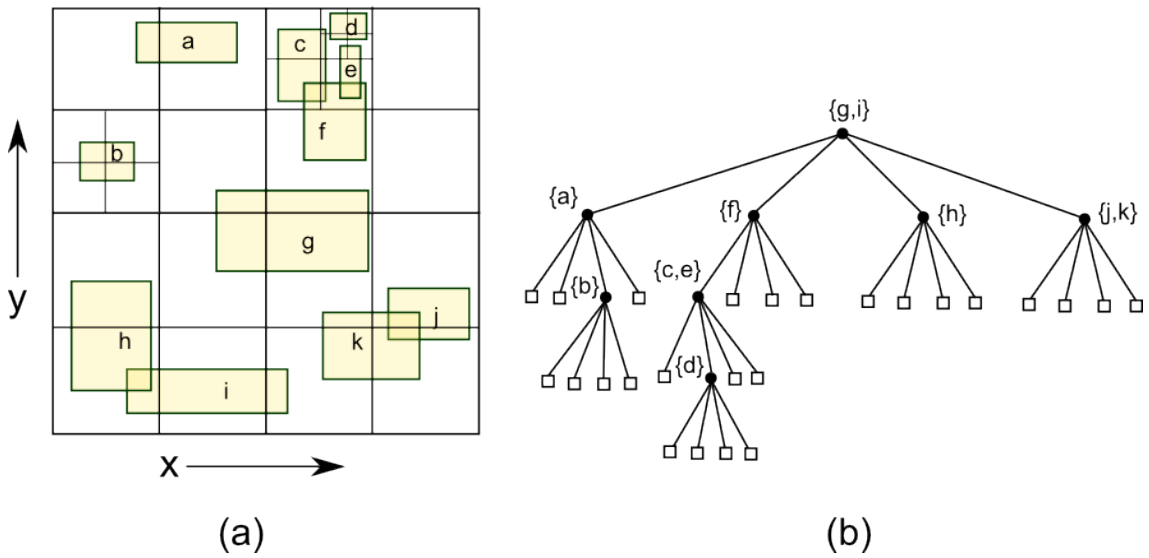

Figure 47: A PR quadtree used for rectangle data. (a) shows the partition of space while (b) shows the representation in the quadtree. The bordered squares stand for nodes with no rectangles inside.

The lower a rectangle can be sorted into the tree, the more rectangles can be excluded from a point or region query. This must be the goal of the data structure. This is why the region in the above example is split up further until no rectangle can be sorted at an even lower position into the tree.

This data structure has a problem though: all rectangles that intersect with any of the division lines of the cells have to be checked, even if they are far away from the actually queried region. If for example a point-in-polygon check is performed on the region around rectangle *d* in figure 47, the rectangles *g*, *i*, *f*, *c* and *e* have to be checked for containment of the point because these are in the upper nodes of the tree. Also, at which level the rectangles are sorted into the PR tree is not dependent on their size but on their position. More precisely, the level of the node that defines the division line the rectangle intersects with.

Figure 48 illustrates this problem: polygon A will be sorted into a node that is a direct child of the root node, despite its small size.

Because a PR kD tree only splits the plane in half per level, the rectangles are sorted into the tree a little bit deeper and thus it does a slightly better job at excluding rectangles that are not in the queried region. Polygon *i* from figure 47 for example would be stored in the second level of the tree instead of in the root node if the plane were first split along the x-axis.

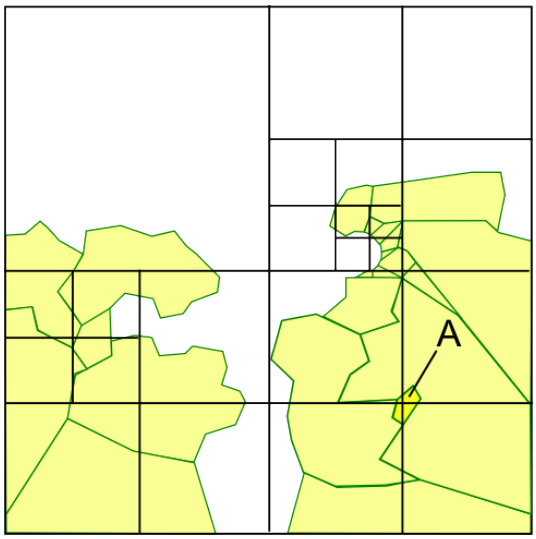

Figure 48: A two dimensional landscape shown with a quadtree. Polygon A is highlighted.

Kedem introduced the q*uad CIF tree* in the context of VLSI (very large scale integration) technology [K1] in which he offered a solution to partly tackle this problem. The quad CIF tree is a quadtree that functions the same way as the PR quadtree described above with the only exception that the rectangles which are completely contained in a cell are not saved in a list (or set) but in two binary trees (also called *bisector lists*): roughly, one binary tree with all the rectangles that intersect with the node's horizontal division line sorted by x-coordinates and one binary tree with all the rectangles that intersect with the node's vertical division line sorted by y-coordinates. This data structure makes it faster to traverse through and discard the excess rectangles. Lai et al. described a similar approach for *2D trees* (which they call *HV/VH trees*) [L2].

Another method is to sort the rectangles of each node by their coordinates and use the scanbeam algorithm on this level to traverse through them faster. However, both methods don't solve the root of the problem: that rectangles are sorted into upper levels of the tree because of their position.

So, if the preconditions mentioned in section 3.6.3 are met, a final solution for this problem (in this application) is to split the polygon up along the division line(s) of a node just like it is done in the grid. Each node and with it all polygons inside of it are split up if a previously defined maximum number of edges is reached. This ensures that the size of a cell automatically adapts to the state of the landscape. To access a given cell in the landscape takes about the same time anywhere on the landscape. Thus, the overall performance of a point query within the landscape can be reduced to the traversal through the PR tree that has a depth of O(log n) plus a constant factor for the access on a certain cell. This helps to meet the requirement to maintain an even processor load during real-time usage of the landscape.

Of course, this is still combinable with any data structure that makes the traversal through the bounding boxes in one cell faster.

Even though the access to single cells in a grid is faster than with a tree, the cell size in a grid is static and thus there is no upper limit for how long the operation *within* a cell will take. This is why the usage of a tree should be preferred in this application for its ability to dynamically adjust the size of each cell; it guarantees that for any given cell, the operation within it will take a constant amount of time.

If the polygons in the game world are supposed to describe more than their enclosed regions for example if the polygons ought to be subjected to gravity or generally be able to move as a whole, they can't be split up arbitrarily. One solution would be to split them up nevertheless but let the pieces of one chunk that ought to stay together during movement link to each other. Another solution is to store polygons which should be able move in another data structure than the rest of the landscape.

Possible for these polygons are also bucketing-methods like the *R-Tree* and derivatives which have not been described in this work. See for example Samet [S5] for an explanation.

5 Conclusion

To create a working polygon-based implementation of a *changeable* 2D landscape for games is not a trivial task. Specifically the implementation of a clipping algorithm (see chapter 3) that is also applicable for games adds more complexity because of the additional requirements mentioned in the introduction, namely to work with integers and to return the same class of polygons that it accepts.

Every possible output of a clipping operation needs to be accepted by any other clipping operation as input. This requires the clipping algorithm to both accept any combination of (weakly) simple polygons as well as to fix up any result that is not initially in the set of simple polygons. As it turned out, there are many edge cases that need to be considered to fulfill this requirement. A clipping operation that for example only accepts simple polygons but in some cases outputs certain weakly simple polygons is not acceptable. Additionally, the clipping operation needs to function correctly with the limited precision of integers. The usage of fixed point numbers instead of floating point numbers revealed a number of problems that already existed for previous implementations based on floating point numbers but were probably overlooked or deemed insignificant (see section 3.3.1).

In an implementation based on integers, it becomes likely that the two polygons intersect each other exactly at the end points of one of their edges as well as that the polygons intersect in (several) line segments instead of just points. Additionally, edges whose midpoint can not be calculated, like for example edges with a length of just one unit one, can become a problem since it is not reliable to determine their containment in the other polygon via the point-in-polygon check (see chapter 3.4). The algorithm presented in section 3.4.2 solves this problem but only for polygons which do not touch themselves at lines, like simple polygons (cf. section 3.6.2). Another important issue is that the shape of the polygon can be altered by *any insertion* of a new vertex which can theoretically lead to *any* number of new intersections between the polygons but also within the polygon to arise. This issue has been discussed and solved in section 3.3.1. The removal of holes in the set of resulting polygons explained in section 3.6.3 again requires the usage of clipping operations, both in order to split the polygons along a line and to subtract the holes from those polygons. This too makes the clipping operation more complex and more expensive because theoretically *any number* of holes can arise during a clipping operation.

In chapter 2, the basic algorithms and checks on polygons like the point-in-polygon check, the calculation of the size or the intersection of a polygon with a line have been described. These algorithms are essential for any implementation of a two dimensional landscape based on polygons, independent of whether the landscape should be changeable or not. Also, the use of spacial data structures which have been discussed in chapter 4 are necessary for any landscape that is supposed to run real-time and consists of more than a couple of polygons.

To summarize, this thesis offers an approach how to implement a polygon-based landscape that can also be changed during the course of the game. This makes it possible to design games that have destructible landscapes while still having the benefits of edge-based landscapes over interior-based landscapes. Of course, the implementation is not limited to be used in games since the employment in games must fulfill additional requirements than previous edge labeling approaches for polygon clipping, not less.

Thus, the implementation can be used in any other area in which polygon clipping algorithms have been used so far and additionally in areas in which the same set of polygons might be continuously clipped (in real-time).

Regarding its application in games, the implementation is only an alternative approach to the pixel-based landscape, not a replacement.

The reason is that a pixel-based landscape is still unbeatable in performance for what it is designed for: small but continuous modifications on the landscape. The time complexity to change areas in a pixel-based landscape behaves linear to the number of pixels that have to be changed and is relatively[7] independent of the actual size of the whole landscape. To subtract a single pixel from the landscape only involves changing a single value in the two-dimensional array, thus runs in O(1). To change a single pixel (or, in this case, a tiny square) from a polygon-based landscape has the same time complexity as any other clipping operation as it is not dependent on the size of area involved but the total number of polygons in the landscape as well as the number of edges of the polygons actually involved.

Because the costs to change an area in a pixel-based landscape grows quadratically to its radius in pixels ($A = \pi r^2$), the same operation on a polygon-based landscape can actually be faster for very large areas at some point because the number of edges of a polygon is linear to the area it encloses ($P = 2\pi r$). However, the normal use case of making changes to the landscape in a game involves the continuous subtraction of relatively small areas like digging through the earth or making explosions. For this thesis not tests have been commenced to find out how big an area would have to be that a clipping operation runs faster in a polygon-based landscape as it was beyond the scope of this work. Though, due to the complexity of a single clipping operation, it is assumed that this would involve areas which are much bigger than anything actually used in games.

Typical simple checks made on the landscape in a game are, amongst others, collision and containment checks to find out if a moving object collides with the landscape and whether it is contained in a (solid) material.

A collision check in a polygon-based landscape is done by intersecting the current movement vector of the object with the polygons in the landscape (see section 2.4) and is thus also dependent on the total number of polygons in the landscape as well as the number of edges of the polygons actually involved. If the line segment intersects with a polygon, the object collided. A collision check in a pixel-based landscape works by checking for each pixel between the current position of the object and the position plus the movement vector if it is a solid material. If there is solid material, the object collided at that point. This check is only dependent on the length of the current movement vector in pixels: if an object only moves 3 pixels in a time step, the two-dimensional array only needs to be accessed 3 times for a complete collision check. Thus, a collision check is much faster in a pixel-based landscape than in a polygon-based landscape for reasonable speeds of objects. The same applies for a containment check. For a polygon-based landscape, a point-in-polygon check needs to be done on the landscape (see section 2.5) while for the pixel-based landscape, only a single value in the array needs to be accessed.

However, the primary reason for a landscape based on polygons is not the performance of these operations. As mentioned in the introduction, modern physics engines only work with geometric shapes like polygons and thus require the landscape to be defined as an edge-based representation. This is because a physics engine does not only simulate collisions and does containment checks. The size, mass and shape of an object must be known to simulate pressure, friction, its kinetic energy and generally forces that act on it.

To calculate the surface at one spot is much simpler with a polygon-based landscape as it is already edge-based while the pixel-based landscape is interior-based. The surface (= edge in this case) in a pixel-based landscape would have to be searched by a (local) edge-detection algorithm first while it is already available in the polygon-based landscape. The size of a certain

7 given that the whole landscape fits in the memory so that no page-faults occur

chunk of material is simply calculated by the Gaussian trapezoid formula (see section 2.2) in a polygon-based landscape. In a pixel-based landscape there is not even a clear notion of an interconnected material chunk. The edges of the said chunk would have to be searched for first, then, every pixel in that chunk would have to be counted separately.

Additionally, it is next to impossible in a pixel-based landscape to actually include the landscape itself in the physics simulation. For example, to make certain parts of the landscape fall off, move around, float on water and especially rotate is simply not possible in the data structure of a two-dimensional array because it is static, like a raster graphic. So, compared to the pixel-based landscape, the polygon-based landscape adds new possibilities to make the landscape more dynamic. After all, this is one of the reasons why many modern platform games don't use the tile-based approach for a landscape anymore.

This thesis is about a polygon-based landscape with the extension of being able to change it as an alternative to a pixel-based landscape. However, no direct comparison in performance regarding both the modifications and the checks on the landscape between the polygon-based and the pixel-based landscape has been made. So the assumptions about the performance made in this conclusion are only based on time complexity of these algorithms. It would be interesting to know how the two different approaches perform in direct comparison but as mentioned above, this would have gone beyond the scope of this bachelor thesis.

Also, the optimizations in performance using spatial data structures have not been covered in detail, especially in respect to the typical operations done in physics simulations, including for example fluid simulation. However, this already reaches into the broad subject of physics simulations in computers in general. This is another topic that could be dealt with in future works.

Apart from the benefits of polygons in physics simulations, there are also a few advantages regarding graphics. Like a vector graphic, any edge-based landscape can be zoomed in without limitations since the level of detail of the landscape is not chained to the resolution of the screen. That is why an edge-based landscape is especially eligible for games that otherwise use 3D graphics (as these are also based on polygons). A pixel-based landscape gets blurry or pixelated if zoomed in, like a raster graphic. Also, it should be noted that modern hardware is optimized to render polygons. While it is not a problem at all to render the whole pixel-based landscape as a texture, a polygon-based landscape adds the possibility to render it more dynamically. As the example *Klonoa: Door to Phantomile* (see figure 4) shows, the game landscape is not limited to be displayed orthogonal to the camera viewing direction. Furthermore, it is possible to use vertex and geometry shaders on the polygons or add certain special effects to specific material chunks.

To summarize, a landscape based on polygons is more versatile than a landscape based on a two-dimensional array, but at the cost of complexity in implementation and performance.

References

[B1] R. A. Finkel and J. L. Bentley. 1974. Quad Trees A Data Structure for Retrieval on Composite Keys. *Acta Informatica 4*, 1-9 (1974)

[B2] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (September 1975), 509-517

[D1] Revar Desmera. 2009. Boolean Areal Geometry. http://revar.livejournal.com/81627.html (last visited: January 2011)

[F1] James D. Foley and Andries Van Dam. 1982. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[G1] Günther Greiner and Kai Hormann. 1998. Efficient clipping of arbitrary polygons. *ACM Trans. Graph.* 17, 2 (April 1998), 71-83

[G2] Volker Gaede and Oliver Günther. 1998. Multidimensional access methods. *ACM Comput. Surv.* 30, 2 (June 1998), 170-231.

[H1] Pei-Yung Hsiao and Chia-Chun Tsai. 1990. A new plane-sweep algorithm based on spatial data structure for overlapped rectangles in 2-D plane. *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International* , vol., no., pp.347-352, 31 Oct-2 Nov 1990

[K1] Gershon Kedem. 1982. The quad-CIF tree: A data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference* (DAC '82). IEEE Press, Piscataway, NJ, USA, 352-357.

[K2] David W. Krumme, Eynat Rafalin, Diane L. Souvaine, and Csaba D. Tóth. 2006. Tight bounds for connecting sites across barriers. In *Proceedings of the twenty-second annual symposium on Computational geometry* (SCG '06). ACM, New York, NY, USA, 439-448

[L1] You-Dong Liang and Brian A. Barsky. 1983. An analysis and algorithm for polygon clipping. *Commun. ACM* 26, 11 (November 1983), 868-877

[L2] Lai, G.G., Fussell, D. and Wong, D.F. 1993. HV/VH Trees: A New Spatial Data Structure for Fast Region Queries. *Design Automation, 1993. 30th Conference on* , vol., no., pp. 43- 47, 14-18 June 1993

[M1] Francisco Martínez, Antonio Jesús Rueda, and Francisco Ramón Feito. 2009. A new algorithm for computing Boolean operations on polygons. *Comput. Geosci.* 35, 6 (June 2009), 1177-1185

[P1] Yu Peng, Jun-Hai Yong, Wei-Ming Dong, Hui Zhang, and Jia-Guang Sun. 2005. Technical section: A new algorithm for Boolean operations on general polygons. *Computers and Graphics* 29, 1 (February 2005), 57-70

[R1] Ari Rappoport. 1991. An efficient algorithm for line and polygon clipping. *The Visual Computer* 7 (1991), 19-28

[R2] M. Rivero, F. R. Feito. 2000. Boolean operations on general planar polygons, *Computers & Graphics*, Volume 24, Issue 6 (December 2000), 881-896

[R3] Rosenberg, J.B. January 1985. Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.4, no.1, pp. 53- 67

[S1] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. 1974. A Characterization of Ten Hidden-Surface Algorithms. *ACM Comput. Surv.* 6, 1 (March 1974), 1-55

[S2] Ivan E. Sutherland and Gary W. Hodgman. 1974. Reentrant polygon clipping. *Commun. ACM* 17, 1 (January 1974), 32-42

[S3] Stuart Sechrest and Donald P. Greenberg. 1981. A visible polygon reconstruction algorithm. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '81). ACM, New York, NY, USA, 17-27

[S4] Klamer Schutte. 1995. An Edge Labeling Approach to Concave Polygon Clipping. (unpublished)

[S5] Hanan Samet. 2010. Multidimensional data structures for spatial applications. In *Algorithms and theory of computation handbook* (2 ed.), Mikhail J. Atallah and Marina Blanton (Eds.). Chapman & Hall/CRC 6-6.

[S6] Hanan Samet. 1988. Hierarchical representations of collections of small rectangles. *ACM Comput. Surv.* 20, 4 (December 1988), 271-309.

[V1] Bala R. Vatti. 1992. A generic solution to polygon clipping. *Commun. ACM* 35, 7 (July 1992), 56-63

Affirmation

I hereby assure that I wrote this bachelor thesis independently. No other than the given aids have been used. I further assure that this work has not been submitted to any other examination procedure before and that both the version on paper and the version on the electronic storage medium are the same. I also agree to the inclusion of this work into the library of the department of Informatics, University of Hamburg.

_____

Place, Date, Signature