17th International Conference on the Applications of Computer
Science and Mathematics in Architecture and Civil Engineering
K. Gürlebeck and C. Könke (eds.)
Weimar, Germany, 12–14 July 2006

AUTOMATING PRELIMINARY COLUMN FORCE CALCULATIONS IN MULTI-STOREY BUILDINGS

E. Lourens[*], **G. van Rooyen**

[*]*University of Stellenbosch, South Africa*
E-mail: elizmari@gmail.com

Keywords: Column Forces, Building Design, Computing, Multi-storey Buildings.

Abstract. *In civil engineering practice, values of column forces are often required before any detailed analysis of the structure has been performed. One of the reasons for this arises from the fast-tracked nature of the majority of construction projects: foundations are laid and base columns constructed whilst analysis and design are still in progress. A need for quick results when feasibility studies are performed or when evaluating the effect of design changes on supporting columns form other situations in which column forces are required, but where a detailed analysis to get these forces seems superfluous. Thus it was concluded that the development of an efficient tool for column force calculations, in which the extensive input required in a finite element analysis is to be avoided, would be highly beneficial.*

The automation of the process is achieved by making use of a Voronoi diagram. The Voronoi diagram is used a) for subdividing the floor into influence areas and b) as a basis for automatic load assignment.

The implemented procedure is integrated into a CAD system in which the relevant geometric information of the floor, i.e. its shape and column layout, can be defined or uploaded.

A brief description of the implementation is included. Some comparative results and considerations regarding the continuation of the study are given.

1 INTRODUCTION

In civil engineering practice, values of column forces in multi-storey buildings are often required before any detailed analysis of the structure has been performed. One of the reasons for this arises from the fast-tracked nature of the majority of construction projects: foundations are laid and base columns constructed whilst analysis and design are still in progress. A need for quick results when feasibility studies are performed or when evaluating the effect of design changes on supporting columns form other situations in which column forces are required, but where a detailed analysis to get these forces seems superfluous. Thus it was concluded that the development of an efficient tool for column force calculations, in which the extensive input required in a finite element analysis is to be avoided, would be highly beneficial.

The automation of the process is achieved by making use of a Voronoi diagram. The Voronoi diagram is used a) for subdividing the floor into influence areas and b) as a basis for automatic load assignment.

The implemented procedure is integrated into a CAD system in which the relevant geometric information of the floor, i.e. its shape and column layout, can be defined or uploaded.

As an initial investigation the force on a column was calculated based solely on the influence area of the column's Voronoi cell i.e. a geometrical problem is solved without taking any stiffness properties into consideration.

2 THE VORONOI DIAGRAM

2.1 Theoretical Background

2.1.1 Definition of the Voronoi diagram

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of n distinct points in the plane. Each of these points represents a site. The Voronoi diagram of P is defined as the subdivision of the plane into n cells, a cell for each site in P, with the property that a point q lies in the cell corresponding to a site p_i if and only if $\text{dist}(q,p_i) < \text{dist}(q,p_j)$ for each $p_j \in P$ with $j \neq i$.

2.1.2 Computing the Voronoi diagram

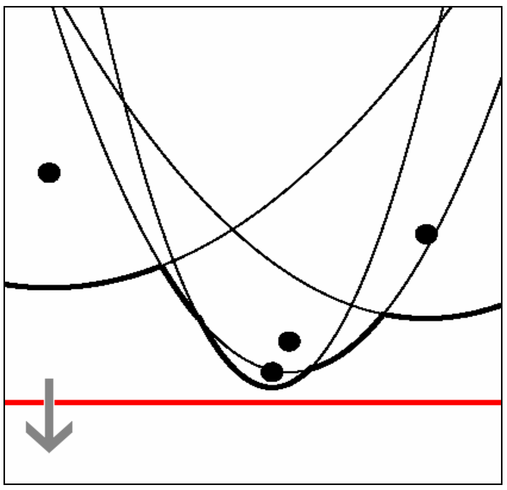
The algorithm used to compute the Voronoi diagram, a plane sweep algorithm commonly known as Fortune's algorithm, has been identified as optimal [1]. Although the algorithm is thoroughly explained in literature, difficulties were encountered during its implementation due to a lack of clarity on a certain implementation step. It was thus deemed necessary to include a description of the algorithm.

The algorithm involves sweeping a horizontal line - the *sweep line* - from top to bottom over the plane. As the sweep line moves downwards over the plane it strikes certain special points referred to as the *event points*. Relevant information regarding the structure of the diagram (i.e. its edges and vertices) is stored at these specific occurrences only. The following paragraphs are dedicated to explaining the algorithm in more detail.

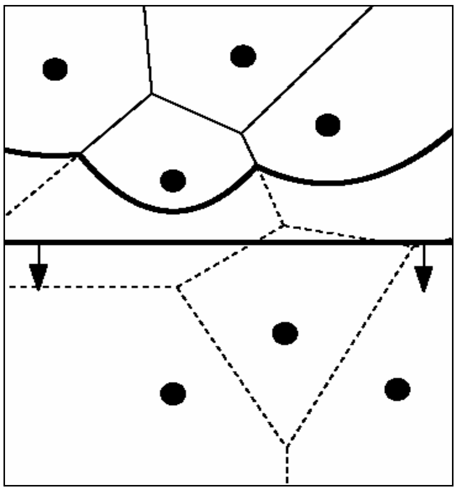
A fundamental concept of the algorithm is the formation and development of the *beach line*: a line situated above the sweep line and consisting of a sequence of parabolic arcs. Referring to Figure 1a, it is easy to visualize that each parabola corresponding to a certain site

p_i above the sweep line bounds the locus of points that are closer to p_i than to the sweep line. Thus the beach line can be seen as the line bounding the locus of points that are closer to any site above the sweep line than to the sweep line itself. The beach line is defined as the line passing through, for each x-coordinate, the lowest point of all parabolas.

As the sweep line moves from top to bottom the edges of the Voronoi diagram are traced out by the *breakpoints* separating the different parabolic arcs that form the beach line. Refer to Figure 1b for clarity.



1a.



1b.

Figure 1a and 1b: The beach line

The previously mentioned event points represent the points where, during the planar sweep, the combinatorial structure of the beach line changes. The first type of event, a *site event*, occurs when the sweep line strikes a new site. At this instance a new parabolic arc is formed which at first is simply a vertical line segment connecting the site to the beach line. As the downward movement of the sweep line progresses, this new parabola widens and the breakpoints at its beginning and end start tracing out a new edge. The edge is initially not connected to the rest of the Voronoi diagram above the sweep line and it is when such growing edges connect to form a vertex, that the second type of event, a *circle event*, takes place.

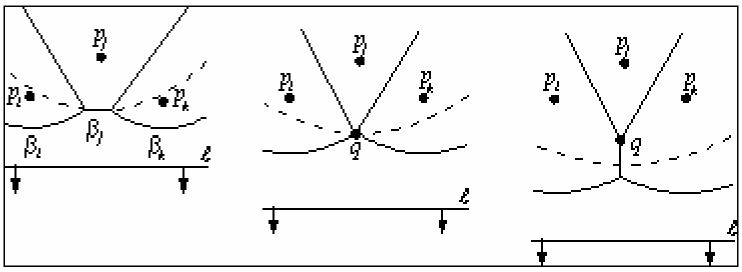


Figure 2: A circle event

Referring to Figure 2, a circle event marks the disappearance of a parabolic arc from the beach line. This happens at an instant where a circle passing through points p_i,p_j and p_k, with the vertex q at its centre and its lowest point on the sweep line, can be drawn. To be more specific, a circle event can be defined as an event where the sweep line reaches the lowest point of a circle through three sites defining consecutive arcs on the beach line.

The construction of the Voronoi diagram can now be perceived to consist of a sequence of a) site events, when new edges start to grow and b) circle events, when two growing edges meet to form a vertex.

2.2 Implementation

The fundamental classes used in the calculations leading to the set of edges and vertices defining the Voronoi diagram are presented in Figure 3. These classes are used by a number of key methods that will now be discussed individually.

determineEventQueue():
By arranging the defined sites from top to bottom an initial list object, representing the event queue, is formed. The list returned by this method only serves as a starting point: the event queue object itself changes continually due to the addition of circle events.

calculate():
This is the core method in which the strategy of the algorithm is applied. Its basic structure is as follows:

```
while the event queue is not empty
    do Remove the event with the largest y-coordinate from the queue
            if the event is a site event
                then handleSiteEvent()
                else handleCircleEvent()
```

handleSiteEvent():
When a site event is encountered the current position of the beach line, a *SortableBeachLineArcList* object populated with the parabolic arcs (*BeachLineArc* objects) as they exist at that instant, as well as a reference to the site event itself, are passed on to this method. The procedure to handle the event is defined as follows:

1. Search the list of arcs to find the arc vertically above the current site. If a) the event queue contains a circle event that has a pointer to this arc and b) the current site lies within such a circle, the circle event must be removed from the queue.

2. Create the new parabolic arc for the site and place it into the arc list in the correct position.

3. Create a new instance of a *HalfEdge* object for the edge separating the current site from the site responsible for the parabolic arc situated vertically above the current site.

4. Detection of circle events: Checks are performed on two sets of triples of consecutive arcs in the arc list. The triple that has the new arc as its left arc is checked to see

whether the breakpoints converge, followed by a check on the triple where the new arc is the right arc. If a circle event is detected, it is inserted into the event queue.

handleCircleEvent():
The method requires the same input as that passed on to the previous one. The event is handled in three steps:

1. Remove a) the disappearing arc from the arc list and b) all circle events that has a pointer to it from the event queue. Create a new breakpoint using the sites responsible for the arcs on each side of the disappearing arc and update the references of these side arcs accordingly.

2. Create a new vertex at the center of the circle causing the event and add it to the set of vertices. Use the new breakpoint (defined in 1.) and the vertex to create a new *HalfEdge* record starting at the vertex.

3. Perform checks and take action in a manner analogous to step 4 under *handleSiteEvent()*, but for the triples of consecutive arcs use a) the triple that has the former left neighbor as its middle arc and b) the former right neighbor.

As a last remark on implementation, it can be said that the algorithm as described above has been shown to handle degenerate cases (e.g. two or more events situated on a common horizontal line; coinciding circle events in situations where four or more co-circular sites exist, etc.) correctly.

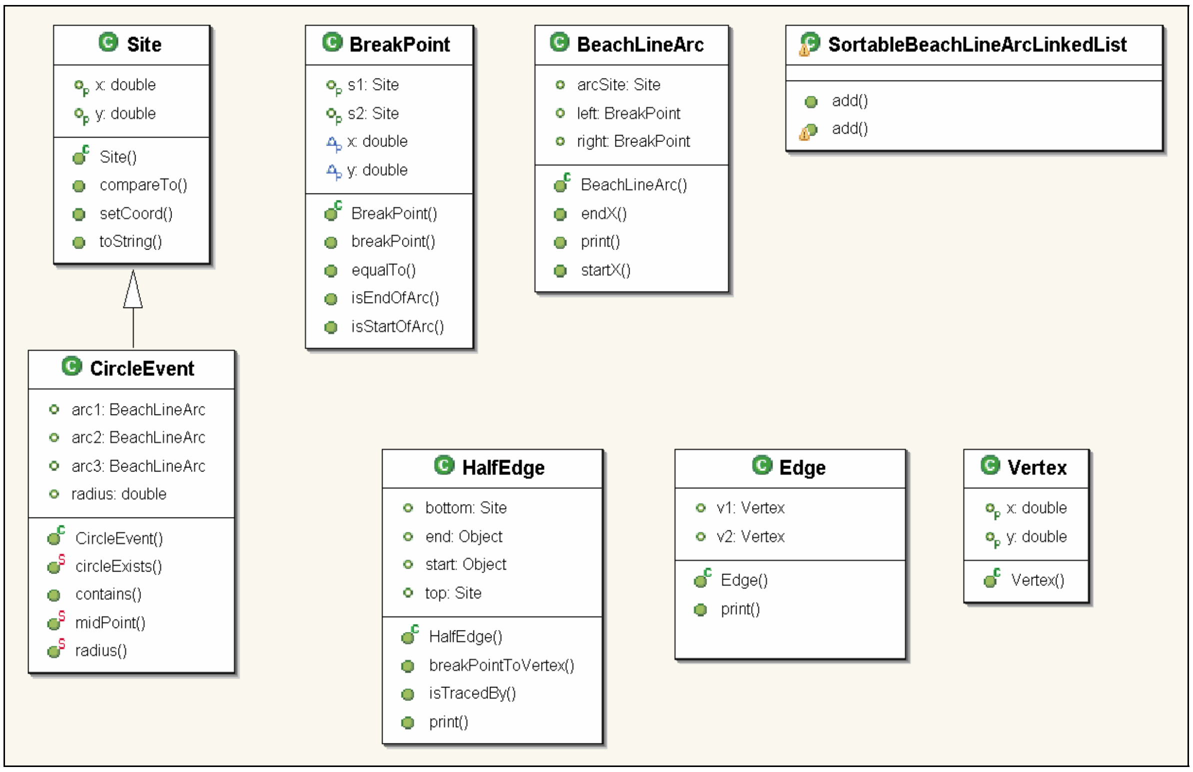


Figure 3: UML diagram of the classes used for calculating the Voronoi diagram

3 INTEGRATION WITH CAD

It was decided to integrate the Voronoi functionality into the CADEMIA system [2] in order to permit the geometrical definition of the building. For this purpose a number of classes were required to tie the new functionality into the existing structure of the CAD software.

3.1 Added classes

3.1.1 Commands

The CADEMIA interface Cmd, defines, amongst others, methods doCmd(), undoCmd() and redoCmd(), and is implemented by the following classes:

DefineOuterBoundary and DefineInnerBoundary:

Through an extension to the *selectByPickIntersect* command existing in the CADEMIA software, these commands permit the definition of an outer boundary of the floor as well as any inner boundaries (e.g. lift shafts) it may have.

AddColumn:

Column objects, representing the *sites* for the computation of the Voronoi diagram, are added to the database.

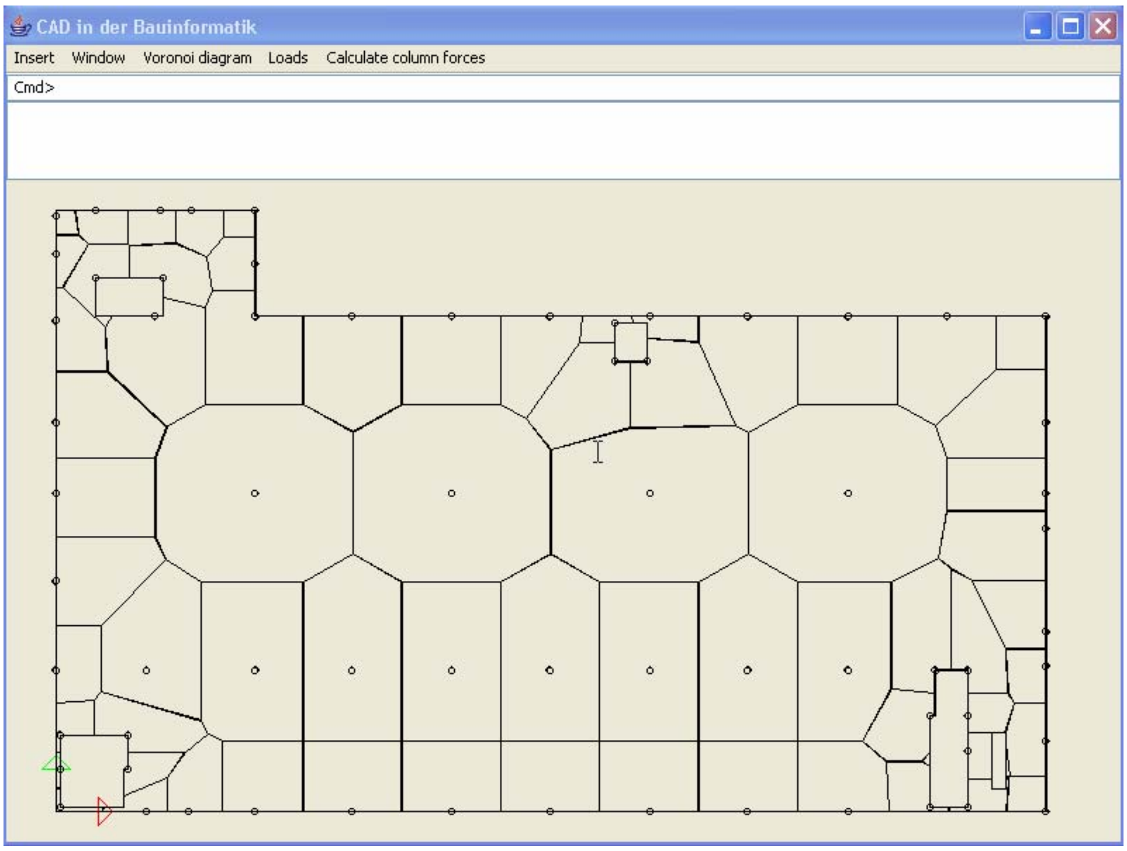


Figure 4: A floor geometry divided into influence areas based on the Voronoi diagram.

Calculate:

The Voronoi diagram is calculated and visually displayed through the graphical user interface as illustrated in Figure 4. Although the implementation of Fortune's algorithm has already been discussed (paragraph 2.2) it is necessary to elaborate on its implementation within CADEMIA. To start with, the computation of the Voronoi diagram is treated as an unbounded problem and the solution consequently produces edges that originally extend past the boundaries of the floor under consideration. This required the addition of another key method to the computation:

trim():
All intersections between the edges and the defined outer boundary and/or the set of inner boundaries are calculated and used to determine which parts of the affected edges are to be thrown away.

Another addition to the computation was necessitated by the need for a set of objects representing the Voronoi cells. These VoronoiCell objects and their contribution to the calculations generating the actual forces on the columns will be discussed at a later stage. For the moment, consider the following:

buildCells():
A set of GeneralPath objects defining the cells of the Voronoi diagram is created by a) identifying the edges and segments of outer and/or inner boundary that contribute to the cell boundary and b) putting them together in the correct order.

AddLoad:

In a manner analogous to that described for defining a boundary, Load objects are created and added to the database. The geometric elements defining the loaded area, e.g. the four line objects of the square in Figure 5, are selected by drawing an intersecting window and assembled into the geometrical object that defines, in conjunction with a kN/m^2 pressure value, the load.

Transfer:

The areas of the different cells are calculated by bounding line integration for linear, quadratic and cubic segments, based on the Gauss divergence theorem. Due to the fact that all geometric objects in CADEMIA can be separated into these three segment types, the integration is exact. The cell areas are then intersected with the areas corresponding to the Load objects (refer to Figure 5) in order to calculate the force transferred to each column.

3.1.2 Components

All elements in CADEMIA's component set implement interface Component. For an element (e.g. an edge or vertex) to be displayed by the graphical user interface, it must implement the methods defined in Component. In this regard classes Column, OuterBoundary, InnerBoundary, Edge, HalfEdge, VoronoiCell and Load were all either directly or indirectly made to implement Component. As regards the boundary classes, as well as classes VoronoiCell and Load, the implementation was done indirectly by extending an already existing component, namely ComponentGeneralPath.

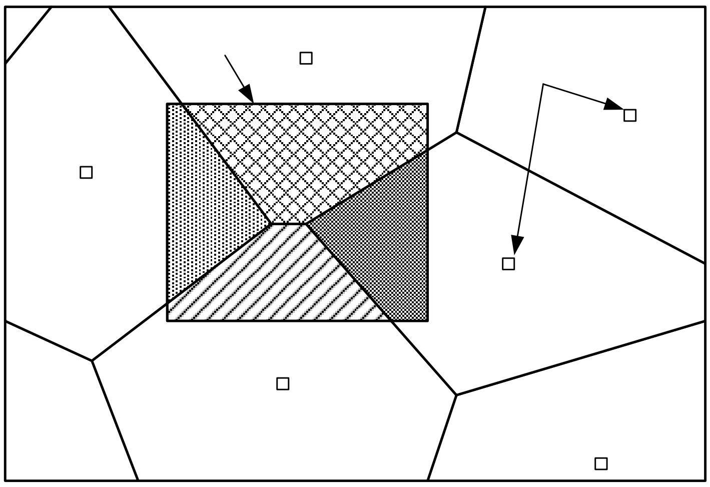


Figure 5: Adding a load and transferring the forces.

4 COMPARATIVE RESULTS

The accuracy of the forces obtained using this automated process was determined through comparison with the results of finite element analyses. Figure 6 shows displacement contour levels and deformations (inlay) for the floor of Figure 4 resulting from a finite element analysis. This specific floor is an industrial coffer slab structure with varying stiffness and self-weight. It generated an average error of 29% for the 67 pairs of column force values. Towards this the internal and edge columns contributed 13% and 33%, respectively. These errors were considerably larger than those obtained with other floors which had uniform stiffness and where the columns were spaced more evenly.

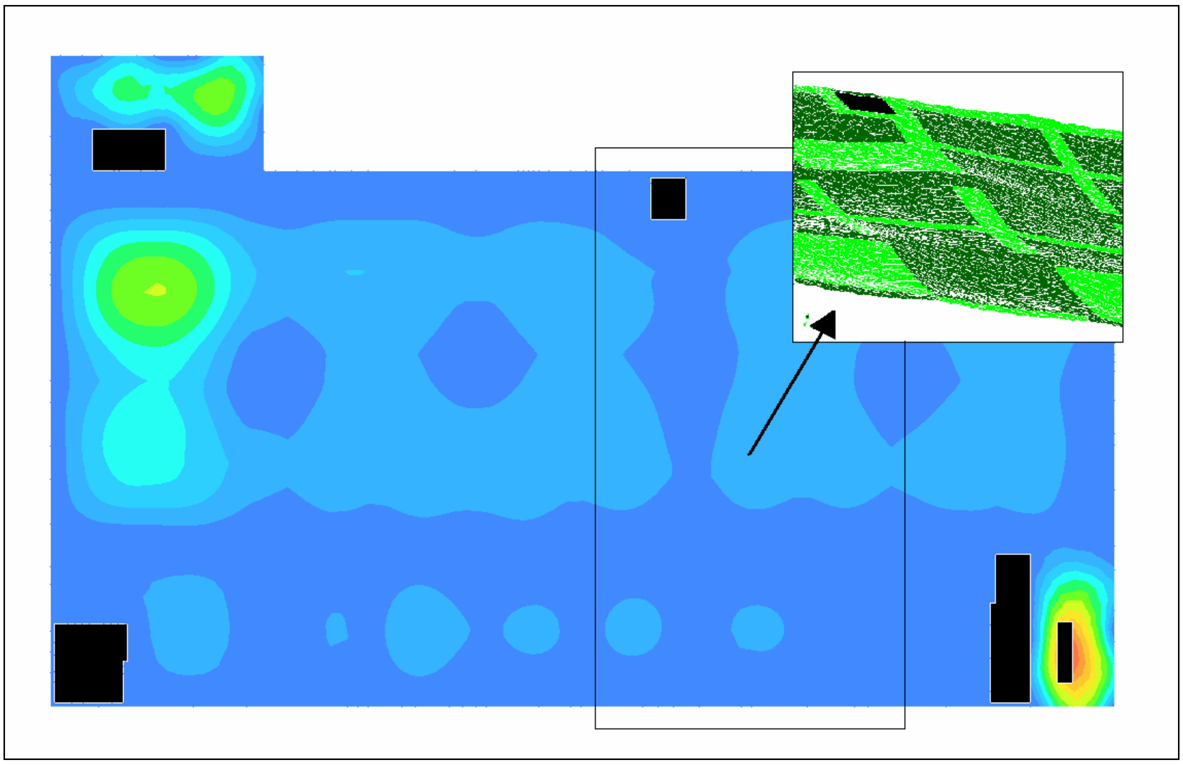


Figure 6: Displacement contour levels for the finite element model of the floor of Figure 4.

Although the finite element results were chosen for benchmarking purposes it is important to note that there exists another set of data for comparison. This set comprises the forces as calculated in industry: rough approximations are generated using rulers to approximate influence areas and accumulating these forces using spreadsheets. Upon reflection, the developed software does exactly this, except faster and more accurately.

A tendency towards larger errors for columns situated on the boundary of the floor was observed. It is our aim to, through the generation of more comparative data, identify more of these tendencies, to investigate their causes and to ultimately try to compensate for them in an effective way.

5 ASPECTS THAT NEED TO BE ADDRESSED

5.1 Line approximations used for curves

There are a number of instances where intersections between GeneralPath objects and Edge objects are required; consider trim() and the need to have a segmented boundary when creating the Voronoi cells. Calculating these intersections poses a problem when the GeneralPath contains non-linear elements and as a consequence the boundaries currently used in the calculations are line approximations of the actual GeneralPath objects. Accuracy can be improved by creating the ability to calculate intersections between quadratic or cubic GeneralPath segments and line objects.

5.2 Building the Voronoi cell

A problem currently under consideration concerns the development of a robust way in which to identify, assemble and combine the different geometrical elements defining the Voronoi cell. The difficulties encountered during the development of the buildCells() method stem from, amongst others, the possibility of more than one area contributing to a single Voronoi cell object. The situation is illustrated in Figure 7. The problem primarily consists of distinguishing between the segments of the boundary, whether an inner or outer boundary, that should be identified as belonging to the set of elements comprising the cell, and those that should not.

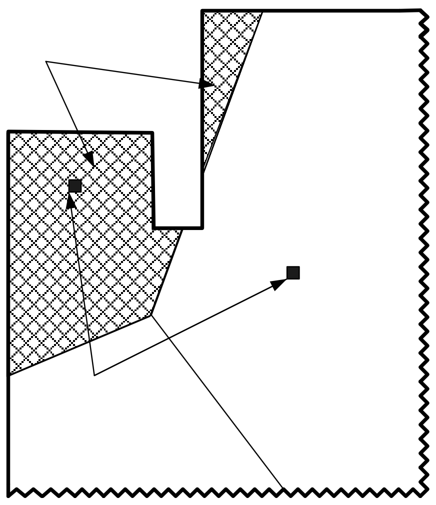


Figure 7: Building the Voronoi cell

6 OUTSTANDING FUNCTIONALITY

6.1 Building definition

The functionality as it exists at present solves a two-dimensional problem and needs to be extended to allow for the definition of the building in three dimensions. It is only when this extension has been carried out that the vertical accumulation of the forces can be calculated – a task that can be dealt with as a flow problem in the column network.

6.2 The effect of walls

Walls are currently modeled using a number of linearly placed columns. The error caused by this simplification as well as its variation with the amount of columns chosen to represent the wall, is a topic for future investigation.

6.3 Horizontal loading

Horizontal forces, imposed on a structure through the workings of earthquakes and wind, are currently not taken into consideration. The possibility of accounting for these forces in the software creates a future study topic.

REFERENCES

[1] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry - Algorithms and Applications,* Springer, Berlin Heidelberg, 1997.

[2] www.cademia.de: CADEMIA – das Open Source CAD System.