Proceedings of the Ninth International Space Syntax Symposium
Edited by Y O Kim, H T Park and K W Seo, Seoul: Sejong University, 2013

SPOT WITH PATHS, AND INTERACTIVE DIAGRAM WITH A LOW COMPLEXITY ISOVIST ALGORITHM

062

Pablo Miranda Carranza
KTH/ e-mail: pablo.miranda@arch.kth.se

Daniel Koch
KTH/ e-mail: daniel.koch@arch.kth.se

Asmund Izaki
Aedas R&D/ e-mail: Asmund.Izaki@aedas.com

Abstract

In an often quoted sentence of his 1976 book "The Architecture of Form", Lionel March drew a clear distinction between science, interested in extant forms, and design, which initiates novel forms. The theories, methods, measures and diagrams of space syntax have often developed following this first more scientific scheme, and they have been concerned with the analysis of existing or projected buildings and cities. This emphasis on analysis is evident in current software, algorithms and measures. But is it possible to think of a space syntax not only as a way of analysing existing situations or validating future designs, but as a form of actually generating architecture?

In our work we have used space syntax at the early stages of the design process, not so much as a form of analysis, but as a sort of architectural diagram. The shift of space syntax into a generative role has demanded a set of conceptual and technical adjustments: from the emphasis on graphic language and visualisation to the need for fast feedback and interaction.

In this paper we present an example from our work, and the framework (technical and methodological) necessary to produce it. The digital diagram we have created deals with the design of a new hospital ward. It represents some basic problems we have encountered in the relation of patients, staff and architecture, which are incorporated into the software through 3 basic interactive entities: isovists (from patients positions), the circulation paths of hospital staff (with a calculation of their visibility relations to the patients), and the arrangement of walls to form rooms. All these 3 entity types are interdependent: isovists depend of walls and positions and the visualisation of staff paths depends on the patients isovists. They are also editable in real time, that is, walls, isovists and paths can be added, deleted, or moved, and the effects of any of these actions visualised at once.

This fast interaction and feedback require efficient algorithms and data structures. In particular we have implemented an algorithm for the calculation of isovists or visibility polygons with a complexity dependent of the size (in terms of visible vertices) of the visibility polygon, rather than being a function of the size of the boundary. This allow us to calculate visibility polygons in real time irrespective of the size of the boundary, may this be a building or a whole city. Our method implements an idea by Asmund Izaki for the calculation of isovists and visibility graphs, based in the use of an underlying triangulation data structure for the search of all visible vertices from a point. Besides the general interest of our approach to the use of space syntax in a generative rather than in an analytical way, we believe that the algorithms for the calculation of visibility polygons or isovists can find application also into existing space syntax software, improving its performance, and in some cases opening the possibility for an extension of its role from forms of analysis to generative ones.

Our software has been developed using the C++ programming language, and it makes extensive use of Open Source libraries such as CGAL, Dime, Qt and Boost.

Keywords: *Isovists, Low complexity, Interactive Software, Algorithm, Design Application*

Theme: *Modelling and Methodology Developments*

1. Introduction

In an often quoted sentence, Lionel March drew a distinction between science, interested in extant forms, and design, which initiates novel forms (March 1976). The theories, methods, measures and diagrams of space syntax have often developed following this first more scientific scheme, and they have been concerned with the analysis of existing or projected buildings and cities.This emphasis on analysis is evident in current software, algorithms and measures. In our work we have considered instead the use of space syntax at the early stages of the design process, not so much as a form of analysis, but as a sort of architectural diagram. The shift of space syntax into a more generative role demands a set of conceptual and technical adjustments, including an emphasis on graphic language and visualisation and the necessity of fast feedback and interaction. At the same time, the implementation of these adjustments requires efficient algorithms and data structures in a larger measure than the analytical approach.

In this paper we present an example from our work, and the technical and methodological framework necessary to produce it. We like to think of our approach as digital diagramming; we develop a set of useful algorithms and visualisation techniques that are assembled into small computer applications, specific to a problem or project. This modular approach to the production of software allow us to test and refine these reusable components, and to provide custom made, simple applications for a design or problem.

The digital diagram we present here deals with the re-design of a hospital ward. A central role in this example is played by polygonal isovists, defining visibility relations within the ward. These isovists need to be draggable and react to changes to the geometry of the plan in real-time. We have consequently implemented an algorithm for their calculation that depends of the complexity of the isovists, rather than being a function of the boundaries of objects, in our case the number of walls in the building. This allow us to calculate isovists in real time irrespective of the number of the visibility obstacles, may this be the amount of walls in a single building, or those of all buildings in a whole city. Our method implements some techniques proposed by Åsmund Izaki (Izaki and Derix 2013) for the calculation of isovists and visibility graphs, based on the use of an underlying triangulation to reduce the complexity of the calculations.

2. Differences between analysis and design software.

There is a clear distinction between a general analysis tool and a design tool; while analysis will have as its goal generalisation, that is, to treat a number of different situations similarly in order to draw general conclusions, or in order to compare an specific case with the inferences drawn from previous analyses, a design tool needs often to reflect the distinct requirements and contingencies of a design brief. Rather than a few general methods that can be used in most circumstances, design projects require approaches that provides specific responses to a design brief or architectural intention: instead of generalisation design requires specialisation. A way to solve the contrasting goals of having general, relevant and well grounded principles, and of being able to use them to answer to specific requirements, is to create generic methods and techniques that can be adapted and assembled to give individualised responses to the particular demands and constrains of architectural projects.

Another important difference between analysis and design tools is in the way these can interact with the design process and its different cycles. Analysis can only be performed on existing proposals; one needs to tentatively design first and then subject the results to analysis in order to validate them. This is not only the case with Space Syntax, but the general case for much of environmental and structural analysis as it is employed in architectural design. Usually such a process does not necessary require fast feedback; most design decisions have been made at the time the analysis is done, and it is only necessary to adjust the proposal.

But if some of the core concerns of Space Syntax representation and measures such as accessibility, visibility or configuration are considered central to a design, then it would be a great advantage to include them as the subject of the ideation process, and to have forms of shaping these qualities directly, rather than by guessing first and validating later. Thus real-time interaction can play an important role in making Space Syntax an integral part of the design process, rather than a measure of its success or failure. At the same time, the implementation of interactive tools requires to address problems of performance and computational complexity.

3. An example: Designing visibility in a hospital ward.

We present an example of an application of these principles in a design context. Mainly carried out as part of a research project named “To see and be seen in health-care environments: user oriented design for visibility and cooperation in spatial systems for health care”, this application has also developed in close relation to a different study of security in buildings. Both projects share a concern about visibility in spatial systems, though clearly of a very different kind, and illustrate how a technique, in the form of algorithms and their implementation in computer code, can become a component that can be reassembled in response to different design problems.

The intentions behind the interactive software was to be able to test simple relations between the configuration of space and its use, particularly the visibility, accessibility and contact between hospital staff and patients. These intentions were translated into 3 basic graphical and software components: first, an schematic drawing of the layout of the ward, editable by removing, adding or modifying walls; second, a collection of isovists, representing the location of patients and the visibility affordances and exposure of those locations; and third, a diagram of the circulation paths of hospital staff and the visibility relations of these paths with patients’ locations, which is calculated and displayed as the number of isovists overlapping different segment of the path. Walls, isovists and paths can be added and deleted, and their effect on each other visualised in real time. (Figure 1).

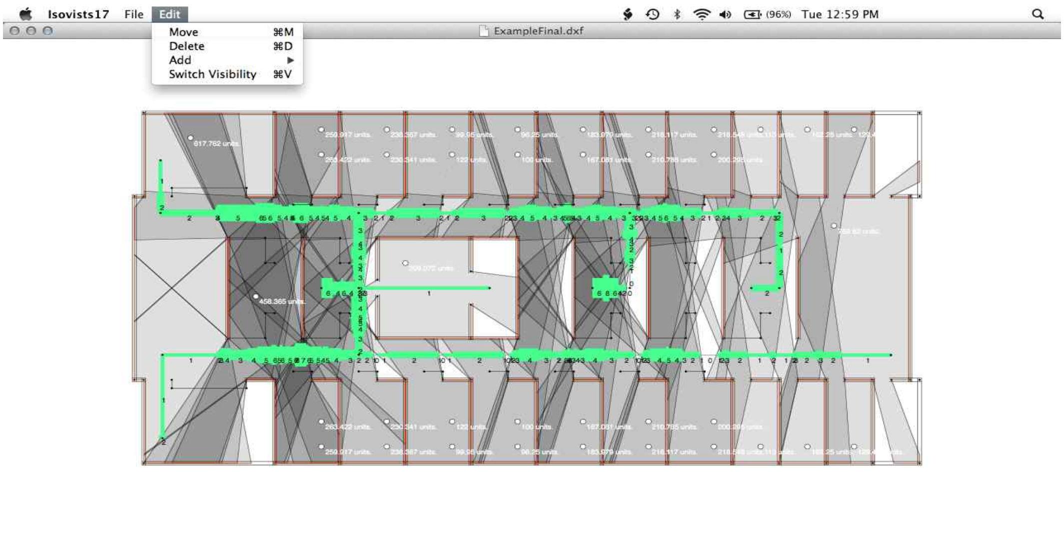

Figure 1: Screenshot of the application with its 3 components: the editable schematic of the plan, the isovists positions and the paths, showing their exposure to the isovists.

All these 3 components, that is, layout diagram, isovists and paths, are interrelated in their software implementations: changes in the layout affect the isovists, and changes in the isovists affect the visibility of paths. The relation between isovists and the layout diagram is particularly relevant and potentially useful in other contexts. It also exemplifies how aspects of performance are central to the development of interactive applications and the need to study the computational complexity of the analysis and visualisation methods used. Thus, we will dedicate the rest of the paper to explain our method for efficiently calculating isovists, and their interaction with an editable representation of the spatial layout of a building.

Our interest on isovists comes from previous developments of tools that deal with polygonal isovists and visibility graphs (Markhede, et al. 2010, Miranda Carranza, et al. 2008). Isovists need little explanation within the context of Space Syntax; the concept of the Isovist was first proposed by Tandy(Tandy 1967) in the context of landscape analysis, and introduced to an architectural setting by Benedikt(Benedikt 1979), who in his paper also proposed a number of measures and properties of isovists relevant to architecture. Another fundamental contribution to the use of isovists is the work of Turner, Doxa, O'Sullivan and Penn, in which they proposed the use of "visibility graphs" (Turner, et al. 2001), a type of diagram and associated measures that has become standard in Space Syntax analysis. Geometrically, algorithmically and as forms of analysis, these different implementations of the concept of isovist are however quite different: while Benedikt's isovists define a precise polygonal region, Turner, Doxa, O'Sullivan and Penn's isovist consists of visibility relations within a lattice. Turner et al. are more interested in the analysis of these visibility relations and their structure, and thus their representation differs substantially from Benedikt's. We are concerned in our work with a description of the isovists in Benedikt's terms. However the method we propose could be easily be adapted to calculate the type of visibility relations involved in Turner et al's visibility graphs. In fact much the basis of our algorithm rests on techniques from the field of computational geometry that deal with visibility relations and queries, which could substantially speed-up the calculation of visibility graphs.

4. Fast calculation of interactive isovists.

4.1 Some definitions.

While isovist play an important role in Space Syntax, the concept of isovist is not exclusive to this domain. In general visibility problems are central to the field of computational geometry, which is concern with the description and analysis of algorithms that can be stated in terms of geometry (Preparata and Shamos 1985). In the context of computational geometry, two points are *visible* to each other if is possible to draw a line segment between them that does not intersect any geometrical obstacle; these obstacles can consist of a polygon, a set of polygons, line segments, planes or other geometric figures (O'Rourke 1994, O'Rourke 2004). Accordingly, the concept in computational geometry equivalent to an isovists is that of a *visibility polygon*: if visibility obstacles are defined by line segments, the *visibility polygon* consists of the locus of all points visible from a point p in a plane (Preparata and Shamos 1985, 322). This region will be defined by a (possibly unbound) polygonal region.

The study of the algorithmic complexity is essential in computational geometry. The complexity of an algorithm can be described as the order of growth of computer resources (memory and processing time) in relation to the growth of the size of its input (for example the number of lines defining a plan, or the number of cells in a tessellation, in some of the algorithms used in Space Syntax). The study of complexity is also known as asymptotic analysis of algorithms, and for representing this complexity it is common to use different types of *asymptotic notations*. One of the most frequent ones is the "big O" -O(f(n))- notation, often employed to represent the time complexity of an algorithm, that is, how the time necessary to perform an algorithm grows in relation to the size of its input. Big O notation is used as an upper bound for complexity, and in general it means that given an algorithm with a complexity of O(f(n)), beyond a certain size of the input, its complexity will grow less than f(n) multiplied by a constant. For example, O(n) would imply a linear time complexity , that is, the time required to run an algorithm will be, in the worst case, and with an n sufficiently large, linearly proportional to its input; O(n2) is a square time complexity, and it would imply that the time needed (in the worst case) by an algorithm would grow as a square of the size of the input. O(1) implies a constant time, that is, an operation that is independent of the size of its input. The study of complexity and its different classes is a fundamental part of the theory of computation and the analysis of algorithms. A good introduction and reference to algorithm analysis can be found in: Cormen, Leiserson, Rivest and Stein's "Introduction to Algorithms" (Cormen 2009).

Thus the following description of our method is based on the definition of an isovist as a *visibility polygon*, and we will analyse the complexity of its calculations in the terms outlined above. We will show how by preprocessing the input data into a triangulation it is possible to reduce the time complexity of the visibility polygon substantially compared with naive or brute force methods. Moreover, we show that it is possible to reduce the complexity of this calculation so it does not depend on the size of the input (the number of obstacles, for example line segments defining walls in our particular case), but on the actual size of the *visibility polygon*, in terms of how many times it overlaps faces of the triangulation. The method explained is an extension of the one originally proposed by Åsmund Izaki (Izaki and Derix 2013). It uses data structures and algorithms from the Computational Geometry Algorithm Library, CGAL, particularly the implementation of constrained triangulations (Pion and Yvinec 2013, Boissonnat, et al. 2002). The use of these data-structures from CGAL allows also to efficiently edit the geometry of the obstacles (the walls in our case), by effectively updating the underlying triangulation. Thus we can calculate isovists in time independently of the size of the data, may this be a building or a whole city, and edit this data (move, delete or add obstacles) with constant complexity, as we will see.

4.2 Background and complexity of calculating the visibility polygon.

The use of triangulations and other convex subdivisions to perform visibility queries has been proposed earlier. Descriptions of a number of algorithms and their complexity can be found in Hershberg (Hershberger and Suri 1995, Hershberger 1989), and some of these can be adapted for example to the fast calculation of the visibility graphs used in Space Syntax. The algorithm we are using is particularly related to some of the basic concepts described in Aronov, Guibas et al. (Aronov, et al. 1998), who employ balanced triangulations to efficiently calculate visibility polygons. The principle for the algorithms explained in their paper is based on the intuition that the complexity of the calculation of a *visibility polygon* should depend on its size, rather than the size of the input data (Aronov, et al. 1998). The algorithm proposed by Aronov, Guibas et al. has a space complexity of $O(n^2)$, and a time complexity of $O(n^2 \log n)$ for preprocessing time, and $O(\log^2 n+k)$ for the calculation of any visibility polygon, where n is the number of vertices of a simple polygon, and where k is the size (the number of vertices) of the resulting visibility polygon. The complexity of the calculation of the *visibility polygon* includes in this case the *location query*, that is, finding out the triangle that contains the point from which we are calculating the *visibility polygon*. The time complexity of the *location query* is in this case $O(\log^2 n)$. The algorithm described by Aronov, Guibas et al. deals with simple polygons without holes. Izaki's algorithm works on a polygon with holes, and our own implementation with any set of line segments (which may or not form polygons).

The problem of using a convex subdivisions for calculating visibility polygons can be seen to consist of 3 separate problems: first, the pre-processing of the input (converting obstacles defined by line segments into triangles, for example); second the *location query* problem, that is, to find in which one of those partitions (triangles in our case) is the point from which to calculate the *visibility polygon*, and third, the calculation of the *visibility polygon* proper.

Our approach is in some aspects less efficient than the one outlined above (particularly on the *location query*), but easy to understand and implement, specially by using the CGAL constrained triangulation. It allow us also to manipulate the input set of obstacles (line segments) in constant time O(1). The complexity of preprocessing the input data into a triangulation is $O(n^2)$ in our case. CGALs 2D triangulations use an incremental algorithm, and its upper bound depends of the *location query* strategy used to insert each vertex of the input one by one into the triangulation of previous vertices. The *location query* we have used is the naive method provided by the CGAL 2D triangulation, which uses a walk strategy (Boissonnat, et al. 2002), and has a worst case complexity of O(n), but which can have a complexity of only $O(\sqrt{n})$ for randomly distributed vertices (Yvinec 2013). This running time of the *location query* can also be improved by using a constrained Delaunay triangulation in CGAL (Boissonnat, et al. 2002). The calculation of the *visibility polygon* is dependant on the number of times the algorithm visits triangles that can be seen from the point defining it. We will proceed to describe in more detail our algorithm for the calculation of the actual *visibility polygon*.

4.3. A detailed description of the algorithm for calculating the visibility polygon

4.3.1. Preliminaries

We will assume that both the preprocessing of the input data into a triangulation and the *location query* have been performed, so we have a triangulation and we know the triangular face in which the point that generates the visibility polygon is placed. From now on, and through the rest of the explanation, we will refer to this point as the *centre* of the *visibility polygon*. The characteristics and relation between obstacles and the triangulation we use (the CGAL constrained triangulation) are as follows:

Obstacles are defined as non-intersecting line segments: these can be linked into closed or open polygonal chains, or they can exist in isolation. We will refer to these segments as *obstacle segments* in the rest of the explanation. The triangulation covers the *convex hull* (Preparata and Shamos 1985, 18) defined by these *obstacle segments*: all vertices of the triangulation will be endpoints of the *obstacle segments*, and all *obstacle segments* will correspond to edges of the triangulation. This is easily achieved through the CGAL constrained triangulation, in which each obstacle segment will correspond to a constrained edge in the triangulation (Yvinec 2013).

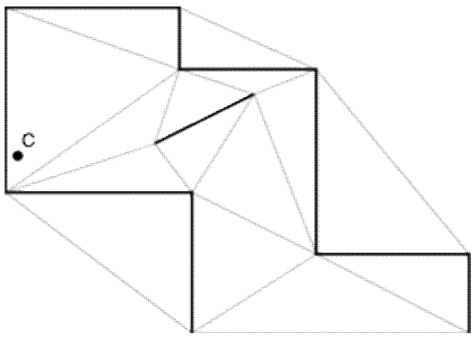

Figure 2: Triangulation of obstacle segments, by using a constrained triangulation. Observe that the obstacle segments form an open polygonal chain and that there is also an isolated obstacle segment. The Convex hull is the closed polygon defined by all outer edges of the triangulation.

The position of the *centre* of the *visibility polygon* is assumed not to be on *obstacle segments*; this is considered a degenerate case and not treated here, though the algorithm can be extended to include it. For simplicity the algorithm described below only considers cases in which the *visibility polygon* is completely *bounded* by *obstacle segments*. Our implementation deals with *unbounded visibility polygons* (visibility polygons with portions that don't meet any obstacle segment and are thus open and of infinite size). We won't explain the particularities of their treatment for the sake of clarity, it is enough to say that these are just especial cases of the explanation given below, which include ways of treating triangulation edges in the convex hull which are not *obstacle segments*, as well as the *infinite edges* used in the CGAL 2D triangulation for defining the exterior of the *convex hull* (Yvinec 2013). The algorithm explained below also assumes a triangulation data structure similar to the one in CGAL, in which a face in the triangulation has access to all its vertices and neighbouring faces in O(1) constant time.

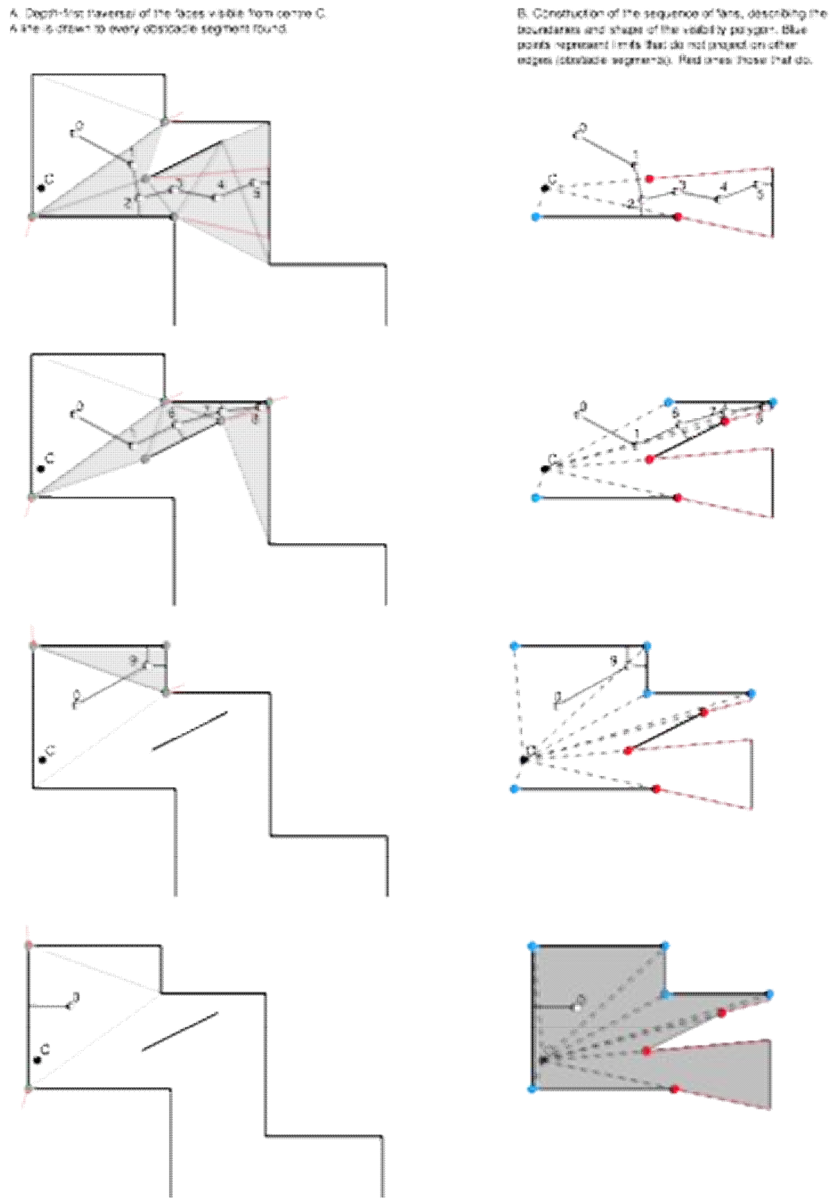

Figure 3: triangle traversal sequence. Step 0 corersponds to procedure VISIBILITY POLYGON(F). Other steps corresponds to calls to VISIT FACE (F, EE, RV, LV). Observe how the algorithm visits only triangles visible from centre C.

4.3.2. The algorithm:

The algorithm consist on a depth-first traversal of all the triangular faces visible from the *centre* of the *visibility polygon*. The result of the algorithm will be the sequence of fans, each entry in this sequence will represent a fan and consist of an edge E of the triangulation, and two corresponding vertices, which we will call the *left limit LL* and the *right limit RL*. These entries define a fan shaped area of the *visibility polygon* from its *centre*: a ray from the *centre* to the vertex *left limit LL* will define the left side of the fan and a ray from the centre to the *right limit RL* vertex, its right side. The *edge* E associated with these two vertices in the sequence, will define the external bound of the fan. In the case of the bounded *visibility polygon* which we are considering, the edges in the *fans* will always correspond to an *obstacle segment*. The way we visit each triangular face in the algorithm makes sure that all the fans in the list are adjacent, and ordered in counterclockwise manner. (Figure 3.B)

To build this list, we implement the depth-first search as a recursive function, which takes 4 parameters as input: a face F, an entry edge EE which belongs to the face F, and two vertices of the triangulation, the *left vertex LV* and the *right vertex RV*, which define the limits of a fan area from the *centre* of the *visibility polygon*. The algorithm can be described through the pseudocode below, which it loosely uses the Pidgin Algol convention: the recursive procedure VISIT FACE,takes the input just described and either adds a fan to the *FANS* sequence or recurses; the VISIBILITY POLYGON procedure starts from the face that contains the *centre* of the *visibility polygon* and calls VISIT FACE function for each face adjacent to this initial face. All traversal operations in the triangulation (vertex in face opposite to an edge, other face on an edge, etc) are depended of how the triangulation data structure is implemented. Using the CGAL 2D triangulation data structure all these operations have a constant complexity O(1). Many other data structures commonly used to represent triangulations have also constant complexity for these operations, such as the the winged-edge or the half-edge data structures.

```
    procedure VISIBILITY POLYGON(F)
input: the triangulation face F, which contains the centre of the visibility polygon
   begin FANS := Ø
         for each (edge e ∈ F) (*edges on a face are ordered counterclockwise*)
            begin of := the other face of e that is not F
                  vr := the rightmost vertex in e when looked from any point inside F
                  vl := the leftmost vertex in e when looked from any point inside F
                  VISIT-FACE(of, e, vr, vl)
            end
   end


   procedure VISIT FACE (F, EE, RV, LV)
input: the triangular face F, one of its edges EE and triangulation vertices RV and LV
   begin if (EE is obstacle segment) then
            begin RL := RV
                  LL := LV
                   E := EE
                  INSERT(FANS, {E, RL, LL})
             end
         else
            begin ov := vertex in triangular face F opposite to edge EE
                  er := edge in F to the right of ov from any point inside F
                  fr := the other face of er that is not F
                  el := edge in F to the left of ov from any point inside F
                  fl := the other face of el that is not F
                  if (ov is to the right of the ray that goes from centre to RV) then
                     begin VISIT FACE(fl,el,RV,LV)
                     end
                  else if                  (*ov is to the left of the ray that goes from centre to LV ) then
                     begin VISIT FACE(fr,er,RV,LV)
                     end
                  else                        (*ov is within the fan defined by centre and RV and LV*)
                     begin VISIT FACE(fr, er, RV, ov)
                           VISIT FACE(fl, el, ov, LV)
                     end
            end
   end
```

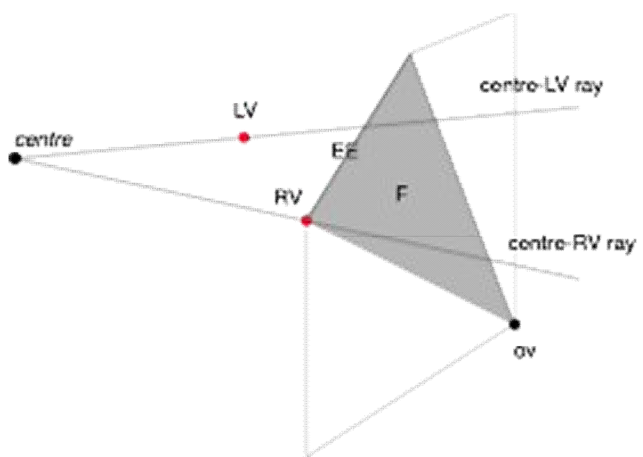

Figure 4: Step 4 from figure 3 in detail (corresponding to the VISIT FACE (F, EE, RV, LV) procedure): since ov is to the left of the centre -LV ray , step 5 will continue only with the edge and face to the left of ov (seen from EE), and with RV and LV as limits.

As we can see from the pseudocode, if all operations dealing with the triangulation have constant complexity, the running time of the algorithm will be dependent of the faces it visits. It is possible that the algorithm visits a face more than once, if this face is cut by more than one of the *fans* defined by the *left vertex LV* and the *right vertex RV* in the above function (as in Steps 5 and 8 in figure 3, for example).

In order to draw or analyse the visibility polygon it is possible to iterate though the *FANS* sequence, once this has been calculated, as follows:

```
  procedure DRAW VISIBILITY POLYGON (FANS)
input: the FANS sequence of fans, each defined by an edge and 2 vertices of the triangulation
  begin for each (f ∈ FANS)
        begin ptr :=    ptl := Ø
              if(f.RL is not vertex of f.E) then
                  begin ptr := intersection of the ray from centre to f.RL with f.E
                        DRAW LINE (f.RL, ptr)
                  end
              else ptr := f.RL
              if(f.LL is not vertex of f.E) then
                  begin ptl := intersection of the ray from centre to f.LL with f.E
                        DRAW LINE (f.LL, ptl)
                  end
              else ptl := f.LL
              DRAW LINE (ptr, ptl)
        end
End
```

This iterates through the *FANS* sequence we have built, and does the following: *f* is an instance of the data structure we use to store the fan data, and *f.E, f.RL* and *f.LL* are the edge, the *right limit* and *left limit* vertices respectively. If *f.RL* is not a vertex of edge *f.E*, then it is a vertex that is projecting on *f.E*. Draw then the projection line from *f.RL* onto *f.E*. Proceed similarly for *f.LL*. Draw finally the line segment which is the visible part of edge *f.E* (the *obstacle segment*). This visible part of *f.E* will either be made of the triangulation vertices at its ends, if *f.E* is completely visible, or otherwise limited by the projections of *f.RL* or *f.LL* (or both) onto the *f.E obstacle segment*.

4.4 Performance test.

As an example of the performance of the method, we have run a simple test, using a file containing all buildings within the city limits of Stockholm, consisting of 49168 line segments. Using a standard laptop with a 2.8 GHz Intel Core i7 processor, it took a bit over 8 seconds to generate the constrained triangulation. Adding and moving *visibility polygons*, as well as editing the data (deleting or adding buildings) could be then performed in realtime (Figure 5).

Figure 5: Real-time isovists with a sample file of all buildings in the centre of Stockholm

5. Conclusions

Our intention with this paper has been to show how the use of Space Syntax as a design tool involves both methodological and technical challenges, and how these are strongly interdependent. An emphasis on graphic and interactive capacities, will necessarily lead to the need of a careful treatment of the computational aspects of their performance. In the paper we have also proposed a style of developing Space Syntax software, which rather than proposing general tools, is instead based on assembling components into software tailored to specific project needs. This will necessarily imply the generation of a common pool of frameworks, methods and algorithms, that can eventually be shared between different practitioners and researchers.

We have also introduced an example of using such an approach, and given a detailed description of one its main constituents, a method for the calculation of isovists with a lower complexity than the methods commonly available in Space Syntax. The paper is also an attempt to introduce in the description of Space Syntax problems some of the rigour common in the field of Computational Geometry. While Computational Geometry has been referenced previously in the context of Space Syntax (Peponis, et al. 1997, Peponis, et al. 1998) the intersection of these two fields deserves a more exhaustive study. Computational Geometry offers an extensive and rigorous treatment of geometrical problems of high relevance to Space Syntax, with a special emphasis on the complexity of their implementation through algorithms. Our proposal to calculate isovists both in this paper and in Izaki's work (Izaki and Derix 2013) shows but an example of the potentials of these intersections.

The software discussed in this paper (OS X version) and its C++ source code distributed under the Open Source GPL 3 license are available at:
http://kth.diva-portal.org/smash/get/diva2:587411/SOFTWARE01.zip.

References

Aronov, Boris, LeonidasJ Guibas, Marek Teichmann, and Li Zhang. 1998. "Visibility Queries in Simple Polygons and Applications." *Chap. 38 In Algorithms and Computation*, edited by Kyung-Yong Chwa and OscarH Ibarra. Lecture Notes in Computer Science: 358-67. Springer Berlin Heidelberg.

Benedikt, M. L. 1979. "To Take Hold of Space: Isovists and Isovist Fields." *Environment and Planning B* 6(1): 47-65.

Boissonnat, Jean-Daniel, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. 2002. "Triangulations in Cgal." *Comput. Geom. Theory Appl*. 22: 5-19.

Cormen, Thomas H. 2009. *Introduction to Algorithms. 3rd ed*. Cambridge, Mass.: MIT Press.

Hershberger, J., and S. Suri. 1995. "A Pedestrian Approach to Ray Shooting: Shoot a Ray, Take a Walk." *Journal of Algorithms* 18(3): 403-31.

Hershberger, John. 1989. "An Optimal Visibility Graph Algorithm for Triangulated Simple Polygons." [In English]. *Algorithmica* 4(1-4)(06/01): 141-55.

Izaki, Åsmund and Christian Derix. 2013. "Visible Polygon Traversal Algorithm (Forthcoming)." In *2013 European Architectural Envisioning Association (EAEA)* 11th Conference. Politecnico di Milano, Milano, Italy.

March, Lionel. 1976. *The Architecture of Form*. Cambridge Urban and Architectural Studies. Cambridge; New York: Cambridge University Press.

Markhede, H., P. Miranda and D. Koch. 2010. "Spatial Positioning Tool: Background, Prototype Software and Some Correlation Data." *The Journal of Space Syntax* 1(1): 14.

Miranda Carranza, Pablo, Asmund Gamlesaeter and Christian Derix. 2008. "3d Isovists and Spatial Sensations: Two Methods and a Case Study." In *EDRAMOVE & SFB TR8 conference on spatial cognition*. Veracruz, Mexico.

O'Rourke, Joseph. 1994. *Computational Geometry in C*. Cambridge; New York: Cambridge University Press.

O'Rourke, Joseph. 2004. "Visibility." In *Handbook of Discrete and Computational Geometry, Second Edition.* Discrete Mathematics and Its Applications: Chapman and Hall/CRC.

Peponis, J., J. Wineman, S. Bafna, M. Rashid, and S. H. Kim. 1998. "On the Generation of Linear Representations of Spatial Configuration." *Environment and Planning B: Planning and Design* 25(4): 559-76.

Peponis, J., J. Wineman, M. Rashid, S. H. Kim, and S. Bafna. 1997. "On the Description of Shape and Spatial Configuration inside Buildings: Convex Partitions and Their Local Properties." *Environment and Planning B: Planning and Design* 24(5): 761-81.

Pion, Sylvain and Mariette Yvinec. 2013. "2d Triangulation Data Structure." In *Cgal User and Reference Manual*. CGAL Editorial Board.

Preparata, Franco P., and Michael Ian Shamos. 1985. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. New York: Springer-Verlag.

Tandy, C. R. V. 1967. "The Isovist Method of Landscape Survey." *Methods of Landscape Analysis*: 9-10.

Turner, A., M. Doxa, D. O'Sullivan, and A. Penn. 2001. "From Isovists to Visibility Graphs: A Methodology for the Analysis of Architectural Space." *Environment and Planning B:*

Planning and Design 28(1): 103-21.

Yvinec, Mariette. 2013. "2d Triangulations." In *Cgal User and Reference Manual*. CGAL Editorial Board.