2012-4

Effects of Variations in 3D Spatial Search Techniques on Mobile Query Speed vs Accuracy

Junjun Yin
Dublin Institute of Technology

James Carswell
Dublin Institute of Technology, jcarswell@dit.ie

Effects of Variations in 3D Spatial Search Techniques on Mobile Query Speed vs Accuracy

Junjun Yin and James D. Carswell

Digital Media Centre, Dublin Institute of Technology, Ireland
{yinjunjun@gmail.com, jcarswell@dit.ie}

Abstract. This paper presents three Spatial Search Algorithms for determining the three dimensional visibility shape (threat dome) at a user's current location in a built environment. Users then utilize this 3D shape as their query "window" to retrieve information on only those objects visible within a spatial database. Visibility shape searching addresses the information overload problem by providing "Hidden Query Removal" functionality for mobile LBS. This functionality will be especially useful in the Web 4.0 era when trillions of micro-sensors become available for query through standard IP access.

Keywords: MSI, Mobile LBS, Spatial Databases, Isovist 3D, Threat Dome

1 Introduction

Visualisation of 3D built environment datasets on commercially available smartphones (e.g. *Google Maps 5* for Android 2.0+) is now reality. This is made possible by rendering the mobile map from a single set of vector data tiles instead of multiple sets of raster image tiles, and allows for smooth and continuous map viewing and scaling from different perspectives using the same set of vector data. Although *Google Maps 5* is not yet photo realistic, the resulting 3D models are close to being geometrically accurate as they are derived from extruding building footprints to known heights for different parts of a building (Fig.1).

Within such a 3D vector dataset of Dublin, we have attached attributes (meta-data) to the various floors, windows, doors of buildings, plus affixed a range of environmental sensor data streams to other scattered locations on a building's façade. Together this provides the beginnings of an *Internet of Things* type environment for testing our developed 2D/3D visibility-based spatial querying algorithms and techniques.

It is recognized that analyzing enormous volumes of data on a mobile device requires addressing the "information overload" problem to reduce display clutter. Allied research into the information overload problem is ongoing, where map personalisation and other semantic based filtering mechanisms are essential to de-clutter and adapt the exploration of the real world to the processing/display limitations of mobile devices [2, 3, 4, 5]. We propose that another way to filter this information is

to intelligently refine the search space by applying *hidden query removal* (HQR) functionality in three dimensions.

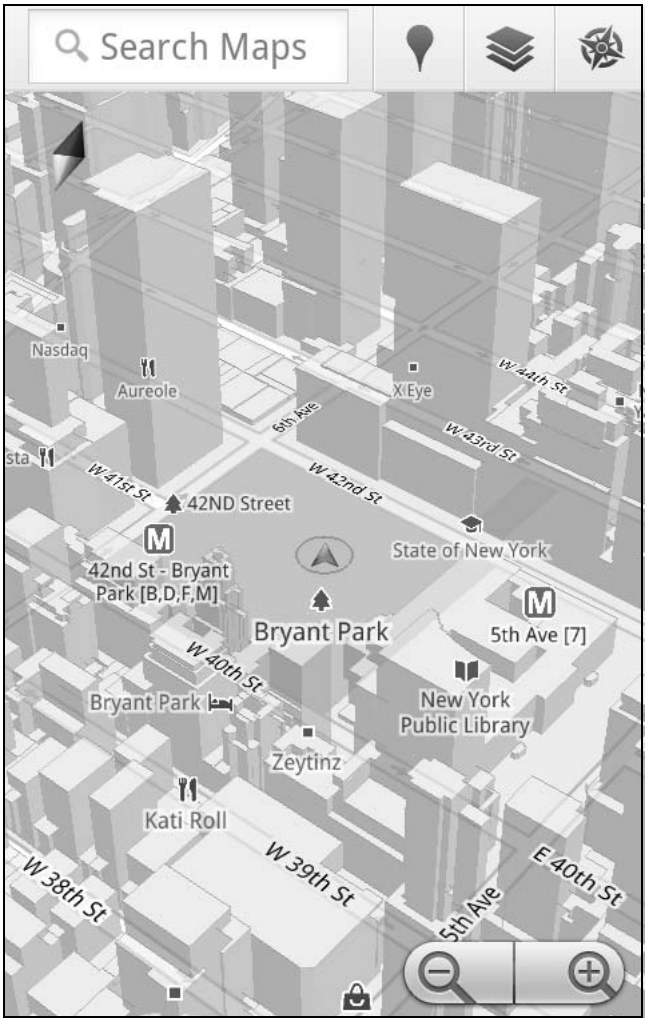

Fig. 1. 3D tilt, zoom, and rotation enabled mobile map displayed by *Google Maps 5 for Android* [1]

The combined effect gives a more accurate and expected query (search) result for Location-Based Services (LBS) applications by returning information on only those objects/sensor enabled "things" visible within a user's 3D field-of-view (FOV) as they move through a built environment. For example, visitors can now explore both their horizontal and vertical surroundings by pointing their smartphones at stores, offices, POIs, or any space to retrieve from the web any recorded information about these objects - answering specific questions such as: "Whose office window is that up there?" or more generally; "What are the air pollution readings along this street?" or

perhaps more interestingly; “Can I see any CCTV cameras from where I’m sitting?” or indeed; “Are they seeing me?”.

However, to make our 3DQ (Three Dimensional Query) prototype function effectively in real-time requires mobile spatial query techniques that extend today’s spatial database technology both on the server and on the mobile device itself. This paper describes the various algorithms developed and results of tests carried out on COTS (commercial off-the-shelf) mobile devices querying a sample 3D vector dataset. Our ultimate goal is to dynamically visualize on a smartphone the 3D query space, or *threat dome*, overlaid in real-time on a 3D mobile map, together with any returned query results (Figure 2).

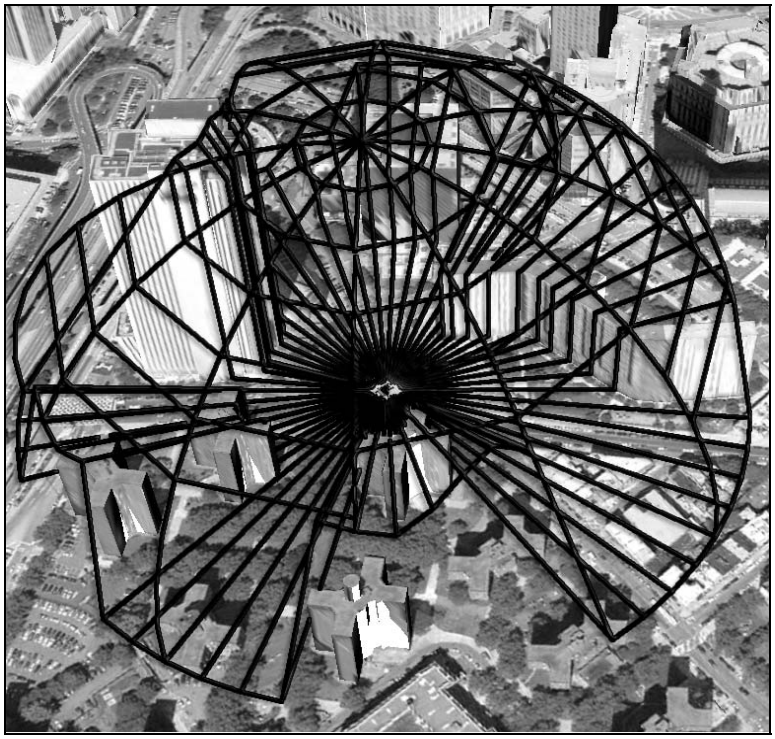

Fig. 2. Threat Dome search space interacting with a 3D cityscape model; only *things* intersecting the solid dome shape get returned by the query.

Since 3DQ is intended to be deployed as a web-based service for mobile users, multiple users can connect and perform location based searches at the same time. Therefore, one significant research challenge was to efficiently and rapidly calculate the underlying visibility query shapes for each user. Also, maintaining the accuracy of the retrieved results is essential to providing a better user experience by reducing the information overload risk.

The remainder of this paper is organized as follows: Section 2 introduces the vector datasets that we utilize in our work. Section 3 describes our main contributions by presenting a comprehensive discussion of the algorithms and implementation of

our 3DQ prototype. This is followed in Section 4 by some evaluations of the performance of 3DQ in terms of speed vs accuracy when using different search algorithms and parameters, and Conclusions and plans for Future Work can be found in Section 5.

2 Vector Datasets Background

Geospatial information is increasingly recognized as the common denominator in both today's "web 2.0" peer-to-peer social network era and tomorrow's "web 4.0" – where it is envisioned that the Internet becomes connected to trillions of micro-sensors placed into real-world objects of all types (i.e. mechanical and non-mechanical), all with their own 128 bit IP address [6]. In other words, an *Internet of Things* that collects and sends time-stamped data to the cloud every second about their location, movement, plus any number of other measureable phenomena – e.g. environmental data such as air/water quality, ambient light/noise data, energy consumption, etc.

It is in this "Big Data" realm where we envisage 3DQ operating most effectively. When the potential of the sensor-web becomes realised, every brick of every building, every cobblestone of every street, every road sign, traffic light, street light, water bottle, beer can, garbage can and flower pot could conceivably be individually communicating their whereabouts and local conditions to the world. In such a world, we believe the ability to filter out, both semantically and spatially, unwanted/unnecessary/unsolicited information while at the same time retrieving task-relevant data for making informed decisions will be paramount.

Three dimensional indexing is a requirement for storing and querying 3D vector objects, which at time of writing limits our spatial database options to Oracle Spatial 11gR2, although PostGreSQL with their anticipated PostGIS 2.0 extension for 3D indexing will be, once available, a useful open source addition to this very short list. However, we've discovered clear limitations of Oracle's spatial query operators when trying to determine the spatial relationships among 3D geometries. These include creating 3D R-Tree indexes on 3D geometries using a minimum-bounding cube. Oracle only considers if these cubes intersect with one another as a method for determining whether their underlying 3D geometries actually intersect. Retrieving the actual 3D location where two geometries (vector objects) intersect in 3D is not yet supported.

For example, an important feature of 3DQ is to detect exactly where the intersection between a generated ray (simulating the 3D pointing direction of a smartphone) and a 3D building occurs. In this case, Oracle derives the intersection point using the 2D spatial operator SDO_INTERSECTION by first projecting the query shape (3D ray in this case) and the target (3D building) onto the ground plane and then only returning the 2D position of this 3D ray/building intersection. Using this information and combining it with the tilt angle and 2D distance to the nadir of the actual intersection point, we are left to compute the actual 3D intersection point of this query ourselves.

In an accurately computed threat dome, the generated dome shape will usually have a large number of surface elements relative to the complexity of the surrounding environment. If we want to consider this dome as a single 3D query shape (surface) in the form of an Oracle SDO_GEOMETRY (in order to make use of the SDO_INTERSECTION query operator), we find that its total number of surface elements will typically exceed the number of elements allowed in the SDO_ELEM_INFO_ARRAY - where it seems an arbitrary maximum of 999 coordinates (i.e. 333 3D points) are permitted.

In our case, where each surface element contains 12 coordinates defining its shape (four 3D vertices), a maximum of only 27 surface elements are then allowed in one SDO_ORDINATE list. As it happens, this is typically far fewer than what is required to accurately describe the boundary of a complete 3D threat dome shape. To get round this limitation, we must first split the complete threat dome into 3 or more sections and then query them individually against the database - instead of creating a single 3D volume as the threat-dome query shape. The returned query results are then a sum of all object/section intersections after first removing any duplication. Ironically, one beneficial consequence of this extra processing is that it encouraged us to mirror the spatial database across multiple servers and then send each individual 3D dome section query to a separate database. The result is a much faster (~2sec.) query process which potentially allows for near real-time threat dome visualisations and searching on 3D mobile maps – our ultimate goal for this work.

Since the introduction of the "Isovist" concept in [7] for describing the 2D visibility shape or 3D visibility volume at a given position, there have been a number developments that employ Isovist-like approaches for urban environment analysis. The notion of a "Spatial Openness Index" (SOI) developed in [8, 9] measures the volume of visual perception within a surrounding sphere from a given point of view, but without defining its shape. Other techniques to measure 2D and 3D visibility in an urban environment are shown in [10], which calculate the visibility of pixel coordinates on Digital Elevation Models (DEMs). Their proposed "iso-visi-matrix" claims to be a very useful from a visual perception viewpoint. Different to these approaches, visibility modelling algorithms developed in [11, 12], calculate the visibility of local landmarks in an urban context. They determine the visibility of a "Feature of Interest" (FOI) for location based services (LBS) that notify users when they are in a position that can actually see those landmarks. 3DQ acknowledges the importance and usefulness of carrying out 2D/3D visibility based analysis in the urban environment and aims to extend this idea by exploiting the actual 3D visibility shape as a query "window" to retrieve only those spatial objects that a user can physically see from a given viewpoint.

When calculating 3D Isovists, a user's visual perception is usually simulated and interpreted as a collection of "sight lines" or "line-of-sight" collisions with spatial objects in the environment [13]. In this regard, the technique of ray casting is a common approach to determine sight line/object intersections where the collective intersection points of collisions eventually form the visibility shape used as the query window.

In most modern computer gaming applications, collision detection techniques are very well developed to determine and render only those visible objects in a game scene to optimize display speed. However, those type of calculations are normally *Boolean value* based operations, which means if the ray hits an object the returned value is "true" and vice versa. For the purpose of computer graphics layered rendering, this technique (without further calculation of the intersection point) are proven to be quite efficient [14, 15, 16]. However, our 3DQ prototype requires more than just determining which objects constitute a scene from a user's viewpoint, we also need the 3D Isovist shape to determine where objects (e.g., built environment, sensors in the Sensor Web) are intersected. Therefore, accurate vertices (intersection points) of the visibility shape are necessarily required to form the corresponding shape in a spatial database for subsequent query processing operations.

3 3DQ Search Algorithms

The 3DQ system adopts a "client-server" architecture to deliver spatial searching as web-services for mobile devices. The services are in RESTful format style, where mobile device as client collects readings from its integrated sensors (e.g. GPS, compass, accelerometer), constructs them into a standard URL, and sends them to the server. Once the server finishes with the query calculations, the responses are organized and sent back in GEO-JSON format, which is OGC standard compatible and completely text based. A more detailed description of the 3DQ system architecture can be found in [17].

In a 2D scenario, the vertical dimension of a built environment is ignored in favour of the geometry of building footprints on the horizontal plane. Ideally, the length of a ray, which simulates a sight line from a user's view point, shall be infinite unless it hits an object along its path. However, to speed up the query calculation, we default the search length (user's perception distance) to 200 meters. In other words, the footprints used to load the built environment around a user's vicinity are limited to a 200 meters radius, thus speeding up considerably the query calculation. Options for users to adjust this search distance are also provided.

An example of a visibility search in a 2D environment (i.e. 2D Isovist) at a given location is shown in Figure 3 (a). The black square represents the user's current location which is picked up from GPS on the mobile device. The surrounding built environment is constructed from the footprints of all building blocks within 200 meters. Benefiting from R-Tree indexing in Oracle Spatial 11g, the retrieval of all buildings from a given location is quite efficient [19]. The filled polygon is the user's 360^{o} visibility shape at that location. The 2D Isovist shape is then utilized as the query window to retrieve all database objects that intersect it. The Isovist construction process is based on the method developed in [18], which is an open source library for fast 2D floating-point visibility algorithms.

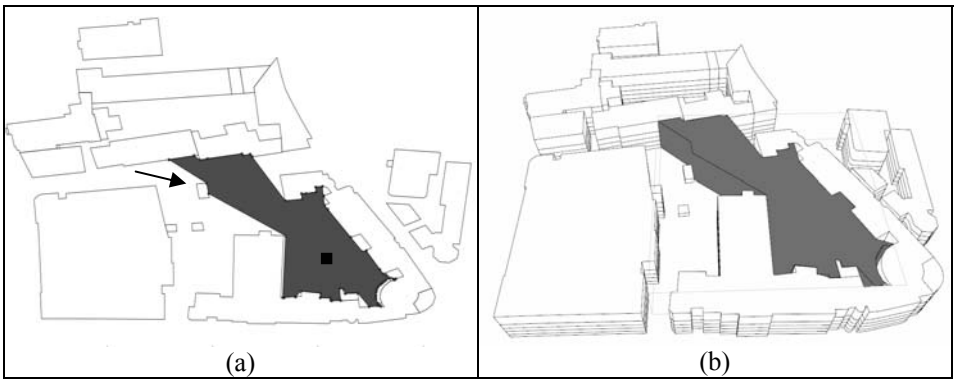

Fig. 3. (a) 2D Isovist view (b) Extruded 2.5D Isovist in a 3D environment

The 2D Isovist query is especially useful for conventional 2D mobile map searches, where it can serve as the preliminary filter to reduce the sometimes overwhelming amount of information available at a given location. However, as Google Maps 5 has progressed 2D mobile maps into 3D, a more realistic look and feel of a built environment is now available. Although, with this added vertical dimension come 3D visibility calculations that are much more complicated than in 2D. For instance, the arrow in Figure 3(a) points to a small building block where a 2D ray gets truncated, but the building's height is much lower than the surrounding buildings so a user's sight line can in fact see over top of this block.

Although fast, to simply extrude a 2.5D Isovist (Figure 3(b)) would be incomplete as it does not pick up on this height difference. In fact, the results retrieved from an extruded 2.5D Isovist would be no different than those returned from a 2D Isovist, as the 3D coordinates for each returned object have the same x and y. Yet, to derive an accurate 3D Isovist, as shown in [20], is far too calculation intensive for real-time searching and therefore not optimal for serving multiple users as a web-service. Therefore, we utilize ray casting techniques on vector datasets using a predefined length (radius) for each ray to save on computation effort. Thus the final query shape of the 3D Isovist appears as the "dome" shape shown in Figure 2 where the vertices that form the dome are the intersection points between each ray and any objects the ray hits.

An example demonstrating how vertices are detected in *Search Algorithm 1* is shown in Figure 4. In this illustration, assume building blocks with different heights are along the path a ray travels. On the horizontal plane, the interval between each ray is predefined at 6° by default (horizontal ray spacing). While on the vertical plane, we first detect what objects the ray hits and then calculate the corresponding 3D intersection point. The next ray along this same direction is initialized with a tilt angle of 15° and so on (vertical ray spacing).

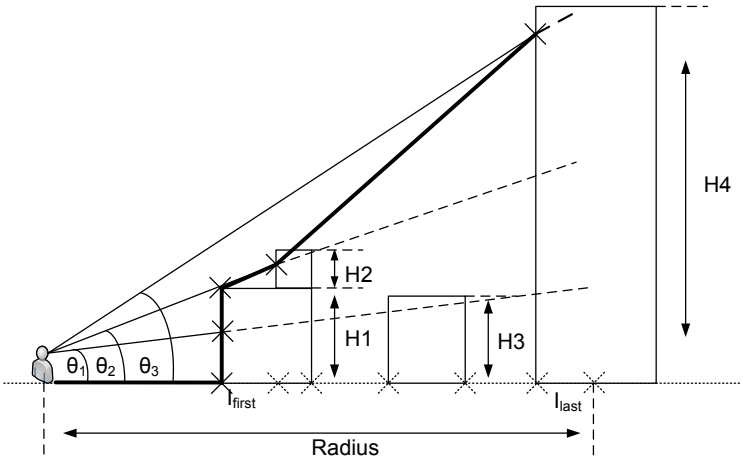

Fig. 4. *Search Algorithm 1* for determining sight line intersections in a 3D environment. The thick black line outlines the final boundary of the threat dome in this direction.

The pseudo code for *Search Algorithm 1* follows:

Algorithm I: Tilting Ray Approach

```
-----------------------------------------

Input:  radius, horizontal ray spacing, tilt angle,
        current location
Output: A 3D threat dome visibility shape

Function RayTilting3D (radius, raySpacing,
initalLocation, tiltAngle):
  Initialize ray generator from initialLocation
  Initialize final shape list: ShapePtCollect
  For each ray start with an initial tiltAngle:
    Initialize list: IntersectionPtCollect
    Get all the intersections and add to list
    Determine the first intersection: intersectionPt
    tiltAngle += AngleInterval (15° default)
    ShapePtCollect.append(intersectionPt)
  Return ShapePtCollect
```

As mentioned, Oracle does not return the exact 3D intersection point when a ray hits the building blocks. Instead, it projects the ray and building blocks onto a horizontal plane and returns a collection of 2D line segments. The intersection point is then determined by getting the first intersection point from all the segments, together with the tilting angle θ. The ray then continues to detect the next intersection point and so on until it stops once titling angle θ reaches 90°. This approach takes advantage of the 3D spatial query operators provided by Oracle Spatial as well as 3D spatial

indexing and serves as a good approximation of the true dome shape. However, it can be seen in Figure 4 that building a threat dome using the tilting angle approach may miss certain intersection points vertically as the ray may overpass a building block because of the gap between any two tilting angles.

To improve on this approach, *Search Algorithm 2* acts like "reverse water-flow", where the ray does not stop at the intersection point but instead continues on to determine the next intersection until it finally stops at the distance specified by the radius (Figure 5). The process starts by determining the intersections on the horizontal plane between each ray and the projected footprints of the 3D building blocks. The actual intersections are a list of line segments and each of them has a pair intersection point $<I_{in}, I_{out}>$. The collection of I_{in} points will be picked up and ordered by their distance from user's location. We then determine the first intersection point of all I_{first}, which represents the first hit between a ray and the objects. The ray restarts from I_{first} and a tilting angle θ_1 is initialized once it reaches the top of the building. The next calculation happens at the next I_{in} point in the list, where if the height of the ray at that point is higher than the height of the building, the process carries on to the next I_{in} point in the list, otherwise, a new tilting angle is established and the same process iterates to the next I_{in} point.

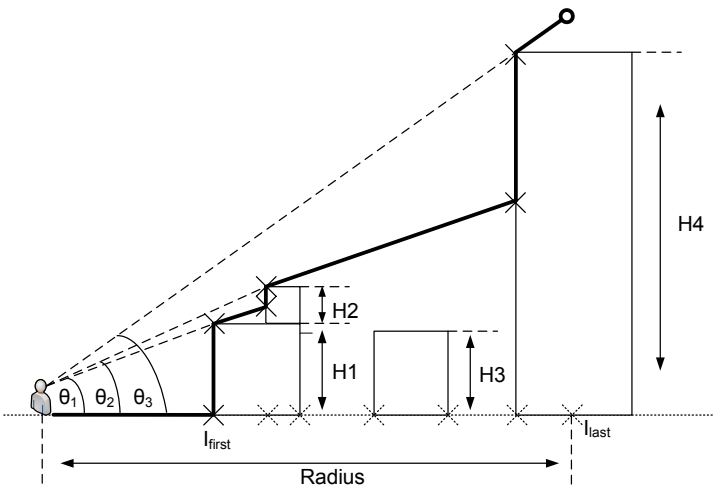

Fig. 5. Search Algorithm 2 for determining sight line intersections in a 3D environment. The thick black line outlines the final boundary of the threat dome in this direction.

Another advantage of *Search Algorithm 2* is that instead of using the tilting angle to generate multiple rays vertically, it only needs to process one ray on the horizontal plane and collect height information of all building blocks along its path. In *Search Algorithm 2*, the 3D building objects, which are represented as solids in Oracle Spatial, are replaced as 2.5D data structures with a height value attached as an attribute to each footprint and therefore uses 2D spatial indexing when deriving the

initial intersection list, which has a simpler and faster data structure than 3D spatial indexing.

The pseudo code for *Search Algorithm 2* follows:

Algorithm II: Reverse Water-Flow Approach

```
Input: radius, horizontal ray spacing, current location
Output: A 3D threat dome visibility shape

Function RaySweeping3D (radius, raySpacing,
initialLocation):
  retrieve all building block geometries that are
    within the radius of a user's current location
  For each ray in the horizontal plane:
    initialize lists: HeightsCollect, DistanceCollect,
    IntersectionPtCollect
    derive all the intersection points from the ray:
      determine first intersection of each collision
    fill the lists
  sort the IntersectionPtCollect according to their
      distances from the initial location
  tg• = Height(I_first)/Distance(I_first)
  initial final shape list: ShapePtCollect
  For each point in the list:
    If Height(I_next)< Dist(I_next)* tg• = newHeight:
      pass
    Else:
      ShapePtCollect.append(I_next, Height= newHeight)
      ShapePtCollect.append(I_next, Height= Height(I_next)
      tg• = Height(I_next)/Distance(I_next)
  Return ShapePtCollect
```

A common feature found in both Algorithms 1 and 2 is defined ray spacing when scanning for objects in the horizontal plane. However, no matter how small the interval is, there is still a risk of missing certain intersection points, which reduces the ultimate accuracy of the final query shape. As noticed in the 2D isovist calculation shown in Figure 3(a), the 2D visibility shape is continuous at filling every visible corner/gap between building geometries. Our third search approach therefore applies Algorithm 2 on top of a 2D Isovist shape. Once a 2D Isovist is calculated, each vertex of the Isovist polygon together with the user's location defines a ray direction with the length equal to the predefined radius. An example is shown in Figure 6, where the rays travel through each of the vertices in a 2D Isovist polygon instead of being evenly specified by a ray spacing value. Although it involves two steps of calculation, it provides a more accurate visibility shape.

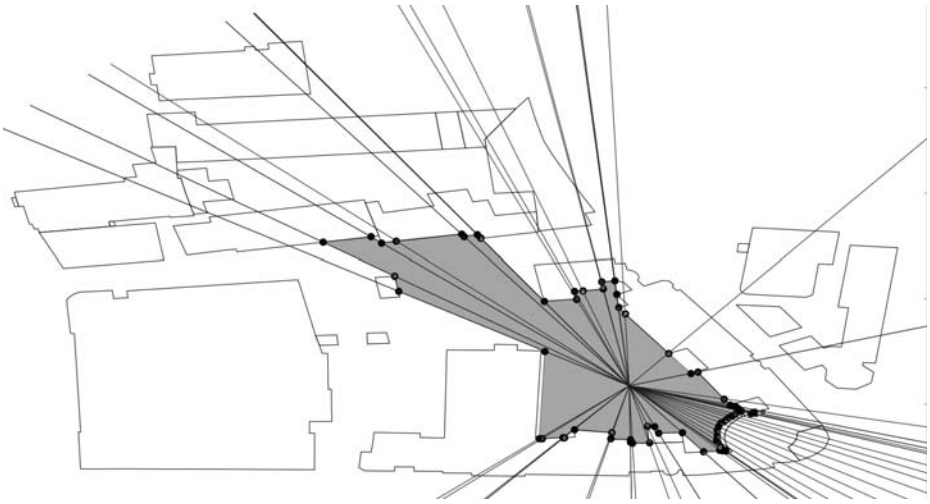

Fig. 6. An example of generating rays through the vertices of a 2D Isovist

The pseudo code for *Search Algorithm 3* is follows:

Algorithm III: 2D Isovist Based Ray Sweeping Approach

```
Input:  radius, user' current location
Output: A 3D threat dome visibility shape

Function RayIsovist3D (radius, initialLocation):
  retrieving all the 2D footprints within the radius
    from current location
  calculating 2D isovist
  initial final shape list: ShapePtCollect
  For each vertices of the Isovist generate a ray
     through the initialLocation:
    applying Algorithm II
  Return ShapePtCollect
```

As mentioned previously, in an accurately computed threat dome, the generated dome will usually have a large number of surfaces relative to the complexity of the surrounding environment. We must therefore split the complete threat dome into 3 sections and then query them individually against the database - instead of creating a single 3D volume as the threat-dome query shape. This led us to mirror the spatial database across three servers and then send each individual 3D dome section query to a separate database. In today's cloud computing era, it may be possible to deploy this approach as a cloud-based service with multiple spatial databases running in different virtual machines at the same time. The result is potentially a much faster query process that allows for near real-time threat dome visualisations on 3D mobile maps.

4 Evaluation of Query Speed vs Accuracy

In this section, we present our performance evaluation of the 3 Searching Algorithms implemented in terms of speed vs accuracy. The datasets used include 2D footprints of DIT campus with actual heights stored as a non-spatial attribute, 3D wireframe building outlines stored as 3D multi-polygons, and extruded 3D solids from the 2D footprints up to the stored heights. Plus 100 3D points simulating environmental sensor "things", which are attached to the surfaces of the building blocks (e.g., windows, walls, and doors, etc.) and other objects (e.g. light posts) in the database. A screenshot of the combined dataset is shown in Figure 7.

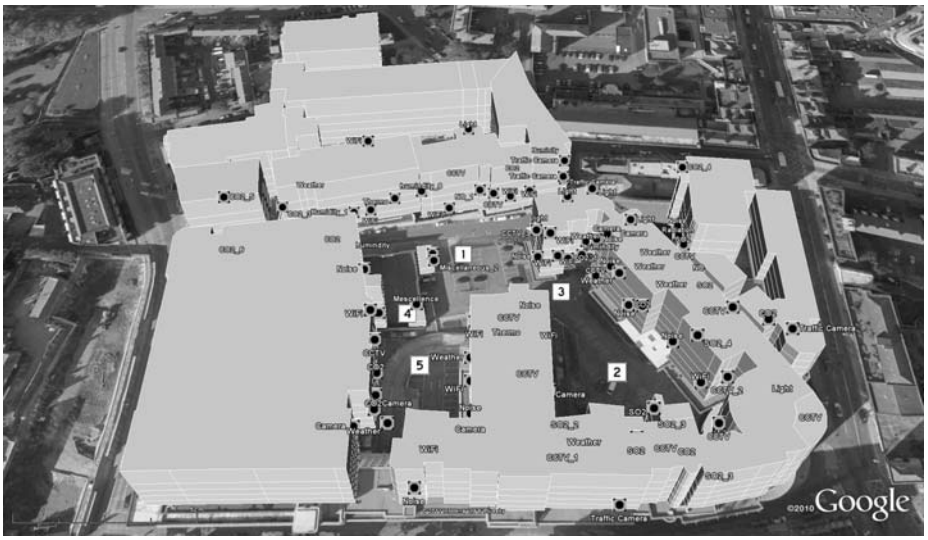

Fig. 7. 3D model of university campus affixed with 100 synthesized environmental sensors.

The evaluation of each search algorithm was carried out at five different locations in the campus. To test our approaches on large datasets as when deployed in a real-world application, the 2D footprints consists of 345,316 polygons with height values attached, and cover most of Dublin city. The datasets are stored in 3 Oracle Spatial databases mirrored on three different machines. More specifically, machine one is running Windows 7 (32 bit) with 4G RAM and Core2Duo 2.8 GHz CPU, the other two machines are running virtual machines under Windows XP (32 bit) with 2G RAM and Core2Duo 2.2 GHz CPU. The programming language is Python 2.7 with the capability of providing OGC standard compatible Geo-JSON output to mobile devices.

Figure 8 shows a comparison of query speed for the three different search algorithms. *Search Algorithm 1* was applied to a limited 3D solid dataset of the nearby campus area while *Search Algorithms 2 & 3* were applied against a complete 2D polygon map of Dublin City with building heights stored as a non-spatial attribute. Notice how 3D querying on 2D datasets proves to be usefully quicker - even though

Oracle Spatial R-tree indexing limits the search space to only those database objects that are within a specified 2D radius in either case.

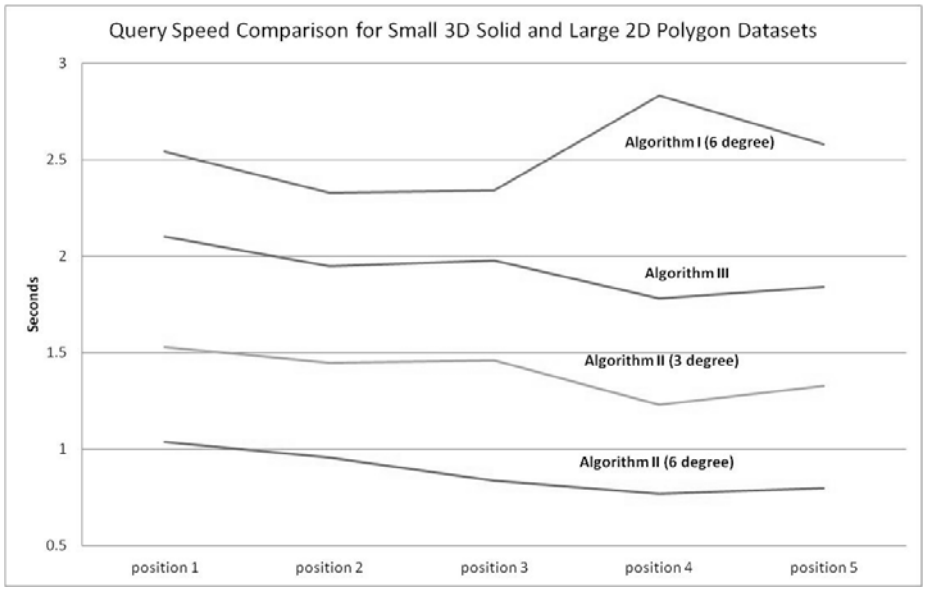

Fig. 8. Comparisons of query speed for different search algorithms at 5 positions on 3D solid and 2D polygon datasets of Dublin

Table 1 shows a comparison of accuracy for all visibility query approaches. Compared to a standard range query, where returning information on all 100 sensored *things* would overload the mobile display at every query position, *Search Algorithm 3* is shown to return the most visible sensors in every query position.

Table 1. Comparisons of accuracy of 3 Search Algorithms at 5 query positions together with the maximum number of sensors actually visible at each position

	Position 1	Position 2	Position 3	Position 4	Position 5
Algorithm I (6^o ray spacing)	19/33	14/32	20/34	5/12	12/26
Algorithm II (6^o ray spacing)	20/33	18/32	20/34	9/12	13/26
Algorithm II (3^o ray spacing)	25/33	26/32	29/34	10/12	20/26
Algorithm III	30/33	29/32	32/34	11/12	24/26

The above query speed experiment was run again on a complete Dublin City 3D solid dataset using all search algorithms. The results of this test are shown in Figure

9. It can be seen that for each search algorithm, the time taken to complete a 3D query on 3D solid data is noticeably longer than when performed on 2D data of the same area with height stored as an attribute. The reason for such a large dip in Algorithm 3 timings at Position 4 is because there are only 12 sensors visible due to the restricted visible search space at this location.

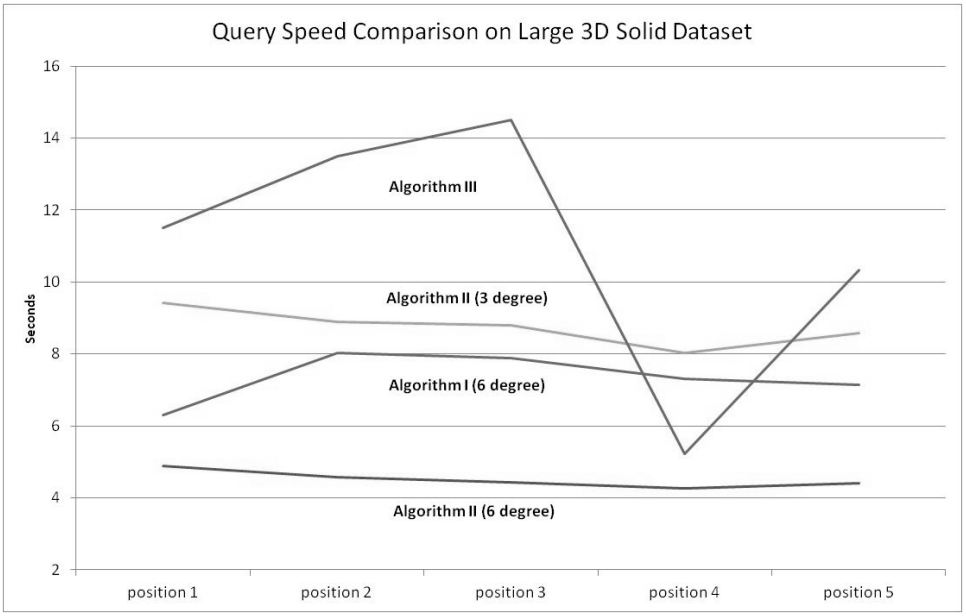

Fig. 9. Comparisons of query speed for different search algorithms at 5 positions on 3D solid dataset of Dublin City

5 Conclusions and Future Work

This paper presented three Spatial Search Algorithms developed in our 3DQ prototype for determining the 3D visibility shape (threat dome) at a user's current location in a built environment. Users then utilize this 3D shape as their query "window" to retrieve information on only those objects visible within a spatial database. Visibility shape searching addresses the information overload problem by providing "Hidden Query Removal" functionality for mobile LBS.

We believe that this functionality will be especially useful in the impending *Future Internet* era where trillions of micro-sensors become available for query through IP access. Analogous to scenes from *Star-Trek*, with Spock scanning his "tri-corder" for readings of life and atmospheric conditions on some strange world, such "situation awareness" queries would also be very interesting to bikers, joggers, walkers, city workers, and all concerned parents and citizens alike on this world who want to know, for example, the health of their immediate environment at any point in time.

Speed and accuracy are two very important requirements of any mobile LBS application. Among the three search algorithms tested, Algorithm 3 shows the most accuracy while Algorithm 2 has the fastest speed. Further study into the trade-offs between speed and accuracy are underway to find the most suitable approach for deploying 3DQ in a real-world mobile eCampus application for student/staff users.

Our ultimate goal is to make 3DQ work in real-time, or near real-time, where the display of the visibility dome shape changes dynamically on top of the 3D map display as a user walks around their city. Currently, the Google Maps 5 visualisation API is not yet publically available for this; however, there are alternatives that can be used for testing. For instance, VisioDevKit [21] provides 3D rendering capability such that any 3DQ visibility shape can be overlayed.

To achieve our real-time threat dome calculation/visualisation goal, improvements to the 3 search algorithms will also be further investigated. As our prototype currently adopts a "client-server" architecture, performance is mainly constrained to the latency of mobile networks. As mobile devices become more powerful as computing platforms, together with some open source mobile spatial database options, implementations of Search Algorithms 2 and 3 entirely on the mobile device will be investigated.

Acknowledgements

Research presented in this paper was funded by a Strategic Research Cluster Grant (07/SRC/I1168) by Science Foundation Ireland under the National Development Plan. The authors gratefully acknowledge this support.

References:

[1] Google Maps 5 (2011): Retrieved from http://googlemobile.blogspot.com/2010/12/next-generation-of-mobile-maps.html, accessed 15-10-2011

[2] S. Di Martino, F. Ferrucci, G. McArdle, G. Petillo (2009): *Automatic Generation of an Adaptive WebGIS* in: 9th International Symposium on Web & Wireless GIS (W2GIS2009), J.D. Carswell et al. (Eds.), Springer LNCS vol. 5886, Ireland, December 2009, pp.171-186

[3] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai and M. Matera (2002): *Designing Data-Intensive Web Applications* in: The Morgan Kaufmann Series in Data Management Systems, Morgan-Kaufmann Publishers, ISBN-13: 978-1558608436, San Francisco, December 2002

[4] E. MacAodih, D. Wilson, M. Bertolotto (2009): *A Study of Spatial Interaction Behaviour for Improved Delivery of Web-Based Maps* in: 9th International Symposium on Web & Wireless GIS (W2GIS2009), J.D. Carswell et al. (Eds.), Springer LNCS vol. 5886, Ireland, December 2009, pp.120-134

[5] D.M. Mountain (2007): *Spatial Filters for Mobile Information Retrieval* in: 4th ACM Workshop on Geographical Information Retrieval (GIR'07), Publisher: ACM Press, Lisbon, November 2007, Pages: 61-62

[6] Ball, M. (2011): *How do crowd sourcing, the Internet of Things and Big Data converge on geospatial technology?* in: V1 Magazine Vol 5, Issue 41: http://www.vector1media.com/-

dialog/perspectives/23362-how-do-crowdsourcing-the-internet-of-things-and-big-data-converge-on-geospatial-technology.html, [Accessed October 11, 2011]
[7] Benedikt M. L. (1979). To take hold of space: isovists and isovist fields. Environment and Planning B, vol. 6, pp. 47-65
[8] Fisher-Gewirtzman D. and Wagner I.A. (2003). Spatial openness as a practical metric for evaluating built-up environments. Environment and Planning B: Planning and Design, 30 (1), pp. 37-49
[9] Fisher-Gewirtzman D., Burt M. and Tzamir Y. (2003). A 3-D visual method for comparative evaluation of dense built-up environments. Environment and Planning B: Planning and Design, 30(4), pp. 575-587.
[10] Morello, E. and Carlo, R. (2009). A digital image of the city: 3D isovists in Lynch's urban analysis. Environment and Planning B: Planning and Design, 36(5) , pp.837-853
[11] Bartie, P., Mills, S. and Kingham, S. (2008). An egocentric urban viewshed: A method for landmark visibility mapping for pedestrian location based services. In Moore, A. & Drecki, I. (Eds.) Geospatial Vision – New Dimensions in Cartography. New Zealand, Springer, pp. 61-85
[12] Bartie, P., Reitsma, F., Kingham, S. and Mills, S. (2010). Advanced visibility modelling algorithms for urban environments. Computers, Environment and Urban Systems, vol. 34, pp. 518-531
[13] SkylineGlobe. (2011). TerraExplorer viewer for 3D earth. Available from http://www.-skylinesoft.com/, [Accessed October 11, 2011]
[14] Ericson, C. 2005. Real-time collision detection. ISBN 1-55860-732-3
[15] Bergen, G. (2004). Collision detection in Interactive 3D environment. ISBN 1-555860-801-X
[16] Watt, A. and Policarpo, F. (2011). 3D Games: Real-time rendering and Software Technology. ISBN 0201-61921-0
[17] Carswell, J.D. (2010). 3DQ: Threat Dome Visibility Querying on Mobile Devices. GIM International, Vol.24, (8), 24, August 2010
[18] Obermeyer, K.J. (2008). The VisiLibity library. Available from http://www.VisiLibity.-org, accessed 11st October, 2011
[19] Ravada, S., Kazar, B.M. and Kothuri, R. (2009). Query processing in 3D spatial databases: Experience with Oracle Spatial 11g. 3D Geo-Information Sciences, pp.153-173, DOI 10.1007/978-3-540-87395-2
[20] 3D isovist (2011). An Grasshopper extension for calculating 3D isovist. Available from http://parametricmodel.com/3DIsovist/32.html, [Accessed October 11, 2011]
[21] VisioDevKit (2011). A real-time 2D/3D rendering engine for mobile devices. Available from http://www.nn4d.com/site/global/developer_resources/apis_sdks, [Accessed October 11, 2011]