Real-Time Visualization of Domain Coverage By Dynamically Moving Sensors

James R. Miller
Electrical Engineering and Computer Science
University of Kansas

Abstract

We describe a collection of algorithms, visualizations, and interactive operations that allow operators controlling the movement of a collection of sensors through an environment to monitor in real time the portion of the environment that can or cannot be seen by some subset of the sensors. The visualization allows preattentive detection of the number of sensors that can see a given location, and the coloring allows the exact identities of the sensors involved to be identified. Two primary ray tracing based algorithms are described. A GPU implementation using CUDA is used that allows all scene updates and sensor movements to be processed and reflected in the display in real time.

1.0 Introduction

Sensors and sensor placement play a central role in military as well as several other general security applications. Such sensors play a critical role in applications related to military situational awareness in combat zones and unfriendly urban areas. Other examples include security cameras for homes and businesses [Garaas 2011], motion detectors in sensitive areas, detecting dangerous materials dispersed into the air [Gruber and Grim 2004], and many others.

Of particular importance in military applications involving unfamiliar environments in which the presence of uneven terrain, buildings, and other major line of sight obstructions exist is the ability to dynamically move a collection of sensors independently throughout an environment while monitoring exactly which parts of the environment are visible to which sensors. As sensors are moved, the field of view and line of sight analysis must be updated and reflected in displays in real time.

In this paper we describe a pair of algorithms, each of which achieves these goals in slightly different and potentially complementary fashions. One approach samples the field of view of each sensor with lines of sight that are truncated at the first surface element encountered. The other algorithm determines the portions of all surface elements in the scene that can be observed by each sensor and "paints" the surface with a dynamically generated colored texture pattern that allows preattentive observation of the number of sensors that can see the area. The colors in the pattern identify the specific sensors that are able to see each piece of surface area.

2.0 Previous Work

Several efforts have targeted various aspects of this general problem. Some methods employ 2D algorithms, but to be effective in the sorts of applications we are addressing, the analysis and display must be 3D, because it is important to take uneven terrain, buildings, and other large structures into account when determining sensor visibility.

The most common 3D approach is to utilize ray casting in some fashion to simulate sensor lines of sight. The challenge is to compute sufficiently detailed information in this way so as to allow at least near real-time analysis while producing high quality and high resolution results.

Livingston and Herbst described an interactive system that allows sensors to be placed in an open environment and basic line of sight analysis to be performed [Livingston & Herbst 2005]. They can determine relationships such as which sensors have line of sight contact with which other sensors. They can determine strings of sensors, each of which can "see" the next sensor in the string. They can determine for a given area of the display how many sensors can see into the area, taking terrain elevations and other properties of the ground that may affect the physical operation of the sensors into account. They do not support non-terrain features such as buildings, nor do they visually identify which sensors can see a given area, only the number of them that can.

Becker described a method to automatically position a collection of sensors in an environment so that all pre-determined critical areas are covered [Becker, Guerra-Filho, & Makedon 2009]. It does not address sensor movement, nor does it attempt to quantify or display the numbers or identities of specific sensors that can see a given area. Its analysis leads to a purely binary "can or cannot be seen".

Garaas presented an interactive tool for simulating the positioning of video cameras in a 3D environment and determining and visualizing the overall coverage of the environment by the collection of cameras [Garaas 2011]. Their approach differs from the others (and from ours) in that it initially creates a voxel grid of some resolution enclosing the entire scene. It then casts rays from a sensor's field of view through the voxel grid, marking each voxel encountered before hitting a surface. They track how many sensors have rays passing through each voxel. Instead of painting the actual objects in the scene according to whether they are visible or not, they "color" the open spaces (i.e., the voxels comprising the air) according to the results of this visibility analysis. Several coloring schemes are supported. For example, a voxel color might indicate the number of cameras that can see the voxel. Alternatively, since their analysis is capable of tracking the range of angles from which a given voxel can be seen, this range can be used to determine the voxel color. Regardless of the specific scheme, the visualization appears as colored air in the open spaces. While this approach is quite different from ours, the display we generate from the first of our two algorithms is somewhat similar in appearance and information content.

The second of our two analysis algorithms leads to a characterization of those sensors that are able to see each object vertex in our scene. We can use this to approximate the extent of a given surface visible to a given sensor, and we will want to visualize not only the number of sensors involved, but also be able to identify the exact ones. We use an approach based on Attribute Blocks [Miller 2007] to present this visualization. Attribute Blocks are used to map onto the surface of an object a visual representation of a set of n attributes whose values are defined continuously across the surface. (In the case of the work here, the n attributes will be visibility flags for n sensors.) The approach uses an OpenGL GLSL shader

program [Rost & Licea-Kane 2010] to dynamically generate and apply a texture whose pattern encodes both the number of attributes being visualized as well as their current values. It is especially effective in applications such as this where the exact quantitative value of an attribute is not of primary importance (although we will show some experimental results where this might be of some value), rather we are concerned with identifying the number and identity of sensors that can see a particular portion of a surface.

The goal is to display a pattern that will allow preattentive determination of the number of sensors that can see the area, and whose colors identify the specific ones. Ideally the pattern should be comprised of regular polygons. To aid in analysis and help ensure equal coverage, each atomic piece of the pattern should correspond to one sensor, and its assigned Attribute Block cell should share an edge with that for each of the other sensors that are also able to see that area. Arguments for the value of edge-sharing are echoed by [Malik, Heinzl, and Gröller 2010] who go on to describe a method based on a uniform tiling of the plane with hexagons that they use for all values of n. Our approach uses instead rectangular Attribute Blocks for the n = 1, 2, 3, and 5 cases. The n = 4 case will be handled using triangular Attribute Blocks. More complete details (as well as issues related to other values of n) are described below.

3.0 Visibility Determination, Interaction, and Visualization

Our over-arching goal is to provide an interactive visual monitoring and directing tool for operators, i.e., individuals who monitor the environment and interactively direct the movement and orientation of the sensors. The operators can view the environment, including the current positions and orientations of all sensors, and visualize those portions of the environment that are or are not currently visible to the sensors. When sensor coverage areas overlap, the visualization will indicate the number of sensors that can see any given area and identify the exact sensors involved. Based on this information, the operators can know how best to interactively reposition and/or reorient one or more sensors (e.g., "drive them around the environment"). As these repositionings are executed, the display is dynamically updated.

The sensor line of sight analysis must take into account potential blocking of all or part of the sensor's view by terrain, buildings, etc. [Livingston & Herbst 2005]. An additional requirement for our application is that the line of sight analysis and display be dynamically updated as sensors are moved throughout the environment.

To achieve real-time display updates, both algorithms we developed utilize GPU-based ray casting. We developed our algorithms using the CUDA toolkit and libraries [NVIDIA 2011]. We first discuss the common representation of sensors used in this application, and then we discuss the two major techniques developed and the visualization techniques used to present the results of the analysis.

3.1 ***Input****: Environment and Sensors*

Our representation of the environment is straightforward, consisting of a collection of an arbitrary number of abstract scene items. Each scene item consists of a piecewise linear

construct akin to OpenGL draw modes (triangle strip, triangle fan, etc.) The scene is read at program startup time from a standard file format. The program does not currently distinguish between buildings, terrain, or other objects – all surface elements are treated the same – however it is possible to augment individual scene items with attributes that might be relevant to certain types of sensors. We return to this idea at the end of the paper.

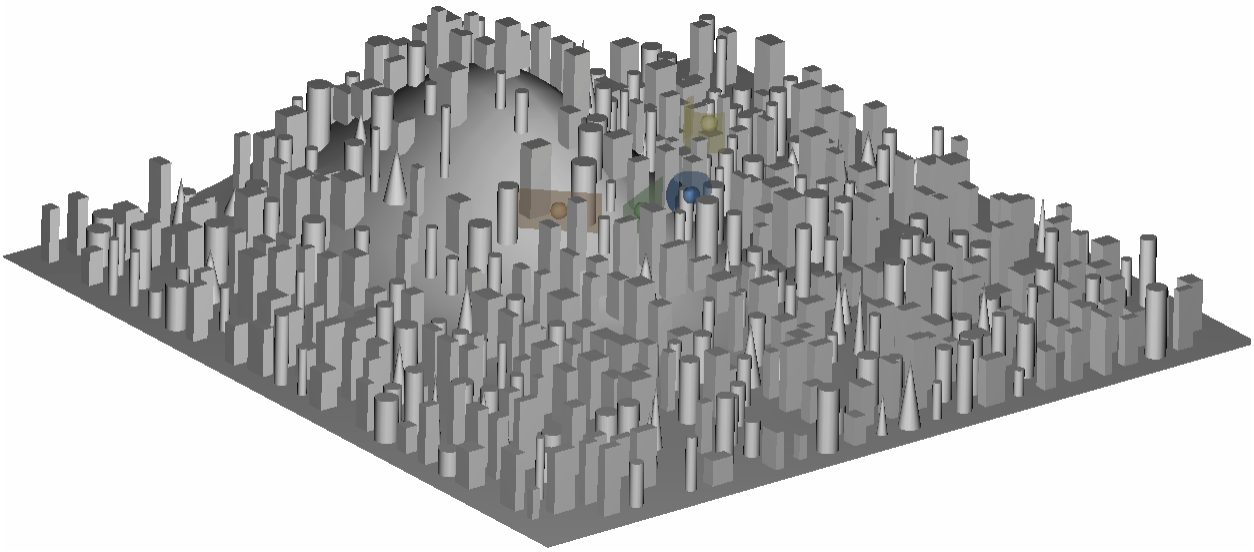

Figure 1: The geometry of the city used for the study. Four sensors are also shown.

The scene used to test the approaches in this paper is shown in Figure 1. We use a neutral color with no texture mapping for now, primarily because we use color and texture for displaying sensor visibility, and we do not want to confuse building color and texture with that created based on sensor visibility.

The scene was constructed by a program of our own design that generates a pseudo-random collection of "buildings" of various sorts placed on a terrain that can be configured to include hills such as the one visible in Figure 1. We used this program in part for performance testing purposes since the buildings and terrain can easily be generated with an adjustable number of vertices (and hence triangles) per building allowing us to experiment with increasingly complex scenes while watching the impact on performance. It also allowed us to observe how the quality of results can improve with increasing vertex density. Specifically, the second of the two algorithms described below operates on a per-vertex basis, so increasing vertex densities produces higher resolution results, but at the obvious cost of performance.

The program also supports sensors of different types. Currently, the primary difference is how the family of sensor lines is determined. Sight lines from a spherical sensor start from a common point (the sphere center). A spherical sensor may emit sensor lines in all directions, or the lines may be restricted to a subset of the spherical surface defined by a minimum and maximum θ and ϕ angle. The sight lines for a cylindrical sensor are defined over a finite subset of an infinite right circular cylinder. All sight lines from cylindrical

sensors are perpendicular to the cylinder axis. The third type of sensor is a planar sensor. All sight lines for planar sensors are perpendicular to the plane and are restricted to a finite rectangular area of the plane.

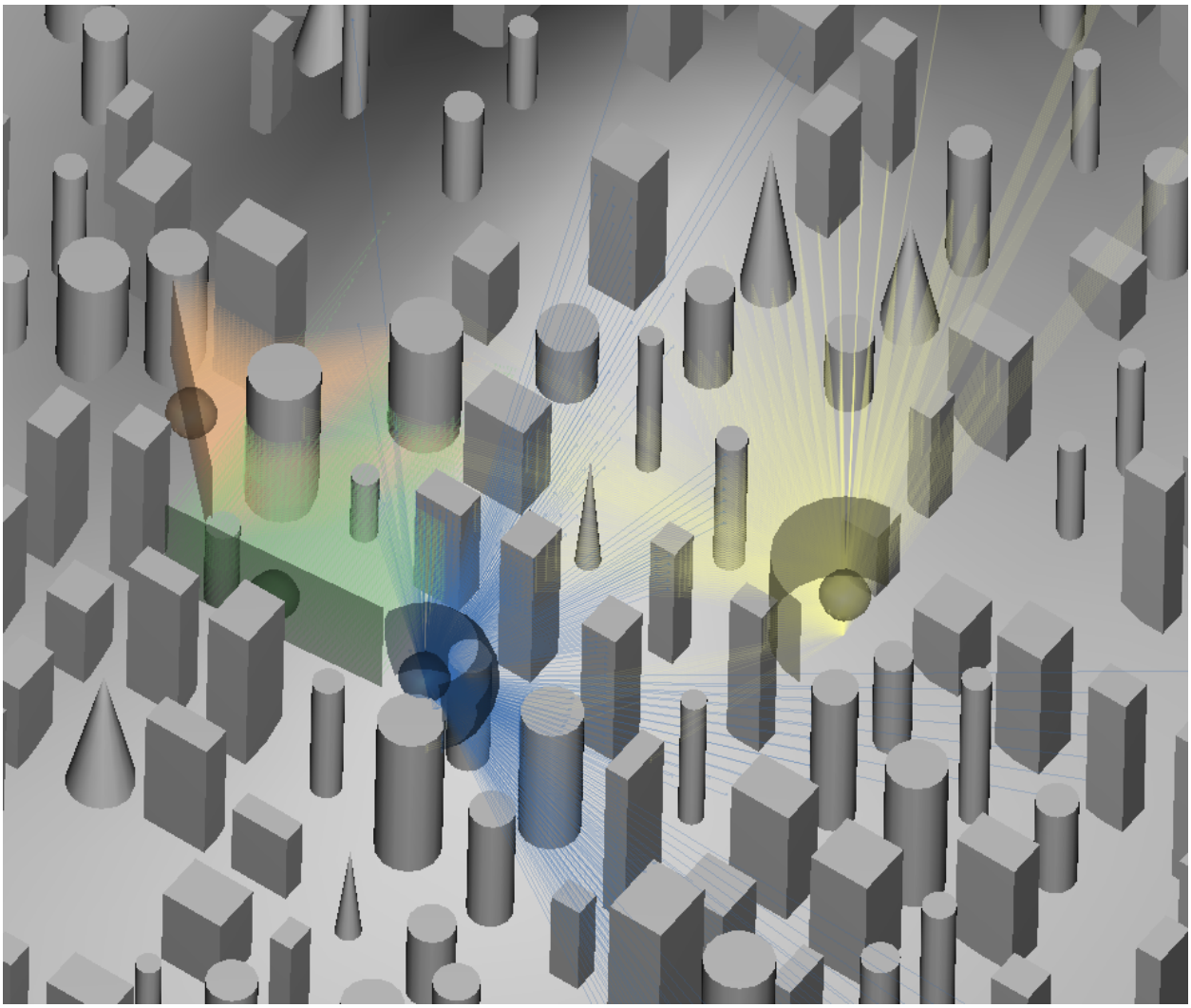

Figure 2: Samples of each sensor type along with a representation of their field of view.

The collection of sensors is read at program startup time. A simple file format is used to read the relevant data. An arbitrary number of sensors can be specified. In Figure 2, we have zoomed into the area of the sensors shown in Figure 1. The figure includes a sample of each sensor type currently supported, and it specifically illustrates their respective fields of view by showing a sampled set of sight lines emitted from the sensors.

3.2 Processing and Visualization

Once the environment and initial sensor descriptions have been processed, two pairs of analysis and rendering algorithms can be used. One samples the field of view of each sensor, tracing a dynamically configurable number of sight lines into the environment,

drawing the portion of each sight line up to its first point of obstruction. The other tests each scene vertex with respect to the field of view of each sensor and the other surfaces in the scene, associating a per-sensor visibility mask with each vertex. From this information, an OpenGL GLSL shader program generates a dynamic colored texture that encodes the exact set of sensors that can see each portion of the scene.

While we normally use one algorithm or the other at any one time, both could be used together. Both algorithms rely on ray tracing, but the second is much more computationally demanding. The following two sections describe each algorithm in turn. Our CUDA-based GPU implementation of the two algorithms operates in real time as the sensors are moved around the scene. Details are in section 3.3.

3.2.1 **Algorithm 1**: Sampling Field of View Sight Lines

A grid is imposed over the field of view of each sensor, and one line of sight ray is traced through each grid point. The ray is traced up to its first intersection point with a surface in the scene. Various display options are available for the trimmed sight lines. While quite different in implementation than the voxel-based algorithm of [Garaas 2011] described earlier, the displays often look similar in that they can be made to "color the open air spaces" visible to given sensors as Figure 2 shows. This is especially obvious for the two "planar" sensors of Figure 2.

3.2.2 **Algorithm 2**: Painting Surfaces with Dynamically Generated Visibility Textures

A two-stage CUDA-based algorithm is used to characterize, for each scene vertex, the exact sensors to which it is visible. In the first stage, vertices are filtered according to the field of view of the sensor. Those vertices found to be inside a sensor's field of view are then passed to the second stage of the algorithm in which a ray is traced from the sensor to each vertex. The use of this two-stage algorithm allowed us to maximize use of bounding volume techniques while minimizing the number of thread processors that sat idle because of early out tests. At the end of the two-stage algorithm, a visibility bit mask is associated with each vertex, and the bit for sensor *i* is set if and only if the vertex is visible to sensor *i* (i.e., if it is inside the field of view, and no surfaces are found between the sensor and the vertex).

The per-vertex visibility flags are interpolated across the triangles of which they are a part. The GLSL fragment shader receives these interpolated flags as floating point values between 0 and 1 for each sensor. Each value can therefore be interpreted as a probability that the pixel is visible to the corresponding sensor. The goal of the visualization is then to convey the exact set of sensors that can see each portion of a building. The general approach is to retain the neutral color of Figure 1 for portions of the environment invisible to all sensors, and to color the rest according to the set of sensors to which they are visible.

Since we can interactively turn individual sensors off and on, we could cycle through (automatically or under user control) the various sensors, coloring portions of buildings visible to each sensor in turn. While our system allows this, our standard approach is to use Attribute Blocks [Miller 2007]. Attribute Blocks can be used to visualize the values of several attributes continuously defined across some domain. A $k_r x k_c$ rectangular pattern is

defined in which each cell displays the value of a single attribute. The $b_r x b_c$ size of each cell in the $k_r x k_c$ pattern can be independently adjusted, either in pixel space or in model space.

For the applications described in [Miller 2007], the user was given interactive controls for adjusting $k_r x k_c$, $b_r x b_c$, and the mapping of attributes to cells in the Attribute Block pattern. In the application here, the "attributes" are the non-zero visibility flags, and we want to always display all non-zero flags. Hence the user has no control over the $k_r x k_c$ pattern used other than being able to turn specific sensors off and on. Instead we dynamically determine in the fragment shader (i.e., pixel by pixel) what the $k_r x k_c$ pattern should be – including a special case in which we locally switch to triangular Attribute Blocks as described below – and assign the visibility flags to cells in the pattern. The user/operator is able to control the cell sizes (i.e., $b_r x b_c$) and whether the sizes and Attribute Block orientation is in pixel space or model space.

Our premise when determining the required Attribute Block pattern is that all visibility flags are of equal importance and should generally be accorded equal display space. Moreover – whenever possible – each sensor cell should share an edge with a cell for all other sensors that can see the area. This arrangement has been found to maximize our ability to conceptualize coverage in an area and is consistent with similar observations made in [Malik, Heinzl, & Gröller 2010].

If we let *n* stand for the number of sensors that can see a given point and use letters to identify sensors, then designing an Attribute Block pattern for *n=1, 2, 3,* and *5* such that each block shares an edge with all the others is trivial to construct. The *n=1* case is simply a solid color; Table 1 shows the patterns for the *n=2*, *3*, and *5* cases.

n	k_r x k_c	Attribute Block Layout
2	2x2	AB BA
3	3x3	ABC BCA CAB
5	5x5	ABCDE CDEAB EABCD BCDEA DEABC

Table 1

The *n=4* case is somewhat challenging because of our shared edge and equal coverage requirements. An alternative that supports the goal of preattentive identification of numbers of sensors that can see an area is to switch to a triangular Attribute Block pattern for the *n=4* case. All sensors are assigned a triangle, and each can easily share an edge with a triangle for another sensor.

The *n=6* case is challenging because that would require a pentagonal tiling, and there is no good, easy-to-generate pentagonal tiling of a plane. The *n=7* case could use a hexagonal tiling which is easy to generate. We have not yet had a need to handle the *n>5* case, so we will not discuss that further in this paper.

We actually use a modified pattern for the *n=3* case. One of our goals was to support preattentive identification of the *number* of sensors that can see a surface, with subsequent interpretation of color allowing operators to identify the specific sensors. By swapping the 'A' in the upper left corner of the *n=3* pattern with the 'B' in the lower right corner, we

produce a pattern that is very distinctive in that we see two L-shaped notches with a diagonal pattern between them. In Figure 3, areas that are visible to 0, 1, 2, 3, and 4 sensors are immediately obvious since the patterns are, respectively, solid neutral color, solid sensor color, L-notched, simple checkerboard, and triangular. Several areas visible to exactly two sensors can be seen. Thus the operator immediately knows the number of sensors covering the area (critical in military applications [Livingston & Herbst 2005]), and further color-based examination reveals which sensors they are.

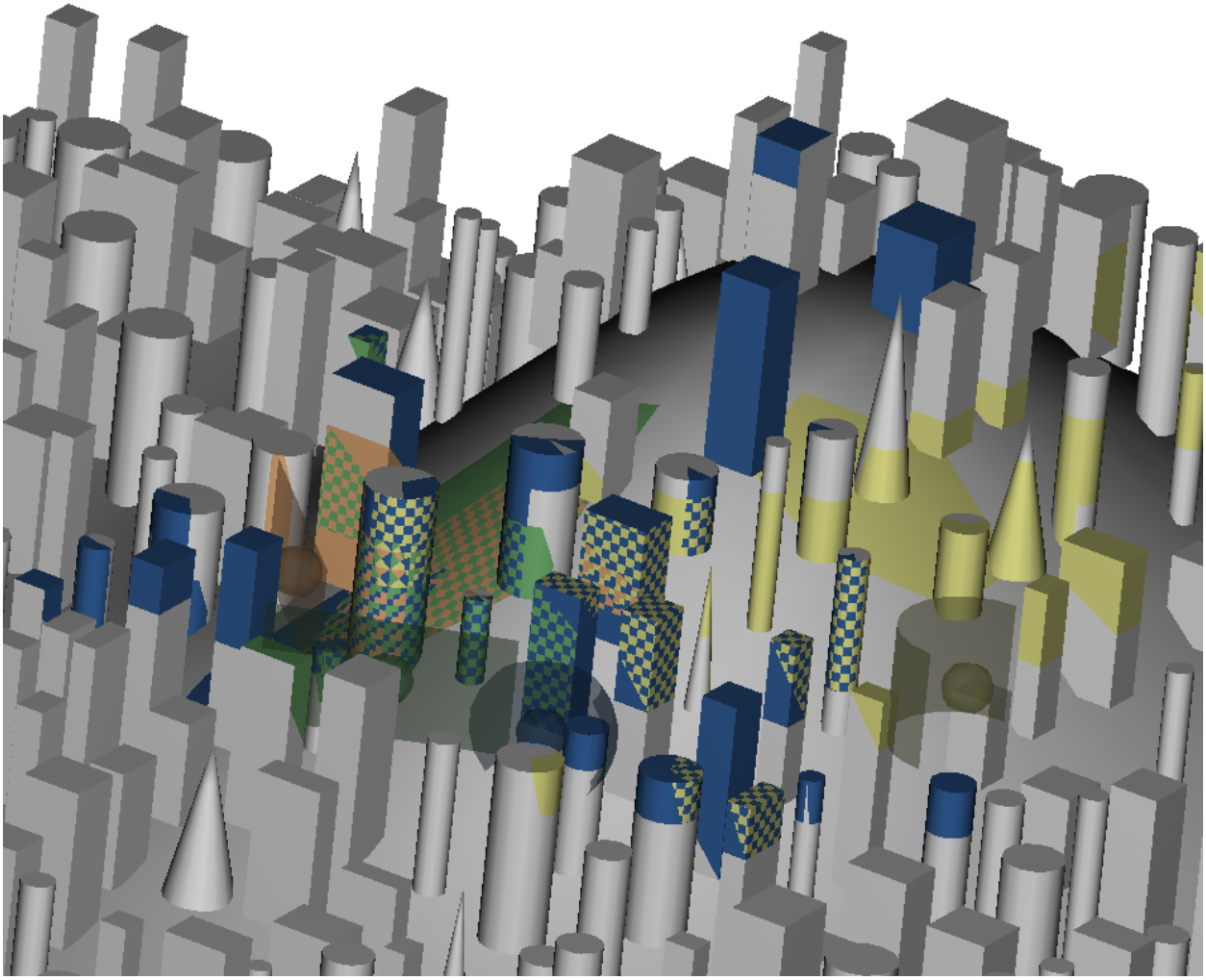

Figure 3: Areas visible to 0, 1, 2, 3, or all 4 sensors are immediately obvious.

Notice that the "*n*" (and hence the structure of the texture) varies from region to region along a surface. Fortunately, all texture computations are done on a pixel-by-pixel basis in the fragment shader, and we never need create a complete standalone texture pattern anywhere. At each fragment (pixel), we know which sensors see it, hence we know what abstract pattern should be applied, and we use modulo arithmetic akin to that described in

the original Attribute Block paper to determine where in the abstract pattern we are, and hence assign the corresponding pixel color.

The Attribute Block patterns themselves can be sized and oriented in screen space or model space. In screen space, the edges of the blocks are always aligned along pixel rows in columns; in model space, they are aligned with axes associated with the local coordinate systems of the individual objects in the scene.

Screen space patterns are often best when the scene is not dynamically rotated because the patterns tend to be more sharply defined. This is especially true when the cell sizes ($b_r x b_c$) are small in order to provide high resolution displays. If the scene undergoes frequent dynamic view manipulations, screen space patterns can be distracting because they are not locked to the object; instead they appear to "slide around". Moreover the fact that the cells in the patterns retain their same pixel size during zooming can sometimes be confusing. Model space patterns are frequently better in these cases since they are defined with respect to the local coordinate system of each object and hence remain locked in place on the object as it moves during rotation and panning. The cell sizes also grow and shrink as expected as the operator zooms in and out. Even when no dynamic view manipulations are involved, model space patterns are frequently preferred when the $b_r x b_c$ cell sizes are larger because they better preserve the sense of 3D surface orientations.

The basic color of each cell inside of a given pattern is determined by the color assigned to the sensor that has been determined to be able to see the object there. This color can be modulated in two ways: by adding a lighting model and by attenuation based on the visibility probability. Recall we interpolate the individual 0 or 1 per-vertex visibility flag across the interior of each triangle. If all three vertices have the same flag, then the triangle is uniformly visible or invisible. However, if one vertex has a classification different from the other two, then pixels in the interior of the triangle have a floating point visibility flag somewhere between 0 and 1. This can be interpreted as the probability that this pixel is visible to the given sensor, and the probability can be used to attenuate the color. We thus have – for each of screen space and model space Attribute Blocks – four choices: no attenuation, lighting model attenuation, visibility probability attenuation, and attenuation by both visibility probability and lighting model. Figure 4(a-h) shows the same scene and sensor placement using each of these eight options in turn. Attenuation by both frequently results in images that are too dark. Hence we usually either use neither or just one.

3.3 Performance and Other Implementation Notes

This tool was designed with the primary goal of delivering real-time performance as sensor placement and orientation was dynamically changed under operator control. We have achieved that with our testing environment here. Our city generation software generates the buildings and terrain in the images shown in this paper with a user-definable resolution in terms of numbers of vertices and triangles. In the simplest scene, the city is defined with 15,411 vertices and 14,714 triangles. The most complex version we tested had 32,139 vertices with 48,096 triangles. Display updates during sensor repositioning and

reorienting were instantaneous for both scenes on a 3.2GHz Intel Core 2 Duo running Linux with a Quadro 600 GPU.

One performance tuning technique that proved to be useful – especially for algorithm 2 – was to break up the field of view of a sensor into a subarray, mapping each subset of the field of view to its own kernel launch. (The first stage field of view filtering of course used the decreased field of view corresponding to each element of the subarray.) Then each kernel was broken down into blocks according to a dynamically determined desired number of threads per block. To facilitate the tuning operation, we added GUI controls to adjust this breakdown interactively.

A variety of other interactive options are available. Operators can turn individual sensors on and off. Current sensor positions can be saved to a file. All the color attenuation options and Attribute Block size and space options are dynamically adjustable.

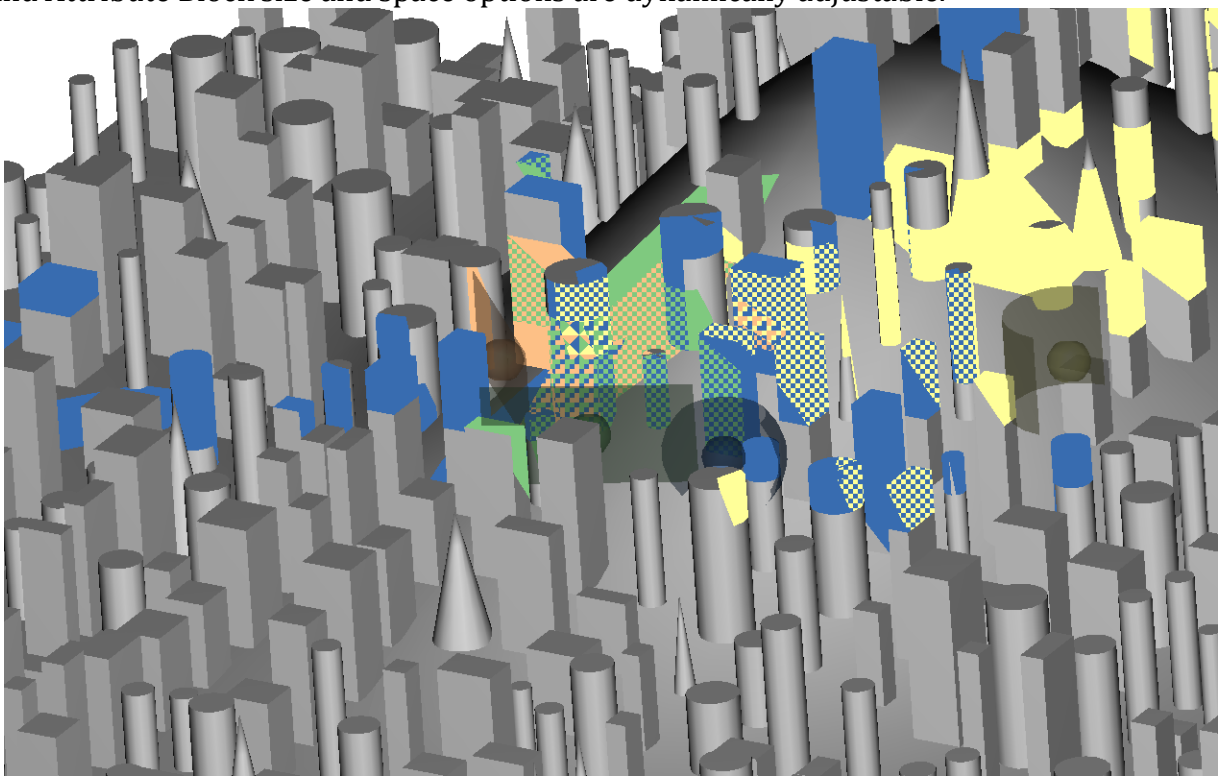

Figure 4(a): Screen Space; Lighting off; Visibility Probability off

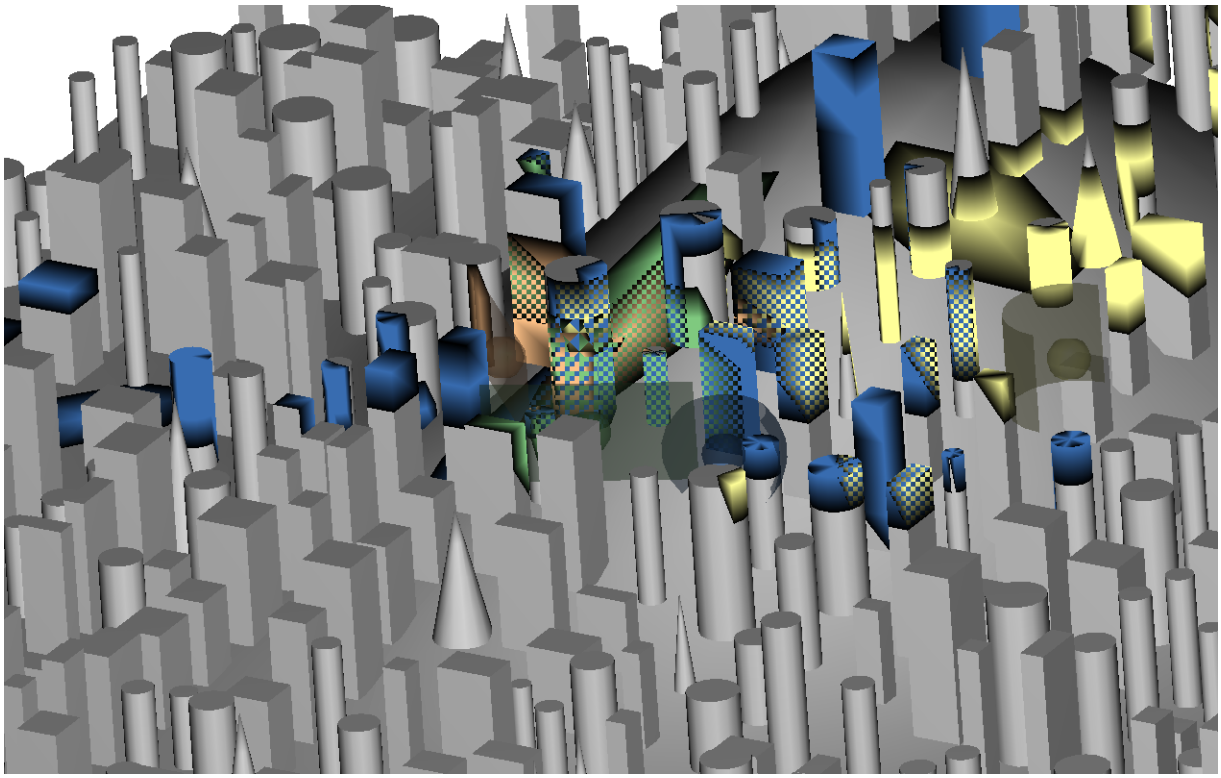

Figure 4(b): Screen Space; Lighting off; Visibility Probability on

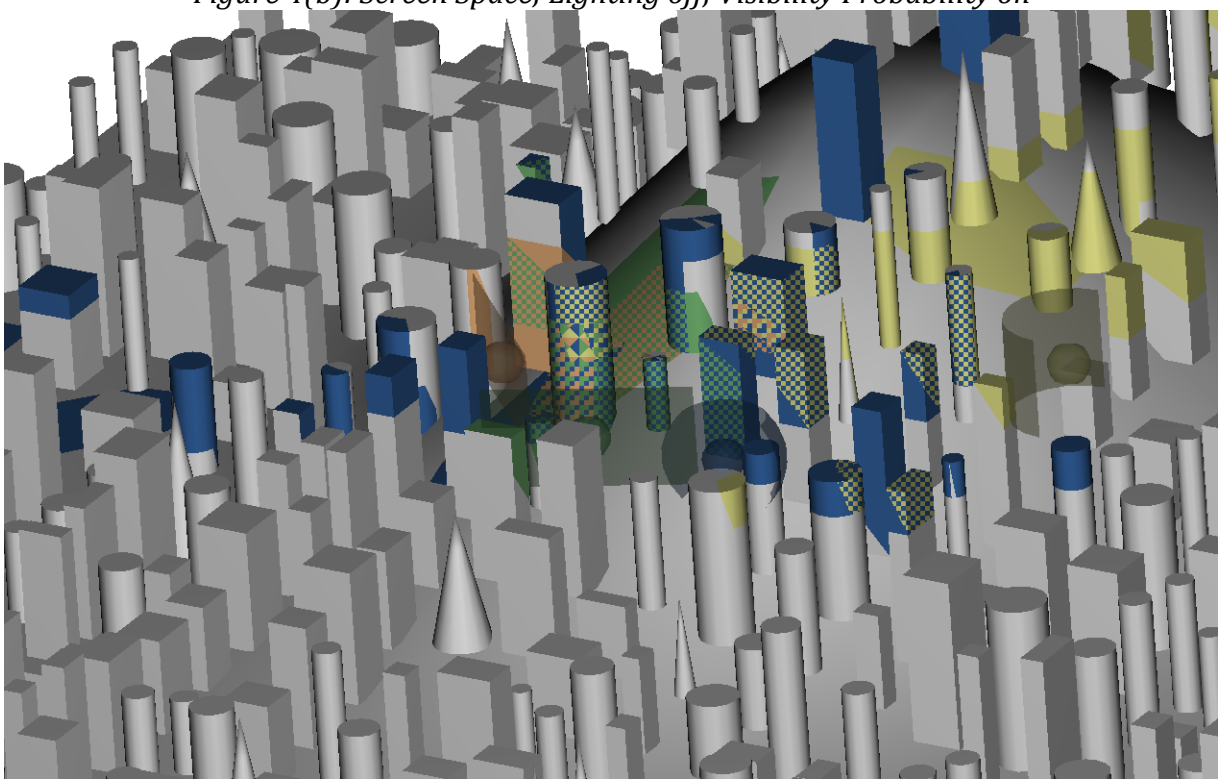

Figure 4(c): Screen Space; Lighting on; Visibility Probability off

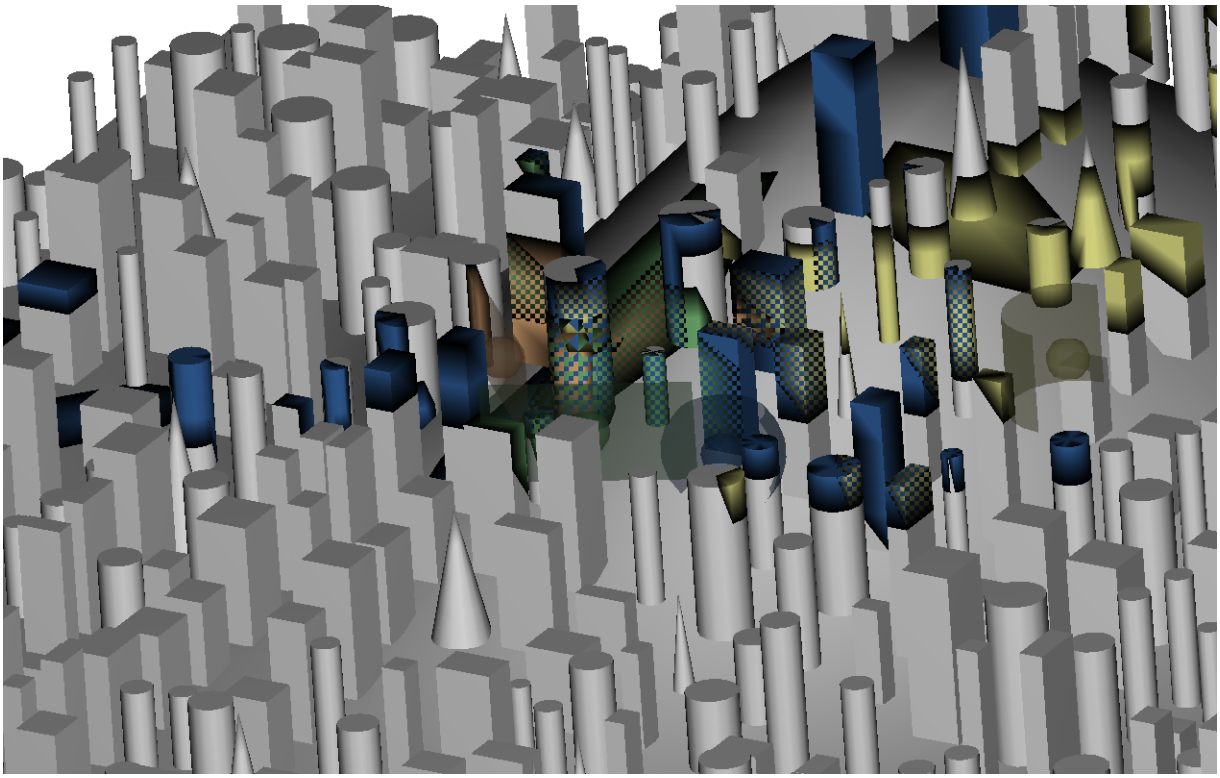

Figure 4(d): Screen Space; Lighting on; Visibility Probability on

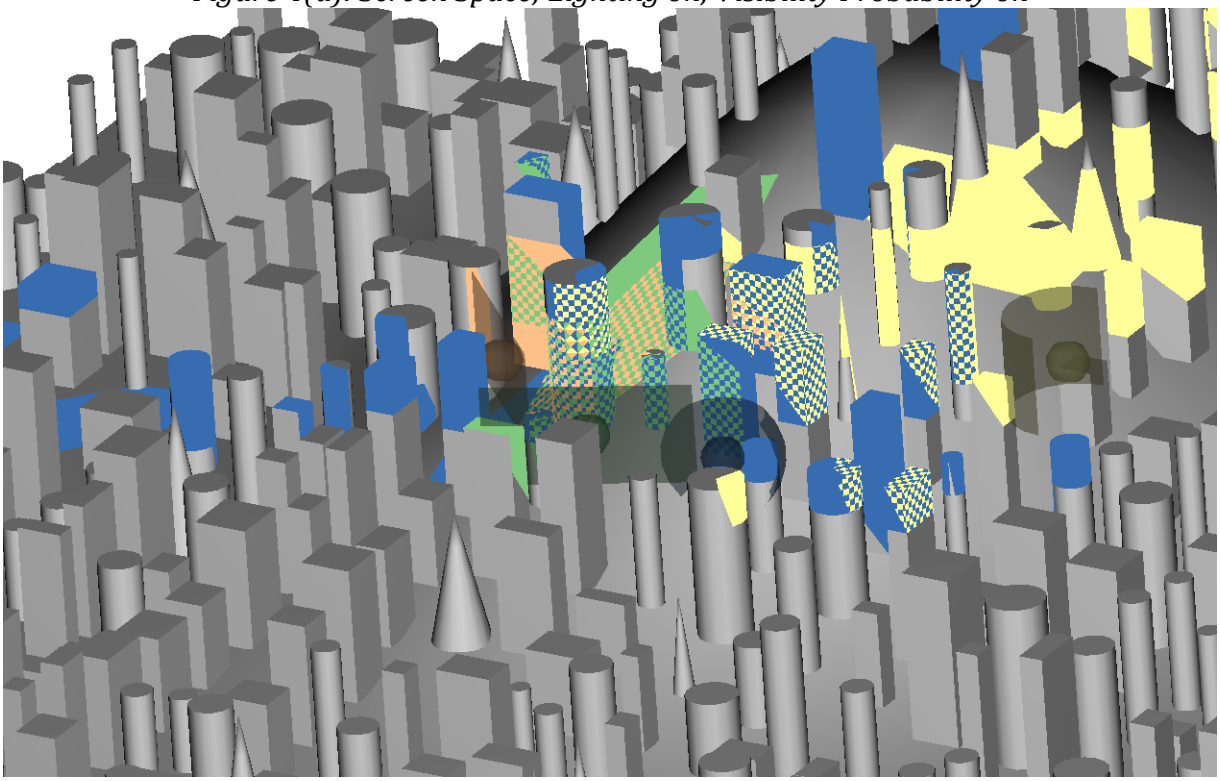

Figure 4(e): Model Space; Lighting off; Visibility Probability off

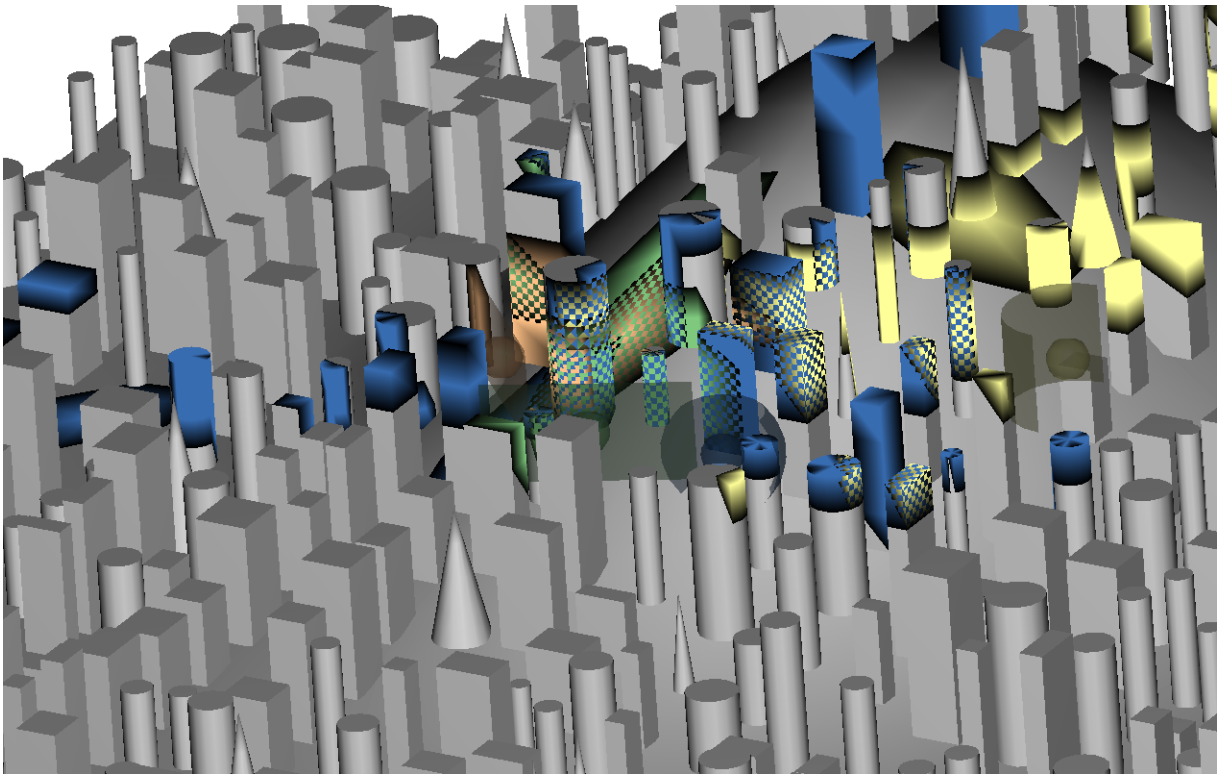

Figure 4(f): Model Space; Lighting off; Visibility Probability on

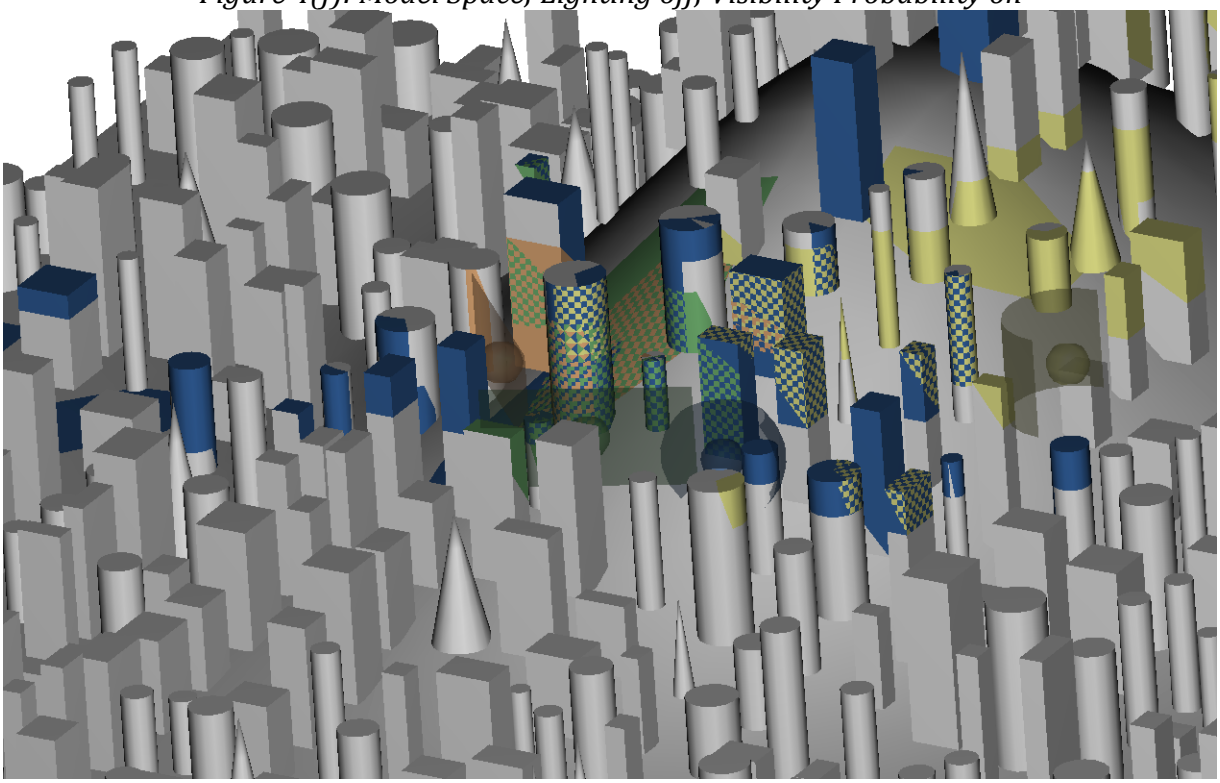

Figure 4(g): Model Space; Lighting on; Visibility Probability off

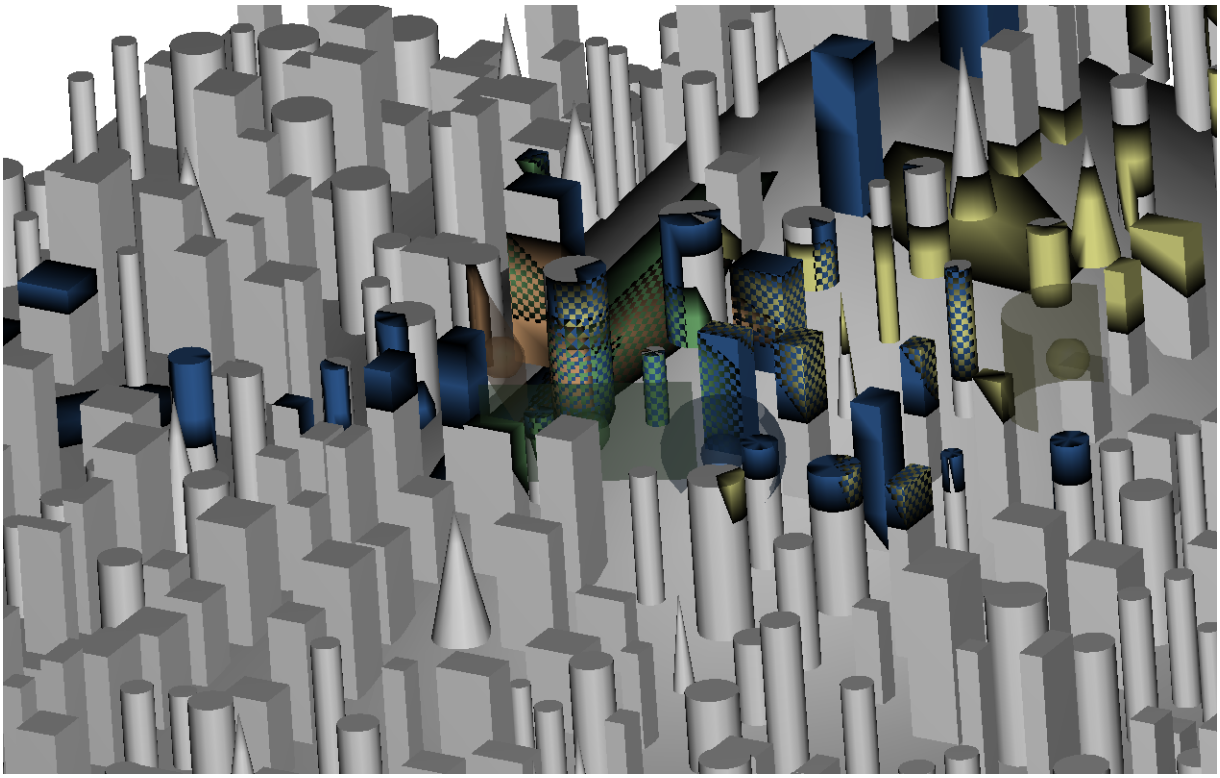

Figure 4(h): Model Space; Lighting on; Visibility Probability on

4.0 Summary, Discussion, and Future Work

We have discussed algorithms and visualization methods that allow operators to decide how sensors should be moved around an environment in real time to achieve required instantaneous domain coverage. Our usage of the system has led us to conclude that the ability to dynamically alter the $b_r x b_c$ Attribute Block sizes, the use of screen space versus model space definitions, and the use of visibility and lighting attenuations, coupled with dynamic view manipulations adds dramatically to the insight one can gain. It takes a short while to get used to the role of the patterns as revealing the number of sensors that can see a region, but once that insight has been achieved, one quickly learns how to manipulate the sensors to achieve desired domain coverage.

We are considering a number of possible extensions. Assigning colors and/or textures to buildings and the terrain might make the scene more realistic, but we need to be careful about over-use of color. Perhaps very muted monochromatic textures could be effective. Another possible extension would be to endow our sensors and scene geometry with additional properties. For example, we could then support sensors that could see through certain types of walls.

Acknowledgments

The colors assigned to the different sensors were selected using the ColorBrewer web site (colorbrewer2.org). I wish to thank the NASA World Wind project team – and especially

Tom Gaskins and Patrick Hogan – for supporting me while prototyping this and other tools for NASA World Wind (worldwind.arc.nasa.gov/java).

References

[Becker, Guerra-Filho, & Makedon 2009] E. Becker, G. Guerra-Filho, and F. Makedon, Automatic Sensor Placement in a 3D Volume, *PETRA '09: Proceedings of the 2nd International Conference on PErvasive Technologies Related to Assistive Environments*, ACM, June 2009, doi 10.1145/1579114.1579150.

[Firey 2007] P. Firey, Visualization for Improved Situation Awareness (VISA), Mitre Technology Program, (pdf of a powerpoint presentation accessed 10 October 2011 from www.mitre.org/news/events/tech07/2630.pdf)

[Garaas 2011] T. W. Garaas, *Sensor Placement Tool for Rapid Development of Video Sensor Layouts*, Mitsubishi Electric Research Laboratories Technical Report TR2011-020, April 2011, 6 pages. (Accessed 10 October 2011 from www.merl.com/reports/docs/TR2011-020.pdf)

[Gruber & Grim 2004] T. C. Gruber, Jr. and L. B. Grim, Visualization of Foreign Gases in Atmospheric Air, *11th International Symposium on Flow Visualization*, August 9-12, 2004, Notre Dame, Indiana, (11 pages).

[Livingston & Herbst 2005] M. A. Livingston and E. V. Herbst, Interactive Operations for Visualization of Ad-hoc Sensor System Domains, *IEEE Mobile Adhoc and Sensor Systems Conference (MASS) 2005*, pp. 341-345.

[Malik, Heinzl, & Gröller 2010] M. M. Malik, C. Heinzl, and M. E. Gröller, Comparative Visualization for Parameter Studies of Dataset Series, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 16, No. 5, Septemer/October 2010, pp. 829-840.

[Miller 2007] J. R. Miller, Attribute Blocks: A Tool for Visualizing Multiple Continuously-Defined Attributes, *IEEE Computer Graphics & Applications*, Vol. 27, No. 3, May/June 2007, pp. 57-69.

[NVIDIA 2011] http://www.nvidia.com/object/cuda_home_new.html (accessed 11 October 2011).

[Rost & Licea-Kane 2010] *OpenGL Shading Language*, R. J. Rost and B. Licea-Kane, Addison-Wesley, (third edition), 2010.