Editor: Mike Potel

Real-Time Visualization of Domain Coverage by Dynamically Moving Sensors

James R. Miller
University of Kansas

Sensors and sensor placement are critical in applications related to military situational awareness in combat zones and unfriendly urban areas. They also play a central role in nonmilitary applications such as security cameras for homes and businesses,[1] motion detectors in sensitive areas, and detecting dangerous materials dispersed into the air.[2]

Of particular importance in military applications involving unfamiliar environments containing uneven terrain, buildings, and other major line-of-sight obstructions 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 move, the field-of-view and line-of-sight analysis must be updated and reflected in displays in real time.

To achieve these goals, I developed a pair of algorithms, each of which achieves the goals in a slightly different and potentially complementary fashion. One samples each sensor's field of view with lines of sight that are truncated at the first surface element encountered. The other first determines the portions of all surface elements in the scene that each sensor can observe. It then paints the surface with a dynamically generated colored texture pattern that allows preattentive observation of the number of sensors that can see the area. (Aspects of an image that are noticed within the first couple hundred milliseconds are generally considered to be preattentive.) The pattern's colors identify the specific sensors that can see each piece of the surface area.

Operational Environment and Approach

My overarching goal is to provide an interactive visual monitoring and directing tool for operators—that is, individuals who monitor the environment and interactively direct the sensors' movement and orientation. This tool should let operators view all the sensors' current positions and orientations and visualize those portions of the environment that are or are not currently visible to the sensors. When sensor coverage areas overlap, the visualization should indicate the number of sensors that can see any given area and identify the exact sensors involved. This information will enable the operators to determine how best to interactively reposition or reorient one or more sensors.

The sensor line-of-sight analysis must take into account potential blocking of all or part of the sensor's view by terrain, buildings, and so on.[3] Also, the analysis and display must dynamically update as sensors move throughout the environment.

Input: Environment and Sensors

My representation of the environment is straightforward; it's 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 strips, triangle fans, and so on). The scene is read during program startup from a standard file format. The program doesn't currently distinguish between buildings, terrain, or other objects—it treats all surface elements the same. However, operators can augment scene items with attributes that might be relevant to certain types of sensors. I return to this idea later.

Figure 1 shows the scene I use to test my method. For now, I use a neutral color with no texture mapping. This is primarily because I use color and texture to display sensor visibility, and I don't want to confuse building colors and textures with those depicting sensor visibility.

To construct the scene, I used a program of my own design that generates a pseudorandom collection of "buildings" of various sorts placed on a

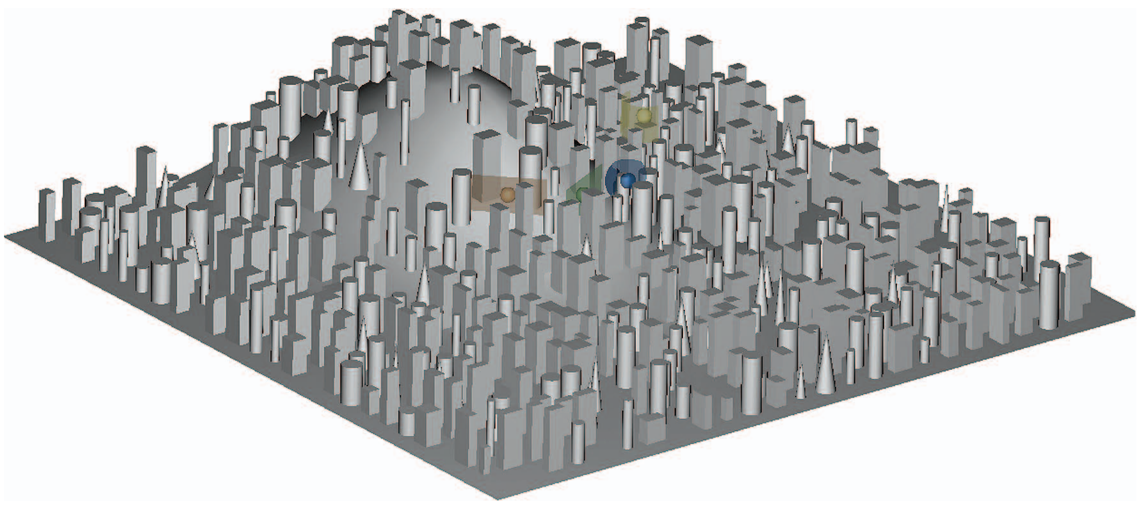

Figure 1. The geometry of the city used for the study. The image also shows four sensors. I use a neutral color with no texture mapping to avoid confusing building colors and textures with those depicting sensor visibility.

terrain that can be configured to include hills such as the one visible in Figure 1. This program can easily generate buildings with an adjustable number of vertices (and hence triangles). This lets me experiment with increasingly complex scenes while watching the performance impact. It also lets me observe how the display quality improves with increasing vertex density. Specifically, my second algorithm operates on a per-vertex basis. So, increasing the vertex density produces higher-resolution results, but at the obvious cost of performance.

The program supports sensors that determine their sight lines differently. For spherical sensors, the sight lines start from the sphere's center and may emanate in all directions or be restricted to a subset of the spherical surface. For cylindrical sensors, the sight lines are defined over a finite subset of an infinite right circular cylinder and are perpendicular to the cylinder axis. For a planar sensor, the sight lines are perpendicular to the plane and restricted to a finite rectangular area of the plane. The operator specifies an arbitrary collection of sensors when the program starts.

Figure 2 zooms in on the area of the sensors shown in Figure 1. The figure includes a sample of each supported sensor type. It specifically illustrates their respective fields of view by showing a sampled set of sight lines emitted from the sensors.

Processing and Visualization

Once my tool has processed the environment and initial sensor descriptions, the operator can use either or both algorithms. To achieve real-time display updates, the algorithms employ GPU-based ray casting. I developed them using the CUDA (Compute Unified Device Architecture) toolkit and libraries (www.nvidia.com/object/cuda_home_new.html).

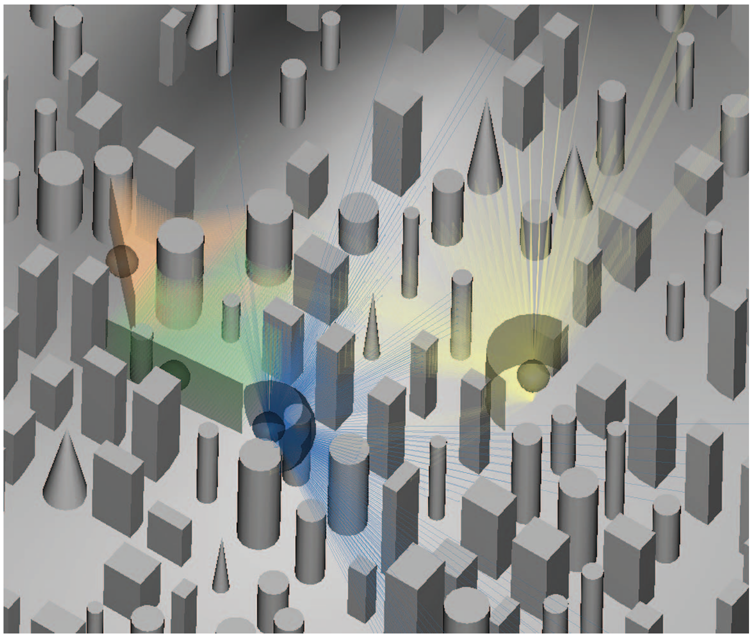

Figure 2. Samples of each sensor type along with a representation of their field of view. I currently use spherical, cylindrical, and planar sensors.

Sampling Sight Lines

My first algorithm imposes a grid over each sensor's field of view and then traces one line-of-sight ray through each grid point. As I mentioned before, it traces the ray up to the ray's first intersection point with a surface in the scene. Various display options are available for the trimmed sight lines.

This algorithm produces displays similar to those created by Tyler Garaas's voxel-based algorithm (see the sidebar), in that they both can color the open spaces visible to the given sensors. This effect is especially obvious for the two planar sensors in Figure 2. However, these two algorithms' implementations differ considerably.

Related Work in Sensor Visibility

Research has targeted various aspects of this general problem. Some methods employ 2D algorithms, but to be effective in the sorts of applications I discuss in the main article, the analysis and display must be 3D. This is because you must take uneven terrain, buildings, and other large structures into account when determining sensor visibility.

The most common 3D approach is to use ray casting to simulate sensor lines of sight. The challenge is to compute information detailed enough to allow at least near real-time analysis while producing high-quality, high-resolution results.

Mark Livingston and Evan Herbst described an interactive system that places sensors in an open environment and performs basic line-of-sight analysis.[1] It determines strings of sensors, each of which can "see" the next sensor in the string. It also determines how many sensors can see into a given area of the display, taking into account terrain (but not nonterrain features such as buildings). The system produces images revealing the number of sensors that can see an area, but it doesn't identify the sensors.

Eric Becker and his colleagues devised a method to automatically position a collection of sensors in an environment so as to cover all predetermined critical areas.[2] It doesn't address sensor movement, and it doesn't 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 can't be seen."

Tyler Garaas presented an interactive tool for simulating the positioning of video cameras in a 3D environment.[3] It determines and visualizes the environment's overall coverage by the collection of cameras. This tool is unique 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. Instead of painting the actual objects in the scene according to whether they're visible, the tool colors the open spaces according to this visibility analysis's results. This method differs considerably from the one that I developed (see the main article). However, my first algorithm generates a display that's somewhat similar in appearance and information content.

A central goal of my research is to display a pattern that allows preattentive determination of the number of sensors, n, that can see the area, and whose colors identify specific sensors. (Aspects of an image that are noticed within the first couple hundred milliseconds are generally considered to be preattentive.) Each atomic piece of the pattern should correspond to one sensor, and its assigned attribute block cell should share an edge with a cell of each other sensor that can also see that area. Muhammad Malik and his colleagues argued for the value of edge sharing in similar visualization schemes.[4] They developed a method based on a uniform tiling of the plane with hexagons that they use for all values of n. My method instead uses rectangular attribute blocks for $n = 1, 2, 3$, and 5. To handle $n = 4$, I use triangular attribute blocks.

References

1. M.A. Livingston and E.V. Herbst, "Interactive Operations for Visualization of Ad-hoc Sensor System Domains," *Proc. 2005 IEEE Int'l Conf. Mobile Adhoc and Sensor Systems* (MASS 05), IEEE, 2005, pp. 341–345.
2. E. Becker, G. Guerra-Filho, and F. Makedon, "Automatic Sensor Placement in a 3D Volume," *Proc. 2nd Int'l Conf. Pervasive Technologies Related to Assistive Environments* (PETRA 09), ACM, 2009, article 36; doi:10.1145/1579114.1579150.
3. T.W. Garaas, *Sensor Placement Tool for Rapid Development of Video Sensor Layouts*, tech. report TR2011-020, Mitsubishi Electric Research Laboratories, Apr. 2011; www.merl.com/reports/docs/TR2011-020.pdf.
4. M.M. Malik, C. Heinzl, and M.E. Gröller, "Comparative Visualization for Parameter Studies of Dataset Series," *IEEE Trans. Visualization and Computer Graphics*, vol. 16, no. 5, 2010, pp. 829–840.

Painting Surfaces

My second algorithm has two stages. The first filters vertices according to the sensor's field of view. It passes the vertices found in that field to the second stage, which traces a ray from the sensor to each vertex.

The two stages let me maximize the use of bounding-volume techniques while minimizing the number of thread processors that sit idle because of early-out tests. At the end of the two stages, the algorithm associates a visibility bit mask with each vertex. The bit for sensor i is set if and only if the vertex is visible to i (that is, it's in the field of view, and no surfaces are found between the sensor and the vertex).

The algorithm interpolates the per-vertex visibility flags across the triangles of which they're a part. A GLSL (OpenGL Shading Language) fragment shader receives these interpolated flags as floating-point values between 0 and 1 for each sensor. So, I can interpret each value as a probability that the pixel is visible to the corresponding sensor. The visualization's goal is then to convey the exact set of sensors that can see each portion of a building. To do this, the algorithm retains the neutral color of Figure 1 for portions of the environment invisible to all sensors. It colors the rest according to the set of sensors to which they're visible.

Attribute blocks. Because I can interactively turn individual sensors off and on, I could cycle through (automatically or under operator control) the various sensors, coloring portions of buildings visible to each sensor in turn. Although my system allows this, my standard approach is to use *attribute blocks*.[4] Attribute blocks can be used to visualize the values of several attributes continuously defined across some domain. I define a $k_r \times k_c$ rectangular pattern in which each cell displays an attribute's value. Operators can independently adjust the $b_r \times b_c$ size of each cell in the $k_r \times k_c$ pattern, in either pixel space or model space.

In a previous article, I described applications in which users had interactive controls for adjusting $k_r \times k_c$ and $b_r \times b_c$, and mapping attributes to cells in the attribute block pattern.[4] In the application I describe here, the attributes are the nonzero visibility flags, and I want to always display all of them. So, operators have no control over the $k_r \times k_c$ pattern other than turning specific sensors off and on. I dynamically determine in the fragment shader (that is, pixel by pixel) what the $k_r \times k_c$ pattern should be and assign the visibility flags to cells in the pattern.

My premise when determining the required attribute block pattern is that all visibility flags are equally important and should generally receive equal display space. Moreover—whenever possible—each sensor cell should share an edge with each cell of all the other sensors that can see the area. This arrangement maximizes the ability to conceptualize coverage in an area and is consistent with similar observations by Muhammad Malik and his colleagues.[5]

Assume that n stands for the number of sensors that can see a given point and that A, B, C, D, and E each indicate a sensor. Constructing an attribute block pattern for n = 1, 2, 3, and 5 such that each block shares an edge with all the others is trivial. The n = 1 case is simply a solid color; Table 1 shows the patterns for n = 2, 3, and 5. (Handling $n \times 6$ so that each cell shares an edge with all the others is considerably more difficult, but such cases haven't proved important.)

I use a modified pattern for the n = 3 case to better support my goal of preattentive identification of the number of sensors that can see a surface. Swapping the A in the upper left with the B in the lower right produces a distinctive pattern: two L-shaped notches with a diagonal pattern between them.

Table 1. Attribute block patterns for n numbers of sensors (labeled A, B, C, D, and E) that can see a given point.

n	$k_r \times k_c$*	Pattern
2	2 × 2	A B B A
3	3 × 3	A B C B C A C A B
5	5 × 5	A B C D E C D E A B E A B C D B C D E A D E A B C

* *$k_r \times k_c$ is a rectangular pattern in which each cell displays the value of a single attribute.*

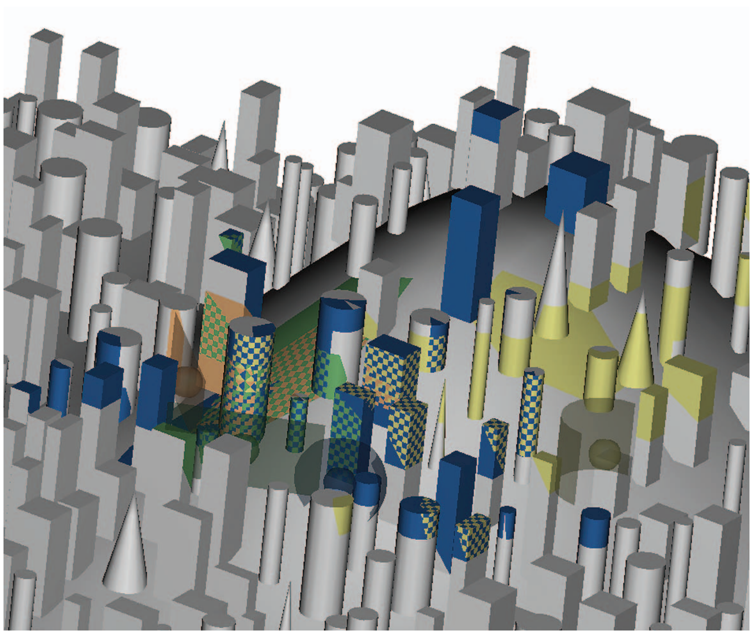
Figure 3. Areas visible to 0, 1, 2, 3, or 4 sensors are immediately obvious. The patterns are, respectively, a solid neutral color, a solid sensor color, L-notched, a simple checkerboard, and triangular.

For n = 4, I switch to a triangular attribute block pattern so that each triangle shares an edge with a triangle for each other sensor.

In Figure 3, areas that are visible to 0, 1, 2, 3, and 4 sensors are immediately obvious. The patterns are, respectively, a solid neutral color, a solid sensor color, L-notched, a simple checkerboard, and triangular. The figure displays several areas visible to exactly two sensors. So, the operator immediately knows the number of sensors covering the area (critical in military applications[3]); further color-based examination reveals which sensors they are.

The n (and hence the texture's structure) varies from region to region along a surface. Fortunately, the fragment shader performs all texture computations on a pixel-by-pixel basis; I never need to create a complete standalone texture pattern. At each fragment (pixel), I know which sensors see it, so I know what abstract pattern to apply. I use modulo arithmetic akin to that described in the original attribute block article[4] to determine where in the abstract pattern I am and hence what pixel color to assign.

Screen space vs. model space. As I mentioned before, operators can size and orient the attribute block patterns in screen space or model space. In screen space, the blocks' edges are always aligned along pixel rows in columns. In model space, they're aligned with axes associated with the local coordinate systems of the objects in the scene.

Screen-space patterns are often the better choice when the scene isn't dynamically rotated, because the patterns tend to be more sharply defined.

Screen-space patterns are often the better choice when the scene isn't dynamically rotated, because the patterns tend to be more sharply defined. This is especially true when the cell sizes ($b_r \times b_c$) 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 aren't locked to the object; instead, they appear to slide around. Moreover, the fact that the patterns retain their same pixel size during zooming can sometimes be confusing.

In those cases, model-space patterns are frequently preferable because they're defined with respect to each object's local coordinate system and hence remain locked in place on the object as it moves during rotation and panning. The individual 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 preferable when the $b_r \times b_c$ cell sizes are larger because they better preserve the sense of 3D surface orientations.

The basic color of each cell in a pattern is determined by the color assigned to the sensor that has been determined to be able to see the object there. I can modulate this color in two ways: by adding a lighting model and by attenuation based on the visibility probability. Recall that I interpolate the individual 0 or 1 per-vertex visibility flag across each triangle's interior. If all three vertices have the same flag, the triangle is uniformly visible or invisible. However, if one vertex has a classification different from that of the other two, then the pixels in the triangle's interior have a flag somewhere between 0 and 1. I can interpret this as the probability that this pixel is visible to the given sensor, and I can use the probability to attenuate the color. So, I have—for both the screen-space and model-space modes—four choices: no attenuation, lighting-model attenuation, visibility probability attenuation, and attenuation by both the lighting model and visibility probability. Figure 4 illustrates the choices, using model-space attribute blocks.

Implementation and Performance

In the simplest scene, the city is defined with 15,411 vertices and 14,714 triangles. The most complex version I tested had 32,139 vertices with 48,096 triangles. Display updates during sensor repositioning and reorienting were instantaneous for both scenes on a 3.2-GHz Intel Core 2 Duo running Linux with a Quadro 600 GPU.

I've achieved my primary goal of delivering real-time performance as sensor placement and orientation dynamically change under operator control. Assigning colors or textures to buildings and the terrain might make the scene more realistic, but operators must be careful about overusing color. Very muted monochrome textures might be effective. Another extension could be to endow the sensors and scene geometry with additional properties. For example, my method could then support sensors that can see through certain types of walls.

Acknowledgments

To select the colors assigned to the different sensors, I used the ColorBrewer website (http://colorbrewer2.org). I thank the NASA World Wind project team—especially Tom Gaskins and Patrick Hogan—for supporting me while prototyping this and other tools for NASA World Wind (http://worldwind.arc.nasa.gov/java).

References

1. T.W. Garaas, *Sensor Placement Tool for Rapid Development of Video Sensor Layouts*, tech. report TR2011-020, Mitsubishi Electric Research Laboratories, Apr. 2011; www.merl.com/reports/docs/TR2011-020.pdf.
2. T.C. Gruber Jr. and L.B. Grim, "Visualization of Foreign Gases in Atmospheric Air," *Proc. 11th Int'l*

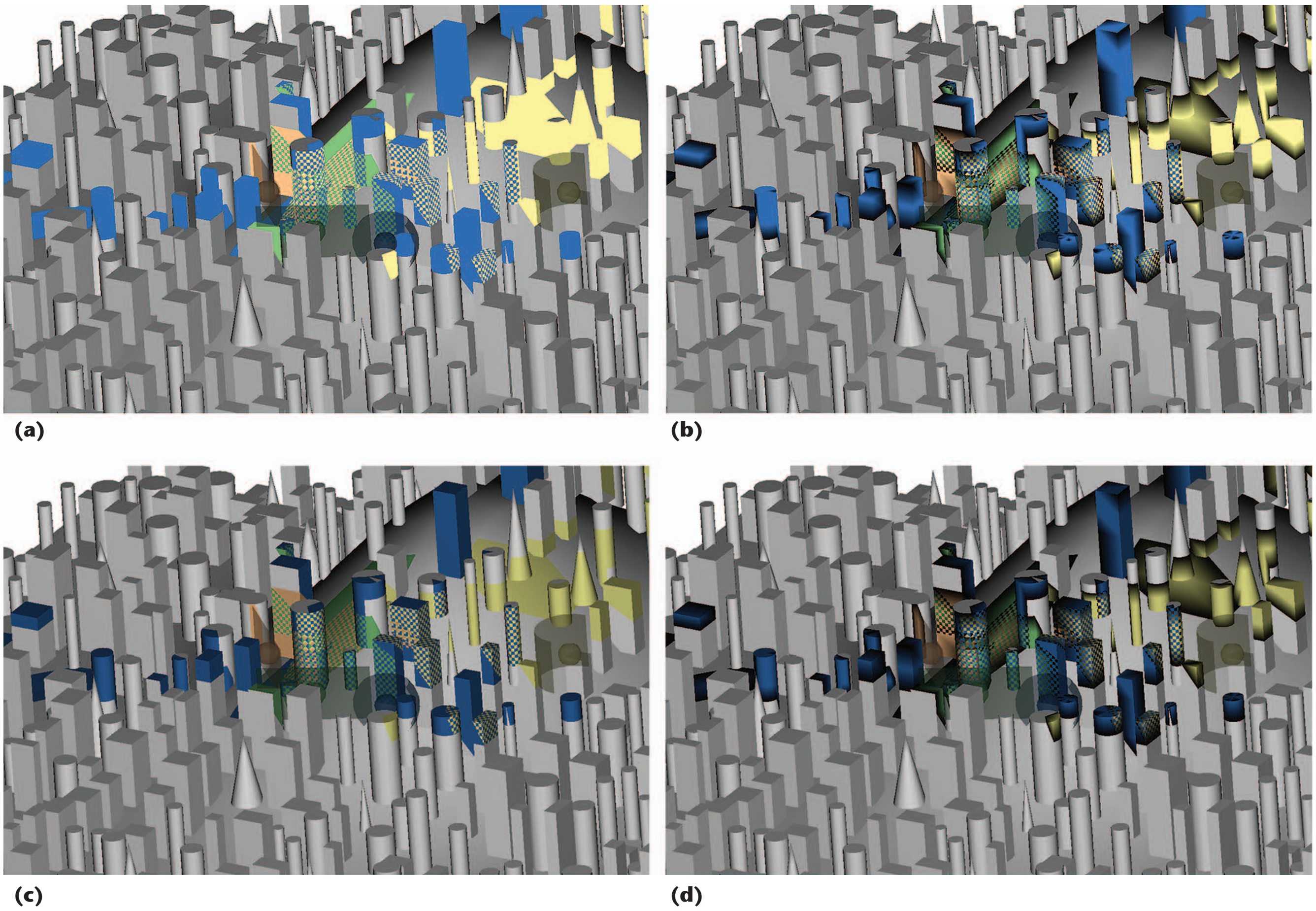
Figure 4. Modulating the color in the model space: (a) lighting off, visibility probability off; (b) lighting off, visibility probability on; (c) lighting on, visibility probability off; (d): lighting on, visibility probability on.

Symp. Flow Visualization, 2004; www.meshoxford.com/Visualization%20of%20Foreign%20Gases%20in%20Atmospheric%20Air.pdf.

3. M.A. Livingston and E.V. Herbst, "Interactive Operations for Visualization of Ad-hoc Sensor System Domains," *Proc. 2005 IEEE Int'l Conf. Mobile Adhoc and Sensor Systems* (MASS 05), IEEE, 2005, pp. 341–345.
4. J.R. Miller, "Attribute Blocks: Visualizing Multiple Continuously Defined Attributes," *IEEE Computer Graphics and Applications*, vol. 27, no. 3, 2007, pp. 57–69.
5. M.M. Malik, C. Heinzl, and M.E. Gröller, "Comparative Visualization for Parameter Studies of Dataset Series," *IEEE Trans. Visualization and Computer Graphics*, vol. 16, no. 5, 2010, pp. 829–840.

James R. Miller *is an associate professor in the Department of Electrical Engineering and Computer Science at the University of Kansas. Contact him at jrmiller@ku.edu.*

Contact department editor Mike Potel at potel@wildcrest.com.