Geographic Grid Registration to Significantly Reduce Range and LOS Calculations

Roger Smith – Modelbenders LLC
gpg @ modelbenders.com

Introduction

Most game objects are interested in two questions, "Who can I see?" and "Who can I shoot?" When this question is asked by a game controlled object, usually referred to as an AI or NPC, it presents a problem of determining which objects are within range. Unfortunately, this can sometimes mean searching through the entire list of objects in the game and subjecting each to a line-of-sight (LOS) or range check. This is hugely inefficient and completely unnecessary.

In virtual worlds, the primary organizing characteristic for interactions between objects is geographic location. Objects tend to interact with those in close proximity – this includes sensor detection, weapon engagement, communication, and the exchange of supplies. Game programmers must manage lists of dynamic (living, breathing, moving) objects so as to retain the geographic relationships between them. The commonly used linked lists and many tree structures do not retain this information. A linked list usually manages all of the objects, but does so in a nearly generic manner so that the order of the list does not contain any useful information. Trees may embed some useful information in the data structure, including geographic location. However, implementations like quadtrees are generally applied to static terrain rather than dynamic objects. This gem describes a simple method for managing object lists using geographic grids as a means of significantly reducing the computational work required for range and line-of-sight decisions. Managing objects geographically can reduce the number of line-of-sight and other related geographic checks on a large battlefield from on the order of 1 million to a few dozen.

Some games enjoy a naturally occurring grid system for managing their objects. When the game level represents the inside of a dungeon, building, or space station, the world is naturally divided into rooms. A "portal engine" will take advantage of this to identify near objects according to the room where they are located. Visible or interactable objects exist only in the local room or an adjacent room with a portal leading to it. Though the game may actually contain several hundred objects, only two or three may be in the room with the object that is looking for targets. This significantly reduces the amount of work required to identify objects with which to interact.

Unfortunately, not all games are so nicely structured. Large open battlefields or spacefields can contain hundreds or even thousands of objects. When there are no walls, mountains, or forests to separate them, all objects effectively exist in the same room. This presents a difficult computational problem for an AI agent that is trying to identify the closest or most desirable targets. Svarovsky described this problem very briefly in the first volume of GPG [Svarovsky00]. In this gem we explore it more deeply, explain why these open spaces are so difficult to deal with, and present a relatively simple solution that has been implemented in a number of simulations. Pritchard also explored tile-based LOS in the second volume of GPG [Pritchard01]. But, he limited his discussion to detectability from the perspective of the human player, leaving the actual detection outcome to the graphic rendering engine and the human eye. This gem includes complete detection decisions for AI agents who rely on range and LOS calculations entirely in the game engine.

Quadtrees and Octrees

Graphics programmers are very familiar with quadtrees. These structures nicely divide a two-dimensional space into four smaller "child spaces". This process can be continued from generation to generation until the entire 2D space is divided into small squares that contain the terrain data (Figure 1). A sensor viewing frustum overlays a specific sub-set of these squares, identifying which grids need to be drawn. Therefore, it is not necessary for computer hardware to render the entire terrain database, when only a fraction of it is viewable by a sensor [Ferraris01].

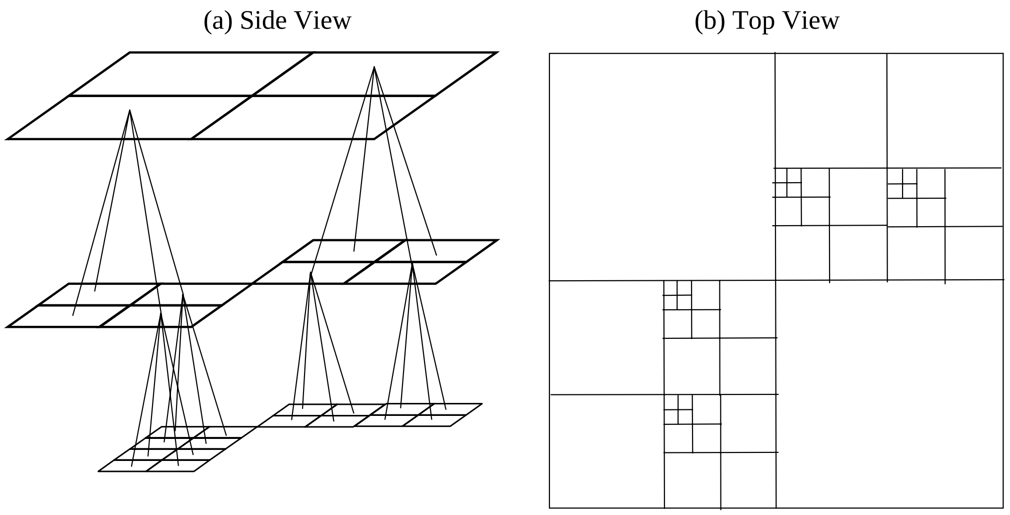


Figure 1 Quadtrees subdivide a two-dimensional space into increasingly smaller areas.

Octrees implement this same idea in three-dimensional space [Kelleghan97]. When the world cannot be simplified into a 2D surface, the subdivision of a 3D world results in eight adjacent spaces, creating data trees with eight children rather than four (Figure 2).

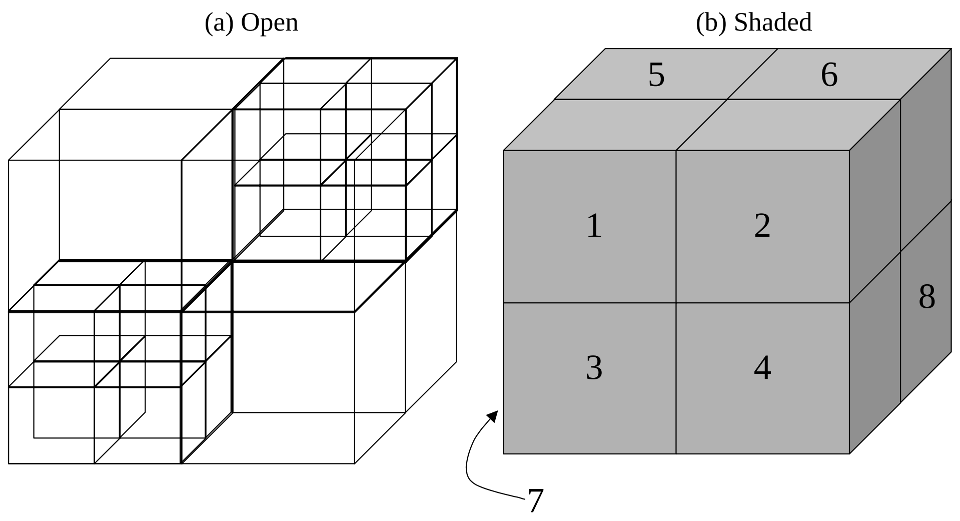


Figure 2 Octrees subdivide a three-dimensional area into smaller volumes.

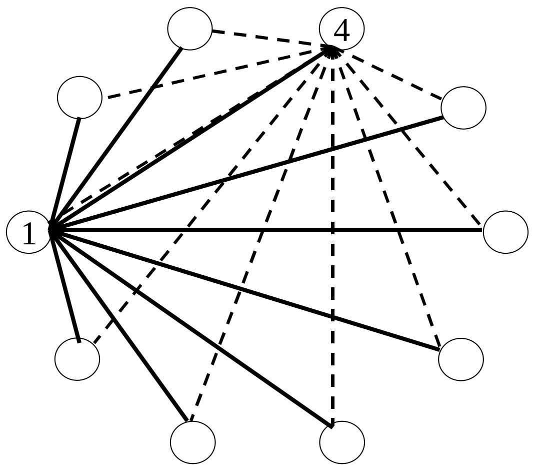
Because terrain, buildings, rivers, and forests usually reside at fixed geographic locations, managing them in a geographic grid has always been a natural approach. This has allowed the use of quad and octrees to rapidly identify the pieces of the terrain that needed to be drawn for a specific scene.

There is a close parallel to this approach for optimizing object-to-object detection for spaceships, trolls, and soldiers that are moving around in the world.

Object Organization

In this gem, object grid registration refers to the need to manage even dynamically moving objects within the same type of grid that is used for terrain data. In some cases, these objects may actually use the very same grid system.

Dynamic objects are constantly changing their locations, as well as other state variables. Individuals and groups of soldiers move from one position to another - advancing and retreating from the enemy. Therefore, from one moment to the next, the list of objects that can be seen or engaged is changing. In a brute force approach to this problem, the game engine will constantly recalculate LOS or range for every pair of objects in the game. As long as the number of objects is relatively small, this approach can be tolerated. For example, if the world contains only 10 objects fighting each other, then the number of LOS calculations is merely 90 - each of the 10 objects calculates LOS to the remaining 9 objects (Figure 3). This is usually referred to as an order n-squared problem – $O(n^2)$. Though this solution is complete, in terms of performance and scalability, it is the worst possible approach. If the number of objects increases to 100, the number of range calculations jumps to 9,900. If there are 1,000 objects in the game, it is nearly one million calculations.



Brute Force = 10 objects * 9 detections = 90 computations

Range Pairing = 9+8+7+6+5+4+3+2+1+0 = 45 computations

Figure 3 Brute force object-to-object detection is an order n-squared problem.

One simple improvement is to implement a pairing scheme in which the range calculated from object 1 to object 4 is used immediately to determine the visibility or sensor detection from 1 to 4 and from 4 back to 1. This means that objects only need to calculate range for the other objects that are later than them in the list of objects. This cuts the number of ranges calculated in half (Figure3).

These calculations may be performed each time-step if there is not some other mechanism at work. Additional filters are often used to mitigate this problem, such as keeping track of the last time that an object changed position in order to avoid unnecessarily recalculating the range between the same two positions again. But, this requires retaining some information about the previous range calculation between specific pairs of objects. This falls short of significantly reducing the complexity of the problem.

Line-of-sight is a geographic question. Arriving at an efficient solution requires a geographic approach to the problem, just as quadtrees have been used for static terrain information.

Grid Registration

Moving objects must be registered into a geographic grid as they change locations. This means calculating a grid location in addition to their more universal position. As an object moves from one grid square to another, just as a chess piece changes grid positions, it must be registered into the arriving grid and unregistered from the departing grid (Figure 4). The figure illustrates two lists, one for Grid(1,5) and another for Grid(1,8). At time 1234 both of these grids contain two objects. However, at time 1235, Object 1 (O1) has moved to a new position. Therefore, the grid registration process must change O1's registration to Grid(3,6). The additional computational overhead associated with managing the grid location is much smaller than the computations required to perform all of the range computations described previously. One part of this savings is due to the simpler problem of determining grid location versus the more complex calculation of range or LOS. However, a far greater portion of the savings is due to the extremely reduced number of operations that have to be performed.

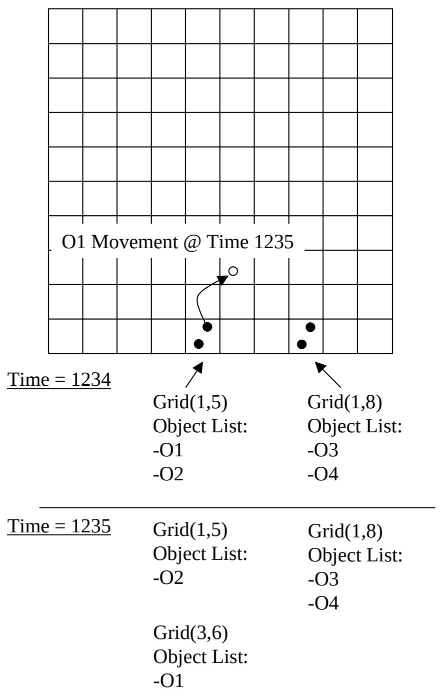


Figure 4 Grid Registration is performed each time an object's new position is in a new grid square.

Object grid registration is an order 'n' problem – O(n). LOS is an order n-squared problem – O(n^2). Therefore, a game that has 100 objects in it will require at most 100 grid registrations during each time step. Without the grid (and disregarding other detection filters), these 100 objects would require nearly 10,000 range calculations each time step. So grid registration exchanges nearly 10,000 range calculations for 100 simpler registration operations.

Weapon or Sensor Footprint Overlay

Grid registration does not entirely eliminate range or LOS calculations. It simply limits it to a much smaller number of objects. As shown in Figure 5, the sensor of the searching object overlays a small number of the grids. These grids can be identified in a number of ways, most of which are described in other articles on quadtrees and terrain database management [Frisken03]. Once this list of grid squares is identified, only objects that are registered within these grids need to be considered for detection or engagement. However, simply because an object is registered into one of these grids, there is no guarantee that the object also falls into the sensor footprint. In Figure 5 it is clear that, in many cases, the sensor does not cover all of a grid square. Objects in the partially covered grids may not actually be within range of the sensor and may need to be excluded from the list of potentially detectable or engageable objects. Therefore, a separate range or LOS calculation must still be done for the much smaller group of objects found in these grids.

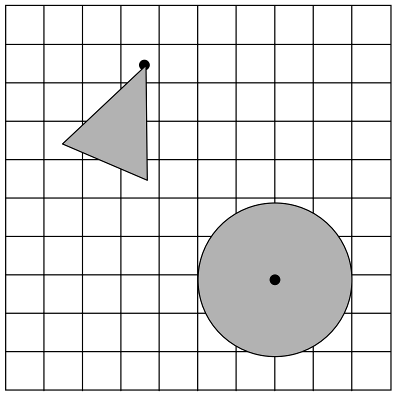


Figure 5 Sensor footprints overlay the grid and identify which squares are potentially in the detection field of the sensor.

The programmer can implement the sensor footprint overlay algorithm such that it clearly differentiates the grid squares that are entirely within the sensor footprint from those that are only partially inside. For some games this can entirely eliminate the need for a range or LOS calculation for objects in grid squares that are entirely contained in the sensor footprint [Pritchard01]. However, for games where 3D terrain is an essential part of the visibility decision of AI's or NPC's, it will remain necessary to perform a separate LOS calculation for each object found in the contained grid squares. Although the grid system can identify which objects are within range of the sensor, it cannot determine whether there is a clear LOS vector from the sensor, through the terrain and obstacles, to the selected object.

Exaggerating the Footprint

Our goal is to minimize the number of objects that must be considered to be viewable by the AI agent. However, in doing this, we must also be careful that we do not inappropriately eliminate moving objects that are on the edge of the sensor footprint. Most games operate on a time step in which an object moves from point A to point B along a constant vector during a single time step. It is possible for the movement of an object to carry it across the edge of the sensor footprint such that it does not exist within the sensor footprint during either of the two discrete times at which calculations are performed (Figure 6a).

To capture these cases, the size of the footprint is usually exaggerated slightly. This allows the sensor to grab objects in grids that are immediately adjacent to the footprint and that may potentially move across its edge. The size of the footprint exaggeration is determined heuristically based on the size of the grid squares, the maximum speed of game objects, and the duration of the time step. The exaggeration must incorporate all grid squares from which an object can reach the edge of the footprint in a single time step. In games where movement precedes detection, the exaggeration will find the objects at their ending position and will use reverse dead-reckoning to calculate the path followed to reach that ending position and determine whether that path crossed the sensor footprint.

There are often two or three instances of footprint exaggeration. The first is an exaggeration of the size of the footprint as was just described. The second is the fact that the grid system is composed of squares and the footprint may be shaped like a wedge or a circle. The process of “squaring the circle” brings in area that is not really within the footprint. Finally, the “squared circle” will not necessarily fall exactly along grid boundaries, so it is further exaggerated to include all of the area of the “squared-in” grid squares (Figure 6b).

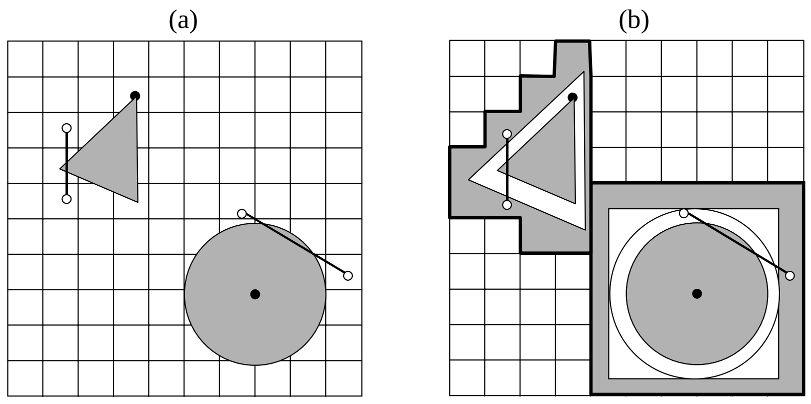


Figure 6 Footprint exaggeration captures objects whose movement vector crosses the edge of the sensor footprint in a single time step.

Conclusion

The method described in this gem is not complex. Once described it seems to be an obvious approach to creating a more scalable detection algorithm. However, the number of projects that continue to rediscover this approach to the problem, and other variations of it, is scandalously high. It is captured in here in an attempt to save future programmers from repeating the work of the past.

The goal is to manage dynamic object locations in a manner that is similar to that used for static terrain objects. These objects enter and leave the game, change positions frequently, and sometimes straddle two or more grid squares. All of these actions make the management of these more challenging, but the payoff in terms of saved processing time far exceeds the cost of implementing geographic grids to reduce LOS and range calculations that are one of the most repetitive calculations in a game.

References

[Ferraris01] Ferraris, Jonathan, “Quadtrees”, GameDev.net. January, 2001. http://www.gamedev.net/reference/articles/article1303.asp

[Frisken03] Frisken, S. & Perry, R., “Simple and Efficient Traversal Methods for Quadtrees and Octrees”, *Journal of Graphics Tools*, Vol. 7, Issue 3, May 2003. http://www.merl.com/people/perry/treeTraversalJGTWithCode.pdf

[Kelleghan97] Kelleghan, M., “Octree Partitioning Techniques”, *Game Developer*, July, 1997.

[Pritchard01] Pritchard, M., “A High-Performance Tile-based Line-of-Sight and Search System”, *Game Programming Gems 2*, Charles River Media, 2001.

[Svarovsky00] Svarovsky, Jan, “Multi-Resolution Maps for Interaction Detection”, *Game Programming Gems*, Charles River Media, 2000.

Biography

Roger Smith is a Chief Engineer with Sparta Inc. and the President of Modelbenders LLC. He develops simulation systems for the military, creates commercial courses on simulation and virtual world technologies, and writes a constant stream of technical papers for books, encyclopedias, journals, and conferences. He has presented several full-day tutorials at the Game Developers Conference and teaches simulation and gaming courses at a number of universities.