**Line of Sight Algorithm for Tile Based Games**   
**by** [**Andy McFadden**](mailto:fadden@amdahl.com)

Courtesy of [Amit Patel](http://www-cs-students.stanford.edu/%7Eamitp)

Taken from a posting to rec.games.programmer on 9 May 1994

In article <67056@sdcc12.ucsd.edu> awiggins@sdcc5.ucsd.edu (Adam Wiggins) writes:

>>myself, I've noticed a problem. If the viewer is close to a wall, it will

>>not be able to see down the wall's length (when it should be able to).

>>So we want to check to see if 'b' is visible from 'a'. The path the

>>bresenham algorithm would take is:

>>......

>>...34.

>>.12...

>>......

>>

>>The problem is then that when it reaches '3', it detects a wall and

>>returns an invisible result.

Here's a couple of items that may be of help. The first is jnh's original table-driven line-of-sight scheme, which works for most things but fails for cases like you describe above. The second is a description of some stuff I was playing with a little while back, which works around the problem to give "correct" shading. It can also be used for line-of-sight calculations.

(This is kinda long.)

From: jnh@ecemwl.ncsu.edu (Joseph N. Hall)   
Newsgroups: rec.games.programmer   
Subject: Shading and line-of-sight calculation \_en\_masse\_...   
Date: 25 Sep 89 17:10:09 GMT

Here is a rough presentation of the technique for calculating shading and visibility that I had mentioned earlier.

...

(summary)

**A Fast Algorithm for Calculating Shading and Visibility in a Two-Dimensional Field**

By Joseph Hall   
Applications Programmer, North Carolina State University

This document copyright 1989 by Joseph Hall. It may be reproduced in entirety for distribution for any purpose, so long as no fee whatsoever is charged for its distribution and no attempt is made to restrict its distribution. No other use is allowed without permission from the author. Permission from the author must be obtained if a substantial portion of this document is to be included in another copyrighted work.

As the author of this document, I hereby release the ALGORITHMS described herein into the public domain. This release does not apply to the actual text of this document.

---

Interactive terminal-based "rogue-like" games such as Hack, Moria, Omega, and, of course, the original Rogue, feature a player character traveling through a maze. The maze usually comprises several levels and is usually laid out on a grid of squares or "tiles." Each tile contains one of several distinct features, e.g., a piece of wall, floor, door, etc., and may also contain objects and/or creatures, if it is not solid.

Hack and Rogue handle lighting and visibility quite simply. All corridors and walls are "visible" once they have been seen. Rooms are square and are either "lit" or "dark." A player carrying a lamp can see with a radius of 1 tile if he is in a corridor (which is always dark) or in a dark room. A player cannot see the occupants of a room until he steps into that room. These conditions eliminate the possible complexity of line-of-sight and shading computations, but detract somewhat from the "realism" of the game.

Moria, on the other hand, allows for line-of-sight considerations. A player can see whatever is standing or resting on a tile is it is both lit and can be seen from his current location, i.e., if there are no "solid" tiles, such as walls or closed doors, intervening. Thus a player can see some of the contents of a room as he approaches its entrance, and more as he gets closer. Moria does not, however, allow for lights of radius greater than one tile, and only the player is allowed to carry a light. Again, all rooms are either lit or not lit, and corridors are dark, although certain player actions can permanently light portions of corridors and permanently light or darken portions of rooms.

One can see the desirability of a more complex scheme, where the player is allowed a lamp of variable radius, other creatures can carry lamps, and rooms are lit by lamps with finite radius. Such a scheme is not trivial to implement, at least from the standpoint of the bookkeeping required, but the greatest difficulty is the amount of calculation required, which can easily take long enough on a microcomputer to remove the interactive feel of the game.

Consider:

Whenever the player moves, and thus his viewpoint changes, the visibility of the entire area surrounding him must be recalculated. This area will be either the visible area on the screen or the portion of it within a limited "sight radius" of the player. A sight radius of at least 25 tiles is desirable, and this could entail calculations for pi \* 25 \* 25 tiles, or about 2000 tiles.

Additionally, whenever a light source moves (when carried by the player or by another creature), the lighting status of the area within the effective radius of the light source must be recalculated. Although a radius of 1-5 tiles is probably optimum for players and other creatures, there may be a number of these light sources on screen at the same time, and larger radii also have some application.

Finally, considerable recalculation is required whenever the solidity of a visible tile changes, e.g., when a door opens or closes.

The obvious approach to all of the above situations is to calculate both visibility and lighting status on a tile-by-tile basis using an ordinary "line-of-sight" routine. That is, for each light source on screen, calculate whether it lights a tile within its radius by seeing whether a line of sight exists between it and the tile; similarly, once the lighting status of all tiles on screen is known, calculate whether the player can see them by checking the line of sight from the player to each of the surrounding tiles.

The difficulty here is that the line-of-sight routine must check each of the tiles intervening between the player/light source and destination. This makes the calculations described above roughly O(n^3), which is generally unsuitable.

A previous posting on USENET suggested using "rays" emanating from the player or light source, one ray to each screen border tile or each tile of limiting circumference. The algorithm involves checking the solidity of tiles along each ray, beginning at the player or light source, and marking them visible until a solid object is encountered. While this is fast and efficient, it is incorrect. To wit:

. | . | |

. . | . . | . |

. . . | . . . \* \* \* \* . . .

@ . x . | @ . x \* \* @ . x \* \* @ . . . . @ . .

(1) (2) (3) (4) (5)

Here, @ is the center of a light source, x is a solid object, '\*' represents a shaded tile, '.' is a lit tile, and '|' is a boundary. (1) shows the system without shading. (2) is the correct shading. (3) is the shading generated by the above algorithm. (4) and (5) are the lines of sight to the border that cause the incorrect shading to be generated. The correct shading will be generated only for the border tiles, and there will be some inaccuracies in the remaining shading.

The author has, however, found an efficient technique that relies on tables of pre-calculated, rasterized shading.

Consider this situation:

. . . \*

. . . . . . \* \*

. . . . . . . . \* \* \* \*

. 3 . . . . . . . . \* \* . 3 \* .

. . 2 . . . . . . . . . 2 \* \* . . . . .

@ . . 1 . . @ . . 1 \* \* @ . . . . . @ . . . . .

(6) (7) (8) (9)

'1,' '2,' and '3' represent solid objects. (7), (8) and (9) are the shading generated by the individual objects. The total shading can be generated by overlaying (7), (8) and (9):

\*

\* \*

\* \* \*

. 3 \* \*

. . 2 \* \*

@ . . 1 \* \*

(10)

Thus the problem of calculating shading for an area can be reduced to one of "summing" the shadows that its individual tiles create. This procedure is straightforward and won't be detailed in this short report.

**How to store** the pre-calculated shadows is a matter to consider, however. One might expect a full set of shadows, say, out to a radius of 32, to occupy an inordinate amount of space, or, if tightly compressed, to present problems in retrieval. But this turns out to be not nearly so bad.

Symmetry considerations, first, reduce the number of shadows that must be stored by a factor of 8, since only one "octant" (45-degree slice), as shown above, need be calculated.

The shadows can be stored as a series of "rasters," using the following representation for each shadow:

byte

1 # of rasters in this shadow

2 #1 start

3 #1 end

4 #2 start

5 #2 end

...

(7), (8) and (9) can be translated as follows:

(7) 1 4-5

(8) 3 4-5 4-5 5-5

(9) 4 4-4 3-5 4-5 5-5

The full set of radius-32 shadows can, in fact, be stored in a readily-accessible table of LESS THAN 9000 BYTES.

...

I have written a prototype that uses this shading technique. Missing certain optimizations in its current version, it still calculates a 32 x 32 area in a relatively-constant 50 milliseconds on an 8MHz 68000. The most efficient conventional LOS-based version that I have been able to write takes about 800 milliseconds. (!)

I am working on a cleaner version of the prototype and table generator and will present them and a detailed report later (a couple of weeks?) in rec.games.programmer.

v v sssss|| joseph hall || 4116 Brewster Drive

v v s s || jnh@ecemwl.ncsu.edu (Internet) || Raleigh, NC 27606

v sss || SP Software/CAD Tool Developer, Mac Hacker and Keyboardist

-----------|| Disclaimer: NCSU may not share my views, but is welcome to.

**Improvements to a Fast Algorithm for Calculating Shading and Visibility in a Two-Dimensional Field**

By Andy McFadden   
Based on ideas by Joseph N. Hall (jnh@ecemwl.ncsu.edu)

(This assumes you have read and understood the original posting by jnh.)

**Introduction**

The line-of-sight (LOS) algorithm used in Moria (written by jnh) does a fast integer computation from the center of the player to the center of the object in question. This works great for something like Moria, where all you're interested in is whether or not you can see a particular object. Small irregularities either won't be noticed or will be accepted as part of the way the game works.

However, I wanted to use his fast visibility algorithm to compute light patterns from multiple sources and visibility updated on every turn. In Moria, you don't stop seeing nearby walls when you move away from them; the LOS rules are only for monsters. What I wanted to do was more like Ultima, where you'd see only what's around you.

The problem is best explained with a picture:

......... .XXXXXXXX

......... ..XXXXXXX

......... --> ...XXXXXX

....###.. ....##XXX

....O.... ....O....

Here, the "O" is the observer's position, the "."s are visible squares, the #"s are obstacles, and the "X"s are areas in shadow.

In this example, the rightmost obstacle is invisible, because a line from the middle of the observer to a position in the middle of the obstacle passes through the obstacle above and to the right of the observer.

For a monster, that's fine; maybe he was hiding around a corner out of sight. For a wall, it makes no sense at all. We can see the FACE of the wall from where we are, so we should be able to see the wall itself. So, what we need is a different table that uses middle-to-face computations instead.

**Issues**

It would appear from this that all we need to do is rewrite the LOS algorithm. Life, unfortunately, is not so simple. For example, the original middle-to-middle algorithm would draw this:

.XXXXXXXX

..XXXXXXX

...####X.

.........

....O....

However, an algorithm that allows visibility if \*any\* of the faces is visible creates a map that looks like this:

.XX.X.XXX

..X.X.XXX

...####..

.........

....O....

Some areas that should be obscured aren't. The problem is that one block obstructs one side of the area, while an adjacent block obstructs the \*other\* side of the area. Both sides of the area are obscured, but by \*different\* blocks. So to implement middle-to-face LOS we need to obscure areas for which both faces are obscured, taking into account that different obstacles block different lines of sight.

The obvious implementation uses different sets of light maps for different faces (i.e. one map that shows which of the left faces is obscured, one that shows which of the bottom faces, etc), but we can do better than that. More later.

Another issue is corners. If we want to have this:

..XX.

..#.X

..O#X

instead of this:

..XXX

..#XX

..O#X

we either need to go from the middle to the corner or repeat the earlier middle-to-middle LOS computation. Doing middle-to-corner isn't so great, since there are situations where there might be only a small part of the corner visible, which we don't want to allow. Allowing middle-to-middle enables the player to see through the small crack in the wall without exposing the entire room. If this is undesirable, just put another block into the corner.

This also allows the player to see the block in the corner, but only when he's on an exact diagonal... so the diagonal blocks will appear and disappear. Ultima IV handled this in a nice way, but sometimes you're allowed to see things that you clearly should not.

One way to resolve this is to treat blocks as occupying less of the square than they actually do... this allows more lenient visibility, but raises the possibility of somebody peeking through "gaps" in a solid wall. I suspect the best way to deal with this problem is to do a second "clean up" pass through the map that identifies corners and un-shadows the corner blocks. I don't know if Ultima IV did this, but it only had a radius 6 display (11x11), so it would not have been very expensive.

If doing middle-to-corner is desirable, it can be added to the shadow map with a minimum of effort. (In fact, the policy could be chosen at runtime.)

**Implementation**

We need to trace the shadows for each object three times, once for the left edge, bottom edge, and middle. Since we're only computing an octant (or, for speed, a quarter; if we want the whole thing we can compute an octant and use reflection to generate the silly table), we don't need the top or right edges.

By using the low three bits of the byte as flags, we can still OR the shadows cast by all of the obstacles together to get the final shadow in one pass. However, only those squares for which all three bits are set are considered to be invisible.

The previous algorithm stored shadows like this:

byte

1 # of rasters in this shadow

2 #1 start

3 #1 end

4 #2 start

5 #2 end

...

The new rasters will be stored like this:

byte

1 # of raster segments in this shadow

2 #1 value (low three bits) + "move over" flag (high bit)

3 #1 start

4 #1 end

5 #2 value + "move over"

6 #2 start

7 #2 end

We need to have more than one raster per shadow, because as we move farther away from a given obstruction it may stop blocking one face of the squares. However, the rasters should still be fairly "smooth", moving from "only bottom obscured" to "bottom/middle obscured" and eventually to "only left obscured" as the rasters move from left to right.

I call these are "raster segments" because they don't represent all of the values on one line. The "move over" flag is used to tell the shadow calculation routine that this segment is the start of a new row (previously this was just assumed). The "value" byte may therefore hold any logically ORed combination of:

0x01 middle obscured

0x02 bottom obscured

0x04 left obscured

0x80 start of new raster

To compute the shadows, the low three bits of the value should be bitwise ORed onto a grid. After shadows for all obstacles have been ORed together, squares whose value equals 0x07 are in shadow, all others are visible. (Don't forget to AND #$7f to get rid of the "start of new raster" mark!)

Obviously, this requires completely separate shadow creation and use routines from the older middle-middle method. By encapsulating the whole thing within a C++ object, it's possible to provide both kinds of tables without the program using them being aware of the difference.

*This should be done as attributes/methods of a "map" object. If a separate object is used, it should be part of the "map" object as a whole, and needs to be able to interact with the "raw" map and the "output" map... doesn't really fit.*

*should have more here...*

fadden@amdahl.com (Andy McFadden) [These are my opinions, not Amdahl policies]

You get what you pay for, if you know what you are doing.

PGP Otherwise, you get what you deserve. RIPEM