LineOfSight

COLLABORATORS			
	TITLE : LineOfSight		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 24, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

LineOfSight

1.1 main

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

By Joseph Hall
Applications Programmer, North Carolina State University
Email: jnh@ecemwl.ncsu.edu

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.