# ImprovedLineOfSight

| COLLABORATORS | | | |
|---|---|---|---|
| | *TITLE* :<br><br>ImprovedLineOfSight | | |
| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
| WRITTEN BY | | November 28, 2024 | |

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| | | | |

# Contents

# Chapter 1

# ImprovedLineOfSight

## 1.1 main

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

By Andy McFadden
Email: fadden@amdahl.com

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
..XXXXXX
...####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... ]