Perspective Based Lingo Mazes: The Director Dungeon Crawl

Andrew M Phelps

Information Technology Dept.
Rochester Institute of Technology
Rochester, NY, 14623
http://andysgi.rit.edu/

Abstract

This article outlines the process of creating a typical 'maze game' in which characters move around via the arrow keys, and must navigate passages defined by the walls of the maze. Further, the view that the character is presented with is one based in first person perspective, meaning that they see the maze as if they were in it. This type of system was popularized by adventure games in the early 1980's like Bard's Tale™, Wizardry™, and Might& Magic™, and has remained a staple of the genre to this day.

This paper is split into two major sections, the first describing the basics of the program, including the relevant data structures and engine setup, and the second linking the underlying engine to the display system producing the illusion of perspective. There is also a future work section describing a few uses for an engine such as this, and links and references to other maze-game implementations.

1 THE MAZE GAME ENGINE CORE

1.1 TUTORIAL FILE SETUP

The idea of a maze game has been around for a while, and was popularized in the 1980's by one of my favorite games of all time, A Bard's Tale by Interplay Productions [198x]. Bard's Tale also included character advancement and combat in a pseudo-Dungeons & Dragons type storyline, and it should be noted that this engine implements absolutely none of that, it is only concerned with movement and rendering. This engine is provided as a stepping-stone for games of that ilk, or for many other genres, it is concerned primarily with the 'how' of the maze rather than the 'why'.

To begin, download the demo files and unzip them into a directory of your choice. Start the movie. Upon starting the movie you will be placed into a most incredibly boring maze, one that has no walls. Given that this tends to be rather unimpressive, click the "load" button. Select the "maze01.txt" file that was included in the download package. The world we will reload, and you will be faced with the screen in figure 1.

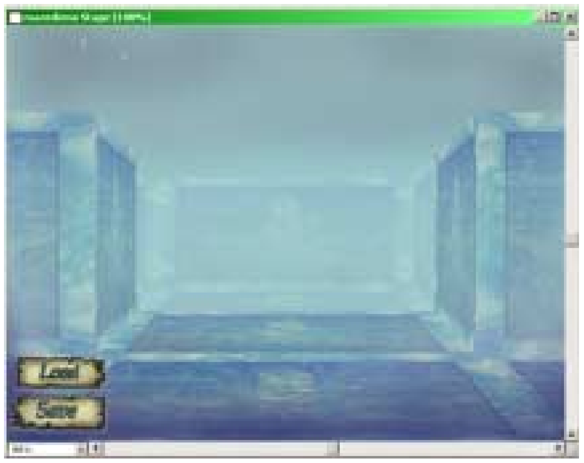

Figure 1: Screenshot of the Maze Engine version 1.0 in action.

This is the demonstration maze that the engine ships with, and you can explore it using the map in figure 2. The 'up' arrow moves forward, and the 'right' and 'left' arrows turn you right or left. If you get lost you can always reload and you will be placed back at the 'start' position, on the "X" in figure 2, facing north ('up' on the map).

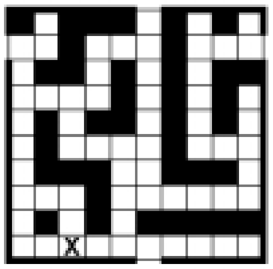

Figure 2: Map of the sample maze. Start position is (3,1).

1.2 DATA STRUCTURES & GLOBAL VARIABLES

At the root of all maze games is a rather common data structure, a two-dimensional array. This is the common structure used in more traditional 2D or 'top down' games, and have been implemented in Lingo by Rozenfeld [2000] as well as countless other authors. This engine creates a two dimensional array, but goes one step further in that it stores in each position of the list a property list that describes the position or 'square' of the current location. The process of list creation and manipulation has been well documented by other authors like Small [1996] and thus this paper offers little explanation on these basic features.

The original array is created by calling the `Make_Maze` handler in the "Maze Tools" script. This handler in turn calls the `Create_Maze` handler, which creates the array.

You can make a properly formatted maze square using the `Make_Maze_Square` handler, you can add it to the maze list with `Add_Square_To_Maze`, you can get the property list from a specific location within the maze array by using `Get_Square_From_Maze,` and you can get the associated value for any `key` in the property list calling the `Get_Value_From_Square` handler.

Finally, the entire creation of the array is governed by the global variables declared in StartMovie. Table 1 describes their use in the overall construction of the engine.

Table 1: Global Variables inside Maze Engine

VARIABLE	DESCRIPTION
max_row	Maximum number of rows allowable in the maze.
max_column	Maximum number of columns allowable in the maze.
current_pos_x	Current position of the character, along the X-axis.
current_pos_y	Current position of the character along the Y-axis.
MAZE	A pointer to the array describing the maze.
facing_dir	String containing the direction the character is currently facing ('north', 'south', 'east', or 'west').
my_right_dir, my_left_dir, my_back_dir	String variables containing the directions to the characters right left and back. These can be calculated by knowing the facing direction, but I store them separately in the case than a future version of the engine expands on the number of possible directions.

1.3 MULTIPLE LEVES AND FILE ABSTRACTION

After all of this setup and instantiation the movie calls `Populate_Maze` and `Fake_Maze`, which are handlers designed to load the default world that you get when you start the movie. This presents an interesting problem and it is one that is common to this type of application, how to store the maze or series of mazes that make up a game. One solution is the `Fake_Maze` handler, which is essentially a script that inserts a hard-coded level into the maze. This is, of course, less than ideal, as it requires the programmer to create a script for each maze or level. Additionally, it requires that the author of this script know something about how the maze is structured, and it is rare that the level designer and / or content writer is also the programmer.

This engine also makes use of a second solution to this issue, storing maze descriptions in separate text files. It does this by reading and writing the array to text files using the fileIO Xtra that ships with Director. This is done by calling `save_maze` and `load_maze`, which are bound to the save and load buttons within the interface. These in turn format and save the text, or read and load the array, respectively. How each handler calls the next can be seen in the trace diagram in figure 3. This method of 'wrapping' the fileIO xtra, originally developed by Kurtz [1999], makes it easy to reuse this block of code over and over, as we have done here at R.I.T. on numerous projects.

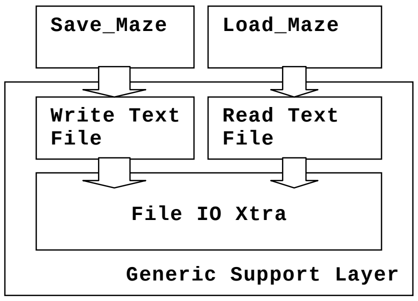

Figure 3: Calling hierarchy for load / save functions.

1.4 MOVEMENT AND COLLISION DETECTION

The final responsibility of the engine core is to account for the movement of the character and the inability of the character to move through walls (at least in most cases, the classic games of the genre generally incorporate hidden walls and teleports in addition to standard collision detection). Moving through the maze is fairly straightforward, and is implemented in the `Move_Me` handler . This handler begins by deciding whether the character is moving forwards or back, and then querying the array for the description of the square the character is currently occupying. If in the property list for the square the value for the characters current facing direction is a blank, then the character is allowed to proceed forward. If this value is returned as a wall, then the character is not allowed to move. This is the reason for describing each square in a property list, by doing so you can determine whether there are any obstacles in the way to block movement.

Assuming the character is allowed to move forward, then the `Move_Me` handler calls `Find_Direction` that modifies the global `current_pos_x` and / or `current_pos_y` variables to reflect a move on either the X or Y-axis respectively. Additionally, this handler contains the logic to make the maze 'wrap around' by checking to see if the X or Y of the character has exceeded the maximum number of rows or columns defined at engine startup.

2 THE ILLUSION OF PERSPECTIVE

2.1 BASIC PERSPECTIVE AND LINGO QUAD IMPLEMENTATION

The real distinction of this system from the more traditional maze systems that have been implemented in Lingo is not so much the code in how it works, but the way in which it is drawn to the stage. This first person perspective view has a long history, and was probably one of the reasons for the success of the games that originally implemented it, like Bard's Tale [198x] (see figure 4).

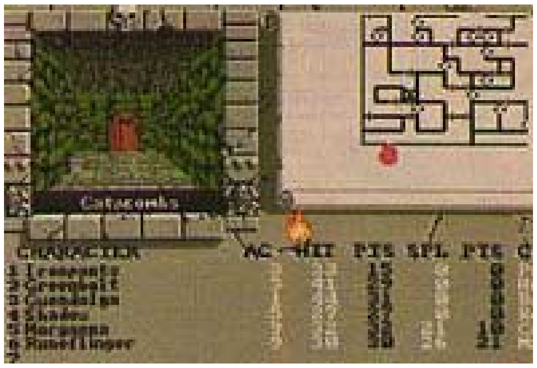

Figure 4: The Bard's Tale III, by Interplay Productions.

This view could be achieved by a traditional 3D engine, projecting the planes against a center of projection and using a lot of neat math tricks. Instead, this engine uses none of this math, it works instead in the simplest possible manner, manipulating only the visibility property of a number of sprites to produce the illusion.

The first step in creating this illusion is 'quadding' the sprites on the stage. What this really means is calling the quad function over and over and positioning the corners of a series of sprites to produce the illusion of depth. This is done in the Position_Walls handler, and while this script hard codes the values for the positions of the walls relative to a 640x480 window, there is nothing preventing a more advanced solution that uses the scale of the window to determine the coordinate points. In any event, the sprites are positioned using the quad function such that they appear in perspective. If there were only 3 sprites possible (ie the character could only see the square he was currently on) and they were not textured with bitmaps, it would look like the original test application in figure 5.

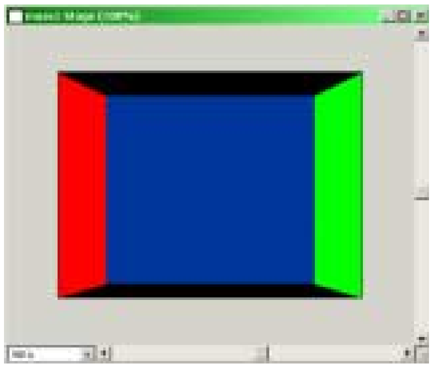

Figure 5: Original Maze Test (left wall in red, forward all in blue, right wall in red showing the placement and quad locations of individual sprites).

All the program is responsible for is turning these sprites on and off, which it does through the visibility property. In essence, all of the walls are there all of the time, its just that they are sometimes 'clear'. The program looks at the square the character is currently occupying, and quesries that square for the `my_left_dir`, `my_right_dir`, and `facing_dir` values, if these come back as 'wall' then it renders the sprite as visible, if the lookup returns a 'blank' then the sprite is invisible.

2.2 ADDITIONAL DEPTH SORTING

This is, however, only part of the illusion. The shot above of the original application is rather unconvincing as view into a world, primarily because it fails to allow a character to see into the distance. In order to allow a character to see 'further', we align more sprites towards the center of the stage. The closer we are to the center, the closer we are to the vanishing points and the smaller the sprites. This engine is set to allow a character to view the square it is on and two additional squares forward. Note that just like the original sprites, these sprites are always present, and their visibility is manipulated in similar fashion in relation to the property list for the square. The engine simulates the position of the character by adding temporary values before lookup to the characters `facing_dir`.

2.3 OTHER VISIBLE SURFACES

The final steps in the illusion are the pieces of walls that are parallel with the front of the viewers square, and the floor. The floor is very simple, as this engine assumes that the floor is always present, thus there is no need to check for a visibility flag for the floor, and the demonstration world ships with no ceiling. The little pieces of the walls are accounted for by adding yet another round of sprites, to account for the widening field of vision from the characters perspective (see figure 6). These values are calculated again by querying the array for 'wall' or blank' but offsetting the characters position not only along the forward direction, but also to the left and right of this value.

2.4 RECURSIVE SOLUTIONS

Needless to say, hard coding each pass of what is visible and not visible based on list lookup is not an ideal solution. There would most likely be a recursive solution based on a `depth_view` parameter that determines how many squares away a character is to see (in outdoor worlds this is commonly used to determine the difference between day and night, or to temporarily 'blind' characters to only the square they are currently on). While not implemented here, it should be trivial based on the implementation above to devise such a system, and probably to position the sprites automatically as well. This version of the engine is presented as a teaching tool and as such does not contain the more abstracted code.

All of this lookup and manipulation occurs in the `Render_Maze` handler, where case after case and lookup after lookup are performed. Because Director seems to have fairly good performance in list lookup, and because the engine is not trying to smoothly pan the transition, there is little to no lag from the users point of view. After this lookup the `Render_Maze` handler calls `Draw_Maze` which actually manipulates the visibility of the sprites.

2.5 MOVING AND SPINNING

Characters move by using the arrow keys and as such the movie binds the `KeyDownScript` constant to the `KeyPressed` handler. This handler checks the incoming ASCII value to ascertain which arrow was pressed, and then calls the `Spin_left`, `Spin_right`, or `Move_Me` handler as appropriate, passing forward for the up arrow key and backward for the down. This is the same Move_Me handler that was described previously, but at the end of the `move_me` handler is a call to the `Render_Maze` function to update the view.

The final touch to the whole program is the ability of the character to change their forward direction, which is performed in the `spin_left` and `spin_right` handlers. These handlers change the forward direction, and then calculate the other three global variables by calling `Set_Init_Dir`. This again calls the rendering loop after the spin, so that the stage is updated.

3 CONCLUSIONS

This engine proves Director a capable tool for creating games that were incredibly difficult to program a few short years ago. While not offering the peak of performance, the Lingo environment can still be used to maximum advantage by playing to its strengths, and this engine does exactly that. This system is a fairly open, generic engine devised to support a first person perspective maze game. With different graphics laid atop the core code, this engine could be used in many different types of systems to provide users with interesting puzzles. Additionally, it can be retooled as the basis for a much more complicated game, and is a suitable teaching tool for students who need to understand the basics of lists, multi-dimensional arrays, and property lists.

4 FUTURE WORK

This engine forms the basis for many different types of games, and while this author is partial to role-playing games, this engine could be used in a variety of scenarios. If this engine were to be adapted into a game, it would be necessary to add support for additional characters within the maze. The functionality for character positions would need to be abstracted out from a single set of global variables and into a character-specific binding. Additionally this engine provides what would probably be a good test bed for AI algorithms, tasking them with moving through the maze with the fewest number of moves, or with a specific goal pattern in mind. Students in my courses have extended the engine to include support for an object system, objects left on squares remain until they are moved, and squares can contain up to a fixed number of objects. They have also created a global repository of maze files, which can be accessed though the Director multi-user server, or through web based solutions such as Directors `GetNetText` functionality to download text files from remote source. If this engine was ever used in a production game it would be necessary to encrypt the maze descriptions somehow since it would be far too easy for a player to simple change a few walls and gain the treasure. My personal goal would be to see this system incorporated into a multi-player game using the Director Multi-user Server, allowing each player to play a member of an adventuring party. This engine represents the foundation of all of these projects; if you take the ideas presented here please contact the author at amp@it.rit.edu.

Acknowledgments

I would like to acknowledge my colleagues at the Rochester Institute of Technology, and in particular Professor Steve Kurtz, who shares my interest and passion for writing graphics programs and extending Lingo into this arena. He was responsible for the original implementation of the code that 'wraps' the fileio extra, and many of my colleagues at R.I.T. have been using similar implementations for a number of years. I would also thank the team at Interplay, for 'Bard's Tale', in addition to dropping my average in 6th grade English by at least a letter grade, that was one of the first games that really hooked me on the genre of electronic entertainment.

References

Rosenweig, Gary. (2000) *Advanced Lingo for Games*. Indianapolis, Indiana: Hayen Books.

Small, Peter. (1996). *The Magic of Lists, Objects and Intelligent Agents*. New York, New York: John Wiley & Sons.

Kurtz, Steve. (1999) *Wrapping the FileIO Xtra for Code Re-use*. Internal documentation at the Rochester Institute of Technology, Department of Information Technology: http://www.it.rit.edu.

Annotated Bibliography

[LL] = Lingo Lists & Property Lists [MG] = Maze Games [LMG] = Lingo Specific Maze Game [P] = Perspective views [GT] = Game Theory

Allis, Lee, et al. (1997) *Inside Director 6 with Lingo*. Indianapolis, Indiana: New Riders Publishing. [LL]

Edgerton, P.A & W.S. Hall. (1999) *Computer Graphics: Mathematical First Steps* Essex, England: Prentice Hall. [P]

Gross, Phil and Jason Roberts. (2000) *Director 8 Demystified: The Official Guide to Director 8 Shockwave Internet Studio*. Berkeley, California: Peachpit Press. [LL][P]

Holder, Wayne and Doug Bell. (1998) *Java Game Programming for Dummies*. Hungry Minds, Inc. [MG][GT]

Rosenweig, Gary. (2000) *Advanced Lingo for Games*. Indianapolis, Indiana: Hayen Books. [MG] [LL] [LMG] [GT]

Rosenweig, Gary. (2000) *Using Director 8 Special Edition*. Indianapolis, Indiana: Que. [LL]

Small, Peter. (1996). *The Magic of Lists, Objects and Intelligent Agents*. New York, New York: John Wiley & Sons. [LL][GT]