Software Design and Development Major Solo Project Proposal

William Hales 2011

Brief Description of the Solution

A top-down view game where the aim is to move a smily-face on a grid of squares until it reaches the end square, without encountering any unfavourable outcomes which restart the level. Several levels in progression with increasing complexity are required to be completed to finish the game.

Each square of every level is made of a single substance. These 'substances' include *wall*, which blocks player movement, *floor*, which the player can walk on, and *spike pits*, which cause an unfavourable event when stepped on (restarts the level). Moving objects such as the player and monsters are stored separately to the squares of the grid, but still move square-by-square.

Interactive squares such as *buttons* (which can open *doors*) also exist, adding an element of puzzle to the game. Button can both help the player finish a level or make it impossible or harder to do, depending on how they are used and placed in a level.

The player controls his/her character using the arrow keys on his/her keyboard. The game does not run in 'real-time', but instead runs one 'step' at a time as the player moves. For example, a player can spend as much time as they want trying to figure out a puzzle, because monsters can only make one step towards the player for every step the player makes. No events occur whilst the user is idle.

Finally monsters, which are similar to the character (but with a characteristic sad-face) try to hunt the player down using specially written AI. Upon reaching the character, they restart the level. Monsters can both be trapped and released by the player, and may be necessary to solving certain puzzles (eg by a monster walking over a button).

Important Notes

This project will use the SDL (Simple Directmedia Layer) libraries to handle graphics and keyboard input. These libraries are free and open source.

No SDL code will be written into the game's source code (with exception to the parts of my code that tell SDL where to draw pictures onto screen, initialise, read input, etc). Instead it will be 'included' at the top of the file with a standard #include C compiler statement, so that there is no fear of confusing my work with that of other people's.

SDL provides (for game development) a more advanced graphical interface API than Visual Basic does. Whilst Visual basic is designed for standard WIMP user-interfaces with buttons and text-boxes, SDL merely allows the coder to 'draw' pictures on-screen at the desired location.

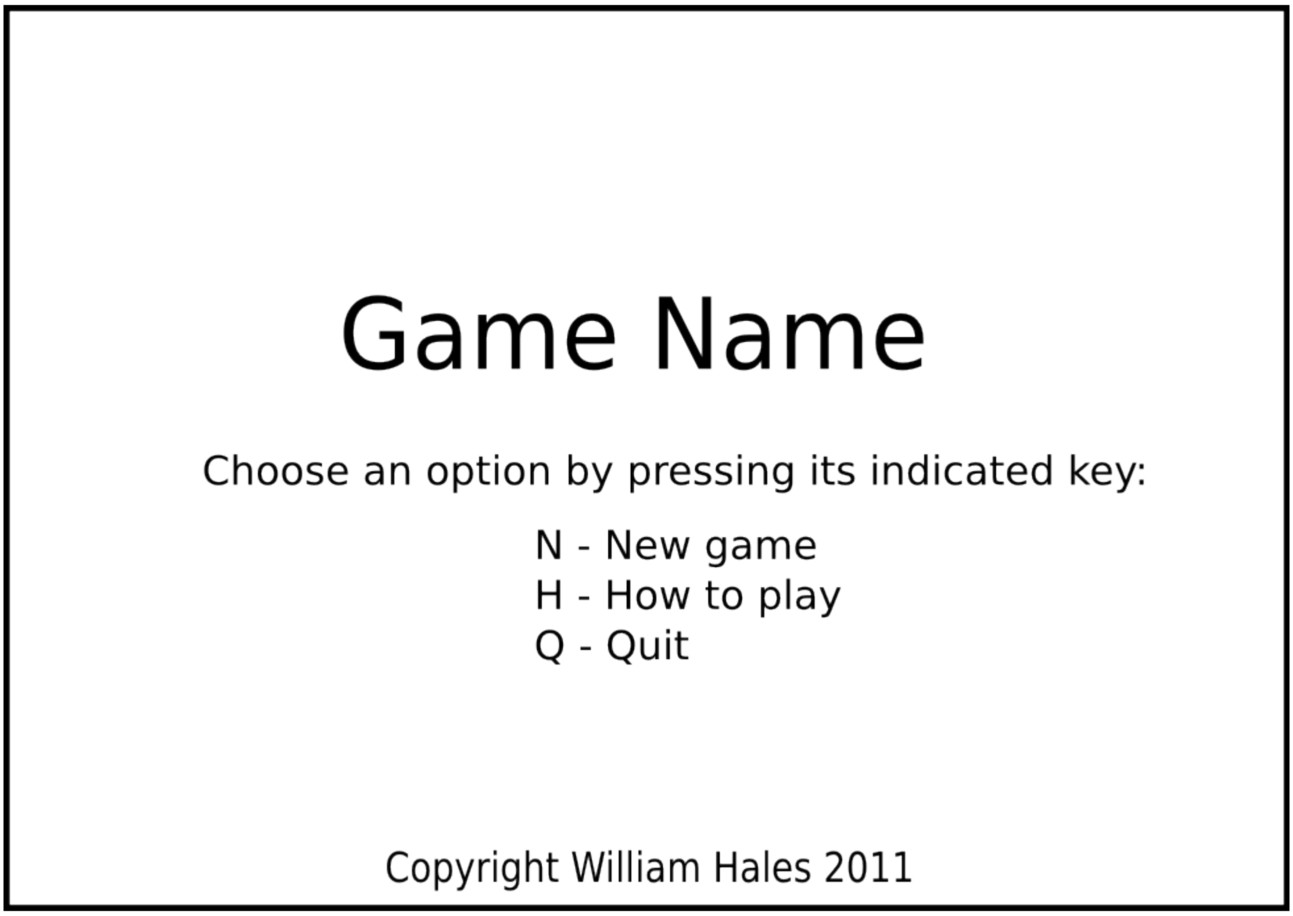
Its advantages include that it is faster, does not suffer visual 'tearing' and it does not restrict the developer to use only the Visual Basic language. These stem from the fact SDL is designed for games.

The SDL libraries also provide features such as CD-ROM management and sound management that I will not use in this project.

Finally, I use the words 'map' and 'level' synonymously throughout this proposal.

Mud Maps

Title menu screen:



Game Name

Choose an option by pressing its indicated key:

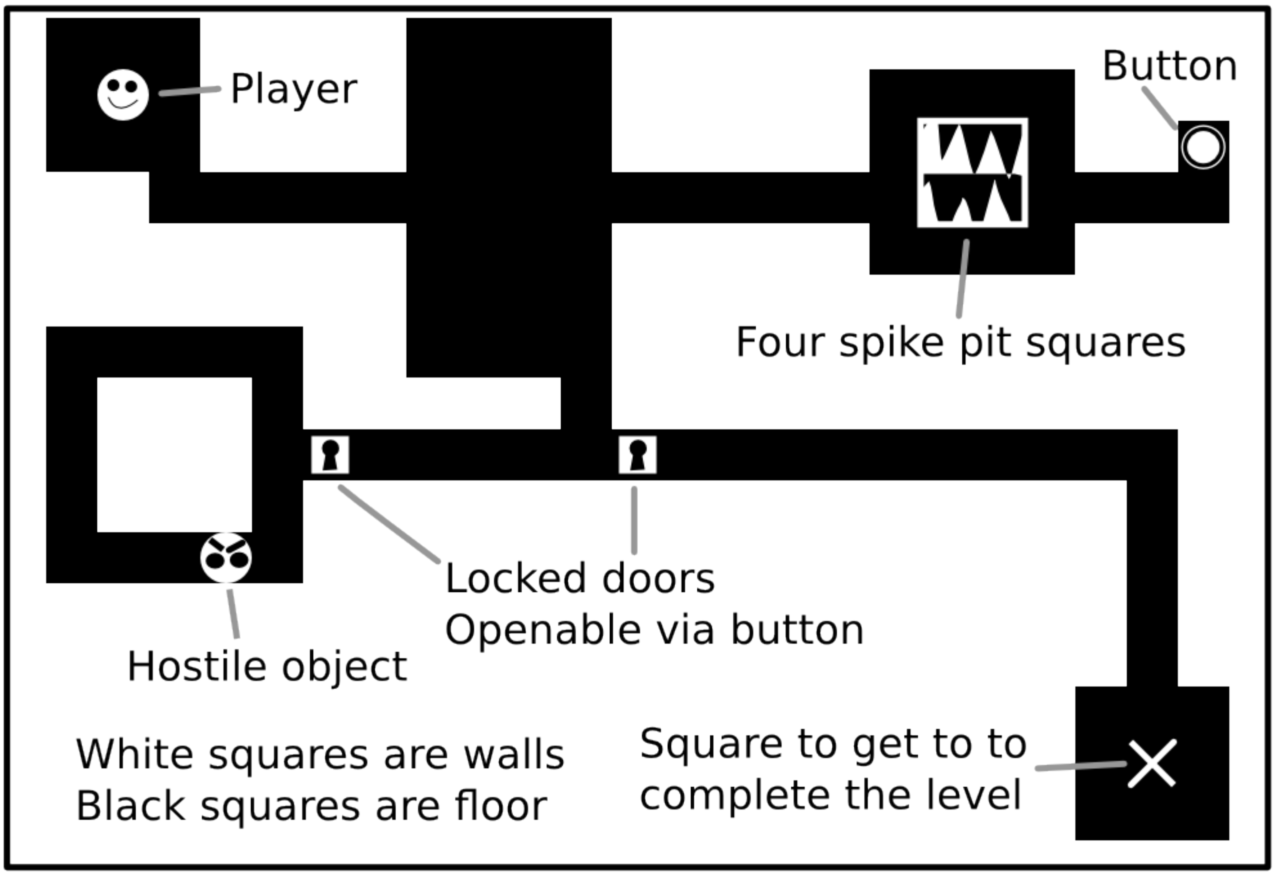
N - New game
H - How to play
Q - Quit

Copyright William Hales 2011

Course of events for the user:

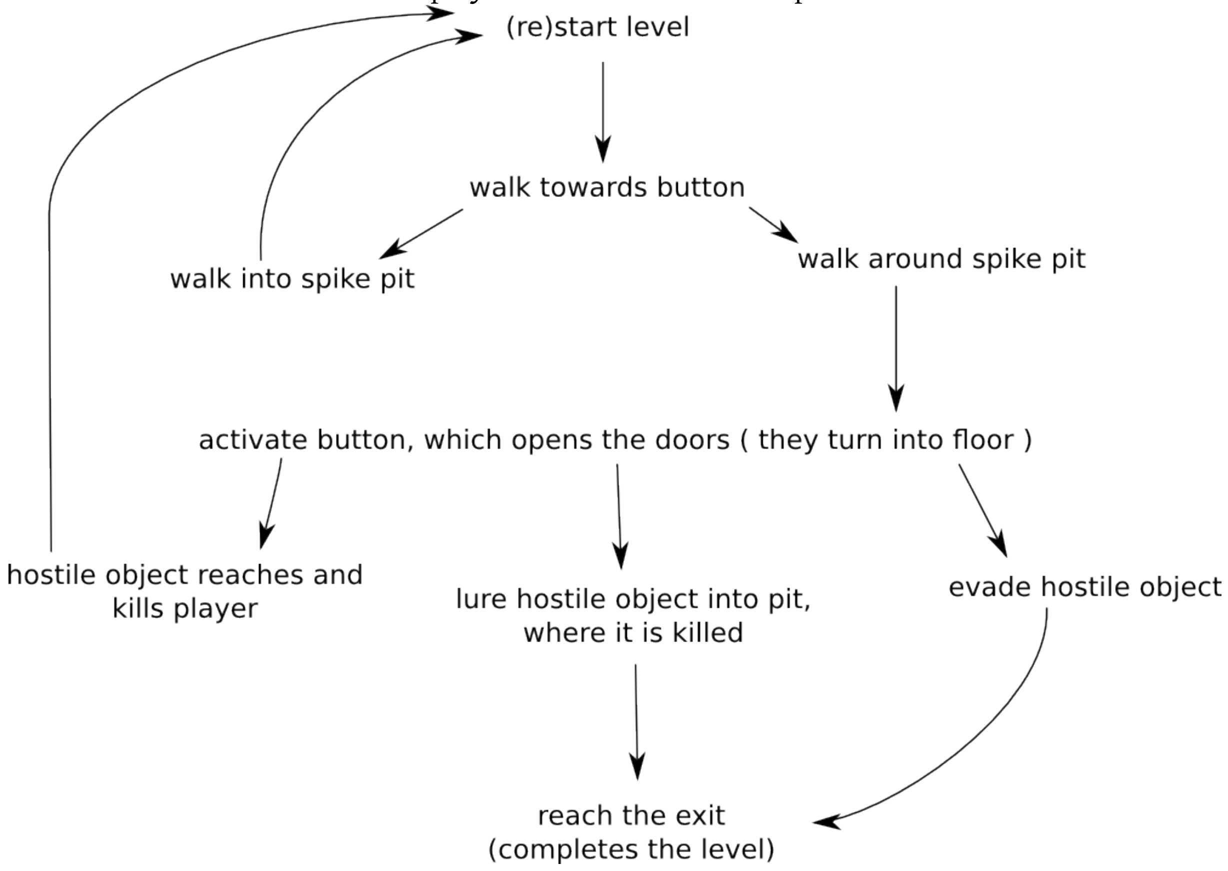
1. Starts program
2. Chooses an option at the main menu
    1. Optionally chooses to look at 'how to play' screen
    2. Returns to main menu
3. Chooses to play the game
4. Is presented with the first level
5. User attempts to work out how to solve the puzzle
    1. Attempts the level
    2. Fails
6. User succeeds
7. User is presented with increasingly harder and more complex levels introducing new ideas
8. User completes all levels
9. User wins the game, and is presented with a picture of a trophy to show their success

In-game level example (note how all map contents are squares on a grid):

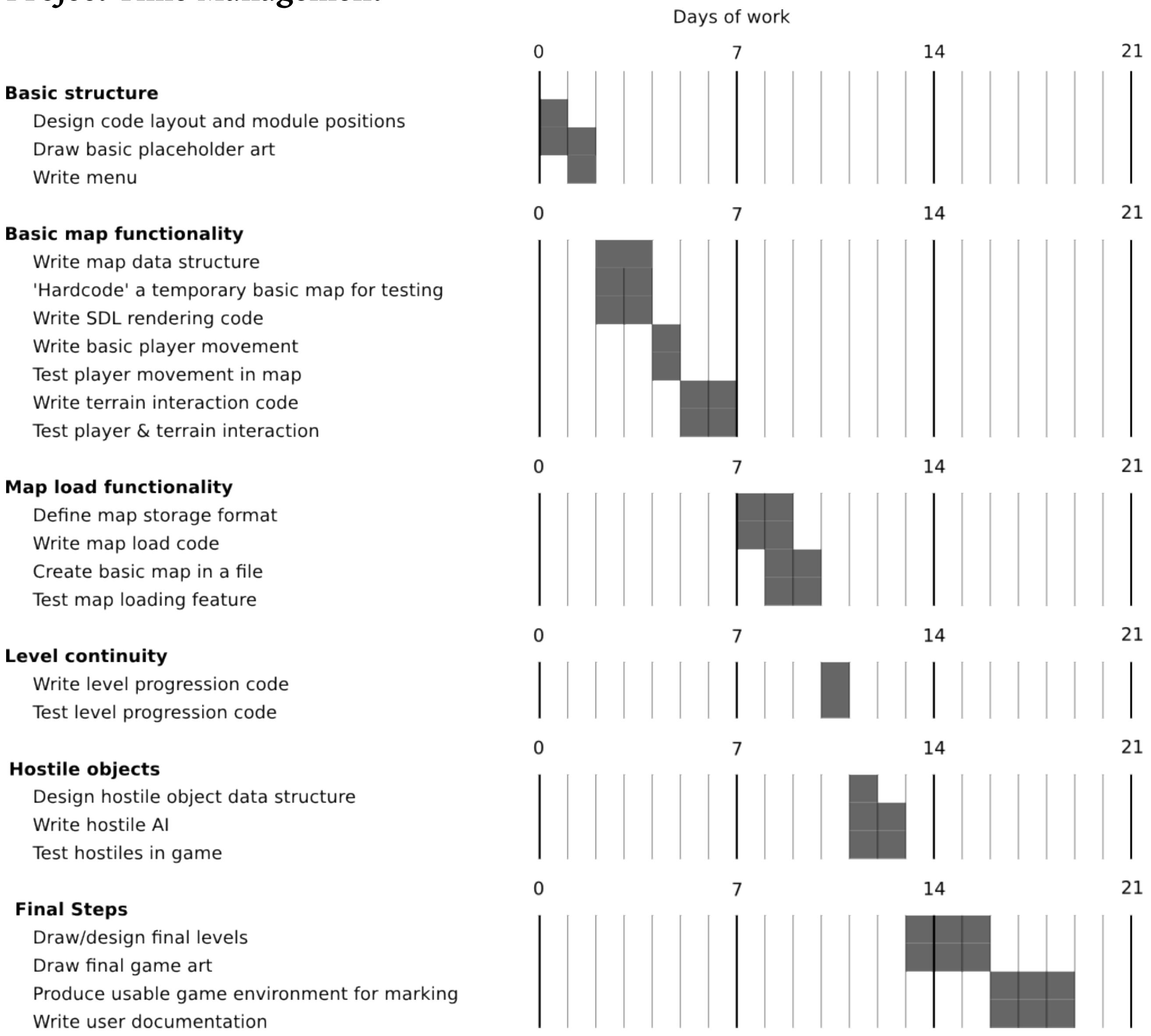


The monster will want to get to the player, and the player will want to get to the exit. If the object reaches the player, the level is restarted. If the player reaches the exit, the level is completed and the next one is loaded.

There are a few routes of action the player could take in this map:


(re)start level
walk towards button
walk into spike pit
walk around spike pit
activate button, which opens the doors (they turn into floor)
hostile object reaches and kills player
lure hostile object into pit, where it is killed
evade hostile object
reach the exit (completes the level)

Project Time Management



The Gantt chart above separates each algorithm from another's development days. Realistically multiple could be completed in the same time period, reducing the forecasted three weeks required to complete this project. The above chart would facilitate a busy three weeks where I am also working majorly on other subjects' work.

It is also important to note that I have successfully written similar code previously in a test-game. I have already developed algorithms to load levels from disk files and code to tell SDL how to render the level, which are shown in the *Key Algorithms* section of this paper.

To aid with the top-down modular approach I intend to use, dummy code is used throughout the proxy to emulate expected output of other modules until they are finished. An example is of the map loading algorithm, which is temporarily substituted with a hard-coded map during the development and testing of basic map functionality.

Finally each module has testing time allocated to determine whether or not the code fits my requirements. If it does not, it must be re-written and/or improved until it does so.

Levels of Development and Complexity

Base level

The foundation of my program will be its main module.

This level includes the organisation of where each module will rest in the code and the menu screen of my game.

Writing this level alone provides no way for the user to play or experience the game.

Primary level

This layer contains basic map data storage, game-play (input) and rendering code. The player can move his/her object around a map, but cannot interact with the terrain and can only play one level. There is no way to lose.

Effectively a one level maze game. This would be the minimum required to create a successful project.

Secondary level

Map interaction (such as doors and buttons) is introduced, as is multiple levels and a way of loading them externally from files so that they do not have to be hard-coded.

With map interaction the player can now be punished for making bad decisions (ie trapping themselves or walking into a pit of spikes).

Levels can be edited by editing their files using a simple text editor such as notepad. Each 'tile' of the level will be represented by a number, and coordinates of where the player starts will be stored at the end.

This level of development is a good fall-back point if circumstances become tricky with the tertiary level.

Tertiary level

Monster (hostile) objects are introduced, namely an unhappy face that tries to touch the player, causing the map to restart. Monster AI is written to allow monsters to intuitively (to a degree) hunt the player down the best the level allows them to do.

Monster location coordinates will need to be stored after the player coordinates in the map files.

This is my targeted level of development.

Extra level

In this level, items which the player can pick-up and use are integrated. This includes food, which can be used to distract monsters, and keys for unlocking doors.

This would require a lot more coding and data structure definition than the previous levels, and would require a modification of the file format. Much more depth would be added to the game, but more time and effort would be required than would be practical.

Key Algorithms

Some algorithms are written in psuedocode, others in C. I have modified psuedocode algorithms slightly to make them easier to read.

Key Concepts used in my algorithms

The common 'linear' or one-dimensional array contains a list of values, accessible under one name but with a different number called an index. For example, the following would display the name 'jane' on screen in most languages:

```
names = bob jane waldo
print names[2]
```

Note that in C, indexes of arrays start at zero, not one. This basically means that 'bob' is not names[1] but instead names[0], and names[2] would actually print 'waldo':

```
Index:   0     1     2
Value:  bob  jane  waldo
```

C supports multi-dimensional arrays. This simply means that more than one index can be used to access the value of an array. In a 2D array, two numbers are used. It is easiest to visualise this as a grid:

```
        X Coordinate
<----------------------------->
   0      1      2     3        ^
0 Fred   Jane   Oscar Lucas     |
1 Hales  Will   Tara  Satwick   | Y coordinate
2 Ned    Kelly   Mr   Osland    |
3 Food   Water  Bread  Nose     v
```

So *names[1][0]* would be 'Jane', and *names[3][2]* would be 'Bread' (here I have chosen to notate my coordinates as names[x][y]).

I use this concept to store the levels used in this game. For every square on screen, a value is stored in a 2D array (like above) that determines whether it is a wall, door, a bit of floor, a button etc.

Technically every square is represented by a number, so I might choose *1* to be open floor and *2* to be a wall. Visualising the data inside of the array as numbers instead of words such as 'door' and 'floor' also makes loading levels from files easier.

Player Movement

On a basic level (pun intended), we simply check to see if the the square in the direction the player wants to move is passable or not.

Note that variables in CAPITALS are pre-determined numerical values that would be unreadable if written directly in.

```
BEGIN
      dim keypressed // An integer

      // The following while loops through all of the keys the user has
      // pressed ( as opposed to an IF statement, which would only do one )
      // in case the user has quickly hit two or more keys in succession.

      WHILE SDL_TheUserHasPressedAKey returns true
            keypressed = SDL_GetKey  // Get the numerical value of this key

            // Playerx and playery are the coordinates of the player on
```

```
                // the grid.  The level's squares are stored in the 2D array
                // called mapsquares[][]

                IF keypressed is equal to KEY_LEFT
                      IF levelsquares[ playerx – 1 ][playery] equalto BLOCK_FLOOR
                            playerx = playerx – 1 // Move player to the left
                      ENDIF
                ELSE IF kepressed is equal to KEY_DOWN
                      IF levelsquares[ playerx ][playery] equalto BLOCK_FLOOR
                            playerx = playerx – 1 // Move player to the left
                      ENDIF
                ELSE IF
                      etc etc for UP and RIGHT
                END IF
          END WHILE
    END
```

Monster AI

The monster AI works in three 'modes':

Mode 1: No obstructions

First the x and y distances between the monster and the player are calculated:

```
    distancex = abs( monsterx – playerx ) // Abs makes sure the value is positive
    distancey = abs( monstery – playery )
```

Next the algorithm checks to see which measurement is greater. It will then travel along the longer dimension (either horizontally or vertically) until the point where both dimensions are the same. At this point the monster will zig-zag towards the player, because the two distance variables oscillate between being the same and one being smaller.

If the desired direction is blocked (not floor) the algorithm enters mode 2:

Mode 2: Moving along a wall

The monster will follow the newly hit wall towards the player along the applicable axis (X or Y).

The monster will continue to do this until either the wall it was travelling along disappears, or a corner is hit. The former initiates mode 3, whilst the latter re-starts mode 2.

Mode 3: Going around a corner

This mode simply moves the monster in the direction that a wall was being checked for in Mode 2 by one block, before returning to Mode 1. This ensures that the monster actually gets around the corner.

C code

Be warned that this is the single most complex part of my project. It is written here to prove that the algorithm is achievable, however this algorithm would require testing and tweaking to work acceptably.

Output-wise, this algorithm only moves the monster one square in any direction every time it is called. Its only inputs are values previously set by either the algorithm itself, or by another part of code (eg the code that initialises the monster's initial position).

This algorithm is designed to plan ahead, so that the next time the module is called, the monster knows which way it is moving and if it is following a wall or not. Traditionally the path of an object on a grid is completely pregenerated using an A-sharp algorithm variant, but this is beyond my skillset. (See https://secure.wikimedia.org/wikipedia/en/wiki/A*)

Most information is kept track of inside of the *monster* structure. An instance of this structure, called *themon* is used in this example.

```
// Give numerical values to directions
#define LEFT  1
#define RIGHT 2
#define UP    3
#define DOWN  4

// Give numerical values to level square types
#define BLOCK_WALL = 1
#define BLOCK_FLOOR = 2


// Structures are like arrays, but we name each value rather than using an index
// This structure holds information about a monster
struct monster
{
	// X and Y coordinates on the level grid
	int xpos;
	int ypos;

	int mode;  // Either 1, 2 or 3

	int direction_moving;   // Has values 1,2,3 or 4 to represent directions
	int direction_checking; // See the #defines at the top
}

// This struct holds level information.  Values that are be unused in this
// algorithm are omitted
struct level
{
	// Array containing contents of level
	int blocks[40][30];
	// Here we assume the level is 40 squares wide and 30 tall
}


// Moves the position of the monster
// Parsed arguments:
//   - the monster structure with its contained values
//   - the level strucutre containing level blocks
// Returned values:
//   1 if move was successful
//   0 if move was blocked by wall, did not move
int move_monster( struct monster *themon, struct level *themap )
{
	// The switch-case statement is a multiway selection, like we did in class
	// If it is hard to understand, visualise it as many ELSE IFs
	switch ( themon->direction_moving )
	{
		// In each we check if the desired direction is blocked by a wall
		// Walls are assumed to be the only impassible thing in this algorithm
		// but in the actual game doors and other things will probably be
		// included
		case LEFT:
			if ( themap->blocks[ themon->xpos - 1 ][ themon->ypos ] == BLOCK_WALL )
			{
				// Wall in the way
				return 0;
			}
			else
			{
				themon->xpos = themon->xpos - 1;
			}
			break;
		case RIGHT:
			if ( themap->blocks[ themon->xpos + 1 ][ themon->ypos ] == BLOCK_WALL )
			{
				// Wall in the way
				return 0;
			}
			else
			{
				themon->xpos = themon->xpos - 1;
			}
			break;
		case UP:
			if ( themap->blocks[ themon->xpos ][ themon->ypos - 1 ] == BLOCK_WALL )
			{
				// Wall in the way
				return 0;
			}
			else
			{
				themon->ypos = themon->ypos - 1;
```

```
                        }
                        break;
                case DOWN:
                        if ( themap->blocks[ themon->xpos ][ themon->ypos + 1 ] == BLOCK_WALL )
                        {
                                // Wall in the way
                                return 0;
                        }
                        else
                        {
                                themon->ypos = themon->ypos + 1;
                        }
                        break;
        }
}


// Makes the monster's movement decisions
// Parsed arguments:
//   - the monster structure containing monster values
//   - the player structure, only used for getting player positions
//   - the structure containing level contents
// Returns no values
void tick_monster( struct monster *themon , struct player *theplayer, struct level *themap )
{
        // The player structure is very similiar to the monster one
        // These distance variables are used in most of the modes
        int distancex = abs( themon->xpos - theplayer->xpos );
        int distancey = abs( themon->ypos - theplayer->ypos );
        // Remember abs() turns negative numbers into positive ones

        if ( themon->mode == 1 ) // Mode 1: No obstructions
        {
                if ( distancex > distancey )
                {
                        // Move along X axis ( left-right )
                        if ( themon->xpos > theplayer->xpos ) themon->direction_moving = LEFT;
                        else themon->direction_moving = RIGHT;
                }
                else
                {
                        // Move along Y axis ( up-down )
                        if ( themon->ypos > theplayer->ypos ) themon->direction_moving = UP;
                        else themon->direction_moving = DOWN;
                }

                // Attempt to move monster.  If unsuccessful, enter mode 2
                if ( move_monster( themon, themap ) == 0 )
                {
                        themon->mode = 2;
                        // Also leave info to say which direction is the wall
                        // for mode2 to use
                        themon->direction_checking = themon->direction_moving;

                        // Finally give mode 2 a direction to keep moving
                        if ( direction_moving == UP || direction_moving == DOWN ) // || means OR
                        {
                                if ( themon->xpos > theplayer->xpos ) themon->direction_moving = LEFT;
                                else themon->direction_moving = RIGHT;
                        }
                        else
                        {
                                if ( themon->ypos > theplayer->ypos ) themon->direction_moving = UP;
                                else themon->direction_moving = DOWN;
                        }
                }
        }

        // Notice that this is IF, not ELSE IF
        // This is incase mode is set to 2 inside of mode 1, otherwise we the monster
        // would not move at all during that run of this module

        if ( themon->mode == 2 ) // Mode 2: Moving along a wall
        {
                // Move the monster, and check if we have run into a corner
                if ( move_monster( themon, themap ) == 0 )
                {
                        // We use this in a second
                        int old_direction_moving = themon->direction_moving;

                        // We should go the opposite direction of the wall we
                        // are following/checking
                        switch ( themon->direction_checking )
                        {
                                case LEFT;  themon->direction_moving = RIGHT; break;
                                case RIGHT; themon->direction_moving = LEFT; break;
                                case UP;    themon->direction_moving = DOWN; break;
                                case DOWN;  themon->direction_moving = UP; break;
                        }

                        // But nomatter what, we now follow along the wall we tried
                        // to move into
                        themon->direction_checking = old_direction_moving;
                }

                // Check to see if the wall has disappeared
                // note && = and, || = OR
                if ( themon->direction_checking = RIGHT && themap->blocks[ themon->xpos + 1 ][themon-
>ypos] == BLOCK_FLOOR
                  || themon->direction_checking = LEFT  && themap->blocks[ themon->xpos - 1 ][themon-
>ypos] == BLOCK_FLOOR
                  || themon->direction_checking = DOWN  && themap->blocks[themon->xpos][ themon->ypos
+ 1 ] == BLOCK_FLOOR
                  || themon->direction_checking = UP    && themap->blocks[themon->xpos][ themon->ypos
- 1 ] == BLOCK_FLOOR )
                {
                        // Enter mode 3
                        themon->mode = 3;
                        // Tell it to move where the wall once was
                        themon->direction_moving = themon->direction_checking;
                }
        }
        // For mode 3 we do use an ELSE IF so that mode 3 cannot run
        // after mode 2 until this module is called again
        else if ( themon->mode == 3 ) // Mode 3: Going around a corner
        {
                // Nomatter what, moving is possible if in this mode
                // Otherwise the long check in mode 2 would have failed
                // That means we do not have to check if the move was
                // successful or not

                move_monster( themon, themap );

                themon->mode == 1;
        }
}
```

Software Specifications and Requirements

Operating System

The project is planned to be written for i686 (normal) Linux. A USB stick containing Linux and the final project along with instructions on how to boot from it to play/run the game will be provided.

If time permits, it may be possible to create a Windows build of the game. The SDL libraries are cross platform, which means they work on all operating systems, but minor changes may have to be made to my code to operate in different operating systems.

An example of this is how Windows uses CRLF line endings, which would interfere with my level-loading module designed for Linux's LF line endings. This example problem would be trivial to fix and would not affect a user until they decided to edit a level themself.

Programming Language

The whole project will be written in the C language. The project will be compiled with the Gnu Compiler Collection (GCC) which is a free and open source compiler system for the language C.

If needed, instructions on how to compile the source code will be provided.

The C language provides a number of advantages over the Visual Basic language (namely dynamic allocation and referencing), but most of all Visual Basic is not designed for games whilst C is a good choice for such a project.

Hardware Requirements

Hardware requirements for the game alone:

- A pentium 2 or higher processor of preferably i686 architecture
- A colour monitor with a screen resolution at or above 640x480
- At least 100MB of ram (free, not including that taken up by the operating system).
- A keyboard (a mouse is not used in the game)

Hardware requirements for the final product on a USB stick:

- USB port
- BIOS capable of booting from a USB device

All of the black school Lenovo Thinkcentre desktop computers (those that for example inhabit all of K13) have specifications that meet and exceed all of these requirements. Most computers manufactured post-millennium meets these requirements.

For security reasons, government issued Year 9 laptops have their ability to boot off USB devices password protected, so they cannot be used to run the program.