Developer Documentation for ‘The Echidna Menace’

1 - Code Licence

This project and all of its contents are distributed under the terms of the GNU General Public Licence, Copyright William Hales 2012. A copy of this licence is included with this game in the file ‘gpl.txt’ .

1.1 - GPL Boilerplate

```
This game, code, resources and documentation are part of The Echidna Menace

The Echidna Menace is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

The Echidna Menace is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with The Echidna menace.  If not, see <http://www.gnu.org/licenses/>.
```

2 - C language concepts used in this game

The C language has a few concepts not immediately understandable by developers only knowledgeable of languages such as Visual basic. This sections briefly explains them.

2.1 - #include

'#include' statements are not actually part of the C language, but are instead lines of text to tell the compiler what to do. When another source file (eg 'level.c') is 'included' its entire contents are pasted into the current file, where-ever the line with '#include "level.c" is written.

My project also 'includes' some files that I have not written, such as the SDL libraries and standard libraries such as <stdio.h>. These 'header files' contain code I use in my project but I did not write myself.

2.2 - Pointers

In the C language the developer has control over how information is stored into the computer's memory.

Working with variables is no different from any other language:

```
int age_of_dog;
age_of_dog = 5;
```

However in situations where an unknown amount of things/variables need to be generated, pointers need to be used. Pointers are a type of variable but instead of containing a value like '5' they contain the location (address) of another variable. Hence they 'point' to it.

Eg:

```
int fred_age = 5;
int *pointer_to_fred_age;  // The '*' before the variable name signifies it is a
                           // pointer.  It is still considered an int (integer)
                           // because that is what it points to

pointer_to_fred_age = &fred_age;  // The '&' tells the computer to get fred_age's address,
                                  // as opposed to its value

// Both of the following lines would output the exact same message
printf("Fred, you are %d years old\n", *pointer_to_fred_age );
printf("Fred, you are %d years old\n", fred_age );
```

2.3 - Malloc

Malloc allows you to create variables that you did not write directly into your code.

Say for example the user can ask to store an unknown amount of dog names - instead of having a thousand dog name variables coded in, you simply request the memory for them with malloc for as many as you need.

```
// Ask malloc to create a space for cat_age.  We tell malloc to find some space the size
// of an integer, because that is what cat_age is.
int *cat_age = malloc( sizeof(int) );
```

Notice that *cat_age is a pointer (hence the '*'). That's because malloc returns an address to where it has found some memory for cat_age to be stored.

2.4 - Structures

Structures are a way of making your own data types that contain multiple other variables. In databases they are known as 'records'.

For example you could create the structure 'human':

```
struct human
{
   int age;
   int number_of_eyeballs;

   char name[20];
};
```

Now we can create humans like we create other variables such as integers

```
struct human fred;
struct human joanne;
```

The variables inside each of the human structures can be used as you would any other:

```
fred.age = 20;
fred.number_of_eyeballs = 1;

joanne.number_of_eyeballs = -20;
joanne.age = fred.number_of_eyeballs;
```

2.5 - Multi Dimensional arrays

First of all, note that in the language C arrays start at index 0. This means if I made an array five big, it would have the indexes 0,1,2,3,4 not 1,2,3,4,5 .

The simplest array is called a linear array:

```
int cat_ages[5];
cat_ages[0] = 20;
cat_ages[4] = 3;
cat_ages[5] = 9; // This line would be invalid -- see above
```

This simplest type of array is known as a 'one-dimensional array' because you can visualise it as values on a line:

```
cat_ages[0]----cat_ages[1]----cat_ages[2]----cat_ages[3]----cat_ages[4]
    20     ----    0      ----    0      ----    0      ----     3
```

In two dimensional arrays we specify *two* index numbers, a bit like coordinates on a 2D plane. I use 2d arrays to store information such as the graphics used in my game (as pictures are just a 2d grid of colours) and the level itself, which is a 2d array of walls, floors, doors etc.

```
// An example two-dimensional array
int tiles_of_the_level[40][30];  // 40 wide, 30 high

tiles_of_the_level[22][3] = 1;
```

3 - Project Layout

This project is divided into sections called modules. A top-down approach has been used in its design, where the execution starts with 'top-level' modules which then runs/call smaller lower level modules, before returning execution back to the top.

Never do modules circularly call each other in my code. Ie if module A calls module B, module B is designed so that it can never call module A, as that would be a legal but 'circular call'. This is mainly done to simplify the layout of the program.

For a few sections of code I will explain their workings in this manual. The documentation inside the code for the rest of the project should explain how other things work provided you have read an understood the parts explained here.

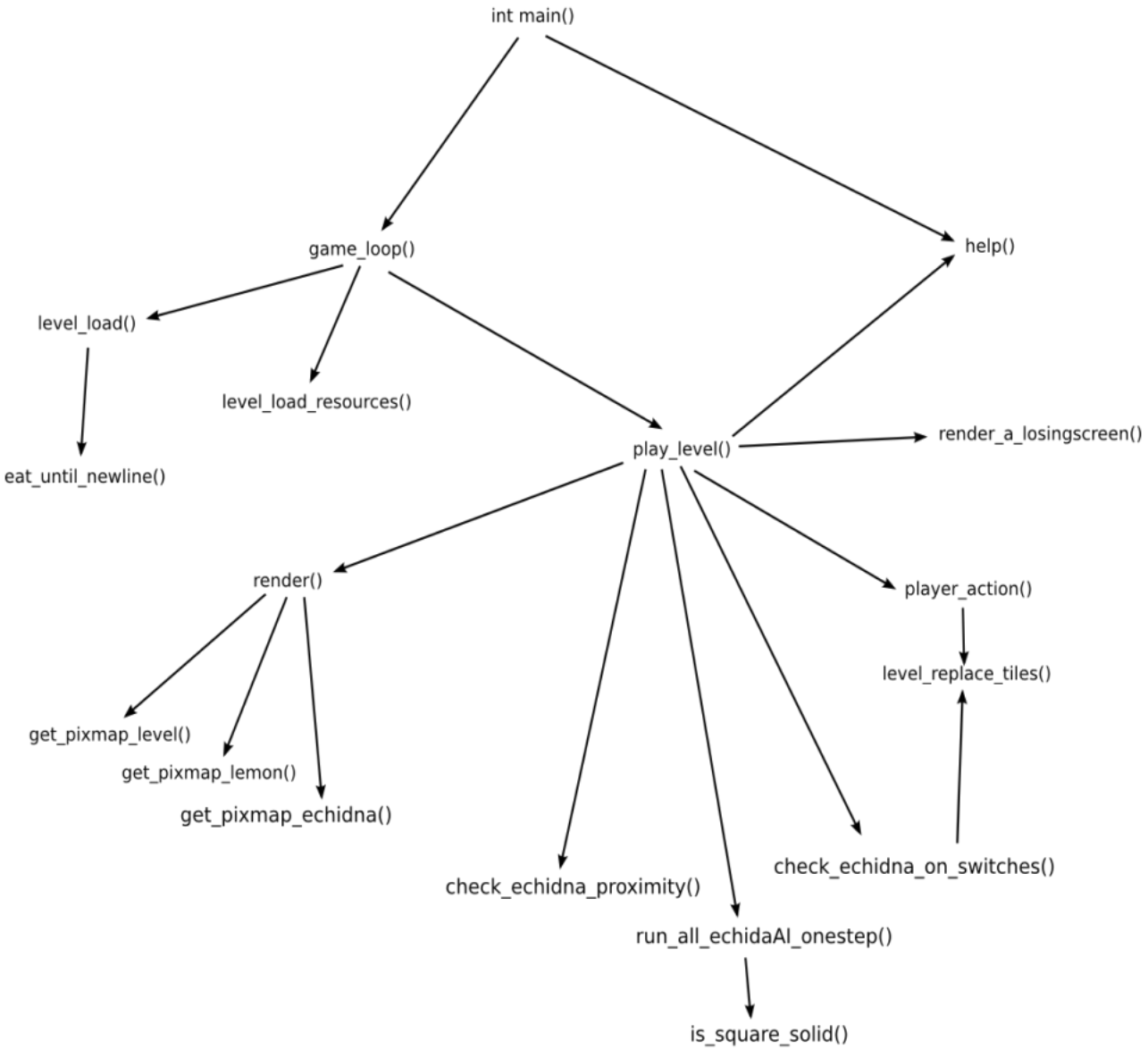
3.1 - Code modules layout

The first module run when you start the game in the main() module. This module does some work, which includes running or 'calling' other submodules.

The diagram below shows which modules use what other modules in the project. For example, the game_loop() module calls the level_load_resources(), level_load() and play_level() modules, each of which then may call other modules as shown on the diagram.

This diagram shows every module in the game I have written except for the error-sdl() module, which gets run if SDL finds a reason the game cannot be played and it has to

exit. It is called by various parts of the code, but it does not provide any structural value to the layout of the program.



3.2 - main() module

When you start the game the first thing to run in the main() module. This main module does two things:

1 - Initialises SDL

2 - Loops the menu

3.21 SDL initialisation

SDL is used in this project to handle the window on-screen, allow the code to draw or ‘blit’ pictures onto this window and to handle input. To use SDL first we must initialise it.

Initialising SDL is actually a simple process, but in the project it is wrapped in error handling code in case something goes wrong:

```
// Initialise SDL
printf("Initialising SDL... ");
if ( SDL_Init( SDL_INIT_VIDEO ) == -1 ) error_sdl("Could not initialise SDL");
atexit(SDL_Quit);
```

SDL_Init() is fed the input SDL_INIT_VIDEO telling the SDL libraries that the code will want access to the ‘video’ sections of SDL’s code. This ‘video’ code allows the project to do things on the screen of the computer, such as show picture of the menus and level.

Next a ‘surface’ is created for the game to work with:

```
// Set surf_screen
SDL_Surface* surf_screen = NULL;
surf_screen = SDL_SetVideoMode( 640, 480, 8, SDL_SWSURFACE | SDL_ANYFORMAT | SDL_DOUBLEBUF
);
if ( surf_screen == NULL ) error_sdl("Could not set videomode");
```

An SDL surface is akin to an image, made of pixels. The window the game is inside is treated as a surface in SDL, as in actual fact what you see on a computer screen is merely an image.

SDL_SetVideo_Mode() creates a window (640 pixels wide, 480 pixels high) and returns the address of where it has stored the surface for that window, which is then stored in the pointer “surf_screen”.

At this point a window now appears in front of the user.

3.3 - File Layout

For clarity reasons the code in the project as well as the graphics and level resources are all split into separate files.

3.3.1 Project Code

All files ending in ‘.c’ are source code for the game. Although split over many files, the code could have just been put all into the one file and would work, but that would create a cumbersomly long single file that would be difficult to navigate.

3.3.2Graphics

All of the files ending with ‘.png’ contain the graphics used in the game. You can open and change them with any image editor/viewer.

entities.png - graphics for the lemon and echidnas

failscreen_??.png - various whole-screen images displayed when the player gets killed

helpscreen.png - help screen image

tiles.png - all of the various tiles used in maps, including walls, doors buttons and spikes

titlesceen.png - the menu screen

3.3.3 Levels

Any file ending in .level is a game level. The game loads them according to their number, but they must have the rest of the name (‘level_’) as well else the game will ignore them.

3.3.4 Misc

There are two more files in the game’s folder - ‘echidna_game’ and ‘Makefile’.

‘echidna_game’ is the actual compiled executable of the game you run to play it. In Linux it has no file extension, but in Windows it would end in ‘.exe’.

The ‘Makefile’ is a set of instructions telling a program called ‘make’ how to compile the program. Compilation is handled later in this manual.

4 - Deeper code analysis

4.1 - Levels, how to make them and change them

4.1.1 - Level data structure

Levels are stored inside of the level structure:

```
struct level_struct
{

   char squares[SCREEN_WIDTH_SQUARES][SCREEN_HEIGHT_SQUARES]; // Squares of the level

   // Player details
   int player_face;  // Picture number ( player has 4 different looks )
   int px, py;  // Player X and Y position, on the tiles
   int sx, sy;  // Player X and Y starting positions

   // Resources
   SDL_Surface * surf_entities; // Player and echidna pictures
   SDL_Surface * surf_tiles;    // Level tiles/squares

   // A various selection of losing screens
   SDL_Surface * surf_losingscreens[5];

   // Up to ten echidnas
   struct echidna echidnas[10];

};
```

This structure stores several key things:

- Player information
- Echidna information
- Graphics resources
- The level's contents itself

4.1.2 Level files

To play a level, a file containing a description of that level's contents is first loaded. The simplest is the first level: *level_01.level*

```
The first introductory level
4 10
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 3 1 1 1 1 1 1 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0
 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0
 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The first line of a level file is completely ignored by the game - you can type anything you want in it.

The second line contains the player's starting position, the first number being his X coordinate and the second his Y coordinate. Remember that the coordinates in C start from 0 (ie the top-left square of a level is 0,0 not 1,1).

After the first couple of lines is what appears to be a giant pile of numbers. If you look closely at the block of numbers you may notice they strike a resemblance to the first level.

Each of those digits is actually a two digit number - it is just that ' 1' is the same as '01', and the extra white-space makes it easier to read and change.

Each two-digit number represents a different tile or square each level is made up of. In this level only three different types of tile are used:

01 - Floor

00 - Wall

03 - Exit staircase

The full list of values is outlined in *level.c*:

```
// Row 1 - main squares
#define T_WALL  0
#define T_FLOOR 1
#define T_SPIKE 2
#define T_EXIT  3

#define T_BLUE_BUTT_UP 4   // unpressed button
#define T_BLUE_BUTT_DOWN 5 // depressed button
#define T_BLUE_DOOR_UP 6   // door unlocked by a button
#define T_BLUE_DOOR_DOWN 7 // door locked and impassible

#define T_YELL_BUTT_UP 8
#define T_YELL_BUTT_DOWN 9
#define T_YELL_DOOR_UP 10
#define T_YELL_DOOR_DOWN 11

#define T_RED_SWITCH_ON 12
#define T_RED_SWITCH_OFF 13
#define T_RED_DOOR_UP 14
#define T_RED_DOOR_DOWN 15
```

Here is a more complicated level that was modified further before releasing the game:

```
The first witnessing of echidnas
5 7
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0
 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 0 0
 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 1 1 0
 0 1 0 0 1 1 1 1 114 1 1 1 1 3 1 0 1 1 0
 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 1 1 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
 0 1 0 0 0 0 0 0 012 1 1 1 0 0 0 0 0 1 1 0
 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0
 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0
 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11 1 3
9 3 0
```

Here two digit numbers are used. Make sure these numbers line up perfectly, else the game WILL NOT behave expectantly.

You may notice there are two extra lines after the level. These lines tell the game that this map has two echidnas (one per line). Echidna information is stored as:

xcoordinate ycoordinate facenumber

Where *facenumber* is a number from 0 to 3 denoting which of the four faces that particular echidna uses. If you forget this number this may act strangely.

4.1.3 - Editing levels and making your own

You can edit the level files with any old text editor - notepad included. I would recommend you edit the first level to start with, as it is the simplest.

Be warned however that VERY LITTLE error checking code is implemented into the game regarding loading levels, so if you stray from the correct level format, the game will not load the level as you expected.

Things to be wary of:

- changing the dimensions of the level
- not making the numbers line up
- using numbers for tiles that do not exist
- placing echidnas inside walls, which can lead to the game hanging (crashing)

The maximum amount of levels in the game is hard coded in *game_loop()* of *game.c*:

```
// Run through each of the levels

   for ( level_number = 1; level_number <= 10; level_number++ )
   {
          int level_state = PLAYER_IS_OK;
```

Change the 10 to any number you want. If the game tries to open a level file that does not exist, it will immediately close.

4.2 - Level loading code

A few concepts of my level loading code are a little obscure to those not intimate with the language C.

```
// Attempt to open the file
   FILE *levelfile = fopen( levelname, "r"); // r = reading mode
```

When a file is opened, its location is stored in a pointer. In this case it is called ‘level_file’.

The C language’s file handling makes it so that the program remembers where you are up to in a file in-between the times to you data from it.

So for example, when you open the file you’re reading position is the beginning of the file. The first line of a level however is junk text not used by the game, so we have to skip this.

To do this I wrote a function called eat_until_newline:

```
void eat_until_newline( FILE *currentfile )
{
   char letter='a';
   while ( letter != '\n' ) // The '\n' represents the invisible line ending characters
(either CRLF or LF)
   {
          // Every time a character is read, the current position inside of the file is
also moved forward
          // one character
          letter = fgetc( currentfile );
   }
}
```

eat_until_newline will read the file, one character (letter) at a time until it has detected it has finished a whole line, then stops. In text files, an invisible character called either CR, LF or both is placed at the end of each line to signify a line of text has ended, and this subprogram quits upon detecting that.

After running eat_until_newline once, we are now up to the second line of the file. This line contains the player’s position. Reading the two numbers out is easy with a formatted expression:

```
fscanf( levelfile, "%d %d", &level->sx, &level->sy );
eat_until_newline(levelfile);
```

‘fscanf’ is a standard C function that I have told to read data in the format “%d %d” from the file. That is, two numbers (**d**igits) separated by a space. It then stores those two numbers in level->sx and level->sx, which is where the starting positions for the player are stored.

The eat_until_newline function is called again as fscanf has only been told to gobble up two numbers, but not the end of line symbol. If we do not proceed past the end of line

symbol, then the next time we read text from the file we will get that nasty symbol instead of what we wanted.

Now we move onto the main part of the level loading code:

int row, column; // Current position

```
    for ( row=0; row < SCREEN_HEIGHT_SQUARES; row++ )
    {
           for ( column=0; column < SCREEN_WIDTH_SQUARES; column++ )
           {
                  /* we read each two letter number into a three letter character array
                   * ( string )the third character is used to represent the end or
                   * 'termination' of the string */
                  char tempstring[3];
                  fgets( tempstring, 3, levelfile );

                  // atoi() is a standard C function used to convert a string of numbers
                  // into an integer

                  // form into an actual number ( integer )
                  level->squares[column][row] = atoi( tempstring );
           }

           // We need to make sure that we finish the current line of numbers/characters in
           // the level file, and are up to the next, else bad things may happen
           eat_until_newline( levelfile );
    }
```

The first FOR loop is looped for every row of the text file. Inside is another FOR loop that loops through every ‘square’ of a level (per row).

Each time the inner FOR loop runs, two of the file’s characters are read. That is because each tile in the level is represented by a two digit number. These two characters are then converted into an actual number which is then stored in the level structure where it can be used to play the level.

Finally eat_until_newline is called again to gobble up the nasty invisible endline/newline characters every line. If extra spaces or letters are on the end of each line for the level’s tiles/squares, it will also gobble them up too, avoiding possible errors cropping up.

The last thing the level loading code does is load up the echidnas:

int ech_no; // Echidna number

```
  for ( ech_no = 0; ech_no < 10; ech_no++ )
    {
           level->echidnas[ech_no].this_echidna_exists = 0;
           level->echidnas[ech_no].mode = 1; // Start in hunt mode

           // Now load the echidna values in
           if ( fscanf( levelfile, "%d %d %d", &level->echidnas[ech_no].xpos, &level-
                        >echidnas[ech_no].ypos, &level->echidnas[ech_no].face  ) == 3 )
           {

                  level->echidnas[ech_no].this_echidna_exists = 1;

                  printf("Echidna loaded at x:%d y:%d\n", level->echidnas[ech_no].xpos,
                                                level->echidnas[ech_no].ypos );
           }
  }
```

A couple of lines are too wide for this document, and are split over two lines in the above example.

The echidna loading code runs ten times, each time trying to gather the numbers from the end of each level file telling it where the echidnas are located.

Once again fscanf is employed to read out formatted information: in this case “%d %d % d” (three space separated numbers).

The code will only activate an echidna if fscanf succeeded in reading all three numbers. This is so levels can have different amounts of echidnas without harm.

4.3 - AI of the Echidnas

The echidna AI has two modes:

- hunting the player
- following a wall

The echidna AI is completely separated into the ‘echidna_ai.c’ file to separate it from the rest of the code because it is so complex and long (around 246 lines).

4.3.1 - MODE_HUNT

In hunting mode, the echidna:

1) Measures the vertical and horizontal distance between it and the player

(the diagonal distance is never measured)

2) Checks which of these distances is larger

3) Moves in the direction that would reduce the larger of the two distances.

In a perfect world, this would be all the Echidna AI needs. Unfortunately the hunt AI cannot cope with obstacles and walls.

If an Echidna hits a wall in HUNT mode, it enters wall following mode.

Before this mode however, the direction it is moving in changes in a clockwise direction. To the Echidna’s perspective this would mean always turning to its right when it hits a wall.

4.3.2 MODE_FOLLOW_WALL

The Echida tries to follow the wall until either:

1) The wall dissapears

2) It finds itself in a corner.

If the wall disappears it will step around it and then resume HUNT ai.

If it finds itself in a corner, it will turn its direction 90 degrees clockwise and now follow the new wall. Note that a dead end is simply seen as two corners by this AI, and will make the Echidna turn around.

5 - Compilation Instructions

5.1 - Requirements

This game requires the following things to compile:

- the main SDL libraries & header files
- SDL_image libraries & header file
- The GNU C compiler collection (GCC)

All of these resources are already available in the environment on the USB stick. See the instructions in the user manual to learn how to enter this environment.

5.2 - Guide

The game is setup to automatically compile if the ‘make’ command is executed inside of the directory containing the game’s source code.

The easiest way to run ‘make’ is to:

1) Open up any of the .c source code files by double clicking on them

2) Press SHIFT+F9 on your keyboard

3) Wait for the game to compile

Messages will appear at the bottom of the text editor’s (Geany’s) window, telling you if the compiler succeeded or not. If an error was encountered, it will tell you what type of error and at what line number in what file this error exists in this same area.

The newly compiled game executable can either now be run by pressing F5 inside of the Geany text editor, or by locating it (inside of the ‘source’ folder, not ‘game’) and double-clicking on it.

6 - Data Dictionary

Most of the data is passed throughout the game inside of the level structure. A single instance of this structure (called ‘currentlevel’) is passed around through the modules.

A second structure called ‘echidna’ is used to store individual echidna information. This structure is stored inside of the level structure.

Below are the definitions of the values inside these two structures and the comments explain their purpose, or where to read the code to further investigate their purpose.

6.1 - Level struct (level.c)

```
/* The level structure is passed to the rendering module, so everything that needs to be
seen
 * on-screen is also stored here as well
 *
 * Eg:
 *    - player position
 *    - resource locations ( graphics )
 */
struct level_struct
{

   char squares[SCREEN_WIDTH_SQUARES][SCREEN_HEIGHT_SQUARES]; // Squares of the level

   // Player details
   int player_face;  // Picture number ( player has 4 different looks )
   int px, py;  // Player X and Y position, on the tiles
   int sx, sy;  // Player X and Y starting positions

   // Resources
   SDL_Surface * surf_entities; // Player and echidna pictures
   SDL_Surface * surf_tiles;    // Level tiles/squares

   // A various selection of losing screens
   SDL_Surface * surf_losingscreens[5];

   // The screen to display when the game ends
   SDL_Surface * surf_winning_screen;

   // Up to ten echidnas
   struct echidna echidnas[10];

};
```

6.2 - Echidna structure (level.c)

```
// An echidna is one of the monsters

 // The structure is define here in level.c so it can be loaded from the level files
 struct echidna
 {
```

```
  int xpos, ypos; // Position on map
  int face; // Which of the echidna pictures to use

  int mode; // As described at the top of the document

  int this_echidna_exists; // When this is equal to 1, the echidna is alive and on the
level

  // These movement directions are used in the AI code
  signed int move_x;
  signed int move_y;
  // ditto here
  signed int check_x;
  signed int check_y;

};
```