Final Report

By Team 007

This is the final report for the Penguin Game project. Here we'll define the various data structures used, different files and functions created and description of where they are used. This report also contains the summary of the project and final algorithm used by the game in autonomous play.

Data Structures

The following are the data structures we found suitable and convenient to use accordingly in our game.

1.Player

```
typedef struct
{
	char name[16];
	int numberOfPenguins;
	penguin *penguinList;
	int playerID;
	int score;
}player, player_t;
```

Above is defined our player structure and its members. All of the members are self-explanatory except for the *penguinlist. This is a pointer to struct penguin (i.e defined in penguin.h file). It create a list of penguins (instances of Penguin struct) and puts them into an array called penguinList.

2.Penguin

```
typedef struct
{
	int position[2];
}penguin, penguin_t;
```

This is a simple structure that is used to hold the position of the penguins.

3.Tile

```
typedef struct
{
	int fishNum;
	int active;
	int isPenguin;
	int playerID;
} tile_t;
```

The structure defined above is used in the program to store and read data from it regarding the current number of fish on the tile, player id of the player present on the tile (0 if no player is present) and other members are used for cross reference, thus preventing any bugs or errors in the game.

4. Board[m][n][4]

```
board[row][col][0] - numFish (up to 3, number of fish on the tile)
board[row][col][1] - isPenguin (0 if there's no penguin on the tile 1 if there is)
board[row][col][2] - active (whether it's active or not, 1 is it is, 0 if not)
board[row][col][3] - playerID (id of the player on the tile, 0 if there's no one)
```

The 3-dimensional array, where m and n are dimensions of the board, is a result of initial ideas. Initially project was said to be based on two dimensional array, later it was 3-dimensional array presented above, used to store the data about the current state of the game. In contrary to the movement phase, placing phase was not rewritten using structures as it was almost fully functional when the existence of structures was discovered by the group. It is used to make any operations during placement phase in penguins_placing.c as it or its pointer is passed to every function in that file.

Algorithm

These were two separate algorithms used to place and move penguins during automatic mode.

1. Placing phase automatic

During placing phase the algorithm was designed to choose the best position on the board that would allow huge variety of movements in movement phase and proximity of tiles with plenty of fish on it. All functions related to the algorithm are located in placing_penguins.c file.

First of all, number of tiles with one fish is checked using function *int tiles_with_one_fish(int m, int n, int board[m][n][4], int total_penguins_to_be_placed, int mode)* which in automatic mode returns number of spaces available for penguin to be placed.

```
    int spaces_available = 0;
    int i, j;

    for(i=0; i<m; i++) //Checking if there is sufficient number of tiles
for penguins
    {
        for(j=0; j<n; j++)
        {
            if(board[i][j][0] == 1 && board[i][j][3] == 0) //If there 1
fish and no penguins
            spaces_available += 1;
        }
    }
```

This function takes as parameters 3-dimensional array board, its m,n dimensions. In automatic mode each turn one penguin is placed, so *total_penguins_to_be_placed* is always equal to 1 and *mode* is set to 2 (1-interactive, 2-automatic), because this function is also used in interactive mode in a slightly different way.

```
int spaces_available = 0;
    int i, j;

    for(i=0; i<m; i++) //Checking if there is sufficient number of tiles
for penguins
    {
        for(j=0; j<n; j++)
        {
            if(board[i][j][0] == 1 && board[i][j][3] == 0) //If there 1
fish and no penguins
            spaces_available += 1;
        }
    }
```

If there is no appropriate tile for a penguin to placed, error message is displayed.

Knowing number of spaces available, 2-dimensional array of those tiles and its pointer is created.

```
    int tiles_empty[spaces_available][3]; //Stores information about all
empty tiles
    int *pointer_to_tiles_empty = &tiles_empty[0][0];
```

Subsequently array is filled using function *int fill_tiles_empty_array(int m, int n, int board[m][n][4], int spaces_available, int *pointer_to_tiles_empty)* that takes as parameters 3-dimensional array board with its dimensions m,n, number of spaces available for a penguin to be placed and pointer to the array described above.

```
int fill_tiles_empty_array(int m, int n, int board[m][n][4], int
spaces_available, int *pointer_to_tiles_empty)
{
    int i, j;
    int number_of_tiles = 0;//Number of tiles currently detected as good
for placing

    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            if(board[i][j][0] == 1 && board[i][j][3] == 0)
            {
                *(pointer_to_tiles_empty + 3*number_of_tiles) = i;
//Assign m coordinate
                *(pointer_to_tiles_empty + 3*number_of_tiles + 1) = j;
//Assign n coordinate
                *(pointer_to_tiles_empty + 3*number_of_tiles + 2) = 0;
//Set points to 0
                number_of_tiles++;
            }
        }
    }
    if(number_of_tiles != spaces_available)
    {
        printf("Skipper: Kowalski, analysis.\n");
        printf("Kowalski: Not all or too many tiles were taken into
consideration while filling scorearray for automatic placing. \n");
        return 2;
```

```
    }
    return 0;
}
```

Each tile that can be chosen, is assigned 3 values. First two of them are its m and n coordinates on the board. The third one is number of points, which is initially 0 for each tile.

Then, number of points is changed to select the best tile. Coordinates of the best file are stored in array:

```
int best_tile[2]; //Coordinates m,n of the best tile to place a penguin
on
```

Function that overrides number of points is *void assign_points(int m, int n, int spaces_available, int board[m][n][4], int tiles_empty[spaces_available][3], int *pointer_to_best_tile)* that takes as parameters 3-dimensional array board with its dimensions m,n, array tiles_empty and pointer to array with coordinates of best tile.

```
    int sum_fish_in_row[m];//Total number of fish in each row
    int sum_fish_in_column[n];//Total number of fish in each column
    int places_occupied_in_row[m];
    int places_occupied_in_column[n];
```

At first, sum of fish and penguins placed in each row and column are calculated.

```
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            if(board[i][j][1])
            {
                places_occupied_in_row[i]+=1;
                places_occupied_in_column[j]+=1;
            }
            sum_fish_in_row[i] += board[i][j][0];
            sum_fish_in_column[j] += board[i][j][0];
        }
    }
```

Finally, points for each tile are assigned.

```
    for(i=0; i<spaces_available; i++)
    {
        tiles_empty[i][2] -= abs(i-half_spaces_available); //Setting
penguins as much int the center as possible
        int m_pos = tiles_empty[i][0];
        int n_pos = tiles_empty[i][1];
```

```
        tiles_empty[i][2] +=
10*(sum_fish_in_row[m_pos]+sum_fish_in_column[n_pos]);
        tiles_empty[i][2] -=
100*(places_occupied_in_row[m_pos]+places_occupied_in_column[n_pos]);
        if(tiles_empty[i][0] == 0 || tiles_empty[i][0] == m-1)
        {
            tiles_empty[i][2] -= 250;
        }
        if(tiles_empty[i][1] == 0 || tiles_empty[i][1] == n-1)
        {
            tiles_empty[i][2] -= 250;
        }
    }
```

At first, very small amount of points is subtracted from each file according to its relative position. The ones in the middle lose fewer points than the ones that are at the outskirts of the board. Then for each fish in the row and column of a potentially best tile 10 points are added, so as to favour rows and columns with a plenty of fish. Subsequently, 100 points is subtracted from the sum for each penguin that is currently occupying the same row/column as the checked tile. Finally, 250 points are subtracted if the tile is located at the boundary of the board.

```
    high_score = tiles_empty[0][2]; //Setting first element of an array
as a temporary best
    *(pointer_to_best_tile) = tiles_empty[0][0];
    *(pointer_to_best_tile + 1) = tiles_empty[0][1];

    for(i=1; i<spaces_available; i++)
    {
        if(tiles_empty[i][2]> high_score)//Set new best tile and save
its position
        {
            *(pointer_to_best_tile) = tiles_empty[i][0];
            *(pointer_to_best_tile + 1) = tiles_empty[i][1];
            high_score = tiles_empty[i][2];
        }
    }
```

Then scores of all tiles are compared and the one with the highest value is chosen as the one where penguin will be placed and its coordinates are saved. Later, functions unrelated to the algorithm override the board and save the file with a penguin placed in a chosen place.

2. Movement phase automatic

To store the data for the board we use a 2-dimensional array of structures called board (tile_t). This is how we allocate memory for the board.

```
board = (tile_t**)realloc(board, rows * sizeof(tile_t *));
//int row_count = sizeof(*arr);
if (board == NULL)
{
    printf("Memory failed to allocate\n");
    exit(1);
}
for (size_t i=0; i<rows; i++)
{
    board[i] = (tile_t*)calloc(columns, sizeof(tile_t));
    if (board[i]==NULL)
    {
        printf("Memory failed to allocate\n");
        exit(0);
    }
}
```

Then we create an array of structure player_t to store the player information (such as name, number, score, etc). We dynamically allocate memory for it and then read and store the information from the file.

```
int numOfPlayers = num_of_players(inputfile);
player_t *playerList = malloc(sizeof(player_t)*numOfPlayers);
for (int i = 0; i < numOfPlayers; i++)
{
    // playerList[i] = malloc(sizeof(player_t));
    playerList[i].numberOfPenguins = num_of_penguins(7, board, inputfile);
    playerList[i].penguinList = malloc(sizeof(penguin_t)*num_of_penguins(7, board, inputfile));
    if (playerList[i].penguinList == NULL)
    {
        printf("Memory failed to allocate");
        exit(1);
    }
}
read_from_file_automatic(inputfile, board, playerList);
int ourPlayerIndex;
for (int i = 0; i < numOfPlayers; i++)
{
    make_penguins(&playerList[i], board, rows, columns);
    if (playerList[i].playerID == 7) ourPlayerIndex = i;
}
```

Then a random move is generated and applied if it is valid.

```
int randDirection = (rand() % 4) + 1;
int randMove[2];
rand_move_based_direction(randMove, randDirection, board, &playerList[ourPlayerIndex], inputfile);
while(!isValidMove(inputfile, randMove, board, &playerList[ourPlayerIndex], &playerList[ourPlayerIndex].penguinList[randPengNum]))
{
    randDirection = (rand() % 4) + 1;
    rand_move_based_direction(randMove, randDirection, board, &playerList[ourPlayerIndex], inputfile);
}
makeMove(randMove, board, &playerList[ourPlayerIndex], &playerList[ourPlayerIndex].penguinList[randPengNum]);
save_to_file_automatic(board, playerList, inputfile, outputfile);
deallocate_mem(board, columns);
exit(1);
```