His Dark Majesty

It is an age of swords and monsters. The plainsmen face off against the evil horde. Knights, Swordsmen, Pikemen, and Rangers carry cannons into battle against the enemies Footmen, Calvary, Archers, and Shieldmen who bring trebuchets and are joined by Ogres on the ground and Vultures, Eagles, and Griffons in the sky descending from the mountains. It is a dark time for mankind.

His Dark Majesty is a turn-based tactics game inspired by "Advance Wars" and "Reign of Swords" (on the Iphone). HDM can be controlled either by the keyboard or mouse. First you have to choose the unit you want by clicking on it. Then the movement range is highlighted. Choose any yellow highlighted cell to move unit there. If you click on the unit again you will be able to attack if enemy is in range (enemies in range are highlighted in gray). Choose highlighted enemy to attack it. Some units are able to attack after movement. In this case the attack mode is activated automatically after move. To advance a turn click anywhere on the map and answer the question ‘y’.

The code is very simple and fast. If you choose the speed 9 and two computer players the whole battle will take a second or two.

The scenario in the first paragraph is only the default map and unit layout, but you can use the editor to make your own campaigns. You

/* type.h listing begins: */

```
#include "config.h"

#define TERRAIN_PLAINS 0
#define TERRAIN_FOREST 1
#define TERRAIN_WATER  2
#define TERRAIN_WALL   3
#define TERRAIN_MOUNTAIN 4
#define TERRAIN_MAX 5

#define UNIT_TYPE_FOOTMEN     0
#define UNIT_TYPE_SWORDSMEN   1
#define UNIT_TYPE_SHIELDMEN   2
#define UNIT_TYPE_HALBERDIER  3
#define UNIT_TYPE_PIKEMEN     4
#define UNIT_TYPE_ARCHERS     5
#define UNIT_TYPE_CROSSBOWMEN 6
#define UNIT_TYPE_MUSKETEER   7
#define UNIT_TYPE_RANGERS     8
#define UNIT_TYPE_CAVALRY     9
#define UNIT_TYPE_KNIGHTS    10
#define UNIT_TYPE_BALLISTA   11
#define UNIT_TYPE_TREBUCHET  12
#define UNIT_TYPE_CANNON     13
#define UNIT_TYPE_VULTURE    14
#define UNIT_TYPE_EAGLE      15
#define UNIT_TYPE_GRIFFON    16
#define UNIT_TYPE_OGRE       17
#define UNIT_TYPE_MAX        18

#define TERRAIN_MASK 0x1F
#define UNIT_MASK 0xE0

#define BOOL int
#define TRUE 1
#define FALSE 0

typedef unsigned char index_type;
typedef unsigned char unit_type;
typedef unsigned char position;
typedef unsigned char limited_value; // 0-255. Watch out for overflow!
typedef unsigned char special_flag;
enum {
       FLAG_NONE=0x0,
       FLAG_FIRST_STRIKE=0x1,
       FLAG_EXPLOSIVE=0x2,
       FLAG_HEAL=0x4,
       FLAG_BLESS=0x8,
       FLAG_ATTACK_AFTER_MOVE=0x10,
};

struct unit_def {
       unsigned char tile;
       char *name;
       limited_value range_min;
       limited_value range_max;
       limited_value attack;
       limited_value defense;
       limited_value move_cost[TERRAIN_MAX];
       special_flag special;
       //     limited_value price;
       //  - add resources here
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
        //  - add upgrades here
};


struct terrain_def {
        unsigned char tile;
        char *name;
        signed char defense_bonus;
        //  Curses only
        unsigned char color;
};
```

can make any unit you want on either side you want. You may want to change your default to map and unit layout to something more symmetrical. If you do make a campaign you can share it on Cymon’s Games!

His Dark Magesty is written by Jakub Debski.

/* config.h listing begins: */

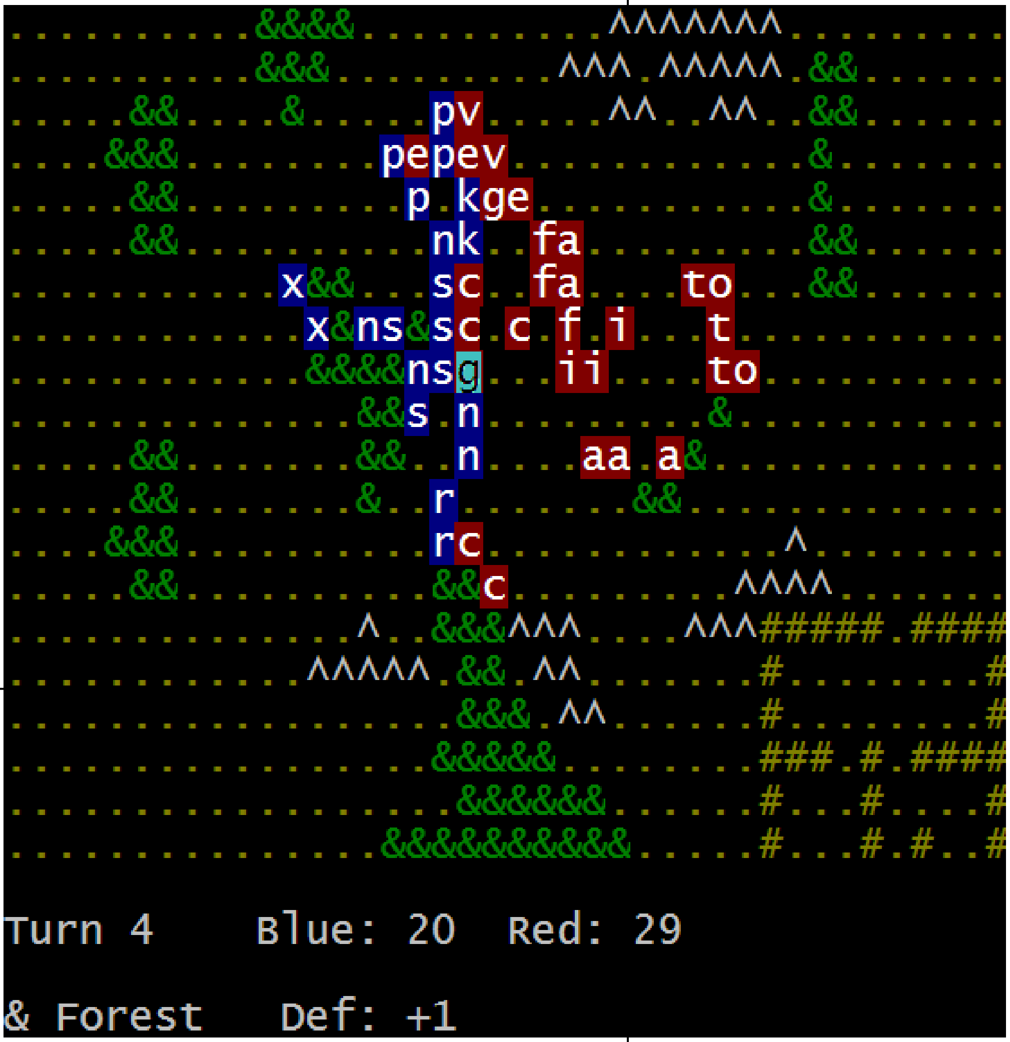
```
#include <curses.h>

#define MAP_SIZE_X 40
#define MAP_SIZE_Y 20

#define UNITS_PER_PLAYER 32
#define MAX_UNITS (UNITS_PER_PLAYER*2)


#define CONTROL_KEYPAD 1
#if CONTROL_KEYPAD==0
#define CONTROL_VI     0
#if CONTROL_VI==0
#define CONTROL_WSAD   0
#endif
#endif

#if CONTROL_KEYPAD
#define KEY_MOVE_UP '8'
#define KEY_MOVE_DOWN '2'
#define KEY_MOVE_LEFT '4'
#define KEY_MOVE_RIGHT '6'
#define KEY_ACTION '0'
#endif
#if CONTROL_WSAD
#define KEY_MOVE_UP 'w'
#define KEY_MOVE_DOWN 's'
#define KEY_MOVE_LEFT 'a'
#define KEY_MOVE_RIGHT 'd'
#define KEY_ACTION ' '
#endif
#if CONTROL_VI
#define KEY_MOVE_UP 'k'
#define KEY_MOVE_DOWN 'j'
#define KEY_MOVE_LEFT 'h'
#define KEY_MOVE_RIGHT 'l'
#define KEY_ACTION ' '
#endif
```



```
..........&&&&..........^^^^^^^.........
..........&&&.........^^^.^^^^^.&&......
.....&&....&.....pv.....^^..^^..&&......
....&&&........pepev............&.......
.....&&.........p.kge...........&.......
.....&&..........nk..fa.........&&......
...........x&&...sc..fa....to...&&......
............x&ns&sc.c.f.i...t...........
............&&&&nsg...ii....to..........
..............&&s.n.........&...........
.....&&.......&&..n....aa.a&............
.....&&.......&..r.......&&.............
....&&&..........rc............^........
.....&&..........&&c.........^^^^.......
..............^..&&&^^^....^^^#####.####
............^^^^^.&&.^^.......#........#
..................&&&.^^......#........#
.................&&&&&........###.#.####
..................&&&&&&......#...#....#
...............&&&&&&&&&&.....#...#.#..#

Turn 4    Blue: 20  Red: 29

& Forest   Def: +1
```

/* editor.c listing begins: */

```
#include "config.h"
#include "types.h"

#define BOOL unsigned char
#define TRUE 1
#define FALSE 0

extern unsigned char terrain
[MAP_SIZE_Y][MAP_SIZE_X];
extern unsigned char terrain_map
[MAP_SIZE_Y][MAP_SIZE_X];
extern unsigned char unit_placement [MAP_SIZE_Y][MAP_SIZE_X];
extern struct terrain_def terrain_prop[TERRAIN_MAX];
extern struct unit_def unit_prop[UNIT_TYPE_MAX];
extern void PlayLevel();
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
extern void TestPlaceUnits();
extern void TestConvertTerrain();
extern position cursor_x,cursor_y;
extern BOOL mouse_active;

unsigned char current_terrain=0;
unsigned char current_unit=0;
BOOL editing_terrain=TRUE;
BOOL editing_blue=TRUE;

int blue_units_left=UNITS_PER_PLAYER;
int red_units_left=UNITS_PER_PLAYER;

void LoadTerrainMap(char *map_name);
void LoadUnitPlacement(char *map_name);

void LoadLevel() {
    FILE *fp;
    unsigned char tile;
    register position x,y;
    fp=fopen("level.ter","rt");
    if (fp==NULL)
        return;
    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            tile=fgetc(fp);
            terrain_map[y][x]=tile;
        }
        tile=fgetc(fp);
        if (tile=='\r')
            tile=fgetc(fp);
    }
    fclose(fp);

    TestConvertTerrain();
    blue_units_left=UNITS_PER_PLAYER;
    red_units_left=UNITS_PER_PLAYER;

    fp=fopen("level.uni","rt");
    if (fp==NULL)
        return;

    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            tile=fgetc(fp);
            unit_placement[y][x]=tile;
            if (tile>='a' && tile<='z')
                --blue_units_left;
            else if (tile>='A' && tile<='Z')
                --red_units_left;
        }
        tile=fgetc(fp);
        if (tile=='\r')
            tile=fgetc(fp);
    }
    fclose(fp);
    editing_terrain=FALSE;
}

void SaveLevel() {
    FILE *fp;
    register position x,y;
    fp=fopen("level.ter","wt+");
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            switch (terrain[y][x]) {
            case TERRAIN_FOREST:
                terrain_map[y][x]='&';
                break;
            case TERRAIN_MOUNTAIN:
                terrain_map[y][x]='^';
                break;
            case TERRAIN_WATER:
                terrain_map[y][x]='~';
                break;
            case TERRAIN_WALL:
                terrain_map[y][x]='#';
                break;
            default:
                terrain_map[y][x]='.';
            }
            fputc(terrain_map[y][x],fp);
        }
        fputc('\n',fp);
    }
    fclose(fp);

    fp=fopen("level.uni","wt+");
    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            fputc(unit_placement[y][x],fp);
        }
        fputc('\n',fp);
    }
    fclose(fp);
}

void ShowEditorMap() {
    unsigned char x,y,tile;
    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            attrset(COLOR_PAIR(terrain_prop[terrain[y][x]].color));
            mvaddch(y,x,terrain_prop[terrain[y][x]].tile);
        }
    }

    if (!editing_terrain) {
        for (y=0;y<MAP_SIZE_Y;++y) {
            for (x=0;x<MAP_SIZE_X;++x) {
                tile=unit_placement[y][x];
                if (tile>='a' && tile<='z') {
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_BLUE*8)|A_BOLD);
                    mvaddch(y,x,tile);
                } else if (tile!='.') {
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_RED*8)|A_BOLD);
                    mvaddch(y,x,tile+0x20);
                }
            }
        }
    }
    refresh();
}

void ShowEditorWindow() {
    unsigned int i;
    clear();
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    ShowEditorMap();
    attrset(COLOR_PAIR(COLOR_WHITE));
    mvaddstr(20,0,"F1-terrain  F2-units  F5-test");
    mvaddstr(21,0,"F8-save     F9-load   F12-clear/fill");
    if (editing_terrain) {
        for (i=0;i<TERRAIN_MAX;++i) {
            if (terrain_prop[i].color!=COLOR_WHITE)
                attrset(COLOR_PAIR(terrain_prop[i].color|COLOR_WHITE*8));
            else
                attrset(COLOR_PAIR(COLOR_BLACK|COLOR_WHITE*8));
            mvprintw(22,i,"%c",terrain_prop[i].tile);
            if (i==current_terrain) {
                attrset(COLOR_PAIR(COLOR_WHITE));
                mvprintw(23,0,"Current:");
                mvprintw(23,11,"%s",terrain_prop[i].name);
                attrset(COLOR_PAIR(terrain_prop[i].color));
                mvprintw(23,9,"%c", terrain_prop[i].tile);
            }
        }
    } else {
        for (i=0;i<UNIT_TYPE_MAX;++i) {
            if (editing_blue)
                attrset(COLOR_PAIR(COLOR_WHITE|COLOR_BLUE*8)|A_BOLD);
            else
                attrset(COLOR_PAIR(COLOR_WHITE|COLOR_RED*8)|A_BOLD);
            mvprintw(22,i,"%c",unit_prop[i].tile);
            if (i==current_unit) {
                mvprintw(23,9,"%c", unit_prop[i].tile);
                attrset(COLOR_PAIR(COLOR_WHITE));
                mvprintw(23,0,"Current:");
                mvprintw(23,11,"%s",unit_prop[i].name);
            }
        }
        if (current_unit==UNIT_TYPE_MAX) {
            attrset(COLOR_PAIR(COLOR_WHITE));
            mvprintw(23,0,"Current: Remove");
        }
        if (editing_blue) {
            attrset(COLOR_PAIR(COLOR_WHITE|COLOR_RED*8)|A_BOLD);
            mvaddch(22,i+1,' ');
        } else {
            attrset(COLOR_PAIR(COLOR_WHITE|COLOR_BLUE*8)|A_BOLD);
            mvaddch(22,i+1,' ');
        }
        attrset(COLOR_PAIR(COLOR_WHITE));
        mvprintw(22,30,"Blue: %d",blue_units_left);
        mvprintw(23,30,"Red:  %d",red_units_left);

    }
    move(cursor_y,cursor_x);
    refresh();
}

void ChooseUnit() {

}

void ClearAfterPlaying() {
    unsigned char x,y;
    for (y=0;y<MAP_SIZE_Y;++y)
        for (x=0;x<MAP_SIZE_X;++x) {
            terrain[y][x]=terrain[y][x]&TERRAIN_MASK;
        }
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
}

void FillMap() {
    unsigned char x,y;
    if (editing_terrain) {
        for (y=0;y<MAP_SIZE_Y;++y)
            for (x=0;x<MAP_SIZE_X;++x) {
                terrain[y][x]=current_terrain;
            }
    } else {
        for (y=0;y<MAP_SIZE_Y;++y)
            for (x=0;x<MAP_SIZE_X;++x) {
                unit_placement[y][x]='.';
            }
        red_units_left=UNITS_PER_PLAYER;
        blue_units_left=UNITS_PER_PLAYER;
    }
    ShowEditorMap();
    ShowEditorWindow();
}

void EditorMoveCursor(position x, position y) {
    cursor_x=x;
    cursor_y=y;
    move(y,x);
    refresh();
}

void EditorAction() {
    unsigned char tile, previous;
    if (editing_terrain) {
        if (cursor_y<MAP_SIZE_Y) {
            terrain[cursor_y][cursor_x]=current_terrain;
            attrset(COLOR_PAIR(terrain_prop[current_terrain].color));
            mvaddch(cursor_y,cursor_x,terrain_prop[current_terrain].tile);
            refresh();
        } else if (cursor_y==22 && cursor_x<TERRAIN_MAX) {
            current_terrain=cursor_x;
            ShowEditorWindow();
        }
    } else {
        if (cursor_y<MAP_SIZE_Y) {
            previous=unit_placement[cursor_y][cursor_x];
            if (previous>='a' && previous<='z')
                ++blue_units_left;
            else if (previous>='A' && previous<='Z')
                ++red_units_left;

            if (current_unit==UNIT_TYPE_MAX) { // removing
                unit_placement[cursor_y][cursor_x]='.';
                ShowEditorMap();
                ShowEditorWindow();
                return;
            } else {
                if (editing_blue) {
                    if (blue_units_left==0)
                        return;
                    else
                        --blue_units_left;
                } else {
                    if (red_units_left==0)
                        return;
                    else
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
                        --red_units_left;
                    }
                }

                tile = unit_prop[current_unit].tile - (editing_blue?0:0x20);
                unit_placement[cursor_y][cursor_x]=tile;
                if (editing_blue)
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_BLUE*8)|A_BOLD);
                else
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_RED*8)|A_BOLD);
                mvaddch(cursor_y,cursor_x,unit_prop[current_unit].tile);
                ShowEditorWindow();
            } else if (cursor_y==22) {
                if (cursor_x<UNIT_TYPE_MAX) {
                    current_unit=cursor_x;
                    ShowEditorWindow();
                } else if (cursor_x==UNIT_TYPE_MAX) {
                    current_unit=UNIT_TYPE_MAX;
                    ShowEditorWindow();
                } else if (cursor_x==UNIT_TYPE_MAX+1) {
                    editing_blue=!editing_blue;
                    ShowEditorWindow();
                }
            }

        }
}

void MapEditor() {
    int key;
    clear();
    ShowEditorWindow();

    editing_terrain=FALSE;
    FillMap();
    editing_terrain=TRUE;
    FillMap();

    LoadLevel();
    ShowEditorMap();
    ShowEditorWindow();

    mouse_on(ALL_MOUSE_EVENTS);
    nodelay(stdscr,TRUE);
    cbreak();

    for (;;) {
        key=getch();
        request_mouse_pos();
        if (mouse_active && (MOUSE_X_POS!=cursor_x || MOUSE_Y_POS!=cursor_y)) {
            curs_set(0);
            EditorMoveCursor(MOUSE_X_POS,MOUSE_Y_POS);
            curs_set(2);
        }
        switch (key) {

        case KEY_F0+1:
            editing_terrain=TRUE;
            ShowEditorWindow();
            break;
        case KEY_F0+2:
            editing_terrain=FALSE;
            ShowEditorWindow();
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            break;
        case KEY_F0+5:
            clear();
            PlayLevel();

            ClearAfterPlaying();
            ShowEditorWindow();

            mouse_on(ALL_MOUSE_EVENTS);
            nodelay(stdscr,TRUE);
            cbreak();
            break;
        case KEY_F0+8:
            SaveLevel();
            break;
        case KEY_F0+9:
            LoadLevel();
            ShowEditorMap();
            ShowEditorWindow();
            break;
        case KEY_F0+12:
            FillMap();
            break;
        case KEY_MOVE_LEFT:
            EditorMoveCursor(cursor_x-1,cursor_y);
            break;
        case KEY_MOVE_RIGHT:
            EditorMoveCursor(cursor_x+1,cursor_y);
            break;
        case KEY_MOVE_UP:
            EditorMoveCursor(cursor_x,cursor_y-1);
            break;
        case KEY_MOVE_DOWN:
            EditorMoveCursor(cursor_x,cursor_y+1);
            break;
        case KEY_MOUSE:
        case KEY_ACTION:
            curs_set(0);
            EditorAction();
            move(cursor_y,cursor_x);
            curs_set(2);
        }
    }
}
```

/* main.c listing begins: */

```
// His Dark Majesty
// PoC strategy game
// Jakub Debski
// v0.5, 2009-06-30
#define GAME_VER "v0.5"

#include "config.h"
#include <time.h>
#include "types.h"

//////////////////////////////////////////////////////////////////////////
// EXTERNS
//////////////////////////////////////////////////////////////////////////

extern void MapEditor();
extern void LoadLevel();
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
#include <assert.h>

#define UNIT_MOVE_RANGE_MAX 5

// Other? - water, desert, road, gate, bridge

unsigned char terrain [MAP_SIZE_Y][MAP_SIZE_X];

// These are not used currently - level is loaded
unsigned char terrain_map [MAP_SIZE_Y][MAP_SIZE_X];
unsigned char unit_placement [MAP_SIZE_Y][MAP_SIZE_X];

///////////////////////////////////////////////////////////////////////////
// AI data
///////////////////////////////////////////////////////////////////////////

// During the evaluation phase
int best_evaluation;
position best_x, best_y;

// level of aggresiveness of AI. 2 is standard. 1 is defensive. 3 is berserk
int ai_aggresive=2;
int game_speed=5;
int turn;

// End of AI data
///////////////////////////////////////////////////////////////////////////

limited_value blue_units_number;
limited_value red_units_number;
BOOL quit_choosen=FALSE;
BOOL blue_human=FALSE;
BOOL red_human=FALSE;

BOOL mouse_active = 1;
BOOL editor_choosen=FALSE;

index_type my_starting_unit;
index_type enemy_starting_unit;

#define UNIT_DEAD 0xFF

struct unit_def unit_prop[UNIT_TYPE_MAX] = {
    {'f', "Footmen", 1, 1,   3,   0, { 3,  5,    10,   10,    99},
FLAG_ATTACK_AFTER_MOVE},
    {'s', "Swordsmen",  1, 1,   4,   1, { 3,  5,    10,   10,    99},
FLAG_ATTACK_AFTER_MOVE},
    {'i', "Shieldmen",  1, 1,   4,   3, { 3,  5,    10,   10,    99}, FLAG_NONE},
    {'h', "Halberdier", 1, 1,   6,   2, { 3,  5,    10,   10,    99},
FLAG_ATTACK_AFTER_MOVE},
    {'p', "Pikemen",    1, 1,   6,   0, { 3,  5,    10,   10,    99},
FLAG_FIRST_STRIKE},
    {'a', "Archers",    2, 3,   4,   0, { 3,  5,    10,   10,    99},
FLAG_FIRST_STRIKE},
    {'n', "Crossbowmen",    2, 4,   5,   0, { 3,  5,    10,   10,    99},
FLAG_NONE},
    {'m', "Musketeer",  2, 5,   6,   0, { 3,  5,    10,   10,    99}, FLAG_NONE},
    {'r', "Rangers",    1, 2,   4,   0, { 3,  3,    5,    10,    10},
        FLAG_ATTACK_AFTER_MOVE|FLAG_FIRST_STRIKE},
    {'c', "Cavalry",    1, 1,   3,   1, { 2,  5,    10,   99,    99},
FLAG_ATTACK_AFTER_MOVE},
    {'k', "Knights",    1, 1,   5,   3, { 2,  6,    99,   99,    99},
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
FLAG_ATTACK_AFTER_MOVE},
    {'b', "Ballista",   3, 4,   6,   0, { 5, 10,    99,   99,    99}, FLAG_NONE},
    {'t', "Trebuchet",  2, 5,   7,   0, { 5, 10,    99,   99,    99},
FLAG_EXPLOSIVE},
    {'x', "Cannon",     2, 6,   8,   0, { 5, 10,    99,   99,    99},
FLAG_EXPLOSIVE},
    {'v', "Vulture",    1, 1,   3,   0, { 3,  3, 3,    3, 3},
FLAG_ATTACK_AFTER_MOVE},
    {'e', "Eagle",  1, 1,   4,   1, { 2,  2, 2,    2, 2},
FLAG_ATTACK_AFTER_MOVE},
    {'g', "Griffon",    1, 2,   7,   3, { 2,  2, 2,    2, 2},
FLAG_ATTACK_AFTER_MOVE},
    {'o', "Ogre",   1, 1,   9,   6, { 5, 10,    99,   10,    99},
FLAG_ATTACK_AFTER_MOVE},
};

struct terrain_def terrain_prop[TERRAIN_MAX] = {
    {'.',"Plains  ",  0, COLOR_YELLOW},
    {'&',"Forest  ",  1, COLOR_GREEN},
    {'~',"Water   ", -1, COLOR_BLUE},
    {'#',"Wall    ",  3, COLOR_YELLOW},
    {'^',"Mountain",  2, COLOR_WHITE}
};

struct unit {
    unit_type type; // 0xFF - dead
    position x,y;
    BOOL moved;
    limited_value hp; // 0-9
    signed char defense_modifier;
    int total_strength;
};

struct unit units[MAX_UNITS];

//////////////////////////////////////////////////////////////////////////
// Selected unit data
// - initialized when the unit is selected by player or by AI
//////////////////////////////////////////////////////////////////////////

BOOL          ai_turn=FALSE;
index_type    selected_unit=UNIT_DEAD;
unit_type     selected_unit_type;
limited_value selected_unit_move_cells_left=0;
struct unit_def *selected_unit_prop;
position      selected_unit_x;
position      selected_unit_y;
limited_value movement_table[MAP_SIZE_Y][MAP_SIZE_X];
struct unit   *selected_unit_ptr;

// Cursor
position cursor_x,cursor_y;

#if defined __NES__ || USE_JOYSTICK
#include <joystick.h>
extern char joy_driver;
#endif

BOOL CoinToss() {
    return rand()&1;
}

void Wait(unsigned char t) {
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    napms(game_speed*20*t);
}

void SetupColors() {
    short i, j;

    for (i = 0; i < 8; i++)  for (j = 0; j < 8; j++) if ((i > 0) || (j > 0))
init_pair(i * 8 + j, j, i);
    init_pair(63, COLOR_BLACK, COLOR_BLACK);
}


void SystemInit() {

#if USE_JOYSTICK
    joy_install(&joy_driver);
#endif
    // Initialization
    initscr();
    if (has_colors())
        start_color();
    noecho();
    keypad(stdscr, TRUE);
    curs_set(2);
    resize_term(24,40);
    SetupColors();
    srand((unsigned int) time(NULL));

    cursor_x=22;
    cursor_y=10;
}

void LevelInit() {
    // Clear units
    register index_type i;
    for (i=0;i<MAX_UNITS;++i) {
        units[i].type=UNIT_DEAD;
    }
    quit_choosen=FALSE;
    my_starting_unit=0;
    blue_units_number=0;
    red_units_number=0;
    turn=1;
}

BOOL OnMap(position new_x, position new_y) {
    if (new_x<MAP_SIZE_X && new_y<MAP_SIZE_Y)
        return TRUE;
    return FALSE;
}

void ShowUnderCursorInfo() {
    struct terrain_def *prop_ptr;

    attrset(COLOR_PAIR(COLOR_WHITE));

    prop_ptr = terrain_prop+(terrain[cursor_y][cursor_x]&TERRAIN_MASK);
    mvprintw(23,0,"%c %s Def: %+d",prop_ptr->tile,prop_ptr->name,prop_ptr-
>defense_bonus);

    if (terrain[cursor_y][cursor_x]&UNIT_MASK) {
        register index_type i;
        for (i=0;i<MAX_UNITS;++i) {
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            if (units[i].type!=UNIT_DEAD && units[i].x==cursor_x && units
[i].y==cursor_y) {
                mvprintw(22,0,"%s      ",unit_prop[units[i].type].name);
                mvprintw(22,11,"%Def: %d%+d Att:%c%d R:%d-%d HP: %d",
                  unit_prop[units[i].type].defense,units[i].defense_modifier,
                  (unit_prop[units[i].type].special&FLAG_FIRST_STRIKE)?'!':' ',
                  unit_prop[units[i].type].attack,unit_prop[units[i].type]
                    .range_min,unit_prop[units[i].type].range_max,units[i].hp);
                break;
            }
        }
    } else
        mvprintw(22,0,"                                        "); //!!! ugly
code warning :)
    refresh();
}

void MoveCursor(position new_x, position new_y) {
    if (OnMap(new_x,new_y)) {
        cursor_x=new_x;
        cursor_y=new_y;
        ShowUnderCursorInfo();
        move(cursor_y,cursor_x);
        refresh();
    }
}

void TurnInit() {
    // Clear units for turn
    register index_type i;
    for (i=0;i<MAX_UNITS;++i) {
        units[i].moved=FALSE;
        units[i].defense_modifier=0;
    }
}

void SelectUnit(index_type choosen) {
    if (!units[choosen].moved) {
        selected_unit=choosen;
        selected_unit_ptr=units+choosen;
        selected_unit_type=selected_unit_ptr->type;
        selected_unit_prop=unit_prop+selected_unit_type;
        selected_unit_x=selected_unit_ptr->x;
        selected_unit_y=selected_unit_ptr->y;
        move(selected_unit_y,selected_unit_x);
    }
}

void ShowMap() {
    unsigned char x,y;
    for (y=0;y<MAP_SIZE_Y;y++) {
        for (x=0;x<MAP_SIZE_X;++x) {
            attrset(COLOR_PAIR(terrain_prop[terrain[y][x]&TERRAIN_MASK].color));
            mvaddch(y,x,terrain_prop[terrain[y][x]&TERRAIN_MASK].tile);
        }
    }
    refresh();
}

void ShowUnits() {
    struct unit *u,*last_unit,*second_player;

    second_player=units+UNITS_PER_PLAYER;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    last_unit=units+MAX_UNITS;

    for (u=units;u<last_unit;++u) {
        if (u->type!=UNIT_DEAD) {
            if (u<second_player) {
                if (u->moved)
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_BLUE*8));
                else
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_BLUE*8)|A_BOLD);
            } else {
                if (u->moved)
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_RED*8));
                else
                    attrset(COLOR_PAIR(COLOR_WHITE+COLOR_RED*8)|A_BOLD);
            }

            mvaddch(u->y,u->x,unit_prop[u->type].tile);
        }
    }
    refresh();
}

void RefreshView() {
    curs_set(0);
    ShowMap();
    ShowUnits();
    curs_set(2);
}

void SelectedUnitMoveTo(position x, position y) {
    terrain[selected_unit_ptr->y][selected_unit_ptr->x]&=TERRAIN_MASK;
    selected_unit_ptr->x=x;
    selected_unit_ptr->y=y;
    selected_unit_x=x;
    selected_unit_y=y;
    terrain[y][x]|=UNIT_MASK;
}

void AddLimitedValueOnDiamondArea(unsigned char *area, position x,position y,
     limited_value range, limited_value value) {
    register limited_value sx, sy;
    register index_type r, i, max;
    // Upper and central part
    sy=y-range;
    max=range+1;
    mvaddch(y,x,'*');
    refresh();
    for (r=0;r<max;++r,++sy) {
        if (sy>=MAP_SIZE_Y)
            continue;
        i=r*2+1;
        while (i-- > 0) {
            sx=x-r+i;
            if (sx<MAP_SIZE_X) {
                mvaddch(sy,sx,'%');
                refresh();
            } else
                break;
        }
    }
    // Lower part
    mvaddch(y,x,'*');
    refresh();
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    while (--r>0) {
        if (sy>=MAP_SIZE_Y)
            break;
        i=r*2;
        while (--i > 0) {
            sx=x-r+i;
            if (sx<MAP_SIZE_X) {
                mvaddch(sy,sx,'%');
                refresh();
            }
        }
        ++sy;
    }
    mvaddch(y,x,'*');
    refresh();
}

BOOL PlaceUnit(position x, position y, unit_type type) {
    register unsigned char i;
    // Find new empty unit
    for (i=my_starting_unit;i<my_starting_unit+UNITS_PER_PLAYER;++i) {
        if (units[i].type==UNIT_DEAD) { // !!!change to pointer?
            terrain[y][x]|=UNIT_MASK;
            units[i].type=type;
            units[i].x=x;
            units[i].y=y;
            units[i].hp=9;
            if (my_starting_unit==0)
                ++blue_units_number;
            else
                ++red_units_number;
            return TRUE;
        }
    }
    return FALSE;
}

void TestConvertTerrain() {
    register position x,y;
    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            switch (terrain_map[y][x]) {
            case '&':
                terrain[y][x]=TERRAIN_FOREST;
                break;
            case '^':
                terrain[y][x]=TERRAIN_MOUNTAIN;
                break;
            case '#':
                terrain[y][x]=TERRAIN_WALL;
                break;
            case '=':
                terrain[y][x]=TERRAIN_WATER;
                break;
            default:
                terrain[y][x]=TERRAIN_PLAINS;
            }
        }
    }
}

void TestPlaceUnits() {
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    register position x,y;
    unsigned char u;
    for (y=0;y<MAP_SIZE_Y;++y) {
        for (x=0;x<MAP_SIZE_X;++x) {
            u=unit_placement[y][x];
            if (unit_placement[y][x]>='a' && unit_placement[y][x]<='z') {
                my_starting_unit=0;
            } else {
                my_starting_unit=UNITS_PER_PLAYER;
                u+=0x20;
            }

            switch (u) {
            case 'a':
                PlaceUnit(x,y,UNIT_TYPE_ARCHERS);
                break;
            case 'b':
                PlaceUnit(x,y,UNIT_TYPE_BALLISTA);
                break;
            case 'c':
                PlaceUnit(x,y,UNIT_TYPE_CAVALRY);
                break;
            case 'e':
                PlaceUnit(x,y,UNIT_TYPE_EAGLE);
                break;
            case 'f':
                PlaceUnit(x,y,UNIT_TYPE_FOOTMEN);
                break;
            case 'g':
                PlaceUnit(x,y,UNIT_TYPE_GRIFFON);
                break;
            case 'h':
                PlaceUnit(x,y,UNIT_TYPE_HALBERDIER);
                break;
            case 'i':
                PlaceUnit(x,y,UNIT_TYPE_SHIELDMEN);
                break;
            case 'k':
                PlaceUnit(x,y,UNIT_TYPE_KNIGHTS);
                break;
            case 'n':
                PlaceUnit(x,y,UNIT_TYPE_CROSSBOWMEN);
                break;
            case 'o':
                PlaceUnit(x,y,UNIT_TYPE_OGRE);
                break;
            case 'p':
                PlaceUnit(x,y,UNIT_TYPE_PIKEMEN);
                break;
            case 'r':
                PlaceUnit(x,y,UNIT_TYPE_RANGERS);
                break;
            case 's':
                PlaceUnit(x,y,UNIT_TYPE_SWORDSMEN);
                break;
            case 't':
                PlaceUnit(x,y,UNIT_TYPE_TREBUCHET);
                break;
            case 'v':
                PlaceUnit(x,y,UNIT_TYPE_VULTURE);
                break;
            case 'x':
                PlaceUnit(x,y,UNIT_TYPE_CANNON);
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
                break;
            }
        }
    }
    my_starting_unit=0;
    enemy_starting_unit=UNITS_PER_PLAYER;
}

limited_value Distance(limited_value x1, limited_value y1, limited_value x2, lim-
ited_value y2) {
    return (abs(((signed)x1 - (signed) x2) + (abs)((signed)y1 - y2)));
}

BOOL InRangeFromPointByAttacker(struct unit *attacker, limited_value x1, lim-
ited_value y1,
    limited_value x2, limited_value y2) {
    limited_value distance;
    distance = Distance(x1,y1,x2,y2);
    if (distance>=unit_prop[attacker->type].range_min && distance<=unit_prop
[attacker->type].range_max)
        return TRUE;
    return FALSE;
}

BOOL EnemyInRange(struct unit *attacker,struct unit *defender) {
    limited_value distance;
    distance = Distance(attacker->x,attacker->y,defender->x,defender->y);
    if (distance>=unit_prop[attacker->type].range_min
        && distance<=unit_prop[attacker->type].range_max)
        return TRUE;
    return FALSE;
}

// Returns UNIT_DEAD if damage kills defender, otherwise damage done
limited_value CalculateDamage(struct unit *attacker, struct unit *defender) {
    signed char defense;
    signed char damage=0;

    damage=(unit_prop[attacker->type].attack * attacker->hp)/9;
    defense= unit_prop[defender->type].defense;
    defense+=
         terrain_prop[terrain[defender->y][defender->x]
&TERRAIN_MASK ].defense_bonus;
    defense*= defender->hp;
    defense/= 9;
    defense+=defender->defense_modifier;

    damage-=defense;
    if (damage<0)
        damage=0;
    else {
        if (defender->hp<=damage)
            return UNIT_DEAD;
    }
    return damage;
}

void CauseDamage(struct unit *enemy, limited_value damage) {
    if (damage==UNIT_DEAD) {
        enemy->type=UNIT_DEAD;
        if (enemy<units+UNITS_PER_PLAYER)
            --blue_units_number;
        else
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            --red_units_number;
        terrain[enemy->y][enemy->x]&=TERRAIN_MASK;
        mvaddch(enemy->y,enemy->x,'+');
        refresh();
        Wait(5);
    } else {
        enemy->hp-=damage;
        --enemy->defense_modifier;
        mvaddch(enemy->y,enemy->x,'0'+enemy->hp);
        refresh();
        Wait(5);
    }
}

// In AI this can be moved to AIAttackWeakestEnemy to skip double damage counting
void SelectedUnitAttackEnemy(struct unit *defender) {
    limited_value damage_attacker, damage_defender;
    BOOL first_strike_attacker=FALSE;
    BOOL first_strike_defender=FALSE;

    // Check for the first strike
    if (selected_unit_prop->special&FLAG_FIRST_STRIKE)
        first_strike_attacker=TRUE;
    if (unit_prop[defender->type].special&FLAG_FIRST_STRIKE) {
        if (first_strike_attacker)
            first_strike_attacker=FALSE;
        else
            first_strike_defender=TRUE;
    }

    if (first_strike_attacker) { // only attacker
        damage_attacker=CalculateDamage(selected_unit_ptr,defender);
        CauseDamage(defender,damage_attacker);
        if (damage_attacker!=UNIT_DEAD && EnemyInRange
(defender,selected_unit_ptr)) {
            damage_defender=CalculateDamage(defender,selected_unit_ptr);
            CauseDamage(selected_unit_ptr,damage_defender);
        }
    } else if (first_strike_defender) { // only defender
        // if defender is in range, then it attacks first
        damage_defender=0;
        if (EnemyInRange(defender,selected_unit_ptr)) {
            damage_defender=CalculateDamage(defender,selected_unit_ptr);
            CauseDamage(selected_unit_ptr,damage_defender);
        }
        if (damage_defender!=UNIT_DEAD) {
            damage_attacker=CalculateDamage(selected_unit_ptr,defender);
            CauseDamage(defender,damage_attacker);
        }
    } else { // both or none
        damage_attacker=CalculateDamage(selected_unit_ptr,defender);
        damage_defender=CalculateDamage(defender,selected_unit_ptr);
        CauseDamage(defender,damage_attacker);
        if (EnemyInRange(defender,selected_unit_ptr))
            CauseDamage(selected_unit_ptr,damage_defender);
    }
}

void AIAttackWeakestEnemy() {
    index_type i, weakest;
    limited_value damage, max_damage;
    weakest=UNIT_DEAD;
    max_damage = 0;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    for (i=enemy_starting_unit;i<enemy_starting_unit+UNITS_PER_PLAYER;++i) {
        if (units[i].type!=UNIT_DEAD) {
            if (EnemyInRange(selected_unit_ptr,units+i)) {
                damage = CalculateDamage(selected_unit_ptr,units+i);
                if (damage>max_damage || (damage==max_damage && CoinToss())) {
                    weakest=i;
                    if (damage==UNIT_DEAD)
                        break;
                    max_damage=damage;
                }
            }
        }
    }
    if (weakest!=UNIT_DEAD)
        SelectedUnitAttackEnemy(units+weakest);
}

limited_value AIEvaluateUnit(struct unit *to_eval) {
    limited_value eval;
    eval=((unit_prop[to_eval->type].attack*2+unit_prop[to_eval->type].defense) *
to_eval->hp)/9;
    return eval;
}

void AIEvaluatePosition(position x, position y) {
    struct unit *u;
    struct unit *last_unit; // in fact after last unit
    int strength;
    int evaluation=0;

    // We cannot move to a position occupied by another unit
    if ((terrain[y][x]&UNIT_MASK) && !(x==selected_unit_x && y==selected_unit_y))
{
        evaluation=-32000;
        return;
    }

    u =units+enemy_starting_unit;
    last_unit=units+enemy_starting_unit+UNITS_PER_PLAYER;

    for (;u<last_unit;++u) {
        if (u->type!=UNIT_DEAD) {
            // Compare strength

            // When enemy is weeker, then the cell is positive.
            // otherwise negative
            strength = abs(AIEvaluateUnit(selected_unit_ptr) - AIEvaluateUnit(u))
                * (60-Distance(x,y, u->x,u->y));

            if (u->total_strength <= selected_unit_ptr-
>total_strength*ai_aggresive)
                evaluation += strength;
            else
                evaluation -= strength;

            // if in range to attack this unit, than it's positive
            if (InRangeFromPointByAttacker(selected_unit_ptr,u->x,u->y,x,y)) {
                evaluation+=500;
                // If catapult, that doesn't attack after move, then stay here
                if (x==selected_unit_x && y==selected_unit_y
                    &&  !(selected_unit_prop->special&FLAG_ATTACK_AFTER_MOVE))
                    evaluation+=2500;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            }
            // If can be attacked here, then negative
            if (InRangeFromPointByAttacker(u,x,y,u->x,u->y)) {
                evaluation-=20;
            }
        }
    }

    // add terrain modifier
    evaluation+=terrain_prop[terrain[y][x]&TERRAIN_MASK].defense_bonus*10;

    // If evaluation is bad for him, then look for some friends

    if (evaluation<0) {
        u =units+my_starting_unit;
        last_unit=units+my_starting_unit+UNITS_PER_PLAYER;

        for (;u<last_unit;++u) {
            if (u->type!=UNIT_DEAD) {
                // The closer to friend the better it is
                evaluation += AIEvaluateUnit(u) * (60-Distance(x,y, u->x,u->y));
            }
        }
    }

    if (evaluation>best_evaluation || (evaluation==best_evaluation && CoinToss
())) {
        best_evaluation=evaluation;
        best_x=x;
        best_y=y;
    }
}


void MovementDrawChangeCellValue(position x, position y, limited_value
move_points) {
    limited_value cost, tested;
    struct unit *u,*last_unit; // in fact after last unit

    if (x>=MAP_SIZE_X || y>=MAP_SIZE_Y)
        return;

    cost = selected_unit_prop->move_cost[terrain[y][x]&TERRAIN_MASK];
    // Enemy units block movement
    if (terrain[y][x]&UNIT_MASK) {
        u =units+enemy_starting_unit;
        last_unit=units+enemy_starting_unit+UNITS_PER_PLAYER;
        for (;u<last_unit;++u) {
            if (u->type!=UNIT_DEAD && u->x==x && u->y==y) {
                cost=99;
                break;
            }
        }
    }

    if (cost<=move_points) {
        tested = movement_table[y][x];
        if (tested<move_points-cost) {
            movement_table[y][x]=move_points-cost;
            // Show possible moves
            if (!(terrain[y][x]&UNIT_MASK)) {
                attrset(COLOR_PAIR(terrain_prop[terrain[y][x]
&TERRAIN_MASK].color+49));
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
                mvaddch(y,x,terrain_prop[terrain[y][x]&TERRAIN_MASK].tile);
            }
            if (ai_turn)
                AIEvaluatePosition(x,y);
        }
    }
}

// Merge these two functions into MovementDraw?
void MovementDrawCellProcess(position x, position y, limited_value ax, lim-
ited_value ay) {
    limited_value cell_value=movement_table[y][x];

    if (cell_value>1) { // This cell is reachable (!0 and has some free points to
move).
        MovementDrawChangeCellValue(x+ax,y,cell_value);
        MovementDrawChangeCellValue(x,y+ay,cell_value);
    }
}

void MovementDrawFirstCellProcess(position x, position y, limited_value ax, lim-
ited_value ay, limited_value sx, limited_value sy) {
    limited_value cell_value=movement_table[y][x];

    if (cell_value>1) { // This cell is reachable (!0 and has some free points to
move).
        MovementDrawChangeCellValue(x+ax,y,cell_value);
        MovementDrawChangeCellValue(x,y+ay,cell_value);
        // Plus the third one
        MovementDrawChangeCellValue(x+sx,y+sy,cell_value);
    }
}

void MovementDraw(position x, position y, limited_value dx, limited_value dy,
limited_value ax, limited_value ay, limited_value sx, limited_value sy, lim-
ited_value distance) {
    register unsigned char i;

    if (x<MAP_SIZE_X && y<MAP_SIZE_Y)
        MovementDrawFirstCellProcess(x,y,ax,ay,sx,sy);

    for (i=1;i<distance;++i) {
        x+=dx;
        y+=dy;
        if (x>=MAP_SIZE_X || y>=MAP_SIZE_Y)
            continue;
        MovementDrawCellProcess(x,y,ax,ay);
    }
}

// If this is the AI turn, then find the best move at the same time
void SelectedUnitPreprocessMovement() {
    limited_value unit_move_points=11;
    register limited_value i;

    // global_movement should be changed to stored local move table
    memset(movement_table,0,sizeof(movement_table));

    MovementDrawChangeCellValue
(selected_unit_x,selected_unit_y,unit_move_points);
    movement_table[selected_unit_y][selected_unit_x]=unit_move_points;

    MovementDrawChangeCellValue(selected_unit_x-
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
1,selected_unit_y,unit_move_points);
    MovementDrawChangeCellValue
(selected_unit_x+1,selected_unit_y,unit_move_points);
    MovementDrawChangeCellValue(selected_unit_x,selected_unit_y-
1,unit_move_points);
    MovementDrawChangeCellValue
(selected_unit_x,selected_unit_y+1,unit_move_points);

    for (i=1;i<UNIT_MOVE_RANGE_MAX;++i) {
        MovementDraw(selected_unit_x  ,selected_unit_y+i,  1,-1,  1, 1, -1, 0,
i);
        MovementDraw(selected_unit_x+i,selected_unit_y,   -1,-1,  1,-1,  0, 1,
i);
        MovementDraw(selected_unit_x  ,selected_unit_y-i, -1, 1, -1,-1,  1, 0,
i);
        MovementDraw(selected_unit_x-i,selected_unit_y,    1, 1, -1, 1,  0,-1,
i);
    }

    // Show possible moves
    move(selected_unit_y,selected_unit_x);
    refresh();
    Wait(5);
}

void AIUnit(index_type choosen_unit) {
    // Escape stops AI move
    if (getch()==27)
        quit_choosen=TRUE;

    // Use precalculated strengths, precalculated moves (?)
    SelectUnit(choosen_unit);

    // Is under attack, when the distance to the enemy is range[min,max]
    best_evaluation=-32000;

    SelectedUnitPreprocessMovement();

    // Now we have the best cell choosen
    if (best_evaluation!=-32000) {
        if (!(selected_unit_x==best_x && selected_unit_y==best_y))
            selected_unit_ptr->moved=TRUE;
        SelectedUnitMoveTo(best_x,best_y);
    }

    RefreshView();
    move(selected_unit_y,selected_unit_x);
    refresh();
    Wait(5);

    // Attack the weakest unit in range
    if (!selected_unit_ptr->moved || selected_unit_prop-
>special&FLAG_ATTACK_AFTER_MOVE)
        AIAttackWeakestEnemy();
}

void CalculateStrengths(index_type starting_unit) {
    limited_value distance;
    struct unit *unitA_ptr, *unitB_ptr;
    struct unit *last_unit; // in fact after last unit
    limited_value strength;

    last_unit=units+starting_unit+UNITS_PER_PLAYER;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
    for (unitA_ptr=units+starting_unit;unitA_ptr<last_unit;++unitA_ptr) {
        for (unitB_ptr=unitA_ptr+1;unitB_ptr<last_unit;++unitB_ptr) {
            if (unitA_ptr->type!=UNIT_DEAD && unitB_ptr->type!=UNIT_DEAD) {
                distance = Distance(unitA_ptr->x,unitA_ptr->y, unitB_ptr-
>x,unitB_ptr->y);
                strength = unit_prop[unitB_ptr->type].attack*2 + unit_prop
[unitB_ptr->type].defense;
                strength *= unitB_ptr->hp;
                strength /= 9; // max.hp
                if (strength>distance) {
                    assert(unitA_ptr->total_strength+strength>unitA_ptr-
>total_strength);
                    unitA_ptr->total_strength+=strength-distance;
                }
                strength = unit_prop[unitA_ptr->type].attack*2 + unit_prop
[unitA_ptr->type].defense;
                strength *= unitA_ptr->hp;
                strength /= 9; // max.hp
                if (strength>distance) {
                    assert(unitB_ptr->total_strength+strength>unitB_ptr-
>total_strength);
                    unitB_ptr->total_strength+=strength-distance;
                }
            }
        }
    }
}

// Calculates the group strength of friends and enemies
// This is needed to decide is it worth to get closer to a particular enemy (it
can be a part of the group!)
void AIPreprocess() {
    // Init AI
    register index_type i;
    for (i=0;i<MAX_UNITS;++i) {
        if (units[i].type!=UNIT_DEAD) {
            units[i].total_strength=unit_prop[units[i].type].attack;
            SelectUnit(i);
        }
    }

    // Calculate
    CalculateStrengths(my_starting_unit);
    CalculateStrengths(enemy_starting_unit);
}

void AIMain() {

    index_type i;

    // Some general AI preprocess
    AIPreprocess();

    // AI for each unit

    // 1. Start with ranged units
    for (i=my_starting_unit;i<my_starting_unit+UNITS_PER_PLAYER;++i) {
        if (units[i].type!=UNIT_DEAD) {
            if (unit_prop[units[i].type].range_max>1) {
                AIUnit(i);
                if (quit_choosen)
                    return;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            }
        }
    }
    // Then normal units
    for (i=my_starting_unit;i<my_starting_unit+UNITS_PER_PLAYER;++i) {
        if (units[i].type!=UNIT_DEAD) {
            if (unit_prop[units[i].type].range_max<=1) {
                AIUnit(i);
                if (quit_choosen)
                    return;
            }
        }
    }
}

BOOL ChooseEndTurn() {
    attrset(COLOR_PAIR(COLOR_WHITE)|A_BOLD);
    mvaddstr(0,0,"End turn? (y/n)");
    nodelay(stdscr,FALSE);
    if (getch()!='y') {
        RefreshView();
        move(cursor_y,cursor_x);
        refresh();
        nodelay(stdscr,TRUE);
        return FALSE;
    }
    nodelay(stdscr,TRUE);
    return TRUE;
}

BOOL PlayerChooseAttackTarget() {
    BOOL target_available;
    unsigned int key;
    struct unit *u,*last_unit; // in fact after last unit
    target_available=FALSE;
    u =units+enemy_starting_unit;
    last_unit=units+enemy_starting_unit+UNITS_PER_PLAYER;
    for (;u<last_unit;++u) {
        if (u->type!=UNIT_DEAD) {
            if (EnemyInRange(selected_unit_ptr,u)) {
                attrset(COLOR_PAIR(COLOR_WHITE+COLOR_RED*8)|A_BOLD|A_REVERSE);
                mvaddch(u->y,u->x,unit_prop[u->type].tile);
                MoveCursor(u->x,u->y);
                target_available=TRUE;
            }
        }
    }
    if (target_available) {
        move(cursor_y,cursor_x);
        refresh();
        for (;;) {
            key=getch();
            request_mouse_pos();
            if (mouse_active && (MOUSE_X_POS!=cursor_x || MOUSE_Y_POS!=cursor_y))
{
                curs_set(0);
                MoveCursor(MOUSE_X_POS,MOUSE_Y_POS);
                curs_set(2);
            }
            switch (key) {
            case KEY_MOVE_LEFT:
                MoveCursor(cursor_x-1,cursor_y);
                break;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            case KEY_MOVE_RIGHT:
                MoveCursor(cursor_x+1,cursor_y);
                break;
            case KEY_MOVE_UP:
                MoveCursor(cursor_x,cursor_y-1);
                break;
            case KEY_MOVE_DOWN:
                MoveCursor(cursor_x,cursor_y+1);
                break;
            case KEY_MOUSE:
            case KEY_ACTION:
                u=units+enemy_starting_unit;
                last_unit=units+enemy_starting_unit+UNITS_PER_PLAYER;
                for (;u<last_unit;++u) {
                    if (u->type!=UNIT_DEAD && cursor_x==u->x && cursor_y==u->y
                        && EnemyInRange(selected_unit_ptr,u)) {
                        SelectedUnitAttackEnemy(u);
                        selected_unit_ptr->moved=TRUE;
                        return TRUE;
                    }
                }
                return FALSE;
            }
        }
    }
    return TRUE;
}

BOOL PlayerContextHit() {
    // 1. If empty and possible to move, then move
    if ((terrain[cursor_y][cursor_x]&UNIT_MASK)==0 && movement_table[cursor_y]
[cursor_x]>0) {
        SelectedUnitMoveTo(cursor_x,cursor_y);
        RefreshView();
        selected_unit_ptr->moved=TRUE;
        if (selected_unit_prop->special&FLAG_ATTACK_AFTER_MOVE)
            PlayerChooseAttackTarget();
        return TRUE;
    } else {
        RefreshView();
        PlayerChooseAttackTarget();
    }
    return TRUE;
}

void PlayerUnitSelectedAction() {
    unsigned int key;

    if (selected_unit_ptr->moved)
        return;

    SelectedUnitPreprocessMovement();
//      SelectedUnitShowReachableTargets();
    move(selected_unit_y,selected_unit_x);
    refresh();

    for (;;) {
        key=getch();
        request_mouse_pos();
        if (mouse_active && (MOUSE_X_POS!=cursor_x || MOUSE_Y_POS!=cursor_y)) {
            curs_set(0);
            MoveCursor(MOUSE_X_POS,MOUSE_Y_POS);
            curs_set(2);
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
        }
        switch (key) {
        case KEY_MOVE_LEFT:
            MoveCursor(cursor_x-1,cursor_y);
            break;
        case KEY_MOVE_RIGHT:
            MoveCursor(cursor_x+1,cursor_y);
            break;
        case KEY_MOVE_UP:
            MoveCursor(cursor_x,cursor_y-1);
            break;
        case KEY_MOVE_DOWN:
            MoveCursor(cursor_x,cursor_y+1);
            break;
        case KEY_MOUSE:
        case KEY_ACTION:
            if (PlayerContextHit()) {
                RefreshView();
                move(cursor_y,cursor_x);
                refresh();
                return;
            }
            break;
        }
    }
}

// End turn if TRUE
BOOL PlayerAction() {
    selected_unit=UNIT_DEAD;
    if (terrain[cursor_y][cursor_x]&UNIT_MASK) {
        index_type i;
        // 1. Start with ranged units
        for (i=my_starting_unit;i<my_starting_unit+UNITS_PER_PLAYER;++i) {
            if (units[i].type!=UNIT_DEAD) {
                if (units[i].x==cursor_x && units[i].y==cursor_y) {
                    SelectUnit(i);
                    PlayerUnitSelectedAction();
                }
            }
        }
    } else
        return ChooseEndTurn();
    return FALSE;
}

void GetPlayerAction() {
    unsigned int key;
    for (;;) {
#if USE_JOYSTICK
        unsigned char joy;
        BOOL was_fire=FALSE;

        while (JOY_BTN_FIRE(joy_read(JOY_1)));
again:
        key=0;
        // Wait for fire release

        joy = joy_read(JOY_1);
        if (JOY_BTN_FIRE(joy)) {
            was_fire=TRUE;
            if (JOY_BTN_RIGHT(joy)) {
                key=KEY_SHOW_INVENTORY;
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            }
            if (key==0)
                goto again;
        } else if (was_fire) {
            // simple fire - context action
            if (cell_in_air>=CELL_ITEM)
                key=KEY_GET_ITEM;
        } else if (JOY_BTN_LEFT(joy)) {
            key=KEY_MOVE_LEFT;
        } else if (JOY_BTN_RIGHT(joy)) {
            key=KEY_MOVE_RIGHT;
        } else if (JOY_BTN_UP(joy)) {
            key=KEY_MOVE_UP;
        } else if (JOY_BTN_DOWN(joy)) {
            key=KEY_MOVE_DOWN;
        } else
            goto again;

#else
        key=getch();
#endif
        request_mouse_pos();
        if (mouse_active && (MOUSE_X_POS!=cursor_x || MOUSE_Y_POS!=cursor_y)) {
            curs_set(0);
            MoveCursor(MOUSE_X_POS,MOUSE_Y_POS);
            curs_set(2);
        }

        switch (key) {
        case KEY_MOVE_LEFT:
            MoveCursor(cursor_x-1,cursor_y);
            break;
        case KEY_MOVE_RIGHT:
            MoveCursor(cursor_x+1,cursor_y);
            break;
        case KEY_MOVE_UP:
            MoveCursor(cursor_x,cursor_y-1);
            break;
        case KEY_MOVE_DOWN:
            MoveCursor(cursor_x,cursor_y+1);
            break;
        case KEY_MOUSE:
        case KEY_ACTION:
            if (PlayerAction())
                return;
            break;
        case 27: // escape
            quit_choosen=TRUE;
            return;
        }
    }
}

void SwitchPlayers() {
    if (my_starting_unit==0) {
        my_starting_unit=UNITS_PER_PLAYER;
        enemy_starting_unit=0;
    } else {
        my_starting_unit=0;
        enemy_starting_unit=UNITS_PER_PLAYER;
    }
}
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
void PlayerTurn() {
    // The player's turn init is during the AI turn. To speed up the player's
turn start.
    ai_turn=FALSE;
    GetPlayerAction();
}

void AITurn() {
    // 1. Evaluate position
    // 2.
    ai_turn=TRUE;
    AIMain();
}

void DiamondDraw(position x, position y, limited_value dx, limited_value dy, lim-
ited_value distance) {
    register unsigned char i;
    for (i=0;i<distance;++i,x+=dx,y+=dy) {
        if (x>=MAP_SIZE_X || y>=MAP_SIZE_Y)
            continue;
        mvaddch(y,x,'*');
        refresh();
    }
}

void DiamondFill(position x, position y, limited_value range) {
    register unsigned char i;
    mvaddch(y,x,'*');
    refresh();
    for (i=1;i<range;++i) {
        DiamondDraw(x,y+i,1,-1,i);
        DiamondDraw(x+i,y,-1,-1,i);
        DiamondDraw(x,y-i,-1,1,i);
        DiamondDraw(x-i,y,1,1,i);
    }
}

void ShowIntroduction() {
    attrset(COLOR_PAIR(COLOR_WHITE));
    clear();
    mvaddstr(0,11,"'His Dark Majesty'");
    mvaddstr(1,18,GAME_VER);
    mvaddstr(2,5,"Strategy game by Jakub Debski");

    mvaddstr(11,0,"- Increase the console font size!");
    mvaddstr(12,0,"- Hit Alt-Enter for Full-screen.");
#if CONTROL_VI
    mvaddstr(13,0,"- Control with HJKL and Space as fire.");
#elif CONTROL_KEYPAD
    mvaddstr(13,0,"- Control with 2,4,6,8 and 0 as fire.");
#else
    mvaddstr(13,0,"- Control with WSAD and Space as fire.");
#endif
    mvaddstr(14,0,"- Choose an unit. ");
    mvaddstr(15,0,"  Choose a place to move it or");
    mvaddstr(16,0,"  choose it again to attack.");
    mvaddstr(17,0,"- Choose empty cell to end your turn.");
}

BOOL CheckEndingConditions() {
    if (red_units_number==0 || blue_units_number==0 || quit_choosen) {
        clear();
        if (red_units_number==0) {
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
            attrset(COLOR_PAIR(COLOR_WHITE+COLOR_BLUE*8)|A_BOLD);
            mvaddstr(3,0,"Blue side wins!");
        } else if (blue_units_number==0) {
            attrset(COLOR_PAIR(COLOR_WHITE+COLOR_RED*8)|A_BOLD);
            mvaddstr(3,0,"Red side wins!");
        }
        mvaddstr(0,10,"'His Dark Majesty'");
        mvaddstr(6,0,"Press Space...");

        refresh();
        while (getch()!=' ');
        return TRUE;
    }
    return FALSE;
}

void ChooseSides() {
    unsigned char key;

    nodelay(stdscr,FALSE);
    attrset(COLOR_PAIR(COLOR_WHITE));

    mvaddstr(7,0,"Blue side as (h)uman or (c)omputer?");
    refresh();
    if (getch()=='h')
        blue_human=TRUE;
    else
        blue_human=FALSE;
    mvaddstr(8,0,"Red side as (h)uman or (c)omputer?");
    refresh();
    if (getch()=='h')
        red_human=TRUE;
    else
        red_human=FALSE;

    mvaddstr(9,0,"Game speed (1-9)? [5 default]");
    refresh();
    key=getch();
    if (key>='1' && key<='9')
        game_speed='9'-key;
    else
        game_speed=4;
}

void PlayLevel() {
    ChooseSides();

    if (mouse_active) {
        mouse_on(BUTTON1_CLICKED);
        nodelay(stdscr,TRUE);
        cbreak();
    } else
        mouse_off(ALL_MOUSE_EVENTS);

    LevelInit();
    TestPlaceUnits();
    for (;;) {
        // First player
        attrset(COLOR_PAIR(COLOR_WHITE));
        mvprintw(21,0, "Turn %d    Blue: %d  Red: %d  ",
            turn, blue_units_number, red_units_number);

        TurnInit();
```

/* Listing continued on next page…*/

/* Listing continued from previous page */

```
        RefreshView();
        MoveCursor(cursor_x,cursor_y);
        if (blue_human)
            PlayerTurn();
        else
            AITurn();

        if (CheckEndingConditions())
            break;

        SwitchPlayers();
        // Second player
        TurnInit();
        RefreshView();
        MoveCursor(cursor_x,cursor_y);

        if (red_human)
            PlayerTurn();
        else
            AITurn();

        if (CheckEndingConditions())
            break;

        SwitchPlayers();
        ++turn;
    }
}

void main() {
    SystemInit();

    ShowIntroduction();
    for (;;) {
        mouse_off(ALL_MOUSE_EVENTS);
        nodelay(stdscr,FALSE);
        mvaddstr(4,0,"(M)ouse or (K)eyboard control?");
        refresh();
        if (getch()=='k') {
            mouse_active=0;
        } else {
            mouse_active=1;
        }
        mvaddstr(5,0,"Play (G)ame or open (E)ditor?");
        refresh();
        if (getch()=='e') {
            if (mouse_active)
                editor_choosen=TRUE;
            else {
                mvaddstr(5,0,"Editor is available only with mouse!");
                refresh();
            }
        }

        if (editor_choosen)
            MapEditor();
        else {
            LoadLevel();
            TestConvertTerrain();
            PlayLevel();
        }
        ShowIntroduction();
    }
}
```

level.ter	level.uni
..........&&&&..........^^^^^^^.........	.......kp................EEGG...........
..........&&&.........^^^.^^^^^.&&......	.......kp................EEE............
.....&&....&............^^..^^..&&......	.......kp.................VV............
....&&&.........................&.......	.......kp...............................
.....&&.........................&.......	....
.....&&.........................&&......	..............................FA........
............&&..................&&......	.......ns.....................FA........
.............&&&&.......................	......xns.....................FCTO......
............&&&&&.......................	.......ns.....................ICT.......
..............&&............&...........	......xns.....................ICT.......
.....&&.......&&...........&............	.......ns.....................IC.O......
.....&&.......&..........&&.............	..............................AC........
....&&&..........&.............^........	..............................AC........
.....&&..........&&&.........^^^^.......	..............................A.........
..............^..&&&^^^....^^^#####.####	........r...............................
............^^^^^.&&.^^.......#........#	........r...............................
..................&&&.^^......#........#	....
.................&&&&&........###.#.####	....
..................&&&&&&......#...#....#	....
...............&&&&&&&&&&.....#...#.#..#	....