Horde

COLLABORATORS

	TITLE : Horde		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Jordi Fita	February 6, 2018	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
a76b968f33d5	2011-07-31	Removed the stupid quote from the horde. Also, the downloads are now images not text-only links.	jfita
c215080f6b3e	2011-04-08	Fixed the use of constant LEVEL_HEIGHT instead of LEVEL_WIDTH in horde.	jfita
3da08916ffa6	2011-04-07	Fixed some typos in horde.txt	jfita
34b7522b4f97	2011-03-28	atangle is now using a new style for directives which don't collide with XML tags. I had to update all games and programs as well in order to use the new directive syntax.	jfita
6cc909c0b61d	2011-03-07	Added the comments section.	jfita
3712b96ce21b	2010-11-03	Added the date for horde.	jfita
01b9406bff84	2010-11-03	Added the mwindows flag to compile the windows version of horde.	jfita
962962656f7a	2010-11-03	Added a the download section to horde.	jfita
8de4e4798876	2010-11-03	Revised the text for horde.	jfita
786988e0b55a	2010-11-03	Added an screenshot of horde.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
000c08edcde4	2010-11-03	Change the use of non-existent true value to the actual TRUE macro in horde.	jfita
dd4bba608e03	2010-11-02	Fixed some typos in accessing the levels where I was using the y coordinate twice.	jfita
86195e167734	2010-11-02	Added the hordlings section to horde.	jfita
b23cac5b645f	2010-11-02	Added a missing break to the south-east direction.	jfita
5e69d0bab95d	2010-11-01	Added the turns, gold, player movement and curses initialization sections to horde.	jfita
ff08c9ac0ea4	2010-10-31	Added the player’s section to horde. Still missing moving and placing fences.	jfita
dbbb056c57ea	2010-10-31	Added the ’level’ section to horde.	jfita
9e03c07f4ce9	2010-10-30	Added the new ’horde’ game. For now only the introduction, Makefile, and license sections are written.	jfita

Contents

List of Figures

1 Introduction

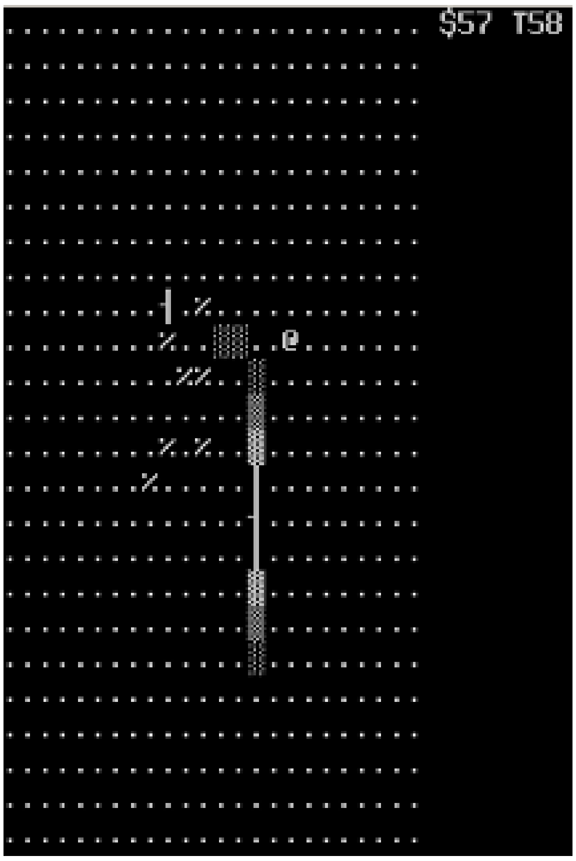

Figure 1: Screenshot of horde

The Horde is an hybrid action-strategy game developed by Toys For Bob in 1994. In The Horde you play as Chauncey, a serving lad who saves his monarch, King of Franzpowanki, from choking on his meal. The King rewards you with his magical monster slayer sword, the Grimthwacker, and a plot of land known as the Shimto Plains. Unfortunately, these particular lands are constantly under attack by “The Horde”, a group of destructive and hungry monsters, called *hordlings*, who are bent on eating everything in the province.

This remake of The Horde is a simpler ASCII-based turn-based version written in C and using the curses library in which the goal is to reach $2000 in gold to pay the taxes while keeping The Horde away from the cattle with the help of the Grimthwacker. At each season’s beginning you receive $10 in gold per cow still alive but it is possible to buy additional cows for $20 the head. Also, to slow down the hungry wave of hordlings, is possible to build, and then improve, fences for $1 apiece. Some of the more expensive fences can even kill hordlings.

Download **Source Code**

2 The Level

Much like chess, the game’s level is a board in which every cell can contain a cow, a piece of fence, a hordling, the player, or be empty. For no particular reason other than to look good on screen, I’ve chosen the level to be 24 cells wide and 24 cells tall.

```
<<constants>>=
#define LEVEL_WIDTH 24
#define LEVEL_HEIGHT 24
```

Given the nature of the curses library, every game's entity — cows, fences, hordlings, and the player — is represented as a single character on screen. Thus, the level can be defined as a multidimensional array of `chtype`, the character type used by curses; generally an integer.

```
<<main variables>>=
chtype level[LEVEL_HEIGHT][LEVEL_WIDTH];
```

Contrary to most representations of 2D coordinates, the first index of `level` is the Y coordinate and the second index is X. I've done this to follow the same convention used in curses, which asks for the Y coordinate first and then X, but more importantly because I am usually going to traverse the array row by row, as for example when drawing the level to the screen.

```
<<main variables>>=
int y;
```

```
<<draw level>>=
for (y = 0 ; y < LEVEL_HEIGHT ; ++y) {
    mvaddchnstr(y, 0, level[y], LEVEL_WIDTH);
}
```

Laying the array row by row, then, makes better use of the cache because this way I access data linearly, although for the relatively small size of the level wouldn't make a noticeable difference.

Before I can draw the level, though, I need to initialize it. I am going to start of with an empty level by setting each level's cell to the empty character, which I've chosen to be a dot (`.`), to follow the conventions of NetHack, a well known ASCII-based game.

```
<<constants>>=
#define EMPTY_CHAR '.'
```

In the case of `level`, I can't use standard C functions such as `memset` to initialize the level because this function accesses the array byte by byte, and `level` is an integer array. Thus, I need to use `for` loops.

```
<<main variables>>=
int x;
```

```
<<initialize level>>=
for (y = 0 ; y < LEVEL_HEIGHT ; ++y) {
    for (x = 0 ; x < LEVEL_WIDTH ; ++x) {
        level[y][x] = EMPTY_CHAR;
    }
}
```

With the level emtpy, I'll then place some cows. The cows are randomly placed within the center of the level, between row 8 and 13 and between column 8 and 13, to keep them away from the hordlings, that comes from the borders. I am going to place a random number of cows between 1 and 10 solely by setting the cow character, which I've chosen to be `%`, on the level without checking if the destination cell if empty. If there was already a cow in the same cell, this will replace the already existing cow.

```
<<constants>>=
#define COW_CHAR '%'
```

```
<<main variables>>=
int cow;
```

```
<<initialize level>>=
for (cow = 0 ; cow < 10 ; ++cow) {
    level[rand() % 6 + 8][rand() % 6 + 8] = COW_CHAR;
}
```

That means that is possible to start with just one cow, and hence with less money. But it also means that there is less area to defend initially. So, in the end the game balances.

To use the `rand` function, I first must initialize the random number generator with a *seed*, which needs to be different each time the game starts to generate different outcome each play. A good seed value is the time in which the game starts up.

```
<<initialize random number generator>>=
srand(time(NULL));
```

`srand` and `rand` are declared in the `stdlib.h` header; `time` in `time.h`. I need to include both headers.

```
<<headers>>=
#include <stdlib.h>
#include <time.h>
```

2.1 Turns

This version of The Horde is turn based which means that in each turn the player and the hordlings can make a *move*. A move can either be an actual movement to an adjacent cell, building a fence, or buying a cow. The player is the only that can make the last two.

Based on the number of turns the player lasts against The Horde, the game sends in a new wave of hordlings or even increases the difficulty. To know the number of turns played, I need a new variable. Using an unsigned integer to make the number of turns practically unlimited.

```
<<main variables>>=
unsigned int current_turn;
```

When the game starts, this variable is set to zero: the first turn.

```
<<initialize level>>=
current_turn = 0;
```

For each game's loop, I need to increment the turn counter by one.

```
<<update turn counter>>=
++current_turn;
```

2.2 Gold

As stated, the game's objective is to reach a minimum of $2000 in gold.

```
<<constants>>=
#define GOLD_GOAL 2000
```

That means that the game must keep account of the current amount of gold. Given the defined upper limit of gold, using a `short` to store the amount is enough. I can't use an `unsigned` because it is possible to reach a negative amount due to attacks from the hordlings, in which case the game ends and the player loses.

```
<<main variables>>=
short gold;
```

This amount needs to be zero at the beginning.

```
<<initialize level>>=
gold = 0;
```

The game reward the player with gold once every new *season*, which starts after an arbitrary number of turns. Through trial and error, I've defined the number of turns to start a season to be 150 that gives enough time to stop the current wave and prepare for the next.

```
<<constants>>=
#define SEASON_TURNS 150
```

Thus, when a new season starts the game counts how many cows still remain on the level and increment the amount of gold by $10 for each still alive.

```
<<constants>>=
#define COW_PROFIT 10
```

```
<<new season>>=
if (current_turn % SEASON_TURNS == 0) {
    for (y = 0 ; y < LEVEL_HEIGHT ; ++y) {
        for (x = 0 ; x < LEVEL_WIDTH ; ++x) {
            if (COW_CHAR == level[y][x]) {
                gold += COW_PROFIT;
            }
        }
    }
```

Notice how the first turn being zero forces to start a new season to start right at the beginning therefore allowing the player to start with some money to spend coming from the cows placed randomly. Unfortunately, for the player, it also means that the first wave of hordlings starts to attack immediately.

As with real life, spending or losing money can happen any time, not only at a new season's start. The player can build a fence or buy a cow any turn. Also, when a hordling attacks a player instead of going after the cattle, the player loses some of the money. The game, thus, needs to check each turn if the player lost all her money. I also check if the player reached the goal at the same time, because either way the logic to quit the game is almost the same, and I want to avoid repetition.

```
<<check gold>>=
if (gold < 0 || gold >= GOLD_GOAL) {
    mvprintw(LEVEL_HEIGHT / 2, LEVEL_WIDTH + 1, (gold < 0) ? "LOSE" : "WIN");
    getch();
    quit = TRUE;
}
```

When the game is over, either because the player won or lost, I print out the outcome on the level's right side and then way for a key before quiting the game.

2.3 Status

Besides the position of all entities, the player needs to see other information such as the amount of gold remaining, to be know whether has the possibly to build a fence or buy a cow; and the current turn, to know when a new season starts.

I print the this extra status at the first row next the level.

```
<<draw level>>=
mvprintw(0, LEVEL_WIDTH + 1, "$%i T%i ", gold, SEASON_TURNS - (current_turn % SEASON_TURNS) ↩
    );
```

Instead of the actual current turn I print the number of turns remaining until the next season. With this is easier to know when a new season starts than with the actual turn number, which gives no extra information to the player.

The additional space written after the turn is there to erase any digit left over when the number of turns gets smaller and uses a digit less (i.e., it goes down from 100 to 99, and from 10 to 9.)

3 Player

Following NetHack's conventions, the player is represented on screen as the *at* character (`@`).

```
<<constants>>=
#define PLAYER_CHAR '@'
```

The player is the only game entity than instead of her position being reflected in the `level` array has its own pair of variables, one for the row and the other for the column.

```
<<main variables>>=
int playerx;
int playery;
```

This difference between the player and the other entities is due the fact that the player is the only entity that can be, on some occasions, on top of another entity. This can happen in two situations:

1. At the beginning, the player is randomly placed in the same area as the cows and thus on top of one of them.
2. When the player builds a new fence or buys a cow, it is placed at the same position where the player is located at.

The first case might be removed if instead of allowing the player to be on top of a cow, I remove the cow replacing it with the player. But the second case is trickier to remove, because if I tried to put a new fence or cow in front of the player, for example, then I would need to keep the current player's direction. Instead, I keep the player's position because is easier to understand than the direction.

Therefore, to draw the player at her position I need to move the curses to the right cell on the screen and write the character, all this **after** drawing the level, otherwise the player wouldn't be visible.

```
<<draw level>>=
mvaddch(playery, playerx, PLAYER_CHAR);
```

I also need to set the player's initial position. As said, I am going to place the player at the same area where the cows are, to keep her away from the wave of hordlings.

```
<<initialize level>>=
playerx = rand() % 6 + 8;
playery = rand() % 6 + 8;
```

3.1 Moving Around

To move the player around the level, I am going to use the same keys as used by NetHack, which has two sets of keys.

The first set use the *numeric pad*, where `4`, `2`, `8`, and `6` move the player to the adjacent cell west, south, north, or east respectively. `1`, `3`, `7`, and `9` move the player diagonally.

The second is similar to the set used by *vi* to move the cursor around, with keys `h`, `j`, `k`, and `l` to move the character to the adjacent cell west, south, north, or east respectively. The `y`, `u`, `b`, and `n` keys move the player diagonally.

As the player moves around the screen, the game needs to check whether the player is going to step in a cell within the level's limit, if is going to attack a hordling, and other similar cases. To avoid repeating all this checks throughout the code that received the input, I am going to have a pair of variables that tell the position the player should move given the input.

```
<<main variables>>=
int new_playerx;
int new_playery;
```

Thus, I need to set these variables based on the current player's position as well as the character received from the keyboard. If the key is not a movement command, the new player's position is set outside the level on purpose.

```
<<handle keyboard input>>=
new_playerx = -1; /* set outside the level to know whether the key was */
new_playery = -1; /* a movement command or any other command. */
switch (getch()) {
    /* west */
    case 'h':
    case '4':
        new_playerx = playerx - 1;
        new_playery = playery;
        break;

    /* south */
    case 'j':
    case '2':
        new_playerx = playerx;
        new_playery = playery + 1;
        break;

    /* north */
    case 'k':
    case '8':
        new_playerx = playerx;
        new_playery = playery - 1;
        break;

    /* east */
    case 'l':
    case '6':
        new_playerx = playerx + 1;
        new_playery = playery;
        break;

    /* north-west */
    case 'y':
    case '7':
        new_playerx = playerx - 1;
        new_playery = playery - 1;
        break;

    /* north-east */
    case 'u':
    case '9':
        new_playerx = playerx + 1;
        new_playery = playery - 1;
        break;

    /* south-west */
    case 'b':
    case '1':
        new_playerx = playerx - 1;
        new_playery = playery + 1;
        break;

    /* south-east */
    case 'n':
    case '3':
        new_playerx = playerx + 1;
        new_playery = playery + 1;
        break;
```

With these variables correctly set, I can now check whether the new position is within the level's limits and if the cell in which the player is going to move is empty. If so, the movement is permitted.

```
<<update player position>>=
if (new_playerx > 0 && new_playerx < LEVEL_WIDTH - 1 &&
    new_playery > 0 && new_playery < LEVEL_HEIGHT - 1 ) {

    <<check collision with hordlings>>

    if (EMPTY_CHAR == level[new_playery][new_playerx]) {
        playerx = new_playerx;
        playery = new_playery;
    }
```

3.2 Buying Cows

To buy a cow and place at the player's position, the game needs to check whether there are enough gold to buy a new cow based on their cost.

```
<<constants>>=
#define COW_PRICE 20
```

It must also check if the player's position is actually empty. Remember that the player could be on top of a cow or fence.

```
<<buy and place cow>>=
if (gold >= COW_PRICE && EMPTY_CHAR == level[playery][playerx]) {
    gold -= COW_PRICE;
    level[playery][playerx] = COW_CHAR;
}
```

To place a new cow the user must enter the `c` key. This sets a new cow character at the player's position and also decrease the amount of gold.

```
<<handle keyboard input>>=
    case 'c':
        <<buy and place cow>>
        break;
```

3.3 Fences

The fences the player can put on can have different strength, from the *weakest* to the *stronger*. Both building a new fence and upgrading a fence to be stronger has the same cost of $1.

```
<<constants>>=
#define FENCE_COST 1
```

To make easier upgrading and downgrading the fences' strength, the different strengths have consecutive characters on screen. Thus, with simple increments and decrements I can change a fence's strength. Therefore, the difference between a fence's character and the weakest fence character is the fence's strength.

```
<<constants>>=
#define FENCE_WEAKEST_CHAR 176
#define FENCE_STRONGEST_CHAR 180
```

The characters that I've selected for the fences are from the extended ASCII table, also known as code page 437. The intended output for these characters is only correct when running in a terminal set to this code page; all other configurations will show different characters for the fences. This is a *known issue* that won't be fixed, because there is no characters in the regular ASCII set that looks as a fence to me, and using a suitable character for different configurations isn't worth it.

Building a new fence, then, is a matter of placing the weakest fence on the position where the player is located at, if that position is empty and there is enough gold. This is done when the user presses the `f` key.

```
<<handle keyboard input>>=
    case 'f':
        if (gold >= FENCE_COST && EMPTY_CHAR == level[playery][playerx]) {
            gold -= FENCE_COST;
            level[playery][playerx] = FENCE_WEAKEST_CHAR;
        }
        break;
```

To upgrade an existing fence I have to come up with another way, because the player can't pass over a fence, forcing the user to think better fence placement strategies. Instead, the I will upgrade the fences when the player tries to move to a cell already occupied by a fence. That is, instead of passing over she will upgrade the fence, if there is enough money and the fence is not at the strongest already. Remember how the different fences' strengths have consecutive values and thus a simple increment upgrades a fence.

```
<<update player position>>=
    else if (level[new_playery][new_playerx] >= FENCE_WEAKEST_CHAR &&
        level[new_playery][new_playerx] < FENCE_STRONGEST_CHAR &&
        gold >= FENCE_COST) {
        gold -= FENCE_COST;
        ++level[new_playery][new_playerx];
    }
}
```

4 Hordlings

Like NetHack, this game represents the hordlings on screen using capitals letters from *A* to *Z*, with `A` the weakest hordling and `Z` the strongest.

```
<<constants>>=
#define HORDLING_WEAKEST_CHAR 'A'
#define HORDLING_STRONGEST_CHAR 'Z'
```

At the each season's beginning, the game spawns at least two new hordlings. These new hordlings always appear on the levels borders, one either north or south while the other always east or west. To make things more interesting, every year — every 4 seasons — The Horde will increase the attack's difficulty by adding two hordlings to the wave and by sending stronger hordlings.

```
<<constants>>=
#define HORDLING_LEVEL_UP_TURN (SEASON_TURNS * 4)
```

As with the cows, the game will choose random positions for hordlings and place them on the level each new season. That means that it is possible for a hordling to replace another, and thus having one less hordling than expected. That is a lucky season.

```
<<main variables>>=
int hordling;
int hordling_level;
```

```
<<new season>>=
    hordling_level = current_turn / HORDLING_LEVEL_UP_TURN + 1;
    for (hordling = 0 ; hordling < hordling_level ; ++hordling)
    {
        /* north or south */
        level[(rand() % 2) * (LEVEL_HEIGHT - 1)][rand() % LEVEL_WIDTH] = ↩
            HORDLING_WEAKEST_CHAR + (rand() % hordling_level);
        /* east or west */
        level[rand() % LEVEL_HEIGHT][(rand() % 2) * (LEVEL_WIDTH - 1)] = ↩
            HORDLING_WEAKEST_CHAR + (rand() % hordling_level);
    }
}
```

4.1 Cooooooowsssss

At each turn, every hordling on the screen will move a step closer to its nearer cow. But first, the game needs to know which hordling to update by looping over the level and looking for hordlings in it. If there is no hordling to move, then we can continue with the turn.

```
<<update hordlings position>>=
while (TRUE) {
    int hordlingx = -1;
    int hordlingy = -1;
    <<other hordling position variables>>

    for (y = 0 ; y < LEVEL_HEIGHT ; ++y) {
        for (x = 0 ; x < LEVEL_WIDTH ; ++x) {
            if (level[y][x] >= HORDLING_WEAKEST_CHAR &&
                level[y][x] <= HORDLING_STRONGEST_CHAR) {
                hordlingx = x;
                hordlingy = y;
            }
        }
    }

    if (hordlingx < 0) {
        /* No more hordlings. */
        break;
    }
```

With the hordling position, I need to find the nearest cow to move toward. Also, if the player is nearer to the hordling than any cow, the hordling instead will move towards the player to steal a piece of gold. I do this by comparing the distance in the X and Y coordinates between each cow and the hordling, and the distance between the hordling and the player. Whichever is the shorter distance, becomes the hordling's target.

```
<<other hordling position variables>>=
int distance;
int targetx;
int targety;
```

```
<<update hordlings position>>=
    /* find nearest cow, or use the player if nearer. */
    targetx = playerx;
    targety = playery;
    distance = abs(hordlingx - playerx) + abs(hordlingy - playery);
    for (y = 0 ; y < LEVEL_HEIGHT ; ++y) {
        for (x = 0 ; x < LEVEL_WIDTH ; ++x) {
            if (COW_CHAR == level[y][x]) {
                int new_distance = abs(hordlingx - x) + abs(hordlingy - y);
                if (new_distance < distance) {
                    targetx = x;
                    targety = y;
                    distance = new_distance;
                }
            }
        }
    }
```

Here I am using the `abs` function declared in `math.h`.

```
<<headers>>=
#include <math.h>
```

With the target's position in the `targetx` and `targety` variables, now I can compute the new position for the hordling in order to move in. Like the player, the hordlings only can move a cell per turn.

```
<<other hordling position variables>>=
int new_hordlingx;
int new_hordlingy;
```

```
<<update hordlings position>>=
    new_hordlingx = hordlingx + (hordlingx == targetx ? 0 : (hordlingx < targetx ? 1 : -1)) ↩
        ;
    new_hordlingy = hordlingy + (hordlingy == targety ? 0 : (hordlingy < targety ? 1 : -1)) ↩
        ;
```

Before I can move the hordling I need to check whether there is any impediment on its way. For the case of a hordling, either a fence or the player.

When there is a fence in front of the hordling, the hordling will hit the fence and will weaken it. If the fence was of the weakest strength, then it will vanish.

```
<<update hordlings position>>=
    if (level[new_hordlingy][new_hordlingx] >= FENCE_WEAKEST_CHAR &&
        level[new_hordlingy][new_hordlingx] <= FENCE_STRONGEST_CHAR) {
        --level[new_hordlingy][new_hordlingx];
        if (level[new_hordlingy][new_hordlingx] < FENCE_WEAKEST_CHAR) {
            level[new_hordlingy][new_hordlingx] = EMPTY_CHAR;
        }
```

Otherwise, if the fence wasn't crushed and it is still a bit stronger than just the bare bone fence, is the hordling who will also take a hit. Like the fence, if the hordling was of the lowest strength, it will perish.

```
<<update hordlings position>>=
        else if (level[new_hordlingy][new_hordlingx] > FENCE_WEAKEST_CHAR) {
           --level[hordlingy][hordlingx];
           if (level[hordlingy][hordlingx] < HORDLING_WEAKEST_CHAR) {
               level[hordlingy][hordlingx] = EMPTY_CHAR;
               /* next monster */
               continue;
           }
        }
    }
```

If instead of a fence, the hordling hits the player, it will steal a piece of gold from her. Remember that the player's position is not in the `level` array but kept by two separate variables.

```
<<update hordlings position>>=
    else if (new_hordlingy == playery && new_hordlingx == playerx) {
        --gold;
    }
```

If it is not any of these cases, check whether the cell to move is either empty or has a cow. In both cases, the hordling moves to the new position. If the cell has a cow, then the hordling replaces — or eats — the cow as if it were an empty cell.

```
<<update hordlings position>>=
    else if (level[new_hordlingy][new_hordlingx] == EMPTY_CHAR ||
        level[new_hordlingy][new_hordlingx] == COW_CHAR) {
        level[new_hordlingy][new_hordlingx] = level[hordlingy][hordlingx];
        level[hordlingy][hordlingx] = EMPTY_CHAR;
        /* update the position. */
        hordlingx = new_hordlingx;
        hordlingy = new_hordlingy;
    }
```

Finally, to mark that the hordling has already moved and thus no need to update it again until the next turn, I'll convert its character to lower case. Since the hordlings characters are all uppercase (from `A` to `Z`), converting it to lowercase the previous loop that looks for hordling will skip all already moved hordlings and stop when all of them have moved.

```
<<update hordlings position>>=
    level[hordlingy][hordlingx] = tolower(level[hordlingy][hordlingx]);
}
```

The `tolower` function is declared in the `ctype.h` header that I need to include.

```
<<headers>>=
#include <ctype.h>
```

After all the hordlings have moved, I must change all them back to uppercase to make them ready for the next turn. I'll use the fact that only hordlings are lowercase letters to my advantage and use the `islower` function, also declared in the `ctype.h` header, to find the hordlings and the `toupper` function to convert them back to capitals.

```
<<update hordlings position>>=
for (y = 0 ; y < LEVEL_HEIGHT ; ++y) {
    for (x = 0 ; x < LEVEL_WIDTH ; ++x) {
        if (islower(level[y][x])) {
            level[y][x] = toupper(level[y][x]);
        }
    }
}
```

I am aware that all this looping around the level has a complexity of *O(n^2)*, but given the size of the grid (24x24 cells) this is hardly an issue. If the level was any bigger, a I would advise a different approach, such as keeping a list of hordlings and cows.

4.2 The Foretold Death of a Hordling

Killing hordlings is done in the same way as upgrading fences, and reciprocally how hordlings steal from the player: when the player is adjacent to a hordling and moves towards it, the hordling receives a hit and lowers its strength.

```
<<check collision with hordlings>>=
if (level[new_playery][new_playerx] >= HORDLING_WEAKEST_CHAR &&
    level[new_playery][new_playerx] <= HORDLING_STRONGEST_CHAR) {
    --level[new_playery][new_playerx];
```

If the hordling was of the weakest strength, then it dies and leaves its cell empty, in which case the player now can move in it. That is why checking for collision with hordlings is done before trying to actually update the player's position.

```
<<check collision with hordlings>>=
    if (level[new_playery][new_playerx] < HORDLING_WEAKEST_CHAR) {
        level[new_playery][new_playerx] = EMPTY_CHAR;
    }
}
```

5 Skipping Turns

It is possible to skip to the next season, for instance when the player no longer wants or can build fences due a shortage of gold.

Skipping to the next session is accomplished by pressing the `s` key in a turn that itself doesn't start already a new season.

Keep in mind that the turn counter is incremented once per turn, thus to start a new session, I need to add to the number of turns required until the next season's beginning, short by one turn, which is added at the end of the current turn.

```
<<handle keyboard input>>=
    case 's':
        if (0 != current_turn % SEASON_TURNS) {
            current_turn += SEASON_TURNS - (current_turn % SEASON_TURNS) - 1;
        }
        break;
```

6 The Game Loop

The main game's loop is kept in the `main` function. The whole game loops until either the player reached the goal or the hordlings left the player without money. This is stored in a boolean variable called `quit`.

```
<<main variables>>=
char quit;
```

This variable is set to `FALSE` when initializing the level and is only changed to `TRUE` when the game needs to quit.

```
<<initialize level>>=
quit = FALSE;
```

The main loop, thus, looks like this:

```
<<main>>=
int main()
{
    <<main variables>>

    <<initialize curses>>
    <<initialize random number generator>>
    <<initialize level>>
    while (!quit) {
        <<new season>>
        <<draw level>>
        <<check gold>>
        if (!quit) {
            <<handle keyboard input>>
            <<update player position>>
            <<update hordlings position>>
            <<update turn counter>>
        }
    }

    return EXIT_SUCCESS;
}
```

It is also possible to quit the game at any moment by hitting the `q` key.

```
<<handle keyboard input>>=
    case 'q':
        quit = TRUE;
        break;
}
```

7 Setting up curses

Before the game can draw anything on the screen, it needs to initialize the curses library. First of all, to be able to call any curses function, I need to include its header file.

```
<<headers>>=
#include <curses.h>
```

Then, I need to initialize the main terminal screen.

```
<<initialize curses>>=
initscr();
```

`initscr` will already show an error message and call `exit` if it can't initialize the library. That means I don't need to add any additional check for errors.

This game don't need to show the cursor while writing all the characters around the screen, nor show the entered keys from the keyboard.

```
<<initialize curses>>=
curs_set(0); /* 0 == cursor is invisible. */
noecho();
```

By default, curses will wait until the user presses `enter` before giving any input to the application. As it would be cumbersome to expect the user to press enter every time, I tell curses to give out the key as soon as entered, but at the same time for the library to handle break and other interrupts.

```
<<initialize curses>>=
cbreak();
```

For this game I require the library to wait until the player pressed a key, as this is the basis for each game turn. It turns out that by default, curses already does that and thus I don't need to add any other call.

Finally, once the game is over and must quit, I need to clean up curses. I will use `atexit` to clean up the window when `exit` get called, but I can't pass `endwin` directly as a parameter to `atexit` because the prototypes don't match: `endwin` returns an `int` but `atexit` expects a function that returns nothing. Instead, I'll write a new function, called `cleanup`, with the proper function signature for `atexit` which calls `endwin`.

```
<<curses clean up function>>=
void
cleanup_curses()
{
    endwin();
}
```

Now I can use this function to clean up curses at exit.

```
<<initialize curses>>=
atexit(cleanup_curses);
```

A horde.c

A single C source code module holds all the building blocks I've described earlier.

```
<<*>>=
/*
    Horde - A curses based action-strategy game.
    Copyright (c) 2010 Jordi Fita <jfita@geishastudios.com>

    <<license>>
*/
<<headers>>

<<constants>>

<<curses clean up function>>

<<main>>
```

B Makefile

Being a simple application, an small Makefile would be sufficient to build and link `horde` from the source document.

The first thing that needs to be done is to extract the C source code from the AsciiDoc document using `atangle`. It is necessary, therefore, to have a `atangle` installed to extract the source code.

```
<<extract c source code>>=
horde.c: horde.txt
        atangle $< > $@
```

Then is possible to link the executable from the extracted C source code. Although, I have to take into account the platform executable suffix and the curse library used.

For Linux and other UNIX systems, the suffix is the empty string, but for Windows I need to append *.exe* to the executable.

For Linux and other UNIX systems, the curses library is `ncurses`, but for Windows I'll use `pdcurses`

To know which system is the executable being build, I'll use the `uname -s` command, available both in Linux and also in MinGW or Cygwin for Windows. In this case, I only detect the presence of MinGW because I don't want to add yet another dependency to Cygwin's DLL.

```
<<determine executable suffix>>=
UNAME = $(shell uname -s)
MINGW = $(findstring MINGW32, $(UNAME))
```

Later, I just need to check if the substring *MINGW* is contained in the output of `uname`. If the `findstring` call's result is the empty string, then we assume we are building in a platform that doesn't have executable suffix.

```
<<determine executable suffix>>=
ifeq ($(MINGW),)
EXE =
CURSES = -lncurses
else
EXE = .exe
CURSES = -lpdcurses -Wl,--enable-auto-import -mwindows
endif
```

With this suffix, I can now build the final executable. Of course, I need to link also with the curses library for it to work.

```
<<build horde executable>>=
horde$(EXE): horde.c
        gcc -o $@ $< $(CURSES)
```

Sometimes, it is necessary to remove the executable as well as the intermediary building artifacts. For this, I'll add a target named `clean` that will build all the files built by the Makefile and only left the original document. I have to mark this target as `PHONY` in case there is a file named `clean` in the same directory as the Makefile.

```
<<clean build artifacts>>=
.PHONY: clean

clean:
        rm -f horde$(EXE) horde.c
```

As the first defined target is the Makefile's default target, I'll place the executable first and then all the dependences, until the original document. After all source code targets, I'll put the `clean` target. This is not required, but a personal choice. The final Makefile's structure is thus the following.

```
<<Makefile>>=
<<determine executable suffix>>

<<build horde executable>>
```

```
<<extract c source code>>

<<clean build artifacts>>
```

C License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows: