Pong

COLLABORATORS

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

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
cf20255f108a	2011-08-01	Moved the download links of Pong on the top and made them an image link instead.	jfita
50da214af96e	2011-06-12	Added a parenthesis around the paddle definition.	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
6f75871d17ba	2011-03-10	Merge.	jfita
470514f24839	2011-03-10	Fixed some typos regading Windows in pong.	jfita
6cc909c0b61d	2011-03-07	Added the comments section.	jfita
3c0dd7be5c1d	2010-10-28	Corrected URL to atangle.	jfita
46aa433a1b8b	2010-10-28	Added the download section to pong.	jfita
d5cc1bcb5948	2010-10-28	The correct source language for Makefiles is make.	jfita
7ebebad0c47d	2010-10-28	Redacted the text for pong to be more understandable.	jfita

REVISION HISTORY

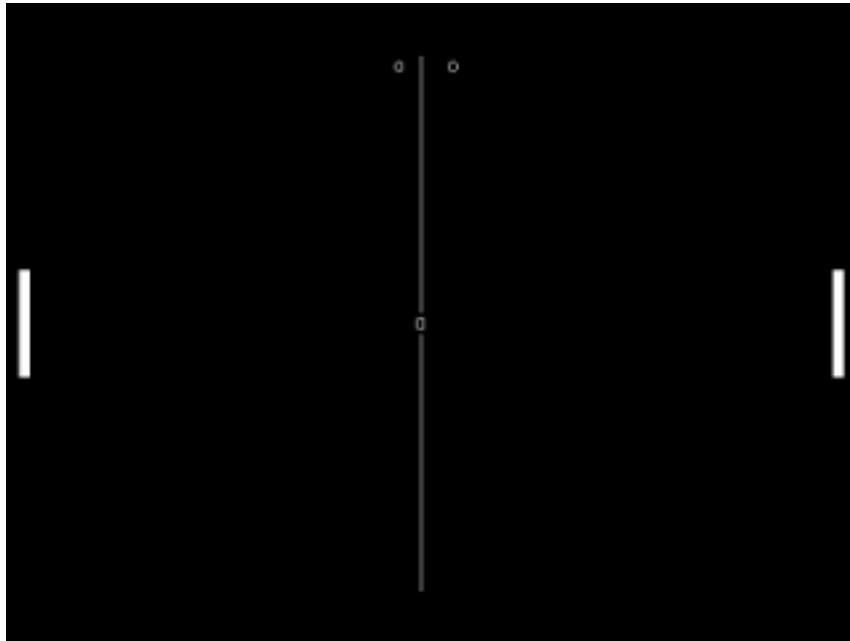| NUMBER	DATE	DESCRIPTION	NAME
eeceb77fda8e	2010-10-28	Merge	jfita
9f46544b23d7	2010-10-27	pong's ball now starts towards the last player who scored.	jfita
854feca7a1b6	2010-10-27	Added the source style and thus highlighting to the Makefile.	jfita
3a0393d06edb	2010-10-27	pong now uses pdcurses in Windows.	jfita
ccc95c8980cd	2010-10-27	Some "boolean" variables where assigned lowercase `true` and `false` values instead of `TRUE` and `FALSE`.	jfita
c7c6a737c9f0	2010-10-27	Added the screenshot of pong.	jfita
8ae1512e69fd	2010-10-27	Completed the first draft of pong.	jfita
6e8053073ef3	2010-10-25	Added the initial version of pong with only the Makefile and license text.	jfita

Contents

List of Figures

Figure 1: Screenshot of pong

Pong is a sports game that simulates a table tennis. The player controls a paddle moving it vertically across the left side of the screen and competes against a computer controlled opponent, who moves the paddle on the right side. Players use the paddle to hit the ball back and forth and a player earns points when the opponent fails to return the ball. The aim is for a player to earn more points than the opponent.

This version of Pong is written in C and uses the curses library to draw the game's screen content to a terminal.

1 Players

The players, both the human and the computer controlled, are represented in the game as an `struct`.

```
<<player struct definition>>=
typedef struct {
    <<player struct members>>
} player_t;
```

To know where to draw the players as well as use their position to check whether they bounce the ball back or miss, the game must store the players' X and Y coordinates. I'll limit the screen's size to 80 characters wide and 25 characters in height, which is the standard size for Windows' console, and thus storing the players' position in a `char` is enough. I don't use `unsigned char` because I need to know when a player tries to go outside the visible screen, and is easier, in terms of logic flow, to rectify a negative coordinate than to prevent it to underflow.

```
<<player struct members>>=
char x;
char y;
```

Players can only move vertically, either up or down, on their side of the screen. They can't move horizontally. To know in which sense on the vertical direction the player is moving, if any, I'll need a `sense` member in the player's structure. This member can have three values:

- `0` if the player is not moving at all,

- `1` if the player is moving down,

- `-1` if moving up.

```
<<constants>>=
#define SENSE_UP -1
#define SENSE_DOWN 1
#define SENSE_NONE 0
```

Given the limited range of value and that the values can be negative, the `sense` field can be a `char` type.

```
<<player struct members>>=
char sense;
```

These values are chosen in such a way that adding the `sense` value to `y`, the game updates the player's vertical position correctly without a need for a check the value of `sense`.

```
<<update player position function>>=
void
player_update_position(player_t *player)
{
    assert(player != NULL && "Tried to update a NULL player");
    player->y += player->sense;
```

To be able to `assert` that `player` is not a `NULL` pointer before updating the position, I have to include their headers. `NULL` is defined in `stdlib.h`.

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

The lowest a player's position can be is 0, which is the screen's first. Conversely, the highest a position can be is 24, which is the screen's lowest row. That is why adding 1 (`SENSE_DOWN`) to the player's position moves it down while adding -1 (`SENSE_UP`) (or subtracting 1) moves it up.

```
<<constants>>=
#define MIN_Y 0
#define MAX_Y 24
```

It is necessary to keep the player within limits when updating the position and set its `sense` to `SENSE_NONE` because the player can't move any further in the current sense.

```
<<update player position function>>=
    if (player->y >= MAX_Y) {
        player->y = MAX_Y;
        player->sense = SENSE_NONE;
    } else if (player->y <= MIN_Y) {
        player->y = MIN_Y;
        player->sense = SENSE_NONE;
    }
}
```

Each player has also an score. Arbitrarily, I've chosen 21 to be the maximum possible score, in which case the game ends and whoever reached this amount wins the game.

```
<<constants>>=
#define MAX_SCORE 21
```

Even If I had allowed a larger score, reaching an score of 255 would be frankly boring. Then, storing the score as an `unsigned char` is enough.

```
<<player struct members>>=
unsigned char score;
```

1.1 Initializing

There are two cases in which the game needs to initialize the players:

1. At the game's beginning.

2. Each time a player fails to return the ball.

Actually, both cases are same, because once a player fails to return the ball, the game starts over again, only keeping the scores. Therefore, with just a single function that sets the player's position with the values passed as parameters and sets the sense to SENSE_NONE to stop movement.

```
<<set player position function>>=
void
player_set_position(player_t *player, char x, char y)
{
    assert(player != NULL && "Tried to set position to a NULL player.");
    player->x = x;
    player->y = y;
    player->sense = SENSE_NONE;
}
```

To initialize, or reset, the player, I just need to call this function with the correct parameters.

```
<<set initial players position>>=
player_set_position(&left_player, LEFT_PLAYER_X, CENTER_Y);
player_set_position(&right_player, RIGHT_PLAYER_X, CENTER_Y);
```

LEFT_PLAYER_X, RIGHT_PLAYER_X, and CENTER_Y are constants that have been chosen using trial and error to fit best within the screen's limits.

```
<<constants>>=
#define LEFT_PLAYER_X 1
#define RIGHT_PLAYER_X 77
#define CENTER_Y 12
```

Besides setting the position, when the game starts it is also necessary to set the players' score to 0.

```
<<set score to zero>>=
left_player.score = 0;
right_player.score = 0;
```

1.2 Drawing

Drawing the players is a simple matter of drawing the characters that forms each player's paddle. I've chosen to draw the paddle using an space but with the terminal colors inverted. That means that on a blank terminal, the paddle will be drawn as a white block.

```
<<constants>>=
#define PADDLE_CHAR (' ' | A_REVERSE)
```

'A_REVERSE` is the way to tell curses that we want the colors reversed. This constant is defined inside curses' header and thus I need to include it.

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

With that, drawing the paddle is just writing the reversed space five times, the paddle's length, as well as a non-inverted space on top and below the paddle. These additional spaces are there to *remove* the remains from a previously drawn paddle in a different position, since a paddle only moves one row at a time either up or down.

```
<<constants>>=
#define BACKGROUND_CHAR ' '
```

```
<<draw player function>>=
void
player_draw(const player_t *player)
{
    assert(player != NULL && "tried to draw a NULL player.");

    mvaddch(player->y - 3, player->x, BACKGROUND_CHAR);
    mvaddch(player->y - 2, player->x, PADDLE_CHAR);
    mvaddch(player->y - 1, player->x, PADDLE_CHAR);
    mvaddch(player->y + 0, player->x, PADDLE_CHAR);
    mvaddch(player->y + 1, player->x, PADDLE_CHAR);
    mvaddch(player->y + 2, player->x, PADDLE_CHAR);
    mvaddch(player->y + 3, player->x, BACKGROUND_CHAR);
}
```

To draw both players, then, call this function with the player to draw as parameter.

```
<<draw players>>=
player_draw(&left_player);
player_draw(&right_player);
```

Drawing the player's score is done outside this function, in the main game's loop, which is also the responsible to draw the rest of the game's background.

```
<<draw players score>>=
mvprintw(SCORE_Y, LEFT_PLAYER_SCORE_X, "%02i", left_player.score);
mvprintw(SCORE_Y, RIGHT_PLAYER_SCORE_X, "%02i", right_player.score);
```

Again, the constants LEFT_PLAYER_SCORE_X, RIGHT_PLAYER_SCORE_X, and SCORE_Y have been picked up by hand based on how good they look on screen.

```
<<constants>>=
#define LEFT_PLAYER_SCORE_X 35
#define RIGHT_PLAYER_SCORE_X 40
#define SCORE_Y 0
```

1.3 Keyboard Controls

The human player is controlled using the keyboard. Unfortunately, in curses the game can only receive an event when a key is *pressed*, but **not** when it is *released*. That means I have two options to move the player's paddle:

1. Require that the player presses repeatedly the up or down key in order to keep moving the paddle.

2. When the up or down key is pressed, set the sense in which to move the paddle accordingly. The sense is kept the same until the player pressed the *same* key again, in which case the paddle stops, or presses the key for the opposite sense.

I've chosen to go with the latter option, because the paddle movement looks smoother and because I don't like to bang my keyboard.

Thus, to move a player with the keyboard, I have to add another function that sets the player's `sense` field according to the key passed as a parameter. The only thing that I need to keep into account is to check whether the `sense` needs to be set to stopped or not depending on its current state.

The keys to move up and down are hard coded. However, I use three sets of keys that are widely used in games:

1. The `up arrow` moves the paddle up and the `down arrow` moves it down.

2. The `w` key moves the paddle up and `s` moves it down.

3. The `k` key moves the paddle up and `j` moves it down.

```
<<set player sense by key function>>=
void
player_set_sense_by_key(player_t *player, int key)
{
    assert(player != NULL && "Tried to set the sense to a NULL player.");
    switch(key) {
        case KEY_UP:
        case 'w':
        case 'k':
            player->sense = (player->sense == SENSE_UP) ? SENSE_NONE : SENSE_UP;
            break;

        case KEY_DOWN:
        case 's':
        case 'j':
            player->sense = (player->sense == SENSE_DOWN) ? SENSE_NONE : SENSE_DOWN;

        default:
            /* Nothing to do.  This key gets ignored. */
            break;
    }
}
```

To use this function it is necessary to get the input from the user and, unless the user wants to quit the game, pass the key received as parameter together with the player that needs to be controlled with the keyboard.

```
<<get keyboard input>>=
key = getch();
if (key == 'q') {
    quit = 1;
    break;
}
player_set_sense_by_key(&left_player, key);
```

2 The Ball

Like the players, the ball is also represented in the game as an `struct`.

```
<<ball struct definition>>=
typedef struct {
    <<ball struct members>>
} ball_t;
```

Again, the game needs to know where the ball is located on the screen in order to properly draw it and also to check whether it needs to bounce or a player scored. Unlike the players, though, the game will update the ball position using sub-character position, i.e., factional positions, to allow for smoother diagonal movements and more control about the ball's speed. That means that the x and y positions are `floats`.

```
<<ball struct members>>=
float x;
float y;
```

When drawing the ball, the game needs to truncate the values of x and y because the curses library requires integer positions to print characters on the terminal.

```
<<draw ball function>>=
void
ball_draw(const ball_t *ball)
{
    mvaddch((int)ball->y, (int)ball->x, BALL_CHAR);
}
```

The character for the ball is a capital O.

```
<<constants>>=
#define BALL_CHAR 'O'
```

Another difference between the players and the ball, is that the ball can move either vertically or horizontally. And, for each direction, it can move in both senses. Therefore, I need to keep two sense members. Both 'float` to match the coordinates' type.

```
<<ball struct members>>=
float xsense;
float ysense;
```

The technique used to update the ball's position is the same used for players, but instead of integers, the senses are floating point numbers. Even so, the same rules apply:

- When `ysense` is negative, the ball moves up; when positive it moves down. Zero stops the ball in the vertical direction.

- When `xsense` is negative, the ball moves to the left; when positive it moves to the right. Zero stops the ball in the horizontal direction.

In the case of `ysense`, though, I allow more that just −1, 0, and 1 as value, because I want to have a ball that changes its speed depending on how it bounces off the players.

```
<<update ball position function>>=
void
ball_update_position(ball_t *ball, const player_t *left_player, const player_t * ↩
    right_player)
{
    <<update ball position variables>>

    assert (ball != NULL && "Tried to update the position to a NULL ball");
    assert (left_player != NULL && "I can't update the ball position without left player");
    assert (right_player != NULL && "I can't update the ball position without right player" ↩
        );

    ball->x += ball->xsense;
    <<check collision between ball and players>>
    ball->y += ball->ysense;
    <<check collision between ball and walls>>
}
```

2.1 Initializing

Much like the players, the ball needs to be initialized either at the beginning or when a player scores. In the case of the ball Y's sense is random and X's sense passed as parameter.

```
<<set ball position function>>=
void
ball_set_position(ball_t *ball, float x, float y, float xsense)
{
    assert(ball != NULL && "Tried to set position to a NULL ball.");
    ball->x = x;
    ball->y = y;
    ball->xsense = xsense;
    ball->ysense = ((rand() % 3) / 4.0f) - 0.25f;
}
```

The Y's sense is random because the game doesn't really change much if the ball starts going straight ahead, up, or down. But having a different and random starting direction adds some variety to the game.

To set the initial position, thus, the game needs to center the ball at the center of the screen. The initial X's sense is arbitrary and I just chosen to go towards the right player just to be positive. But to do that, I first need to set the sense to left.

```
<<set initial ball sense>>=
ball.xsense = SENSE_LEFT;
```

Then, the game will assign the reverse sense to the ball. With that, I can reuse the same function call when a player scores and the ball will start off at the opposite sense, and hence towards the scoring player.

```
<<set initial ball position>>=
ball_set_position(&ball, CENTER_X, CENTER_Y, -ball.xsense);
```

I also need to define the central X position and the sense to the left.

```
<<constants>>=
#define CENTER_X 38
#define SENSE_LEFT -1
```

2.2 Erasing the Past

In contrast to players, the ball can't easily erase itself when drawing on a new position, because it can bounce to a different direction when colliding with the walls or players, and thus we can't readily know which was the last ball's position.

In the ball's case, then, I'll add a new function to erase the current ball's position. This function is almost the same as the one drawing the ball which I've already shown, but the character to draw is the background instead of the ball's.

```
<<erase ball function>>=
void
ball_erase(const ball_t *ball)
{
    mvaddch((int)ball->y, (int)ball->x, BACKGROUND_CHAR);
}
```

This function needs to be called *before* updating the ball's position in the main loop, otherwise it would erase the ball in its new position and leave the old position alone, which is not what I am after.

2.3 Collision with Walls

To check if the ball collides with a wall, the only thing we need is the current ball's Y position and the position of the walls. The position of the walls is always fixed to MIN_Y for the top wall and MAX_Y for the bottom wall and that both walls have infinite length.

When the ball collides with a wall, the only thing the game needs to do is invert the Y's sense. That way, if the ball was moving upwards, it will now change to downwards and vice versa.

```
<<check collision between ball and walls>>=
if ((int)ball->y > MAX_Y) {
    ball->y = MAX_Y;
    ball->ysense = -ball->ysense;
} else if ((int)ball->y < MIN_Y ) {
    ball->y = MIN_Y;
    ball->ysense = -ball->ysense;
}
```

2.4   Collision with Players

The collision with the players is done in a similar fashion as with walls, except that the players do have a limited length. That means that we do not only need to check whether the ball's X position is the same as the player, but also if Y is within the range of the player's length.

To avoid duplication of code, I store the player in which the ball collides to a pointer. When that pointer is NULL, the ball didn't collide with any player. Thus the initial value is of no collision.

```
<<update ball position variables>>=
const player_t *colliding_player = NULL;
```

Then, I check whether the ball collided with the left player. If so, I store the left player's address to the pointer.

```
<<check collision between ball and players>>=
if ((int)ball->x == left_player->x && (int)ball->y <= left_player->y + 2 &&
    (int)ball->y >= left_player->y - 2) {
    colliding_player = left_player;
}
```

The same for the right player.

```
<<check collision between ball and players>>=
else if ((int)ball->x == right_player->x && (int)ball->y <= right_player-> y + 2 &&
    (int)ball->y >= right_player->y - 2) {
    colliding_player = right_player;
}
```

If the ball collided with any player, it needs to invert its X's sense, in the same way it inverted Y's when colliding with walls, and move the ball 2 positions into the new direction. Otherwise, it would look like the ball is on top of the paddle.

```
<<check collision between ball and players>>=
if (NULL != colliding_player) {
    ball->xsense = -ball->xsense;
    ball->x += 2 * ball->xsense;
```

But colliding with a player might also alter the ball's Y's sense. If, at the moment the collision, the player is moving, depending on whether the player is moving at the same sense as the ball or the opposite, it adds or removes some speed to the ball.

```
<<check collision between ball and players>>=
    ball->ysense += 0.25f * colliding_player->sense;
}
```

2.5 Ball Tracking

Besides the keyboard controlled player, the game has a dump computer player that the only thing that does is to set its sense to move toward the same vertical coordinate as the ball, or stops when the ball is in front.

For this to work I need both the current player's and ball's position and then set the player's `sense` member accordingly.

```
<<track ball function>>=
void
player_track_ball(player_t *player, const ball_t *ball)
{
    assert(player != NULL && "Tried to make a NULL player track a ball.");
    assert(ball != NULL && "Tried to track a NULL ball.");

    if ((int)ball->y > player->y) {
        player->sense = SENSE_DOWN;
    } else if ((int)ball->y < player->y) {
        player->sense = SENSE_UP;
    } else {
        player->sense = SENSE_NONE;
```

There is an special case, though: when the ball comes straight towards the player (i.e., ball's Y's sense is 0) and it is just one character in front, the player will start to move in a random sense. With that, as said, the ball changes its direction slightly when the it bounces off the moving player. I do that just to add a random new vertical sense to the ball.

To select the random sense — either 1 (SENSE_DOWN), 0 (SENSE_NONE), or -1 (SENSE_UP) — I get a random number between 0 and 2 and subtract 1.

```
<<track ball function>>=
        if (ball->ysense == SENSE_NONE &&
            (ball->x == player->x + 1) || (ball->x == player->x - 1)) {
            player->sense = (rand() % 3) - 1;
        }
    }
}
```

In order to use the random function, I first need to initialize the random number generator with a *seed*. A good seed is to use the time in which the game starts.

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

And to be able to use these two functions, I need to include their headers.

```
<<headers>>=
#include <time.h> // for time.
#include <stdlib.h> // for srand and rand.
```

To call this function, then, I just need to pass the player to move and the ball to track.

```
<<track ball>>=
player_track_ball(&right_player, &ball);
```

3 The Net

As with any table game, Pong needs to have a net between the two players. For this game it is just a vertical line from the first row to the last on the screen. In this case, the symbol to print to screen is the ACS_VLINE defined in `curses.h`.

```
<<constants>>=
#define NET_CHAR ACS_VLINE
```

The function that draw the net moves at the center of the first row and print a vertical line down to the last using this character.

```
<<draw net function>>=
void
net_draw()
{
    move(MIN_Y, CENTER_X);
    vline(NET_CHAR, MAX_Y + 1);
}
```

4 The Game Loop

Since this game has no menu, the main's game loop is implemented inside the `main` function.

The first thing I need is a variable to know when to stop the game and quit.

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

This variable is set to FALSE at the very beginning and is only set to to TRUE when the user tells that she wants to quit.

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

    quit = FALSE;
    <<initialization routines>>
    while (!quit) {
        <<play game>>
    }
    return EXIT_SUCCESS;
}
```

Before the main game loop starts, the game needs to initialize the curses library, the random number generator, as some of the game's status, such as the players' score and the initial ball's sense.

```
<<initialization routines>>=
<<init curses>>
<<init random number generator>>
<<set score to zero>>
<<set initial ball sense>>
```

After that, the game can begin creating the players, the ball and setting their initial position.

```
<<main variables>>=
ball_t ball;
player_t left_player;
player_t right_player;
```

```
<<play game>>=
<<set initial players position>>
<<set initial ball position>>
```

Then it just waits until the user enters `space`. It needs, thus, a variable to store the player's input.

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

With this variable it can wait until the user presses 'space` or q, in which case the game ends. Before that, though, it is nice to see something on the screen so it draws the players' and ball's current position, the net, the scores and the net.

```
<<play game>>=
clear();
<<draw players>>
net_draw();
<<draw players score>>
ball_draw(&ball);
refresh();

do {
    key = getch();
    if ('q' == key) {
        quit = TRUE;
        break;
    }
} while (key != ' ');

if (quit) {
    break;
}
```

To avoid moving the ball and paddles on the screen too fast to be playable, the game updates the players' and ball's position once every *some* loops. I've chosen the values for the players and ball to be the same, but they could be different.

```
<<constants>>=
#define BALL_SPEED 12 /* number of loop before update. */
#define PLAYER_SPEED 12 /* number of loop before update. */
```

To know when to update the positions, I need a pair of variables to keep track of the number of loops remaining before the players and the ball can be updated.

```
<<main variables>>=
int ball_update;
int player_update;
```

These variables are decremented each loop and when reach 0, the player's position can be updated.

```
<<update players position>>=
if (0 == --player_update) {
    player_update_position(&left_player);
    player_update_position(&right_player);
    player_update = PLAYER_SPEED;
}
```

The same for the ball. But, in the case of the ball, I also need to check whether the ball passed through any of the players. In that case, the game increments the other player's score and ends the match.

```
<<update ball position>>=
if (0 == --ball_update) {
    ball_erase(&ball);
    ball_update_position(&ball, &left_player, &right_player);
    ball_update = BALL_SPEED;

    if ((int)ball.x < left_player.x) {
        in_match = FALSE;
        ++right_player.score;
    } else if ((int)ball.x > right_player.x) {
        in_match = FALSE;
        ++left_player.score;
    }
}
```

At the match's beginning, these update variables are at their maximum value.

```
<<play game>>=
ball_update = BALL_SPEED;
player_update = PLAYER_SPEED;
```

After that, the match starts until one of the players scores 21 points.

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

```
<<play game>>=
in_match = TRUE;
while (in_match) {
    <<play match>>
}

if (left_player.score >= MAX_SCORE || right_player.score >= MAX_SCORE) {
    quit = TRUE;
}
```

In order to know approximately how much a loop lasts, the game will pause some milliseconds each loop after drawing and after updating the positions. The number of milliseconds I've chosen is arbitrary and make the game run fine.

```
<<constants>>=
#define GAME_SPEED 2 /* milliseconds per loop. */
```

```
<<play match>>=
<<draw players>>
net_draw();
<<draw players score>>
ball_draw(&ball);
refresh();
napms(GAME_SPEED);
```

Then we need to get the input from the user in order to update her paddle. The computer player also tracks the ball's position.

```
<<play match>>=
<<get keyboard input>>
<<track ball>>
```

And, finally, update the players and ball, if necessary, and pause again.

```
<<play match>>=
<<update players position>>
<<update ball position>>
napms(GAME_SPEED);
```

5   Setting up curses

Before the game can draw anything on the screen, it needs to initialize the curses library. The first thing to do, thus, is to initialize the main terminal screen.

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

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

Pong don't need to show the cursor while writing all the characters around the screen, thus I set it to invisible. If this fails, I ignore any errors and keep the cursor's visibility.

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

As input, the game needs to be able to get cursor keys events to move the player. In curses this is translated to a call to `keypad`. Also, I don't want to see the input keys on screen, therefore I need to disable the echo.

```
<<init curses>>=
noecho();
keypad(stdscr, TRUE);
```

By default, curses will wait until the user presses `enter` before giving to the application any other pressed input. As it would be cumbersome to expect the user to press enter every time, I make curses give out which key is entered as soon as it is pressed.

```
<<init curses>>=
raw();
```

What is also not acceptable in this game is for it to pause until the user presses a key when calling `getch`. To disable this behaviour, I need to call `nodelay`.

```
<<init curses>>=
nodelay(stdscr, TRUE);
```

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` gets called, but I can't pass `endwin` directly as a parameter to this function because the prototypes don't match: `endwin` returns an `int` but `atexit` expects a function with no parameters that returns nothing. Instead, I'll write a new function, called `cleanup`, with the proper function signature for `atexit` which calls `endwin`.

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

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

```
<<init curses>>=
atexit(cleanup);
```

A pong.c

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

```
<<*>>=
/*
    Pong - An curses based pong game.
    Copyright (c) 2010 Jordi Fita <jfita@geishastudios.com>

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

<<constants>>

<<ball struct definition>>
```

```
<<player struct definition>>

<<clean up function>>

<<draw ball function>>

<<erase ball function>>

<<set ball position function>>

<<track ball function>>

<<update ball position function>>

<<draw net function>>

<<draw player function>>

<<set player sense by key function>>

<<set player position function>>

<<update player position function>>

<<main>>
```

B   Makefile

Being a simple application, an small Makefile would be sufficient to build and link `pong` 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>>=
pong.c: pong.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
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 pong executable>>=
pong$(EXE): pong.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 pong$(EXE) pong.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 pong 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:

```
<<license>>=
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License version 2.0 as
published by the Free Software Foundation.

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

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
```