Motion Controlled Multiplayer Pong

Datorteknik, IS1200

October 10, 2025

Project Members:
[Razmus Nilsson, Razmus@kth.se, 990817-3439]
[Kai Nguyen, kaingu@kth.se, 041006-8357]

Objective and Requirements

With this project we have developed our own version of the classic 1972 game "Pong". In this game, two players compete to score points by ping-ponging a ball back and forth, attempting to launch the ball past their opponent's pad and hitting their goal (wall). First to score 10 points wins. What makes our version a bit different is that we have used two accelerometers as input to allow for motion control of the paddles, which enables an active and engaging multiplayer experience. The two players will control the pads in the Y-axis to defend their respective goals.

The following are the requirements we have met from our original draft to our final project:

- Multiplayer (2-player) and motion control are the main focus of the game. The users can control their respective paddles using an external accelerometer.
- The game features and keeps track of the score to reach a win-condition for one of the players. The winning player is displayed at the end.
- The ball reacts to where on the paddle it hits and changes its trajectory accordingly.
- The entire game is programmed in C.

Following are the original objectives that were omitted from this project:

- The original plan for the project and a big point was to use randomized power-ups that would spawn into the game and change up the gameplay.
- The game only controls in the linear Y-axis instead of both Y- and X-axis.
- We intended to use a bitmap to display the graphics.

Solution

The entire project works exclusively on our provided DTEK-V board, being developed in C. Our two accelerometers of the type MPU6050, act as gyro-sensors. The MPU6050 gives readings on acceleration in the x-, y- and z-axis. For our project, we only read the acceleration for the y-axis. This acceleration value gives us how much distance we need to move the paddle in the game engine. The MPU6050s is connected to a breadboard. Their power sources is then directly coupled with the DTEK-V board. The communication pins of the sensor, SDA and SCL pins, are both connected to GPIO pin 1 and 0 respectively.

To properly communicate with the sensors we had to use an external open sourced library dtekv-i2c-lib[1]. This enables us to properly communicate with I2C devices (our accelerometers) over the DTEK-V's GPIO pins without the use of an external I2C bus, which was the original plan for this project. The sensors are repeatedly read inside the game loop at each iteration to get real-time input from the players. The paddles will move up if the sensor reading is positive and vice versa. Since the sensor readings are very rapid and sensitive, we added a deadzone, which is a range where we ignore any inputs from the sensor. If the sensor readings fall within this range, we change the paddle's velocity to 0 which will make it stand still.

To visually display object onto the screen, we draw onto the VGA using the VGA frambuffer and treat it as an array of pixels with 320x240 resolution. To properly display colors, we use 8-bit colors represented in hex and put it inside the framebuffer at a predefined position. When drawing the paddles, ball and net, since they're all rectangular, we simply use a for loop that starts at a defined y-coordinate, and for each row which represents the height of the object, we will fill the object with a predefined amount of pixels in the x-row, which will represent the width of the object. Each pixel we fill will have one of the 8-bit colors that we defined. What we will effectively put inside the frambuffer is a bunch of 8-bit colors at selected x and y coordinates of the VGA framebuffer.

When handling movements of the draw objects, we simply refresh the screen by keeping track of where each object's previous location is, then draw the object at that location to be black to blend in with the background which effectively removes the objects from view. We then update the object positions, and then we redraw the object at the new location with their intended colors. Since we're constantly doing this act of erasing and re-drawing to keep track of the movement all objects on the screen has a constant flickering visual. Both the score and the "net" is re-drawn constantly as well due to the ball passing over these would act as a eraser if we omitted them from the constant re-drawing. We made this process more performant by constantly refreshing the screen using the in built timer of the DTEK-V board. This way, we can still perform other calculations in the main program, such as reading from the accelerometers, loop without blocking them in the same loop while refreshing the screen. The screen refresh and update will be handled via the timeout event which will in turn trigger an interrupt function. The interrupt function is what holds the logic for the screen refresh and update.

When drawing text and digits onto the screen, we define a 2D array bitmap where each row is a bitmap, consisting of 1s and 0s, that effectively describes how we should draw a certain character or digit onto the screen. If a location in the bitmap is 1, that means we should draw the portion of that character at that location. If 0, we ignore drawing it.

The game is displayed from the DTEK-V board's VGA output into an external compatible display where the graphical interface can be seen.

When handling collisions of the ball to the paddles, which we describe more extensively under verification, we take in consideration the paddles x and y position as well as their height and width, alongside the ball's x, y positions and its size. The ball is considered hit by a paddle if it reaches these conditions:

- The ball's y coordinate is within the range of the paddle's y coordinate and (y + paddle height).
- If ball hits left paddle, the ball is considered hit if the ball's x coordinate is equal to the paddle's (x coordinate + paddle width).
- If ball hits right paddle, the ball is considered hit if the ball's x coordinate is equal to the paddle's x coordinate.

The game ends when one of the player reaches 10 points. The score is counted when the ball passes the screen's left-most or right-most boundary. Here we use the same technique to detect the ball collision with the screen boundaries as we did for collision with paddles.

Verification

To verify that our game works properly as intended we have done multiple and mostly repetitive testing of the game's behaviour. Following will categorise these different cases and how we've handled them:

The core gameplay starts in the main function by drawing and initializing our objects and sensor. It then enters the gameplay loop when button one on the DTEK-V board is pressed. the function `handle_interrupt` will keep the gameplay-loop active until the win condition is met, while it keeps track of updating the graphics and handling that the paddle and ball positioning remains active. Inside of the function `start` the accelerometers will update the current position of the paddles.

An important part of Pong is the ball's behaviour and physics when interacting with the paddles or hitting the walls. Since we don't want the ball to reach out of bounds, we've defined and simply invert the direction of the ball when it hits either the upper or lower bounds, same way the paddles are not able to move out of bounds in their y-axis. The ball's trajectory when bouncing off the paddle is decided by a self defined sine (movement in X-axis) and cosine (movement in Y-axis) approximation, using the Taylor series definition. The max bounce angle is defined by a PI constant / 4, which gives us a max bounce angle of 45°.

When the ball hits the paddle the angle is calculated by taking the difference between the middle of the paddle and the ball's hit position on the paddle. The ball's hit position is then normalized between -1 and 1 which gives us a bounce angle when `hit_position` is multiplied by `max_bounce_angle`. This bounce angle is then taken through our cosine and sinus functions to calculate the new movement in the X- and Y-axis respectively.

The defining feature of our version is the use of external accelerometers. This in addition with adding a second one is as previously mentioned only possible with the external library. Since the library is only defined for a single I2C device, we had to edit it to read from both of MPU6050 devices separately, we read constantly the input from both during the gameplay loop.

Lastly we have done testing that the score updates and is displayed accordingly, which is whenever the goal (wall) of each respective player is hit. We exit the gameplay loop as soon as the winning condition of 10 is reached. The score and winning message both use 2D array-bitmaps to store the characters. The way we draw characters is to draw each row individually. Each element in the first dimension represents a row of 5 pixels (5 bites) wide and in binary is either 1 or 0 (on or off). All elements are 7 pixels in height. We take a string of characters as input and then translate it to pixels with our functions `draw_text` and `draw_char`.

Contributions

Since we chose to omit certain part of the final project the distribution of work shifted during development. Following is how the final work-load was distributed between the two of us.

Kai handled:

- The accelerometer's implementation and behaviour. He handled all external equipment and how the logic would function inside of the game. As well as the button logic to start and restart the game.
- The physics of the ball and its behaviour, especially its trajectory when bouncing of the paddle or the upper and lower bounds.

Ramzus handled:

- The graphics inside of the game and how it updates consistently under the gameplay loop.
- The implementation of the score and the display of text when the winning condition is reached.
- The scoring logic and the winning condition.

Reflections

In our first abstract draft we included certain objectives that have been omitted from the final project. Mainly being the power-up feature that was supposed to give the player that collected it a small temporary advantage. Unfortunately due to time constraints (and being more challenging than originally thought to implement) we chose to remove this feature, since it was not necessary for the core gameplay-loop.

We also wanted to display our objects in game with the use of a bitmap to draw each pixel individually on the screen, this was also omitted due to it simply not displaying properly and also being more difficult to implement than the option we chose to proceed with. We instead chose to display our graphics as blocks of pixels instead, which worked just as well. We did however use two separate bitmaps to display our score and winning message. Since these were only for display and did not have any physics, it worked properly without hassle.

References

1. Olivoz. *dtekv-i2c-lib*, 2025. Available at: `https://github.com/Olivoz/dtekv-i2c-lib`