Università degli Studi di Pisa
Scuola di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Robotica e dell'Automazione (LM-25)

FLUID IN A BOX

Studenti:

- Angelo Massara (641075) - a.massara@studenti.unipi.it
- Mattia Tinfena (535503) - m.tinfena@studenti.unipi.it

Real-Time Systems - A.A. 2021-2022

Contents

List of Figures

List of Algorithms

1

Overview of the Project

1.1 Request

Simulate a fluid in a box using a set of interacting particles. The box can rotate around its center and the user can impart commands to rotate it through the keyboard. The user should also be allowed to change the parameters of the simulation online.

1.2 Introduction

In this project, we simulated the behavior of a fluid in a box, through the cellular automata model. Particularly we focused the attenction on two specific fluids: sand and water. Our aim is to show how the particles which compose the fluid interact with each other when we rotate the box (and hence the frame). The project consists of two parts:

- software
- hardware

The software, is realized as a real-time multi-thread application in C language on the Linux operating system. We used the FIFO scheduling policy, implemenenting precedence costraints. Were also imported Pthread library (to allow multi-threading programmation) and the Allegro library (version 4.2.2) to make the Graphical User Interface, also known as GUI, as shown in the section 4.

For the hardware has been used a Raspberry Pi 4 model B (shown in fig. 1.1) where we installed the Raspberry Pi Operative System (also known as Respbian), based on Debian and optimized for the Raspberry Pi hardware, equipped with adxl345 accelerometer and a 24x32 LED matrix driven by the MAX7219 chip.

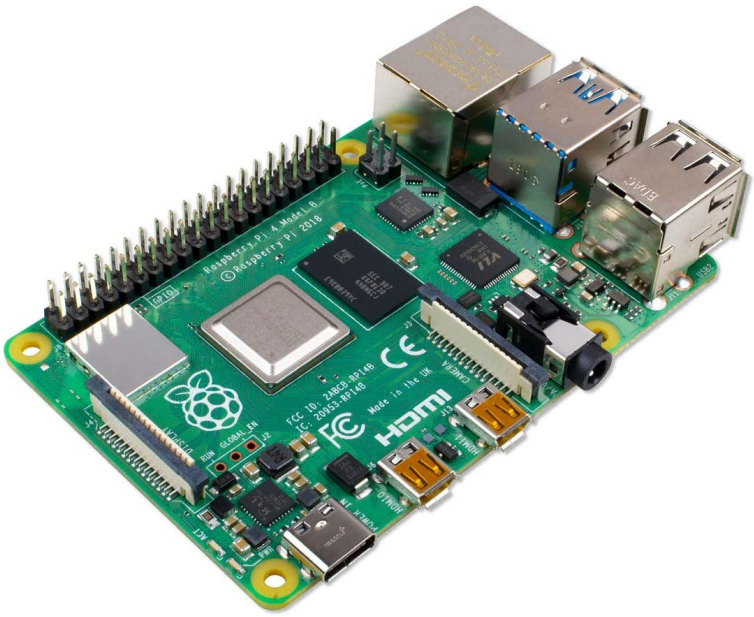

Fig. 1.1. Raspberry Pi 4 MODEL B

2

Physical Model Used

2.1 Cellular Automata

Cause the main aim of this project is not to represent a behavior completely true to life, the physical model we used is based cellular automata, which is a good approximation of the realty but easier to implement. A cellular automaton (pl. cellular automata, abbrev. CA) is a discrete model of computation studied in automata theory. A cellular automaton consists of a regular grid of cells, each in one of a finite number of states. The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state (time t = 0) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function)that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood.
In our practice, we consider the following rules for movement of each particle:

1. if a particle has no other particles below it, then it will fall to the bottom of the box according to the direction of gravity;
2. if a particle sees that the position immediately below is occupied by another particle, then it will check that the position to the right of the latter is free and in that case it will position itself there;
3. if the position on the right will also be occupied, then it will check if the position on the left is free and in that case it will position itself there;

These rules are implemented either for the sand either for the water. The main difference between the movement of sand and that of water occurs when the particle sees that the three positions below it are completely occupied: in the case of sand, The particle will stop in the position immediately above the first one it checked. In the case of water, in order to have a movement more similar to that of the liquid, we impose that in such case the particle will verify before the position to its right is free and if this too will be occupied then it will check the one to its left. Only if these two positions are also occupied the water

particle will stop. A clearer representation can be observed in figure 2.1 and 2.2, taken from the slides of the lesson "Examples of Real-Time Application" of this course(although in that case was preferred the movement to the left instead of the one to the right).

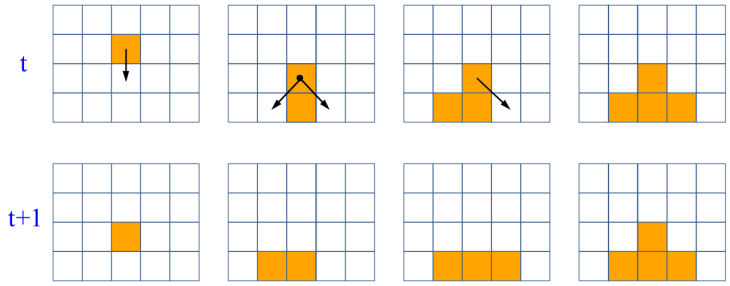

Fig. 2.1. Representation of the Physics of Cellular Automata - Sand

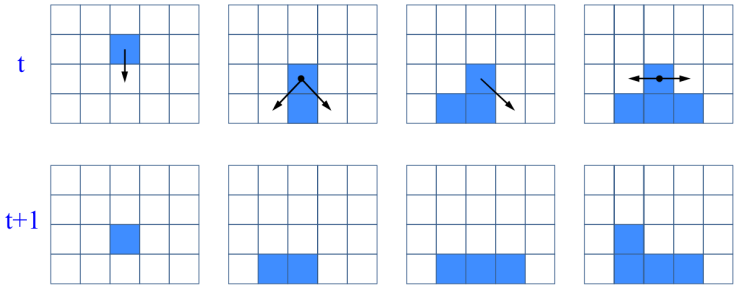

Fig. 2.2. Representation of the Physics of Cellular Automata - Water

2.2 Pitch - Roll angles

Now we introduce the pitch and roll angles, that we called θ and ϕ. As will be explained in Section 5, the accelerometer will return θ and ϕ angles which allow to determine in which direction we are turning the box.

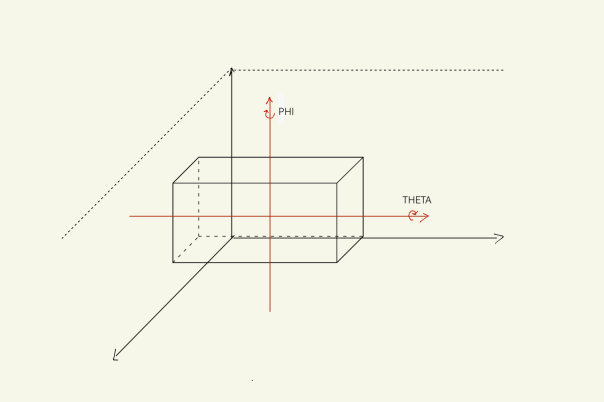

Fig. 2.3. Pitch-Roll angles

2.3 Orientation angle

Knowing θ and ϕ, is possible to calculate the alpha angle (the calculations are shown in the section 5) which allows to discretized four quadrants according to the direction of gravity. This discretization can be seen in the figure below, where the arrows indicate precisely the direction of gravity.

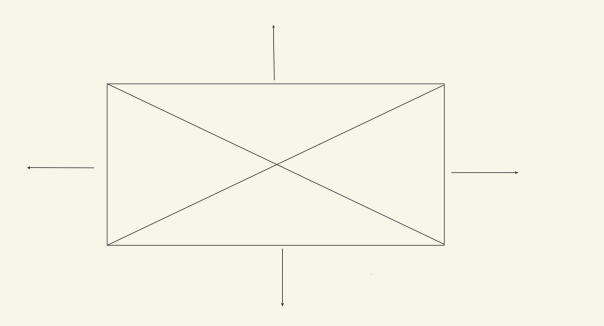

Fig. 2.4. Quadrant discreization

2.4 Gravity and Speed computation

Now we can move on to compute speed and gravity. These vectors were divided into their components along the columns (i.e., x) and along the rows (i.e., y). To compute gravitational acceleration were used the following equations:

$$g_c = g * cos(\alpha) * tilt_c \tag{2.1}$$

$$g_r = g * sin(\alpha) * tilt_r \tag{2.2}$$

It can be noted that both components have been multiplied by a tilt factor so as to consider the actual inclination of the box (for example, if the box is tilted to 90^{o} the tilt factor will be equal to 1, else the tilt factor will be less than 1, so as to reduce the contribution of gravity). The tilt factor depends on the orientation of the box and its components will be calculated as:

$$tilt_c = |\frac{\phi}{\frac{\pi}{2}}| \tag{2.3}$$

$$tilt_r = |\frac{\theta}{\frac{\pi}{2}}| \tag{2.4}$$

Before going to the calculation of the speed, we have to calculate the dt (which is the integration interval) multiplying the TSCALE (that it indicates the speed of the simulation) for the period, in our case 50 ms, therefore 0.05. At this point we can calculate the two components of velocity (for each element) with an iterative formula:

$$v_c^{(k+1)} = v_c^{(k)} + (g_c * \frac{dt}{scale}) \tag{2.5}$$

$$v_r^{(k+1)} = v_r^{(k)} + (g_r * \frac{dt}{scale}) \tag{2.6}$$

where $1/scale$ is a multiplicative factor that serves to slow down particles (we will have a lower scale value in the case of water as we assume that this type of particles has a faster speed than sand) and $v_c^{(k=0)} = v_r^{(k=0)} = 0$. Now we can handle the physics of this system, but it will be explained in the Section 6.

3

Design Choices

3.1 Approach

For this project we preferred to use a bottom-up approach, start from I/O devices and define low-level modules that abstract their behavior and then rise the level of abstraction defining higher-level modules with more complex behaviors, until you reach the goal. Then our analysis starts from the analysis of the data that are returned by the sensors (the accelerometer) and from here the orientation of the box is determined and the physical model is constructed.

3.2 Tasks and Scheduling

For our application we therefore thought of building first of all a task that takes the data from the accelerometer and returns θ and ϕ angles. Then we will have the task of physics which, based on the data collected from the previous task, determines the orientation of the box and calculates the movement of the particles. Then there will be the task which, taking the positions calculated by the physics task, projects them onto the hardware matrix. Finally we will have the user task, which will allow the user to interact with the application, and the graphics task, built thanks to the Allegro library. There are 2 sharing resources, R_0 (θ and ϕ) and R_1 (the mask of the matrix) which are protect with a classical binary semaphore used for exclusion called mutex. With these premises, and also considering that the tasks all have the same period, the scheduling that we considered the most appropriate to implement is FIFO with priority: in particular, we will implement in the following order (with decreasing priority): accelerometer, physics, led matrix and then, at the same priority, user and graphics. The priority are chosen at intermediate level so if in the future will be necessary add other tasks, at higher or lower priority, it will be possible. To make it clearer we have built the following Precendence Graph.

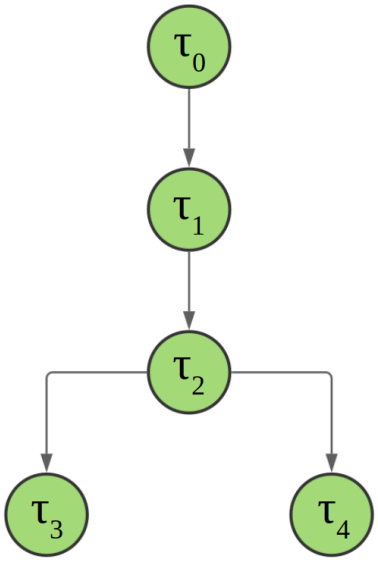

Fig. 3.1. Precedence Graph

4

User Interface

4.1 Allegro Library

The library we imported to realized the Graphical User Interface (GUI) is Allegro. is a recursive acronym which stands for Allegro Low LEvel Game ROutines and it's an open source graphic library for game and multimedia programming. The version we used is Allegro 4, which is the classic library, whose API is backwards compatible with the previous versions, back to Allegro 2.0.

4.2 General description

The background image with writings was created with an online software: we saved it as *.bmp* file to import it and use Allegro functions and is printed on the screen using the blit function. The main window has dimensions 1680x960, while the box, to have a linear match with the physical matrix (which is 32x24), has dimensions 640x480. For the color depth, we chose the *High Color Mode*, i.e. a color depth of 16 bit.

4.3 Commands and Features

We implemented the following commands:

- with the key P the user can play or pause the simulation;
- with the key S the user can switch the fluid to sand;
- with the key W the user can switch the fluid to water;
- with the key O the user can clear the screen from all particles;
- with the key R the user can restart the simulation;
- with the key B the user can add a particle in a random position;
- with the key G the user can increase gravity by a factor of 0.5;

- with the key *H* the user can decrease gravity by a factor of 0.5;
- with the key *T* the user can increase TSCALE by a factor of 0.25;
- with the key *Y* the user can decrease TSCALE by a factor of 0.25;
- with a *RIGHT CLICK OF THE MOUSE* the user can generate a particle at the point where he clicked.

We also implemented some features to manipulate θ and ϕ: in that case the accelerometer is disabled. Specifically, the commands are:

- with the key *RIGHT* the user can increase ϕ by 10°;
- with the key *LEFT* the user can decrease ϕ by 10°;
- with the key *UP* the user can increase θ by 10°;
- with the key *DOWN* the user can decrease θ by 10°;
- with the key *A* the user can reactivate the accelerometer;

Another particular feature we implemented is the radar that represent the gravity direction: the arrow is a *.bmp* file created with the same software used for the background but in this case, to make the pixels around transparent they are marked with the color makecol(255, 0, 255), corresponding to bright pink.

The final result is shown in the Fig. 4.1:

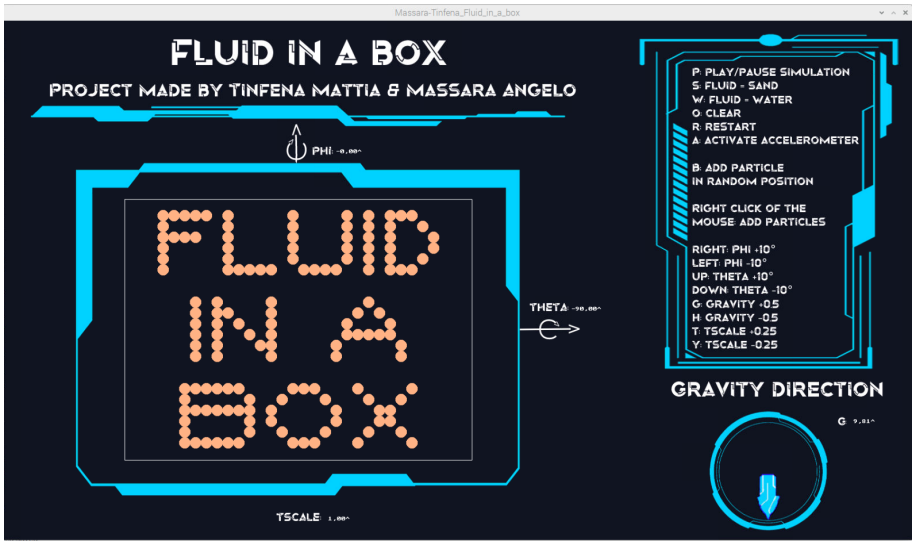

Fig. 4.1. Graphical User Interface

5

Tasks and Shared Resources

5.1 Tasks

5.1.1 Accelerometer task τ_0

Task parameters are:

- name: *accelerometer_read*
- index: 0
- period: 50 ms
- relative deadline: 50 ms
- priority: 13

This task reads the accelerations on the three axis of the accelerometer on the I2C port of the Raspberry Pi using the *WiringPi I2C Library* and save it in three variables named *x, y, z*. To have more reliable results unaffected by noise we made multiple lectures and we consider the average values. If the difference between the actual and the precedent lecture exceeds a certain value Δ we calculate θ and ϕ as:

$$\theta = atan(\frac{x}{\sqrt{y^2 + z^2}}) \tag{5.1}$$

$$\phi = atan(\frac{y}{\sqrt{x^2 + z^2}}) \tag{5.2}$$

5.1.2 Physics task τ_1

Task parameters are:

- name: *physics*
- index: 1
- period: 50 ms
- relative deadline: 50 ms
- priority: 12

This task handles the movement of each particle of the fluid according to cellular automata model. Firstly it reads θ and ϕ returned from the accelerometer task, then, if $\theta > |10°|$ or $\phi > |10°|$ calculate α as:

$$\alpha = atan(\frac{\theta}{\phi}) \tag{5.3}$$

otherwise we consider the box in stand-by mode. We distinguish four cases:

1. if $-135°< \alpha < -45°$ we consider the gravity directed downwards and we check the mask (R_1) from the bottom to the top of the box;
2. if $-45°< \alpha < +45°$ we consider the gravity directed rightwards and we check the mask (R_1) from the right to the left of the box;
3. if $+45°< \alpha < 135°$ we consider the gravity directed upwards and we check the mask (R_1) from the top to the bottom of the box;
4. if $+135°< \alpha < -135°$ we consider the gravity directed leftwards and we check the mask (R_1) from the left to the right of the box.

When the task finds a particle, it computes the next position according to the physical rules described in the section 2.

5.1.3 Led Matrix task τ_2

Task parameters are:

- name: *led_ matrix*
- index: 2
- period: 50ms
- relative deadline: 50 ms
- priority: 11

This task drive the phisical led matrix, the matrix is formed by 12 8x8 led matrices each of wich is driven by the max 7129, an integrate circuit that comunicates to the raspberry pi trough the SPI port. The max 7219 is formed by a 16 bit shift register so you have to send for first 8 bit to indicate the register and then 8 bit for the data, the register field indicate the row (from 1 to 8) of each 8x8 matrix, the data field indicate the decimal value of the row. This task reads the starting coordinates of a 8x8 matrix, calculate the decimal value of a row and write the value in an unsigned char array. After doing this for each matrix send the array through the SPI port. Then it makes the same thing with the other rows until it has read the whole mask (R_1).

5.1.4 User task τ_3

Task parameters are:

- name: *user*
- index: 3
- period: 50ms
- relative deadline: 50 ms
- priority: 10

This task reads the key pressed from user tanks to the allegro library. To ensure that the reading does not block the execution of the program, the key-pressed function has been implemented. This task performs several function as previously shown in the chapter 4.

5.1.5 Graphic task τ_4

Task parameters are:

- name: *graphic*
- index: 4
- period: 50ms
- relative deadline: 50 ms
- priority: 10

This task handle the GUI, during its execution it draw the balls of the fluid, calculate and show the θ and ϕ angles (R_0) in degrees, show the actual value of the gravity and the TSCALE and draw an arrow to indicate the gravity direction. For the ball a specific bitmap is used wich is destroyed after transfer to the main bitmap (called buffer). To show the arrow that indicates the gravity direction 4 sprite (one for each direction) with the bright pink background are used, the dimensions are scaled according to θ and ϕ angles with the function *stretch_ sprite*. In this case we report the code that was written to accomplish this function (only for one direction):

```
1     circlefill(buffer,1414,833,89,makecol(21, 23, 36));
2     switch (dir)
3     {
4       case down:
5           ascale = (PiGreco / 2) / (-theta);
6           larghezza = 180 / ascale;
7           altezza = 180 / ascale;
8           stretch_sprite(buffer, arrowdown,xcentro-(180/2)
    -(larghezza/2), ycentro-(180/2)-(altezza/2), 180/ascale
    , 180/ascale);
9           break;
10    }
```

Algorithm 5.1. Arrow Function

In the figure 5.1 is represented the diagram which show how tasks and resources interact with each other.

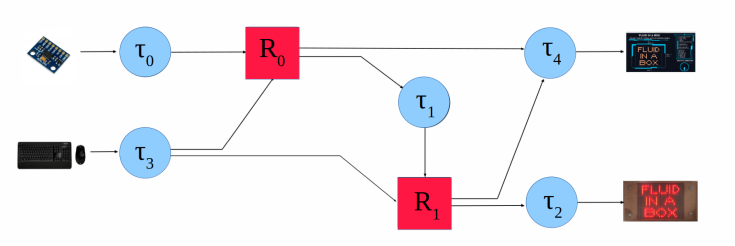

Fig. 5.1. Task-Resource Diagram

6

Experimental Results

The computation times of each task have been measured through the following function that allows you to start a timer when the task starts running and stops when the task ends.

```
long int start_time;
long int time_difference;
struct timespec gettime_now;
float C_min=1000,C_max=0, C_avg=0, total=0;
int iter=0;

void start_timer (void)
{
  clock_gettime(CLOCK_MONOTONIC, &gettime_now);
  start_time = gettime_now.tv_nsec;       // Set start time
}

void stop_timer (void)
{
  clock_gettime(CLOCK_MONOTONIC, &gettime_now);
  time_difference = gettime_now.tv_nsec - start_time;
  float wcet = time_difference;
  if (wcet < 0)
    wcet += 1000000000;             // Rolls every seconds
  wcet /= 1000000;                  // Convert in ms

  iter++;

  if (wcet < C_min && wcet > 0) C_min = wcet;   // set
    minimum time
  if (wcet > C_max) C_max = wcet;         // set maximum
    time

  total += wcet;                  // Compute average time
  C_avg=total/iter;
```

```
29
30   printf("\nC_min= %.3f", C_min);
31   printf("\nC_max= %.3f", C_max);
32   printf("\nC_avg= %.3f", C_avg);
33 }
```

Algorithm 6.1. Timer Function

It was implemented as follow:

```
1  void *task(void *arg)
2  {
3  <local state variables>
4  int ti;
5      ti = get_task_index(arg);
6      set_activation(ti);
7      while(1) {
8      start_timer();
9      <task_body>
10     stop_timer();
11     if (deadline_miss(ti))
12         <do action>
13     wait_for_activation(ti);
14
15     }
16 }
```

Algorithm 6.2. Scheme of a Periodic Task

The results we have obtained are shown in the following table:

Tasks	C_{min}	C_{avg}	WCET
τ_0	0.001	13.025	15.887
τ_1	0.001	0.300	0.648
τ_2	2.100	2.119	2.359
τ_3	0.002	0.004	0.042
τ_4	10.030	10.574	29.883

Table 6.1. Computation Times

Is possible to notice from the previous table that, summing the computational times in the worst case we get that:

$$U = \sum_{i=o}^{n} U_i = \sum_{i=o}^{n} \frac{C_i}{T_i} = 0.97638 < 1 \tag{6.1}$$

where n is the number of task - 1 (5 tasks so $n = 4$), C_i is the Computation Time in the worst case ($WCET$), T_i is the Period (50 ms for each task) and U_i is the Utlization Factor.
Because of $U < 1$ the task set is schedulable with the FIFO algorithm.
Occasionally, deadline miss of the graphical task was detected. However, as it is a soft task, we have found them tolerable, both because they occur rarely and because they do not degrade the performance of the system.
In the following pictures are shown some examples of the operation of the system.

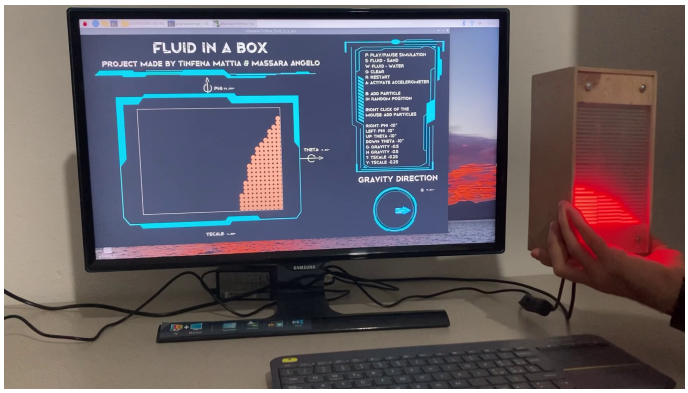

Fig. 6.1. Example 1

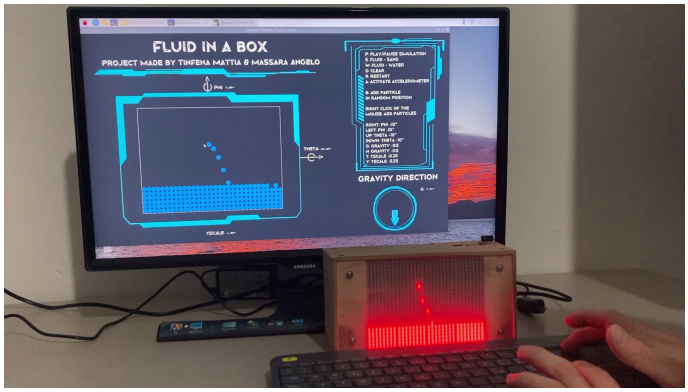

Fig. 6.2. Example 2

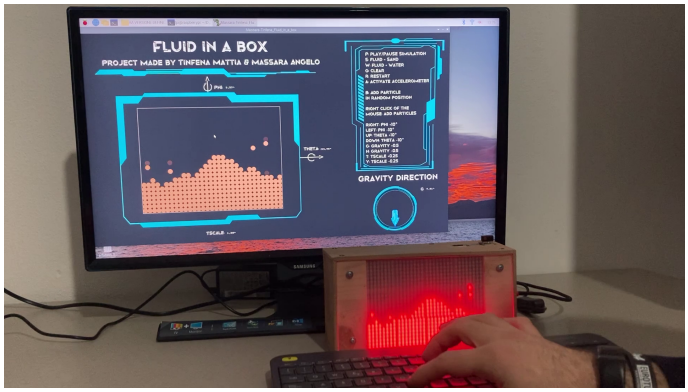

Fig. 6.3. Example 3

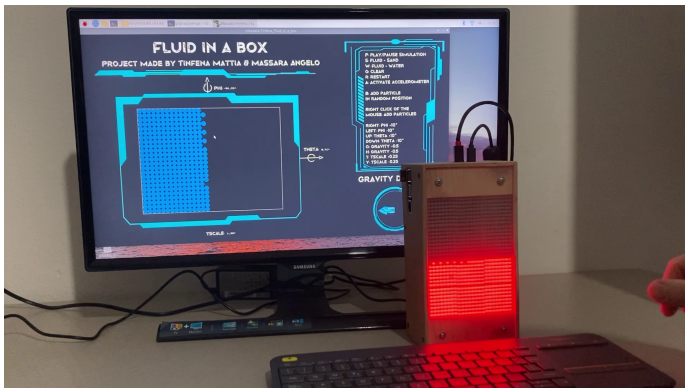

Fig. 6.4. Example 4