Pedro Inácio Rodrigues Pontes

Practice 11
AEDs Laboratory

Belo Horizonte, Brazil

2024

0.1 Introduction

The work in question is a virtualization of the Tower of Hanoi game. Its virtual implementation was done using the Processing Java programming language. The rules of the game are:

- Only one disk can be moved at a time
- Only the top disk of a stack can be transferred to the top of another stack or to an empty rod
- Larger disks cannot be stacked on smaller disks

The objective of this practice was to apply the knowledge about Stacks acquired in lesson No. 11 of AEDs. It is noticeable that the tower functions very similarly to a stack because only the most recent/top element of the tower can be moved, while older/lower ones can only be moved after all more recent/upper ones have been removed.

0.2 Development

To create this virtualization, 4 classes were used:

- Timer
- Pillar
- Disc
- Main

0.2.1 Timer

Records the time. Main code view:

```
public void update(){
  iterations ++;
  if (iterations % framerate == 0){
    seconds ++;
  }
  if (seconds == 60){
    seconds = 0;
    minutes ++;
  }
  formatedTime = String.format("%02d:%02d", minutes, seconds);
  }
```

0.2.2 Pillar

Creates the pillars on which the disks can be stacked. The class constructor was created to be initialized by iteration and stored in a class array - this simplifies and streamlines the code that uses the attributes and methods of the 3 pillars. As there are three pillars in the game i < 3 in the iteration. Constructor view:

```
public Pillar (int i) {
    this.Width = 10;
    this.Height = 490;
    this.xPos = 142.5 + i * (142.5 + Width);
    this.yPos = 100;
  }
```

The x position of the pillars is calculated based on the calculation above, based on the values of i. In the end, three equally spaced pillars will be created.

There is a show() method to display the pillar in the graphical interface

0.2.3 Disc

Creates the discs used in the tower. Follows the same principle of initialization by iteration under some command *e.g., for* and storage in a class array. Constructor view:

```
public Disc (int i, float size){
    this.Width = size;
    this.Height = Width/3;
    this.Width -= i*10;
    this.xPos = (width/2 - Width - 5)/2;
    this.yPos = height - (Height * (i + 1)) - 10;
    this.color1 = random(255);
    this.color2 = random(255);
    this.color3 = random(255);
  }
```

Here, colors are randomly initialized and various calculations are made to define *Width, Height, xPos, yPos*, based on the parameters i and size. Size is used to randomize the dimensions of the discs, specifically the base disc, which influences the value of all others. If size were defined within the constructor, all discs would have random sizes, causing the ordering of larger discs below and smaller discs on top to be lost.

There is also a constructor for when it is necessary to initialize the class without initializing its values, except Width:

```
public Disc(float size){
    this.Width = size;
}
```

Width is initialized to be able to maintain an ordering of these discs with uninitialized attributes based on their size.

There is a show() method to display the disc object in the graphical interface

0.2.4 Main

Main class, it has 9 functions. They are described in the next subsections. Globally, 7 variables are initialized, 6 of which play an essential role in the game dynamics:

- int move
- int referencedDisc = 0
- int totalDiscs = 5
- float size = random(110,135)
- boolean win = false
- boolean menu = true
- boolean initialize = true

The Timer class, the Pillar class array, and the tower array, which is a stack that contains discs, are instantiated.

0.2.4.1 void setup()

Main function. Initializes the size of the Processing graphical interface where the code will be executed, the images and fonts, and a for loop that instantiates the pillars and towers:

```
for (int i = 0; i < 3; i++) {
  tower[i] = new Stack<>();
  pillar[i] = new Pillar(i);
```

0.2.4.2 void draw()

Main function. Calls the other functions. The code is self-explanatory:

```
void draw() {
  if (menu){
    initialMenu();
  }
  else{
    if(win == false){
      game();
    }
    else{
      finalMenu();
    }
  }

  //Verify if the player wins
  if (tower[1].size() == totalDiscs || tower[2].size() == totalDiscs){
    win = true;
  }
}
```

0.2.4.3 void initialMenu()

Loads the initial menu

0.2.4.4 void finalMenu()

Loads the final menu

0.2.4.5 void game()

Initializes the tower discs. They are initialized outside of () because it's necessary to wait for the player to choose the desired number of discs for the game. Performing this initialization only once requires a slightly more complex code. Follows:

```
if (initialize) {
  for (int i = 0; i < totalDiscs; i++) {
    tower[0].push(new Disc(i,size));
    }
  initialize = false;
}
```

The timer is activated until the end of the game. Pillars and discs are displayed. If a disc is clicked by the mouse, it will follow it. This is done by the following code, which is directly related to the () function:

```
Disc d = tower[referencedDisc].peek();
d.show();

// The disc follows the mouse after it is mousePressed
if (move == 1) {
    d.xPos = mouseX - d.Width/2;
    d.yPos = mouseY - d.Height/2;
}
```

0.2.4.6 void moveDisc()

Called within (). Codes that allow the disc to follow the mouse after being clicked and move to a specific tower when there is a subsequent click in its area. Follows the code:

```
if (initialize == false){
  Disc d = tower[referencedDisc].peek(); // Disc that is being referenced
  Disc[] disc = new Disc[3];

  // Create an instance of peek disc for each tower
  for (int i = 0; i < 3; i ++){
    if (!tower[i].isEmpty()){
      disc[i] = tower[i].peek();
    }
    else{
      disc[i] = new Disc(size);
    }
  }

  // The disc follows the mouse after it is mousePressed
  for (int i =0; i < 3; i ++){
    if (mouseX >= disc[i].xPos && mouseX <= disc[i].xPos + disc[i].Width &&
        mouseY >= disc[i].yPos && mouseY <= disc[i].yPos + disc[i].Height) {
      move = -move;
      referencedDisc = i;
      d = tower[referencedDisc].peek();
    }
  }

  // Reposition the disc after a click in an allowed location
  if (mouseX >= d.xPos && mouseX <= d.xPos + d.Width &&
      mouseY >= d.yPos && mouseY <= d.yPos + d.Height) {
    for (int i = 0; i < 3; i ++){
      if (move == -1 && mouseX >= pillar[i].xPos -30 && mouseX <= pillar[i].xPos +
          mouseY >= pillar[i].yPos && mouseY <= pillar[i].yPos + pillar[i].Height &
        tower[referencedDisc].pop();
        d.xPos = pillar[i].xPos - d.Width/2 + 5;
        d.yPos = height - (d.Height * (tower[i].size() + 1)) - 10;
        disc[i] = tower[i].push(d);
        referencedDisc = i;
      }
    }
  }
}
```

The code below the comment *// Reposition the disc after a click in an allowed location* is fundamental for the entire program because it manages the algorithm that enables the movement of a disc from one tower to another, managing the involved stacks. In terms of importance, it is the topmost code. Essentially, it removes the disc from the tower it originated from, sets its x position to the center of the chosen tower to be placed, its y position above the topmost disc of the chosen tower, adds the disc to the placed tower, and updates *referencedDisc* for the code to execute correctly.

0.2.4.7 void changeTotalDiscs()

Called within (). Generates the total number of discs and buttons to increase or decrease them. Appears only in the initial menu.

0.2.4.8 void startGame()

Called within (). Generates the Play button and subsequently starts the game after it is clicked. Appears only in the initial menu.

0.2.4.9 void mousePressed()

Allows (), (), and () to be called only when the mouse is clicked. It's worth noting that these functions also have internal conditions for execution.

0.3 Results

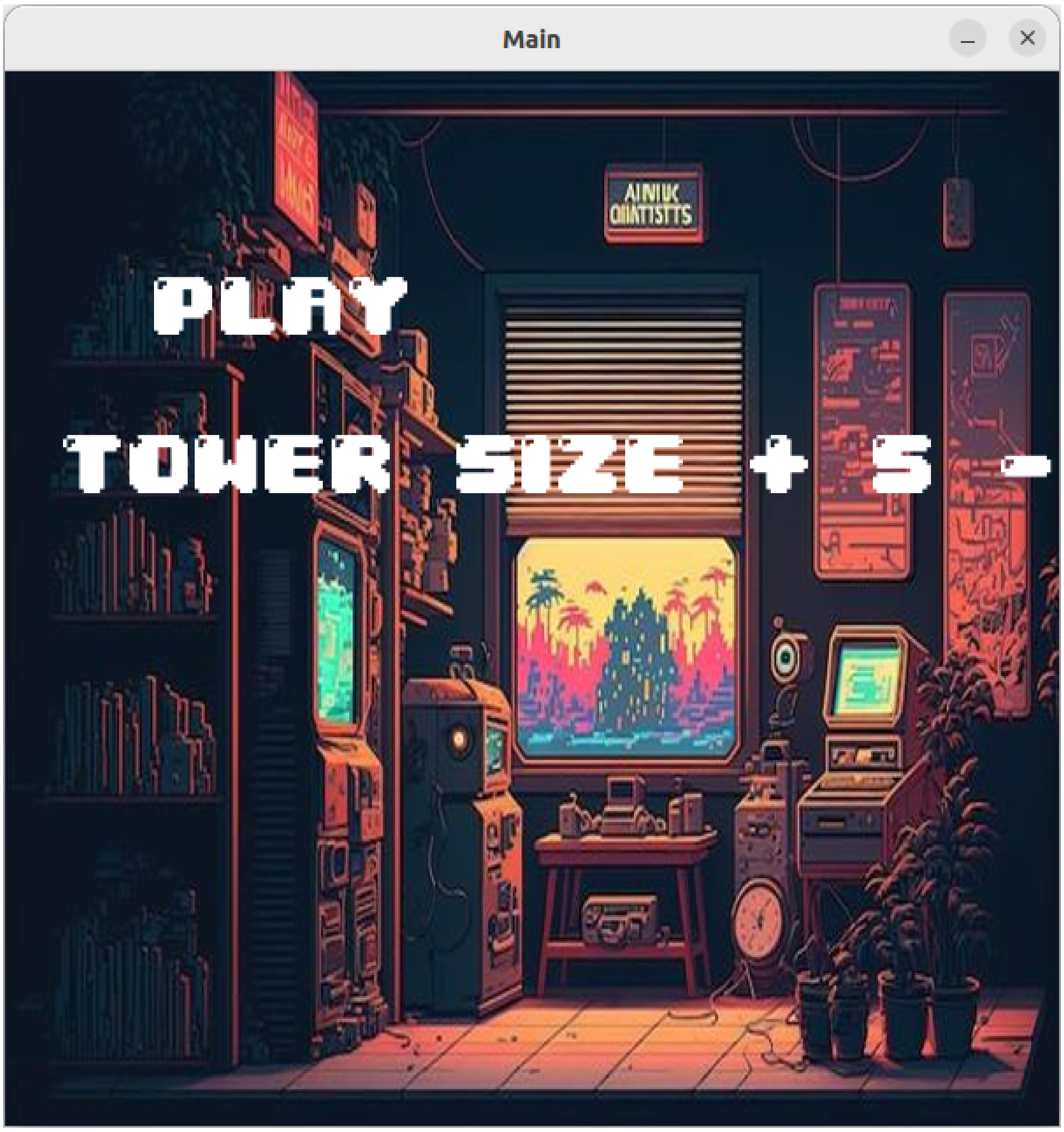

Figura 1 – Initial Menu

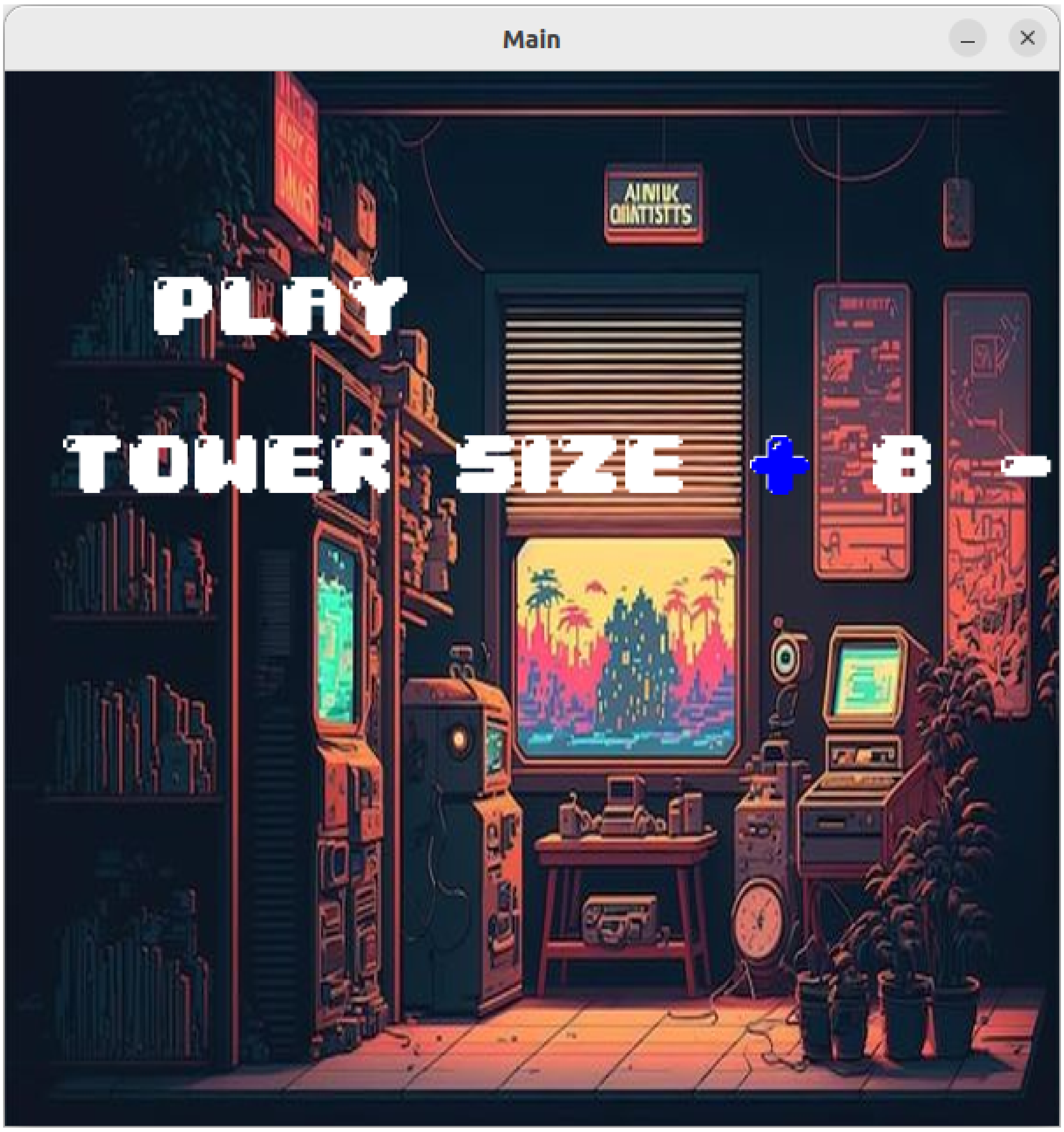

Figura 2 – Button to increase tower size working

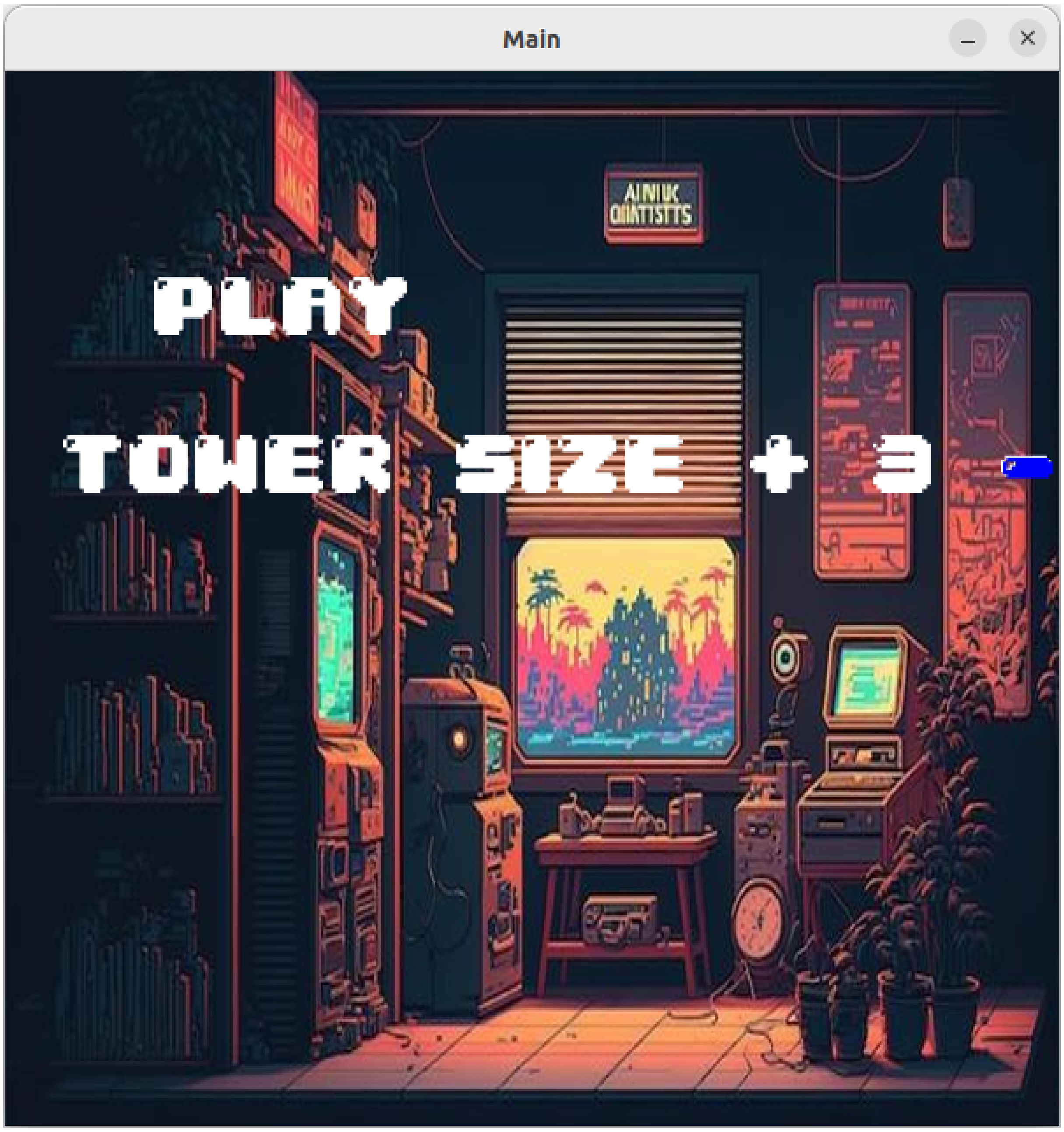

Figura 3 – Button to decrease tower size working

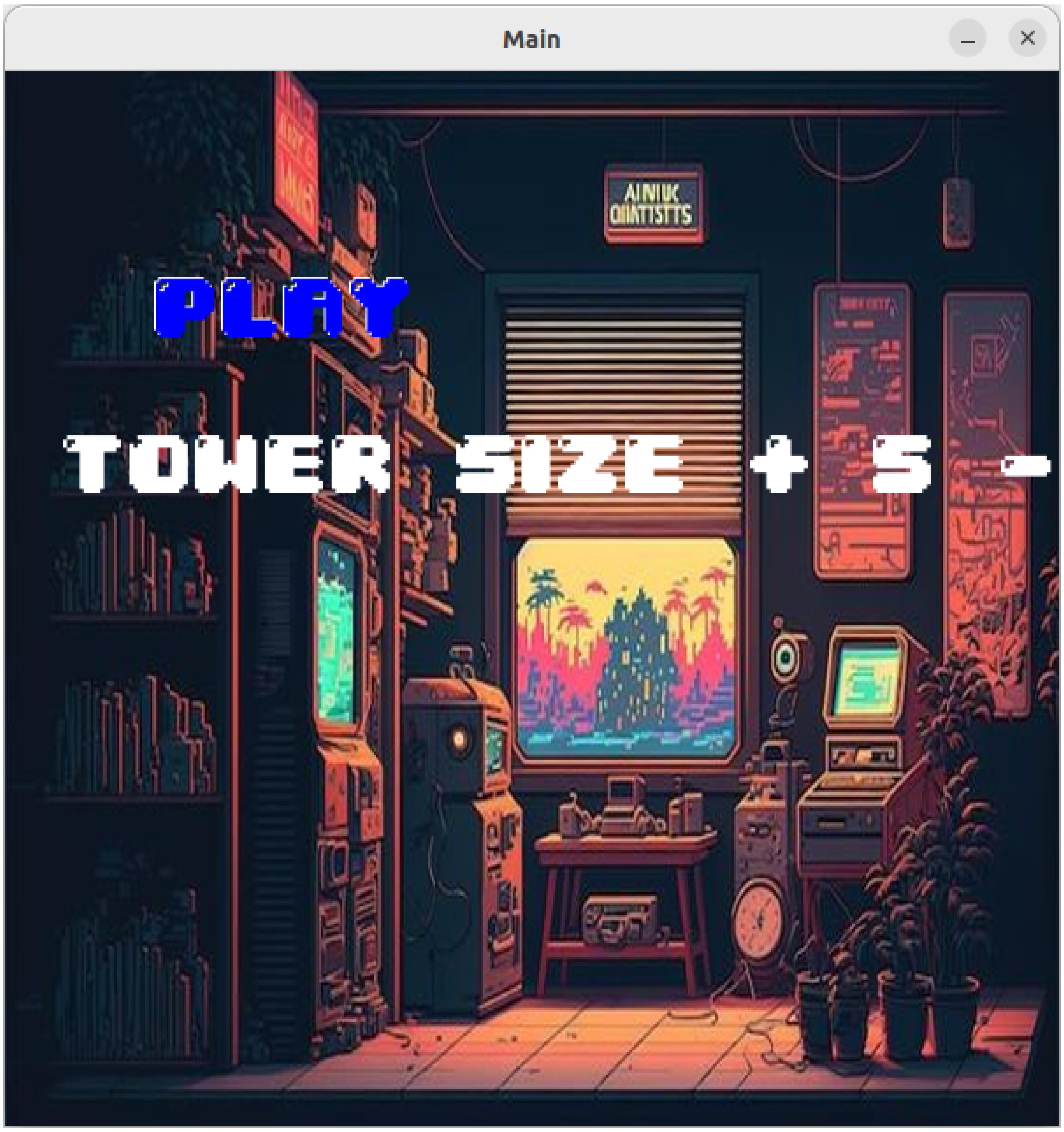

Figura 4 – Play button working

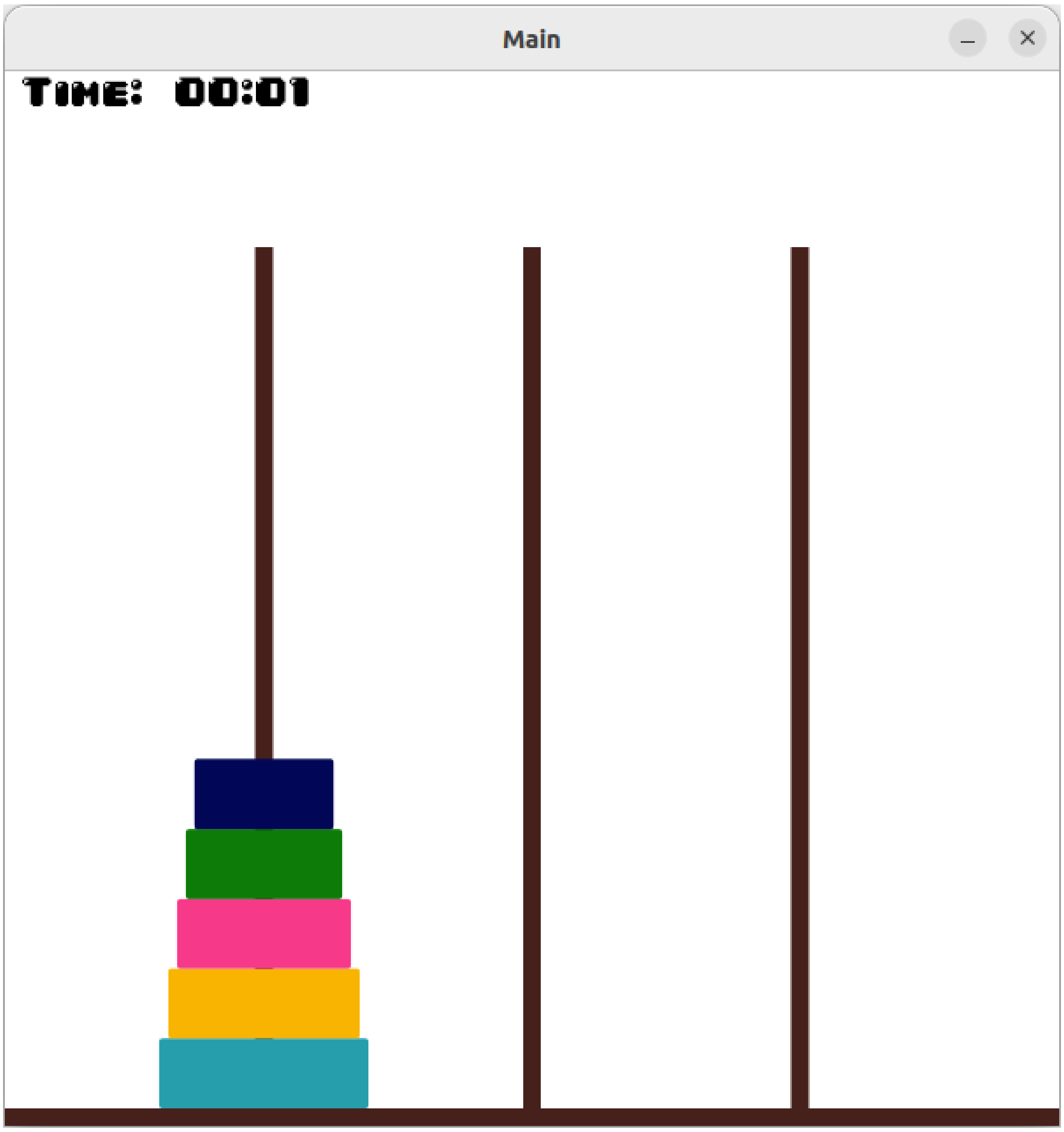

Figura 5 – Initial view of the game

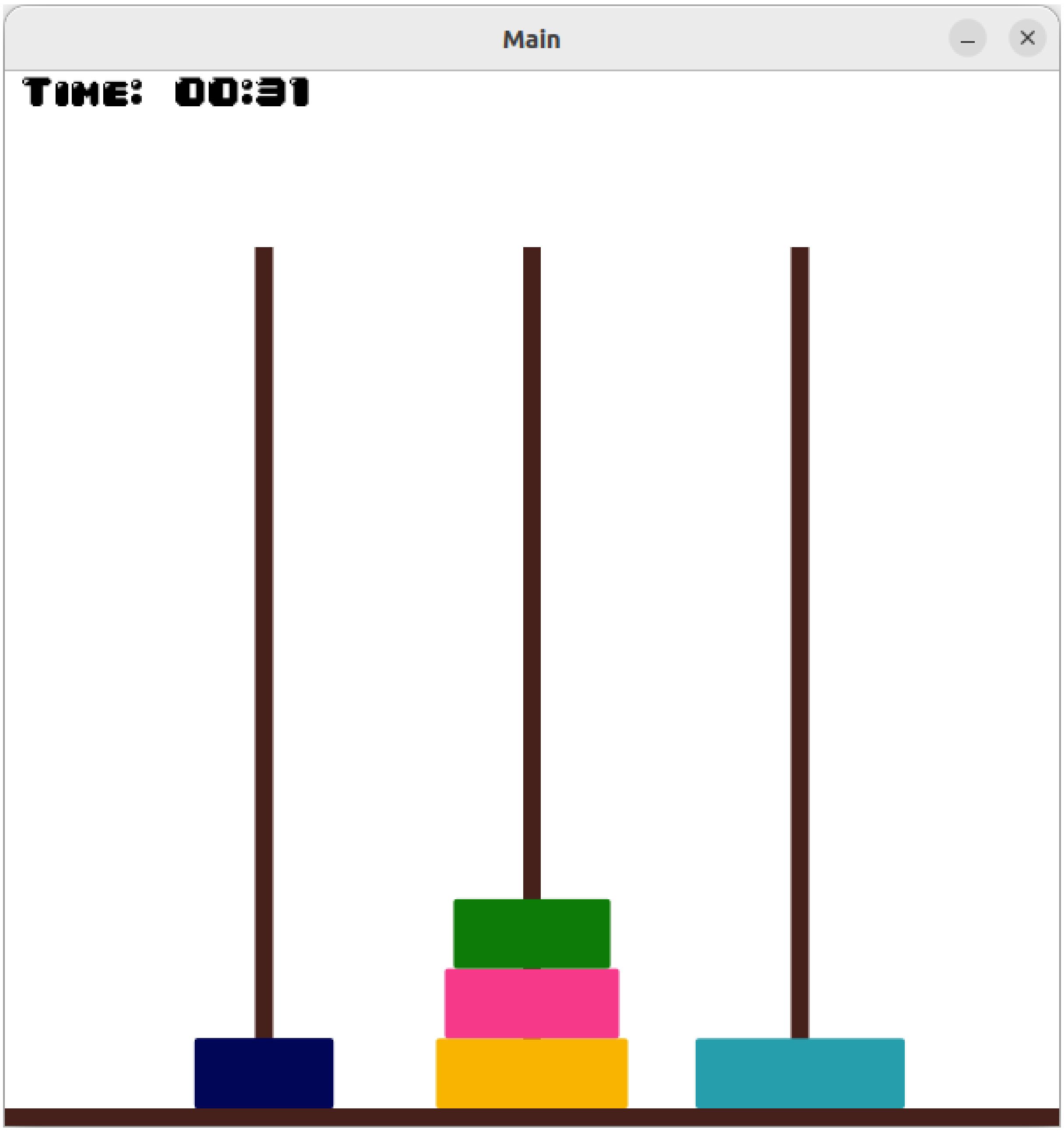

Figura 6 – Moved discs

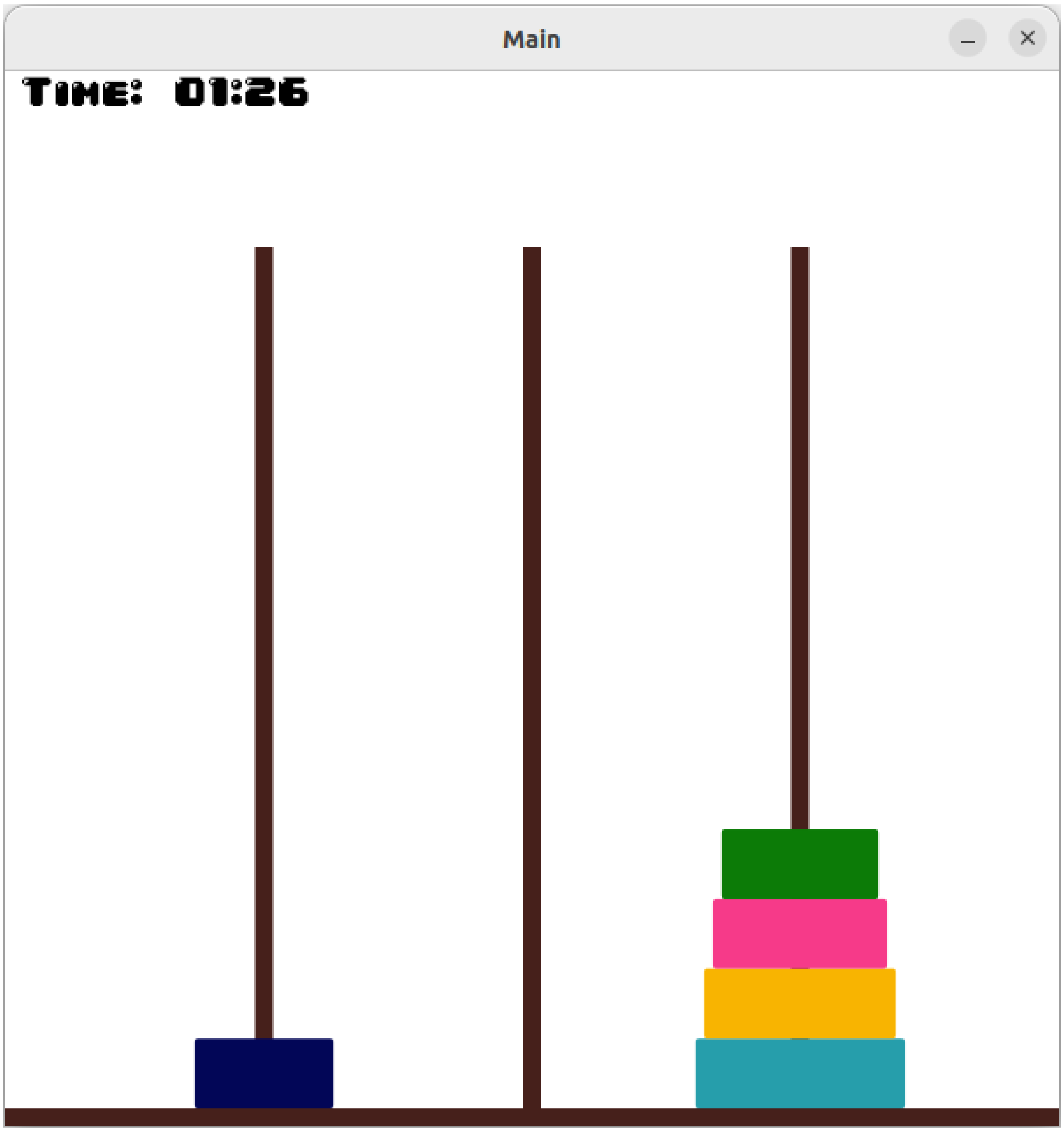

Figura 7 – Nearly ordered tower

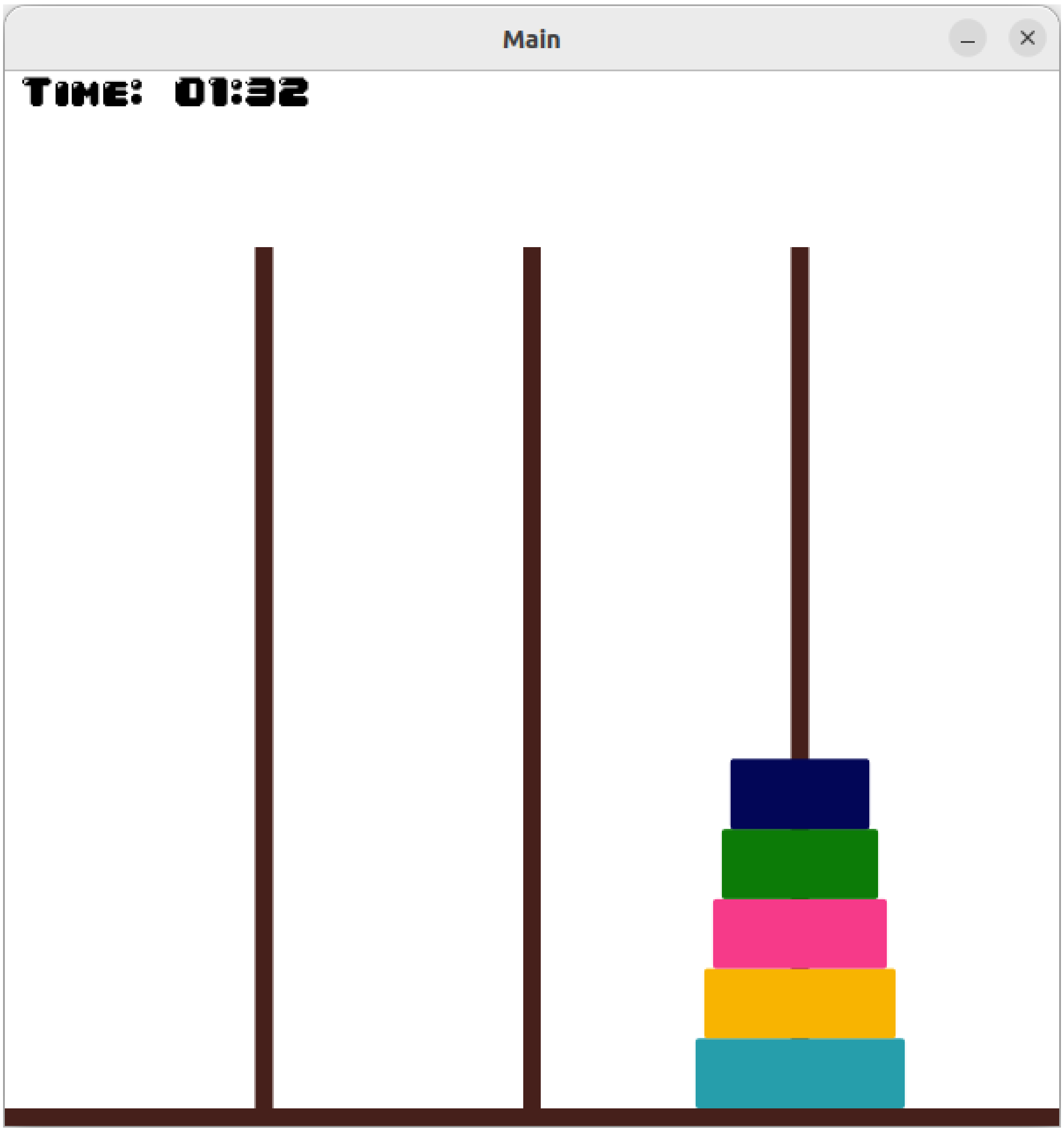

Figura 8 – Ordered tower (moment before the appearance of the final menu)

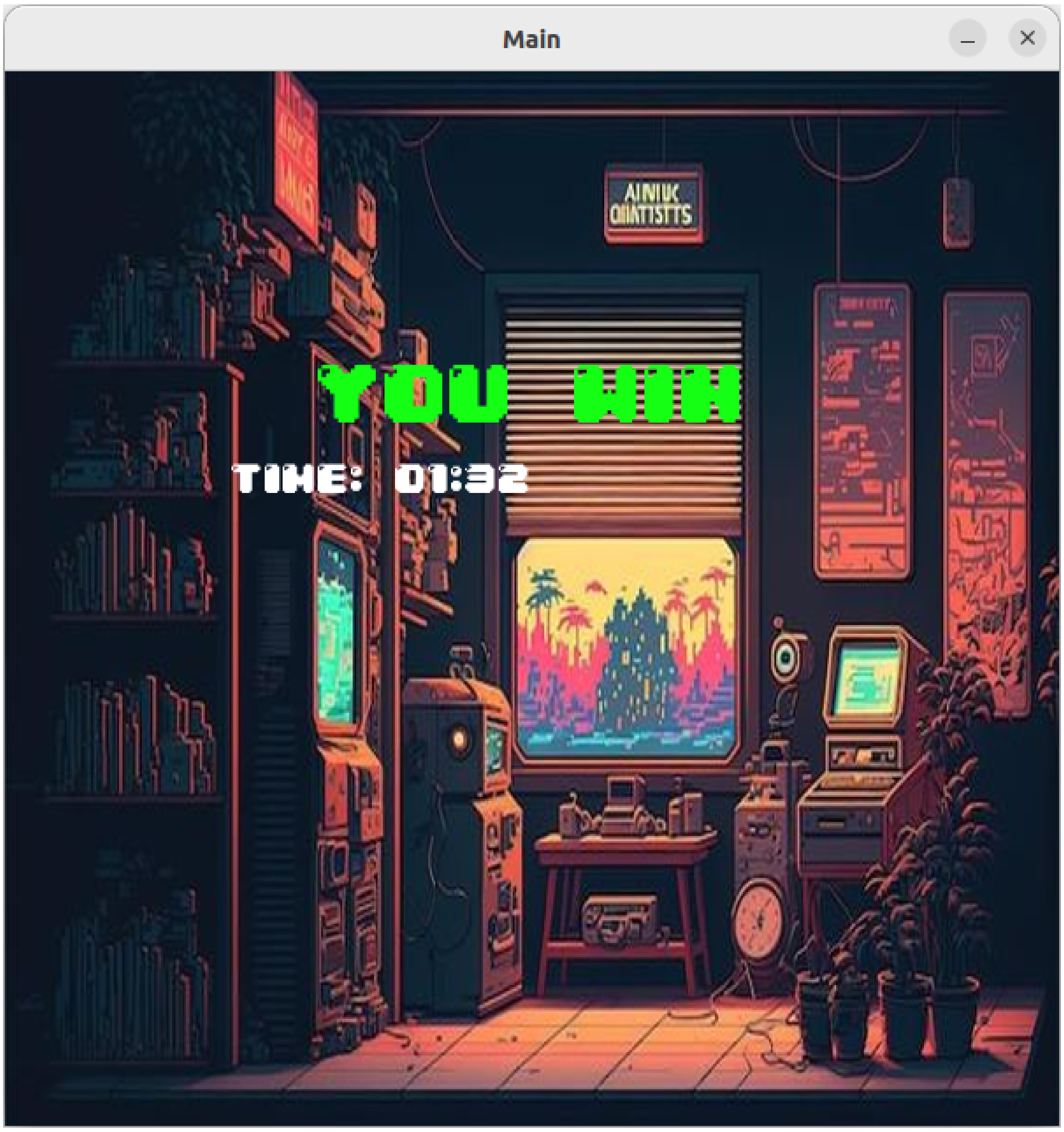

Figura 9 – Final Menu

0.4 Conclusion

- All results are as expected. The game was created correctly by implementing its rules and stacks to store the discs in each tower. The practical learning on stacks was also achieved.
- The main difficulty during the program execution was creating the *moveDiscs()* function to manage the towers and discs. There were several failed attempts until achieving an abstraction that worked correctly. Creating an *index referencedDisc* and the 3 instances

of *peek* disc for each tower was revolutionary for the correct functioning of the function and code. Another difficulty worth mentioning was creating the Disc constructor planned for use within a *for* loop, as the calculations of disc dimensions and positions became too complex due to lack of planning and poor implementation.

- Based on the completion of this experiment, the greatest learnings were the need to plan a code before its creation - which would have avoided the difficulty in creating *moveDiscs()* and the calculations of the Disc constructor - the maximum use of 1 hour and 30 consecutive minutes for focused periods of work (after this, everything becomes confusing and irritability increases), and the improvement in the ability to abstract and use stacks.