LOYOLA UNIVERSITY CHICAGO
COMP 450 – Microprogram/Microprocess – Spring 2015

Tiago de Almeida
tdealmeida@luc.edu

Custom Project

Flappy Birduino

This report covers the implementation of the game Flappy Birduino, a version of the popular game Flappy Bird[1] built using Arduino and an 8x8 LED Grid.
The goals stated in the Project Proposal[2] document were fully reached, and the final version of the game turned out to be pretty enjoyable. As previously foreseen, the hardware part of the project was not really difficult. The software side, on the other hand, was really challenging. More on that in the following sections.

1 Hardware

1.1 Hardware Components

The following components were used for this project:
- 1 Arduino Duemilanove
- 1 Breadboard
- 1 8x8 LED Grid
- 1 Button
- 5 220Ω resistors

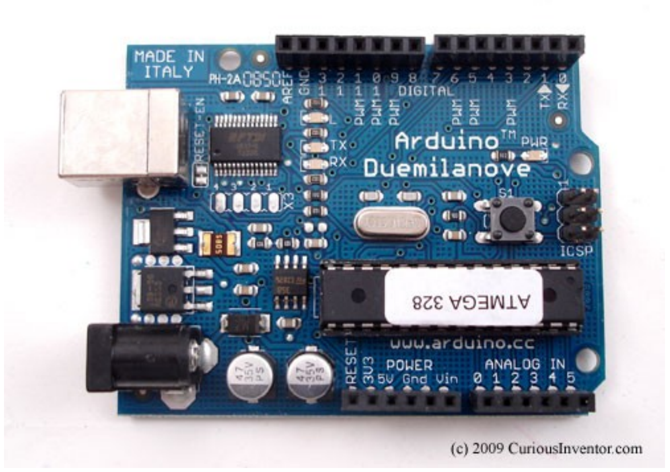

Figure 1: Arduino Duemilanove

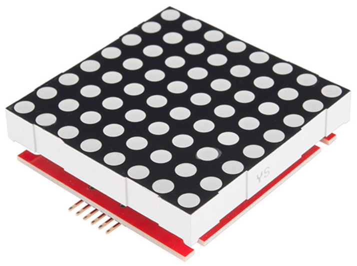

Figure 2: 8x8 LED Grid

Figure 3: Button

1.2 Schematic and Connections

Figure 4 presents the schematic of how the components were connected. Figure 5 shows the actual circuit assembled for this project.

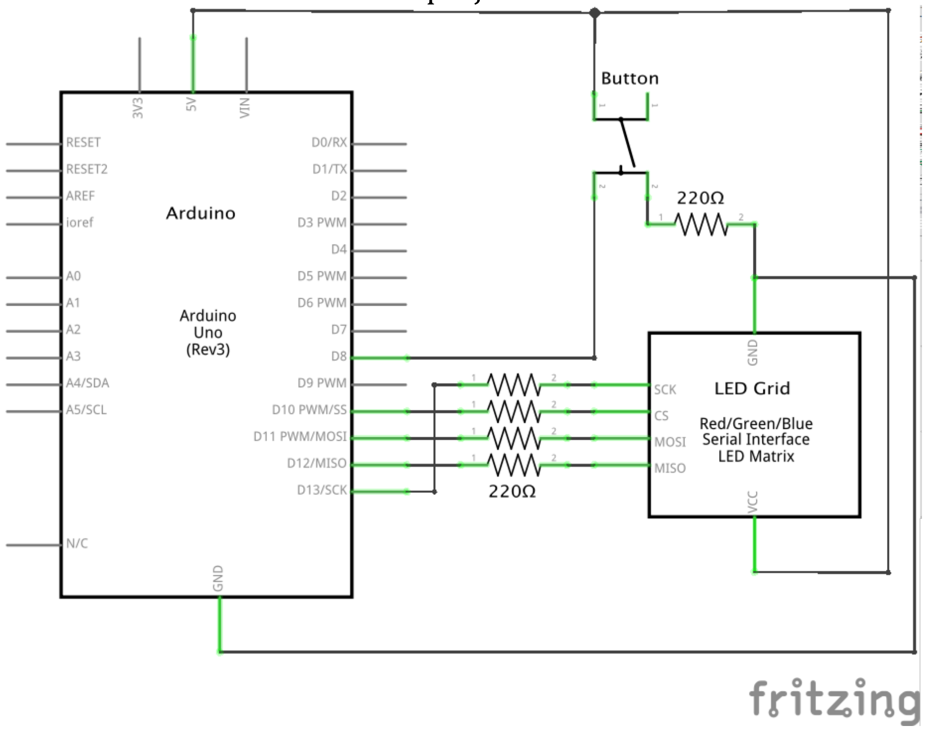

Figure 4: Schematic representation of how the components were connected.

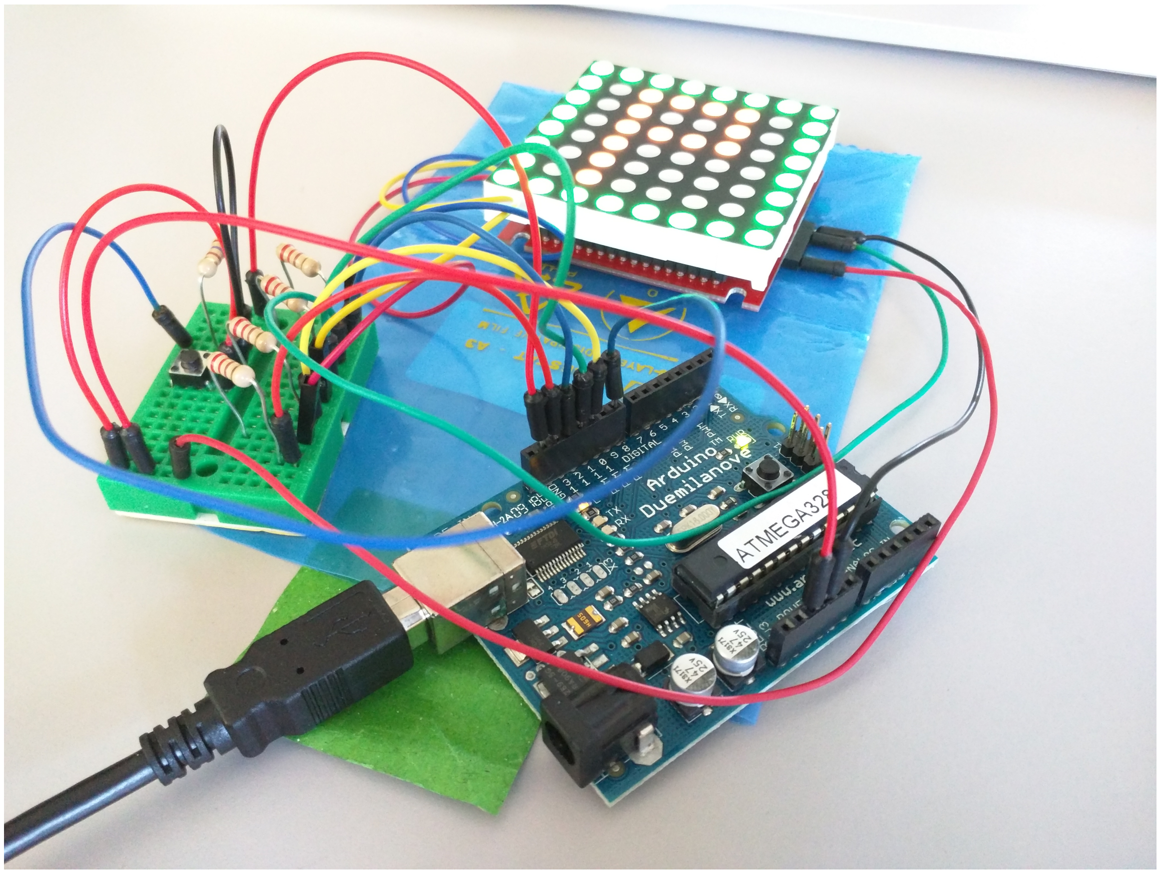
Figure 5: Actual circuit

1.3 Hardware Design Notes

On top of VCC and GND for powering the component, the LED Grid interfaces with Arduino using 4 pins. They control serial input to the Grid, which is performed by shifting data bit by bit into the component. Clock signals are used to indicate and control when input is received and when to turn the LEDs on based on the data read.

The LED Grid was connected to pins 10, 11, 12 and 13 of Arduino, according to the schematic representation (Figure 4). The button was connected to pin 8.

2 Software

2.1 Overview

The software needs to control independent parts of the game. The bird moves up and down, but doesn't move right and left. The walls, on the other hand, move from right to left only, to give the player the feel that the bird is moving through the scenario. Controlling both movements independently, while also checking whether the button was pressed, were some of the big challenges. They were done by using a really useful library for Time and Events[3] control. More on this library is explained in *2.3 Library and Useful Code*.

The software also controls the screen, that is, the LED Grid. This is done by using a 64 position buffer that represents the 64 LEDs on the Grid. Each position is an integer variable that can be set from 0 to 255. Each value technically results in a different color. In practice, however, several colors seem to repeat.

While the game is running, the whole scenario is calculated (new bird position, new walls position), updated in the screen buffer, and then sent to the screen.

2.2 Functions

2.2.1 Screen

These are the functions that control the screen:

`void sendChar(char cData)`
Sends a single byte to the screen by shifting each bit in. This is how we send data to the screen. It's used by `sendFrame()`.

`void sendFrame()`
Sends a full frame (64 position screen buffer) to the screen. Sends each position by calling `sendChar()`.

`void resetGrid()`
Turns the screen off by sending a full frame with all positions set to zero.

`void setGrid(int color)`
Turns the screen on (all LEDs) by sending a full frame with all positions set to `color`.

`void setLED(int x, int y, int color)`
Turn the LED on position (x, y) on, with a chosen `color`.

`void unsetLED(int x, int y)`
Turn the LED on position (x, y) off.

`void initialScreen()`
Sets a specific initial screen for the game. It's a green frame around a big 'P' letter. 'P' indicates that the game is 'P'aused, and the user must 'P'ress the button to start.

2.2.2 Game

Variables about the game are stored in a single struct:

```
typedef struct Game{
     GameState state;
     int score;
     float vy;
     float birdY;
     Wall wallOne;
     Wall wallTwo;
} Game;
```

`GameState state` stores one of two possible states for the game: `STARTED` or `STOPPED`.

`int score` stores the current score for the player.

`float vy` stores the current velocity on the y-axis for the bird.

`float birdY` stores the current position on the y-axis for the bird.

`Wall wallOne` stores the current position on the x-axis for the wall one, as well as a bitmap byte that represents the wall and the hole on the wall, using 0s and 1s.

`Wall wallTwo` does the same as `wallOne`, but for the second wall.

These are the functions that control some general tasks on the game:

`void startGame(boolean doit)`

Receives `true` or `false`. In case `true`, sets all initial variables for the game, such as initial position for the bird and the walls, and the game state as `STARTED`. It also generates the first two walls, and creates the event for updating the bird position every 50 milliseconds, as well as calling the first wall after 2500 milliseconds, and the second wall after 3300 milliseconds. In case `false`, it sets the game state to `STOPPED` and stops the events that update the bird and the walls position.

`void reactToUserInput()`

Checks whether the button was pressed, and acts accordingly, by starting the game in case it was stopped, or by lifting the bird position, in case the game is running. It's called every 30 milliseconds by the Time and Events library.

`void explode()`

Provides a visual feedback to the user when the game is over, by simulating an explosion on the screen, changing the whole screen from blue to red really fast several times.

`void gameOver()`

Writes score to the screen and calls `startGame(false)` to stop the game.

`void setScore(int score)`

Reads the player's score from `Game.score` and writes it on the screen, by calling `setNumber()` for each digit.

`void setNumber(int pos, int number)`
Writes a specific `number` to the screen, on the specified position `pos`.

2.2.3 Bird

void drawBird(int direction, byte yHead)
Receives the bird's head position, and draws it on the screen. It also draws the bird's tail, according to `direction`.

void updateBirdPosition()
Applies a gravity component to the bird's velocity, to make it fall faster. Updates the new bird's position according to its current velocity. Defines the bird's direction, according to the new position. Calls `drawBird()` to update the new position to the screen.

2.2.4 Walls

`void drawWall(struct Wall *wall, byte x)`
Draws `wall` on the screen, on position `x`.

`void eraseWall(struct Wall *wall, byte x)`
Erases `wall` from the screen, on position `x`.

`void moveWall(Wall *wall)`
Updates new position of `wall`. If the new position is off the screen, creates a new wall on the initial position. Checks whether the wall just slammed into the bird. If that's the case, calls `explode()` and `gameOver()`. If not, increments the score.

`void startWallOne() and void startWallTwo()`
Calls the `every` function from the Time and Events library to handle `moveWallOne()` and `moveWallTwo()` every 200 milliseconds.

`void moveWallOne() and void moveWallTwo()`
Calls `moveWall()` for each `wall`.

`byte generateWall()`
Generates a new wall with a randomly sized and randomly positioned hole. Represents the new wall on a bitmap byte.

2.3 Library and Useful Code

The Time and Events[3] library was really useful for this project. It allowed me to control several portions of the software concurrently, as this was fundamental for the game. The functions I used from this library were the following:

```
gTimer.after(2500, startWallOne);
gTimer.after(3300, startWallTwo);
```

Allowed me to define when a function call would be performed. The code continued to run, and I could rest assured that the specified functions would be called later, after the time I also specified.

```
gTimer.every(30, reactToUserInput);
gTimer.every(50, updateBirdPosition);
gTimer.every(200, moveWallOne);
gTimer.every(200, moveWallTwo);
```

Allowed me to specify repeated calls to specific functions in the background. I just had to set the interval, and the functions would be called every specified milliseconds.

```
gTimer.update();
```

Placed inside the main loop(), it updates all created events and allows for their concurrent execution.

```
gTimer.stop(gUpdateEvent);
gTimer.stop(gMoveWallOneEvent);
gTimer.stop(gMoveWallTwoEvent);
```

Stop the specified events. Used when the game stops.

Useful code that can be reused was also written. The following functions, already mentioned and explained on *2.2.1 Screen*, are useful whenever one is using the LED Grid:

```
void setLED(int x, int y, int color)
void unsetLED(int x, int y)
void setGrid(int color)
void resetGrid()
```

2.4 Best Code

Some of the best parts of the logic applied to this project's code have to do with managing the bird and the walls movement. For example:

2.4.1 Bird's Free Fall

Most implementations of this game using the LED Grid that I was able to find online moved the bird in a constant speed down the screen. This is not the same behavior as in the original game, where the bird actually feels the effect of gravity and accelerates towards the ground. To imitate such effect on my implementation, the y-axis of the screen is simulated as having a thousand positions, and not only 8 as on the screen. This is done by representing the bird's Y position in a float variable that goes from 0 to 1. This way, the bird's initial position is 0.5, and I add a gravity component of 0.005 to the bird's velocity every time its position is updated. As a result, the bird's velocity is increased by 0.005 every 50 milliseconds.

```
// initial position (simulated screen size 0..1)
gGame.vy += kG; // apply gravity to velocity
gGame.birdY -= gGame.vy; // calculate new y position
```

The button's lift effect is created by setting the bird's velocity to -0.05 when the bird is falling, or adding this value when the bird is already going up.

```
if (buttonState == HIGH){
     if (gGame.vy > 0)
          gGame.vy = kLift; // initial bounce
     else
          gGame.vy += kLift; // keep adding lift
}
```

Using this wide range of positions allowed me to create this natural effect of free fall, as the bird goes down faster and faster. When writing the bird's position to the screen, all I need to do is to multiply the float variable by 7, and I'll have the translation to the screen's range of positions.

```
// convert to screen position
byte ypos = 7 * gGame.birdY;
```

2.4.2 Bird's Tail and Direction

The bird is built by two parts (two LEDs), the head and the tail. This helps to give a sense of motion to the player. The tail moves in a sort of delayed, looser way. When the head is moving up or down, the tail is one LED "behind". When the head is moving up and stops, reaching the point where it starts falling down again, the tail goes up one LED further than the head. When this happens, the player knows that the bird is about to start falling again. All of this makes the game more natural and enjoyable.

To make this happen, every time the bird's position is updated, the direction it is going to is defined:

```
if(abs(oldY - gGame.birdY)<0.01) direction = STRAIGHT;
else if (oldY < gGame.birdY) direction = UP;
else direction = DOWN;
```

When drawing the bird, the tail's positions is defined as:

```
yTail = constrain(yHead - direction, 0, 7);
```

2.4.3 Generating Random Walls

First, the size of the hole on the wall is randomly generated, from 3 to 6, and expressed as a bitmap. Then, the position of the hole is also randomly selected, and then also expressed as a bitmap. At this point, the wall with the hole is generated and ready to be placed in the game.

```
byte generateWall(){
     byte gap = random(3, 6); // size of the hole
     byte punch = (1 << gap) - 1; // hole’s bitmap
     byte slide = random(1, 8 - gap); // hole's offset
     return 0xff & ~(punch << slide); // the wall
}
```

2.4.4 Moving Walls

Moving the walls on the screen and detecting when they hit the bird was not the biggest challenge. However, the code was built modular enough so that the `moveWall()` function turned out to be really well organized and readable.

The first part detects whether the wall has come past the screen. In this case, it erases the wall from the position 0, and generates a new on, placing it in the initial position. If this is not the case, the wall is erased and written now in the new position.

```
void moveWall(Wall *wall){
     if (wall->xpos == 255) { // wall has come past screen
          eraseWall(wall, 0);
          wall->bricks = generateWall(); // new wall
          wall->xpos = FIRST_WALL_POS;
     }
     else if (wall->xpos < FIRST_WALL_POS) // still visible
          eraseWall(wall, wall->xpos + 1);

     drawWall(wall, wall->xpos); // draws on new position
```

The second part checks whether the wall has hit the bird:

```
     // check if the wall just slammed into the bird.
     if (wall->xpos == 2) { // bird’s X position
          byte ypos = 7 * gGame.birdY; // convert to screen
          if (wall->bricks & (0x80 >> ypos)){ // collision
               explode();
               gameOver();
          }
          else gGame.score++; // no collision: score!
     }

     wall->xpos = wall->xpos - 1;
}
```

3. Conclusion and Final Result

I'm really satisfied with the final result of this project. The game turned out to be really enjoyable and, dare I say, addicting. Every time you play, you want to make a better score.

The use of a big range of positions and a gravity component was fundamental to give the player a sense of natural movement. The tail was also really helpful to create such sensation.

The code is well organized and modular. Basically every basic unit of logic has its own function, making the code extremely readable. The dynamics of all movements on the screen is working better than I expected.

Please watch this short video to see the project working:
https://youtu.be/GQJLi_nnIf4

The code can be found here:
https://github.com/almeidainf/Arduino_Projects/tree/master/Flappy_Birduino

4 Problems / Questions / Future Work

The main problem to solve for this project was the management of concurrent tasks. Fortunately, the Time and Events[3] library was really helpful and made this an easy task.

Learning how to use the LED Grid was easy using the example[4] provided on Arduino's website. It provides the `sendChar()` and `sendFrame()` functions, as well as what needs to be set up in the `setup()` function.

The basic functionality of the game is done. So as future work, maybe levels could be implemented. For example, one starts playing in an easy level, and progresses for harder levels. Level's difficulty could be changed by changing the size of the hole in the wall, the speed in which the walls and the bird move, the proximity of the walls, etc.

5 References

[1] Flappy Bird Game: http://flappybird.io/
The original game was available only for mobile devices, and it was a huge success. This is a web version of the game. It seems to work exactly like the original.

[2] Flappy Birduino Project Proposal: https://tinyurl.com/n3udtgs
My project proposal for this game, before I started working on it.

[3] Time and Events library for Arduino: http://playground.arduino.cc/Code/Timer
Extremely useful library for controlling concurrent tasks on Arduino code.

[4] LED Grid Getting Started: http://playground.arduino.cc/Code/RGBBackpack
Helpful information on getting started with the LED Grid. Also useful code to interface with the grid.

[5] augustzf's Flappy Bird implementation: https://github.com/augustzf/arduino-flappy-bird
This Flappy Bird implementation for Arduino was extremely helpful for my project. I borrowed a lot of the smart components of its logic. I also discovered the so helpful Timer library from this code. However, I changed some things and added others. It uses a library to control the Grid, and I found that this library was not necessary for me, so I took it off and implemented the communication with the LED Grid on my own. I also added the score reporting, ported the code to Arduino Duemilanove, and changed/added a bunch of different things throughout the code.

Version 1.0 (original)
Tiago de Almeida, April 25, 2015