Marioscii: An Interactive Ascii-Based Platformer

Patrick Huston - February 26, 2015

Project Overview

Marioscii is an interactive ascii-graphics platformer that the user interacts with and controls through auditory and motion inputs. The intention of the game is to engage the player in a new, novel, and physical manner.

Results

Marioscii successfully implements motion-detection and audio triggered controls to navigate through ascii-based maps. The character, ‘Mario’, navigates left and right based on the coordinates of the average movement detected by the webcam. To jump, a threshold volume level must be broken. When the game is started, the user is greeted with a simple yet easy to understand menu:

Combining the two inputs described above, the user must navigate ‘Mario’ through several different levels. To complete a level, the user must reach the target tile, indicated in red. When this tile is reached, the next level is rendered immediately, and the character starts from the last coordinates.

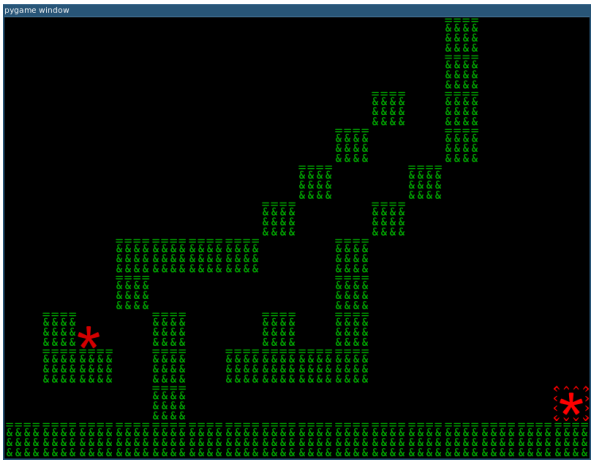


Fig. 2: Level 1

Fig. 3: Level 2

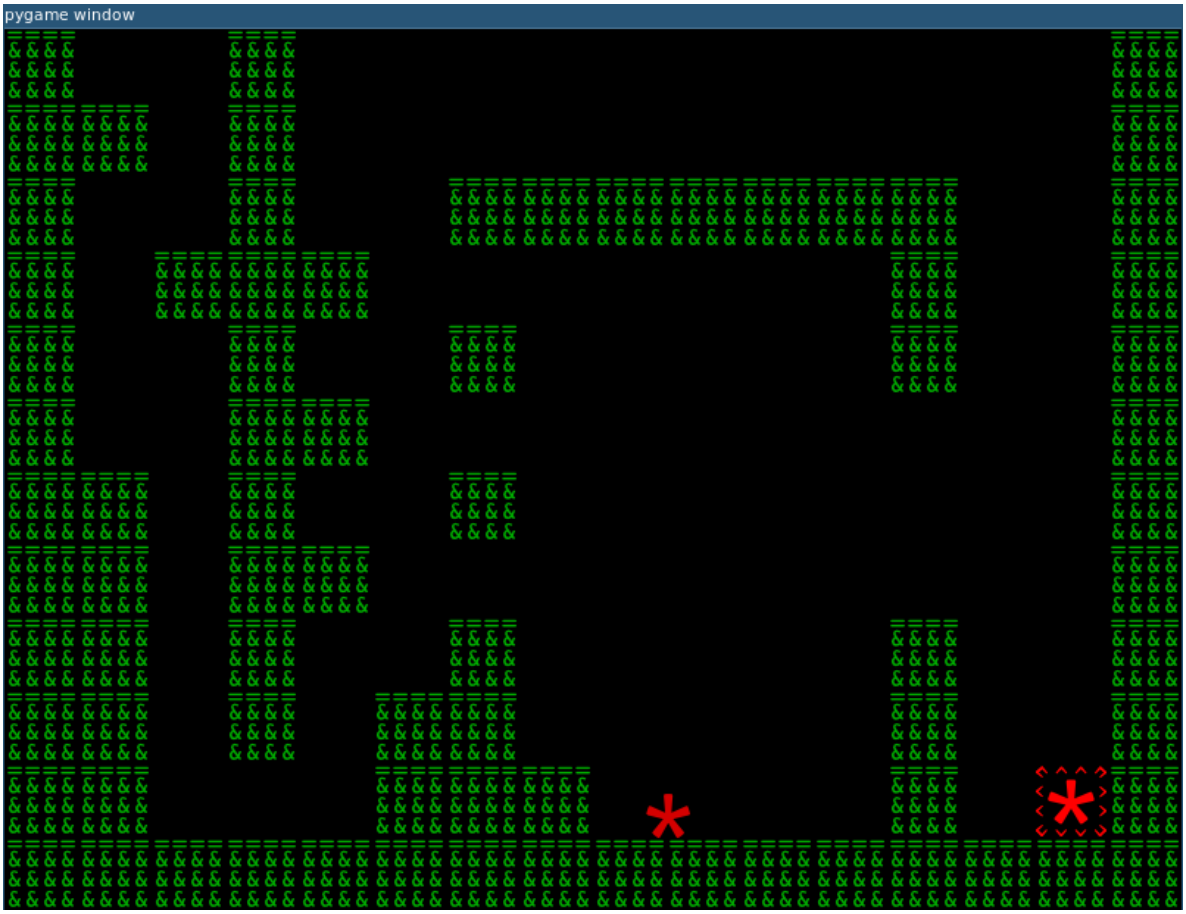


Figure 4: Level 3

In the experience of playing Marioscii, the intention is for the user to get increasingly engaged in the game, and in the process look progressively more and more silly waving their arms and yelling at their computer. Through several playtests, we saw that this goal was achieved.

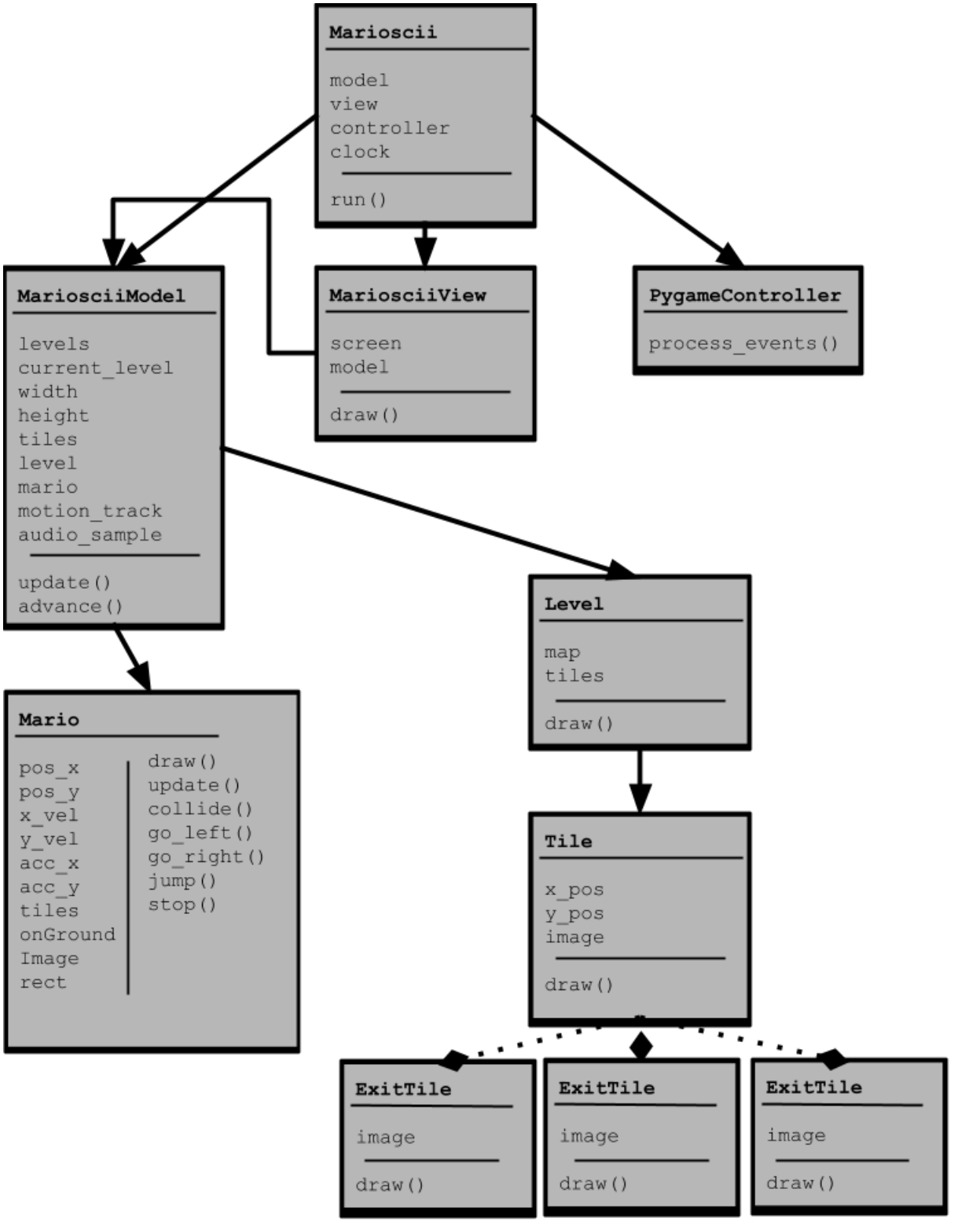
Implementation

The overall structure of Marioscii follows the Model-View-Controller paradigm, which is composed of three main components. The first component is the model, which houses all of the components of the game, their pertinent state variables, and their functionalities. In this case, the model holds information about the character, Mario, the levels, and the tiles. Encompassed in the character class for Mario is information about Mario's current state, and methods that hold directions for updating the state of Mario based on different input events. The next component is the view, which is the portion of the program that is responsible for creating the correct view states for the components in the model. The final component is the controller, which is responsible for listening for events, and responding to these events to change the state of the game.

To create novel and fun inputs for the game, we implemented OpenCV and Alsaaudio to allow the user to control the game using waving motions and yelling. The audio sampling is straightforward - the implementation simply listens for a set period, reads in a set of data, and takes the root mean square to get a sound level for that period of time. The algorithm for motion detection, however, is slightly more intricate. To determine whether the subject in the webcam stream was moving, two frames are read in, converted to numpy arrays, and then subtracted from each other. The resultant frame is a mix of colors and white portions, which represent moving edges. To extract just the moving edges, all colors other than white are filtered out, resulting in frames that only show edges in motion in the frame. Next, for each of these masks, the entire image is shrunk using the implementation of laplacian pyramids built into OpenCV. Finally, each of these shrunk masks is iterated over to get the entire count of white pixels, and thus, determine the coordinates of the average motion. These coordinates can then be utilized to determine whether the user is moving their hands on the left portion of the frame, or the right portion.

One major design decision we had to make was how to generate an ASCII-looking game environment. We first thought about using actual text inside the command line to play the game, but deemed that too complicated to implement. We could have also used the third-party library "pygcurse" to generate the ASCII graphics. Pygcurse is meant for generating text adventure and roguelike games, and it supposedly makes it easy to turn text characters into graphics. It took a while to learn how to use pygcurse and to get it running with a basic platformer game, as there is very limited documentation on how to use the library. After days of toiling with pygcurse, we decided to just use pygame. Instead of generating graphics directly from text, we had to have pictures of text and have pygame display the pictures. Another major design decision revolved around choosing between a tiled map implementation (which uses arrays filled with values representing different tile types), or simply drawing each component separately. Ultimately, the advantages of a tiled array implementation shone through: it's clean, it's sexy, and it's incredibly easy to test out new levels with little to no headache.

UML Diagram


Marioscii
model
view
controller
clock
run()
MariosciiModel
levels
current_level
width
height
tiles
level
mario
motion_track
audio_sample
update()
advance()
MariosciiView
screen
model
draw()
PygameController
process_events()
Level
map
tiles
draw()
Mario
pos_x
pos_y
x_vel
y_vel
acc_x
acc_y
tiles
onGround
Image
rect
draw()
update()
collide()
go_left()
go_right()
jump()
stop()
Tile
x_pos
y_pos
image
draw()
ExitTile
image
draw()
ExitTile
image
draw()
ExitTile
image
draw()

Reflection

Overall, we are satisfied with our final product. It was disappointing that we couldn't get pygcurse to work, but our final product is definitely above our minimum viable product threshold. We had our own methods for unit testing different functions, as we had to tweak certain values of velocity and motion sensitivity to make the game run smoothly. We learned that using a poorly documented third-party library is not always the best idea, especially with beginner knowledge of python. Going forward, we will probably use libraries with caution.

We divided our work by working on separate implementation of the code, with one person working in pygame and the other working in pygcurse. Some issues that arose were merge conflicts if we both iterated on the same piece of code but with different libraries, and we addressed them by working on different branches and keeping each other updated on what stage we were at.