Wakka Wakka - Educational Tool Extraordinaire

Written Report

Hugh Dowdy

December 12, 2014

This program utilizes a Command Prompt (cmd.exe) like interface (we'll call it fake CMD) to familiarize inexperienced users with a preliminary feel for using cmd or Terminal (the Mac equivalent). The user is exposed to a few Computer Science principles during the fake CMD portion, after which the user can play a game I created which is a mix of Pac-Man and Snake. I'm well aware of the fact that the time frames I give throughout this document are extremely vague, but this is a brief synopsis of the process of developing the program and as such is relatively restricted to particularly pertinent and demanding aspects of the project. I would advise looking through the code as opposed to reading this, because the processes are thoroughly documented and commented.

Initial Skeleton

------------------------------------------------------------------------------------------------------------------------

Setup()

Open CMD

loop (to acquire Ghost color, sprite image filename, buttons to bind)

print prompt

store user input

generate images

store in more accessible location for quick reference

------------------------------------------------------------------------------------------------------------------------

Draw()

Initiate

loop

draw using current location

take input

calculate new player location

calculate new delta between Player and NPC

calculate new NPC location w/ intentional error

store new data

collision check

------------------------------------------------------------------------------------------------------------------------

The final code ended up surprisingly similar to the skeleton, though substantially more sophisticated, including timers, triggers and nested classes.

Primary Difficulties

Opening Command Prompt

After a little research, it became evident that it would be rather difficult to pass values to and from cmd.exe, starting with a complicated process just to open the program.

Text positioning

Due to the fact that the solution to using cmd.exe was to generate a similar GUI that redrew each frame continuously, there was an innate issue with positioning in relying on simple commands like text(String,int,int). Particularly with respect to trying to form some resemblance to cmd.exe. A significant amount of time was dedicated to calculations, as well as trial and error; primarily concerning determining font widths and heights, and appropriate positioning based on these factors.

Jagged movement

As with any graphics processing application, frame rate and processing speed come into play. With heavily nested sections, a single iteration can be both resource intensive and generate a discrepancy in frame refresh rate, resulting in uneven image presentation.

Ghosts developing stripes

There was a point when some ghosts were being rendered with vertical lines while others were being drawn properly. Though the code used to generate them was identical, the parameters passed to them had different values.

Integrating Programs

I had initially split the code into two segments: I/O interaction and the game. Once these programs were written I ran into difficulty getting them to reference each other, because they both utilized their own setup() and draw() I was not able to leave them both as they were when it came to uniting them.

Passing user inputs to game code

I ran into a fair amount of trouble when it came to binding keys determined by the user, and later referencing them. I had to pass variables from the I/O portion, where the key bindings were generated, to the game code, which relied on these bindings as inputs.

Collision detection unresponsive

Occasionally the collision detection will have a delayed response or not be triggered at all in a scenario when there has undoubtedly been overlap. This makes the program seem unpolished and come across incomplete. Having said that, this is not particularly frequent.

Returning values

None of the methods I made return a value, i.e. they are all void. Though this doesn't have any serious implications in this particular application, I would like to have included a few for variety of functioning/code structure (though too much variety in code structure would definitely be an issue).

Solutions

Opening Command Prompt

In order to compensate for the lack of compatibility between processing I/O and cmd.exe, I made my own mimic of the Command Prompt, which serves the purpose of perhaps removing some of the stigma or dismay that many new comers feel toward Command Prompt and Terminal.

Text positioning

As mentioned in the difficulty section, the only solution to this was thought, time, and trial and error. Between counting pixels, some simple algebra and "eyeballing it" I find that I did a decent job representing the terminal feel.

Jagged movement

My solution to this was to execute the command responsible for determining the ghosts' direction of motion less frequently. This is due to the fact that the interactions between Wakka Wakka and the ghosts is constantly changing and since the ghosts are required to move either horizontally or vertically at a given time, they would constantly be switching between the two axes of motion.

Ghosts developing stripes

This was resolved when I changed the speed of the ghosts to integers. The parameters used in rendering the ghosts are integers and as such, having a position that was defined by an int(float) expression resulted in marginal but constant discrepancies and errors.

Integrating Programs

In the end, I decided to make the splice the I/O setup program with the game program rather than just calling one inside the other. The primary, executable program more closely resembles the I/O program, because I decided to turn the game into a class that contained other classes (the rendering segments).

Passing user inputs to game code

The primary issue from this came in the form of inconsistent data types associated with keystrokes, as well as trying to keep congruence in my setup code, converting everything into strings so that I could have an array holding all the user inputs. Whether I would do this a different way in the future, or maintain the consistency, I relied on methods to convert the pertinent segments I needed from the strings. The setup code does a fairly decent job of ensuring that the correct data types are stored, but if the user does not follow the instructions, they're likely to get an error since I have implemented minimal error recovery.

Collision detection unresponsive

I made a clock for debugging using a start time, current time, and the difference between them. I was able to compare the lengths of different methods and found that essentially all of them run between 0-3 milliseconds. I found that the method for drawing Wakka Wakka took the longest at ~ 9 milliseconds, due to the fact that the vertices are determined using a handful of simple trigonometric calculations. In finding the most resource intensive I was able to relocate the collisionTest() method in an attempt to remove a bit of the lag. At best, the collisionTest() method approximates the ghosts as circles, certainly resulting as a source of error. Improving on the collision detection is included in the Additional Features section below.

Returning values

Since this is a fairly simple program that doesn't need to call on others, or be called on itself, having the methods return a value is unnecessary, though it would be another way to achieve some of the commands in the program. Currently I make alterations to some variables/execute some methods that are nested in a different location: String.class.method().

Prospective Additional Features

Decreasing power pellet size as they age

This would add another level of complexity to the game and scoring, though fairly simply implemented. Making the awarded score proportional to the size of the pellet would also instill a sense of urgency in the user.

Error recovery

Without it, an ignorant user or a misplaced finger could result in a crashed program.

Variable number of ghosts

Another aspect of the game that could add complexity and excitement, adding ghosts as the game progresses is an interesting prospect.

Changing ghost attributes with time

Including variable ghost speed could not only make the game more difficult and entertaining, it offers additional personification of the NPCs; exhaustion level for instance.

eatGhost()

Coupled with adding ghosts, having the ability to remove ghosts when they're vulnerable (blue) as in Pac-Man would substantially increase the entertainment value of the game.

Wrapping window

I inadvertently came upon this idea while testing, when my code for ensuring that Wakka Wakka stays within the bounds of the window set his position to the opposite edge of the window.

More sophisticated collision detection

It would certainly improve the functionality of the program to have a collision test that better approximates the shape of the ghosts and that falters less often or not at all.

Referenced Text

I spent a large amount of time on the processing and Java APIs, figuring out syntax, functionality, looking at examples and related methods. Most of the time I didn't know the functioning of a method, but I knew what I wanted to do and a Google search, referring to the processing and oracle websites often quickly gave me a starting point. Aside from those there are only two instances that I remember utilizing someone else’s code outside of looking at examples. I did not use anyone else's code intentionally.

Jitterbug mission

I built my Wakka Wakka class based roughly on the structure of the Jitterbug class, particularly with respect to updating position.

Blinky

I added quite a lot to the Blinky program, which I turned into the Ghost class. Perhaps the largest adjustment is determining the shortest distance to Wakka Wakka and the powerPellet() method.

Saving a keystroke to a growing string

Since I had to make my own CMD GUI and wasn't sure on how to store a temporary string or save a keystroke to a string, I was quite lucky to fall upon a forum in which someone did just that. I'd like to stress that essentially the only portion of that code that remains is the storage of a key stroke and the Boolean statements used to determine whether a depressed key is coded or the enter and backspace buttons.

<https://www.processing.org/discourse/beta/num_1223627482.html>

Taking the first character from a string

As mentioned in the difficulties section, I relied again on a forum for help in converting a string to a character. I was well aware that you can't turn a string into a single character, but I was able to find a method that takes a desired character, determined by position in the string, from a string and save it to a char. And though String.charAt(int) is a well document method, I feel conflicted in saying that I didn't get the idea from someone else.

<http://stackoverflow.com/questions/7853502/how-to-convert-parse-from-string-to-char-in-java>

My take away from this section is that StackOverflow is a phenomenal resource.

List of Methods

class WakkaWakka

void CMDScreen()

void CursorLocation()

void draw()

void Declare()

void KeyAssign()

void keyPressed()

void setup()

class Game

void endgame()

void nextScreen()

void powerPellet()

class Ghost

void ghost()

void lookyloo()

void mousePressed()

void move()

void powerPellet()

class WakkaWakkaDude

void collisionTest()

void getInput()

void move()

void oneWakka()