Project 1: Centi--- I mean Millipede Infraction Protection Simulator

Released: 11:59 PM Thursday, September 27th, 2018.

Due: ~~11:59 PM Sunday, October 28th, 2018.~~

Extended: 11:59PM Sunday, November 4th, 2018.

We will create a nice interactive game based loosely on the hit arcade game Centipede.

Introduction

Hello! Can yo—*static*—ee? Can you hear me? I'm contac*static* from the past! I'm calling from the year 1981! Is this 2018? Is the future as great as we all expect it to be? Hoverboards? World peace? Flying ca— What's that you say? You're breaking up… 2018 is a trash year?? No holographic video games… just cheap VR to feed people ads? Dang.

Well… maybe we can brighten up our future by looking at the relative simplicity of the past. There's this great game called Centipede by Ed Logg and Dona Bailey. It was originally programmed by Dona Bailey, who, when she realized her mundane work involved the same type of processor, and therefore the same assembly language, as games, went to work for Atari.

Aside: Dona Bailey gives a short description of her inspriration and process for designing the original Centipede here.

Hey! Who else knows some assembly? That's all of you! Times haven't changed much, eh?

Creating "MIPS" (in MIPS)

Here, we won't create Centipede. No no. We will create MIPS: Millipede Infraction Protection Simulator. You see, we are designing a simulation for a robot that eradicates the millipedes from our garden. That's very different from centipedes. More legs n'at.

Simply, we will create a game where the player controls this robot. The bot will be able to fire projectiles toward the millipede. Each time the millipede is struck, it will shrink. When the last segment is gone, the millipede ceases to threaten our garden (and let's just say it is magically transported away) and we win!

We will, naturally, write our program entirely in MIPS assembly. To aid us, we will, of course, make use of the system calls available to us via our MARS simulator. Furthermore, we will be using the provided LED display. You can find this under **Tools/Keypad and LED Display Simulator** in the MARS menu bar.

There are several pieces of code and logic we will be making use of, and will be described in detail further down this document:

- Random Numbers (looked at in Lab 4)

- Input Polling (Involving bitmasking, and will be provided to you)
- LED Device (Memory mapped hardware, code to interact with this is provided)

The rest will be good ol'-fashioned programming. (emphasis on ol'-fashioned)

The Game Design

Millipede

The MIPS game will be very inspired by the basic Centipede mechanics. There will be a, *ahem*, *millipede* that will navigate through the level. The level itself will be a two-dimensional grid made up of single pixels. The millipede will have a particular length (10) while being a single pixel wide and move left to right across the board.

It will start with its tail to the top-left and its head toward the right and move to the right until it encounters either an obstacle or the boundary of the level. When it encounters a collision, it will move down (ignoring anything in its path) and change direction. Each segment of the millipede will simply move by occupying the space of its adjacent segment, exactly in the manner of the original game.

When the millipede reaches the bottom and can no longer move down when it decides it needs to do so, it will instead be moved to its initial starting point (top-left) while retaining its current size. It will then move to the right under the same rules.

Level Obstacles

The level will be a 32 by 32 grid. At least 40 or so Mushrooms will be randomly placed within the field. The millipede nor the player can move into a space occupied by a mushroom. Once every second, a new mushroom at a random location is generated.

You do not have to be rigorous about ensuring that the random location is an empty space. That is, if you randomly select a location that already has a player, millipede, or mushroom, then you can simply do nothing and wait for the next second to pass.

Player Movement

The player is a single pixel that can move in any normal direction (up, down, left, right.) The player cannot move through a mushroom. The player is bounded by the level and stops when the player collides with the edges of the screen.

If the player intersects the millipede (or is collided with by the millipede), the player loses.

Projectiles

There can only be one projectile on the screen at a time. The player fires a projectile with the "fire" key (the b key) which creates a projectile directly one pixel above the player. This projectile moves upward at any speed you desire as long as its movement is noticable (it is not an instaneous laser) and faster than the player itself.

When the projectile hits a mushroom, that mushroom is destroyed and the projectile is also destroyed. This allows the player to fire again.

When the projectile hits the millipede, the length of the millipede is shorted by one. If the millipede's length is equal to one when struck, the player wins. The projectile is destroyed and the player is able to fire again.

When the projectile hits the top boundary of the level, the projectile is unceremoniously destroyed, allowing the player to fire again.

Otherwise, the projectile moves upward and the player cannot fire a second projectile until a collision occurs.

Other Considerations

You may make the color of any element (millipede, mushroom, player, and projectile) any color you wish as long as none are the same color. I find that red for the millipede, blue for the player, green for a mushroom, and yellow/orange for the projectile works well. But, you may experiment.

Losing

When the millipede collides with the player (or the player collides with the millipede), the game will end and the player will lose.

Winning

When the millipede ceases to exist because the player hit each segment of the millipede, and at the point that the player hits the last remaining segment, the player wins.

Helpful Tidbits

Random Numbers

See Lab 4 Part A.

LED Display

See Lab 5 Part B.

Game Loop

Our game loop will be an infinite loop that periodically checks for updated inputs (button presses) and also periodically updates our game's state.

Every 100ms (1 tenth of a second), it will adjust the current frame. It will move characters and projectiles around and check for win or loss conditions. Our `update` function likely calls several other `update` functions for each game object (millipede, player, projectile, etc.)

```
last = getSystemTime()
while true
  handleInput()

  time = getSystemTime()

  # Get elapsed time since our last update in milliseconds
  elapsed = time - last

  # Each of our frames will be 100ms
  # That means, we are running at 10 frames per second
  # (more or less)
  if elapsed > 100
    # We won't worry too much about missed
```

```
    # frames or elongated frames
    last = time
    update()
  end
end
```

Input Handling

Our game loop has a `handleInput` function. Here, we will look at our hardware to detect whether or not an input was pressed. For this type of game, we can simply modify a variable for each direction and the one "action" button.

The code to handle input will be provided to you.

Check out this simple LED input handling program (which also exhibits a game loop!): simple_led_input.asm

Millipede Movement

Here are some helpful diagrams to depict how the millipede movement works. The normal movement of the millipede, colored red here, when it encounters the level boundary.

When it hits a "mushroom", indicated here in green.

When it hits a "mushroom", indicated here in green, and there is a mushroom below. Just ignore that mushroom.

Project Stages

In order to help you we aware of your progress, I will recommend a series of mile markers to help you divide up the work. You can, of course, ignore these if you wish. However, if you find you need some direction, by all means follow along.

You have over four weeks to complete this project, and I've divided this into four stages. You could consider accomplishing each stage for each week. (Or do everything in the last three days at your own peril! :)

Stage 1 - Millipede Movement

Draw the millipede with a length of 10. Starting at the top left and moving initially to the right. We don't have any level obstacles, so just make it go from the top-left to bottom following the normal millipede rules outlined above in the Millipede section of the game design portion.

Ensure that it, when it reaches the bottom, resets to the top-left. It will basically move right and left until reaching a bottom corner. And then resetting to the top-left and doing the exact same thing.

This stage involves establishing memory variables to keep track of the millipede. Which is some array of coordinates. Moving the millipede involves figuring out where the next segment goes and adjusting each segment as it moves.

Stage 2 - Level Generation

With your millipede in place, add some obstacles. Here, you will introduce a random element. As a hint, you will likely want a function somewhere that can add a random mushroom to the playing field, and then call that within a loop a certain number of times. It is here that you can also add the ability to periodically generate new mushrooms according to the rules found within the Level Obstacles section of the game design portion above.

Ensure that your millipede now navigates the obstacles appropriately. It is completely fine if the millipede goes through a mushroom **as it moves down** since it would be difficult to determine what behavior it should have in this case. Let's just koolaid-man it for that particular situation (ignore level collisions when moving millipede down).

Hint: You don't need any variables to remember where the mushrooms are. Why is that?

Stage 3 - Player Movement and Controls

With our basic enemy movement and level generation done, this is when I would recommend worrying about the player. Add a player, and also variables that pertain to the player. We also need to add input handling. When the "left" key is down, move the player left. Similarly handle "up", "down", and "right". Ensure that the player cannot move into a mushroom. Also, add your lose condition when the player or millipede collide with each other.

Since we do not have projectiles, we don't have a way of clearing the mushrooms. So we might get stuck, randomly. That's ok! You'll get to that.

Stage 4 - Projectiles and Win-Condition

Now… we get to our win condition. We will worry about only a single projectile at a time. According to our game design above, we only fire when there isn't already an existing projectile visible on the screen. Very classic arcade rules!

Obviously, handle the "fire" input key (the 'b' key, oddly) to create a projectile above the player.

Our projectile then travels upward. Handle the collision with mushrooms by destroying both mushroom and projectile (allowing the player to fire again)

Then, handle the collision with the millipede by shortening the millipede (what's a good way of doing that? There are a couple of choices. I'll let you decide.)

Finally, handle the win condition. When the projectile hits a single length millipede… you win!

Rubric

```
Documentation:

- Name included in source code
- Any bugs described
- Code commented reasonably enough

Millipede:

- Drawn correctly
- Moves in a particular direction automatically
- Handles hitting the level boundary and reversing direction
- Handles hitting a "mushroom" and reversing direction
- Handles hitting bottom corner

Player:

- Drawn correctly
- Moves through the input keys
- Cannot pass through level boundary
- Cannot pass through mushrooms

Mushrooms:

- Spawn randomly at start
- Spawn randomly at some reasonable interval (1 second recommended)

Projectiles:

- Spawn above player when action key pressed
- Drawn (in a distinguishing color)
- Move upward correctly
- Destroy mushroom upon hitting one (and then projectile disappears)
- Destroys a segment of the millipede upon collision
- Can only shoot one at a time

Gameplay:

- Win condition displayed when millipede size goes to 0
- Lose condition displayed when millipede and player collide

Extra Credit:

- Up to 5 points for any extra feature that is well described and commented
```

Submission

Submit your project zipped up as `dww9_project1.zip` or similar with your own username.

CS 447

- CS 447
- dwilk@cs.pitt.edu

- wilkie

CS 447 (26147); T-Th 9:30AM; 123 Victoria