|  |  |
| --- | --- |
| **Name: Karina Cuadrado** | **Fall 2017: CMPEN 351** |
| **Final Project: Writeup** | **Due: December 13, 2017** |

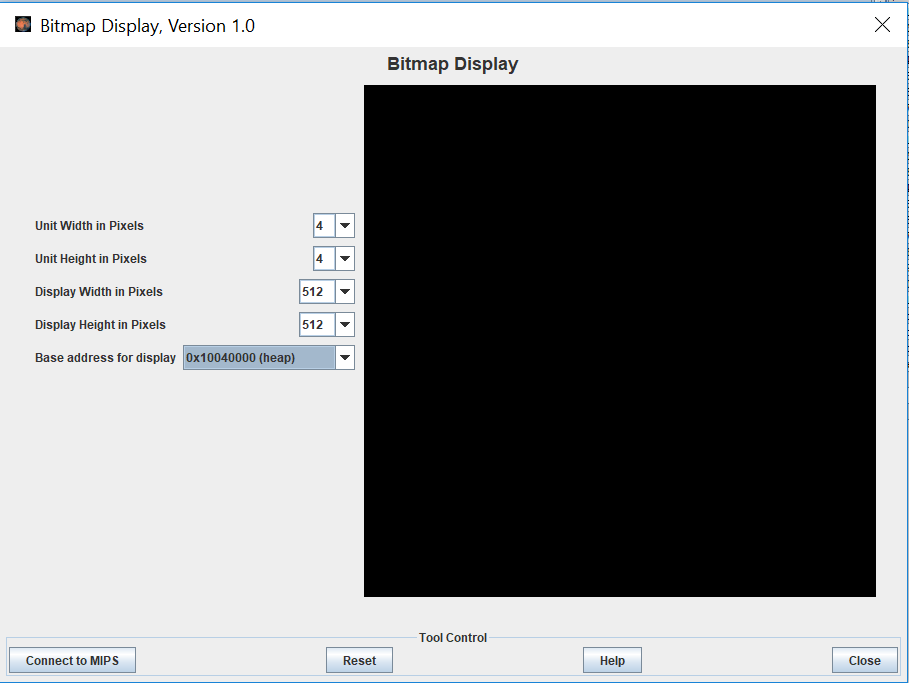
**Title: Donkey Kong 0.5**

**Section 1**

This program is an early draft of what the arcade version of Donkey Kong is. There is no princess to be saved, there aren’t any hammers to be used nor ramps to climb while you avoid obstacles. Instead, Mario sits in the right corner of the screen as Donkey Kong, on the left corner, throws barrels at him. You, the player, must press the spacebar key to jump and avoid as many barrels as you can! The program keeps a count for you so you can show off to Princess Peach and your friends.

**How to setup the program**

1. Using the application MARS 4.5, open the “.asm” file.
2. Open the “Tools” tab at the top and click on Bitmap Display.
3. Configure the display as shown:



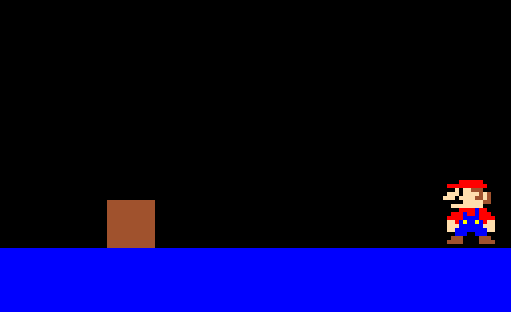
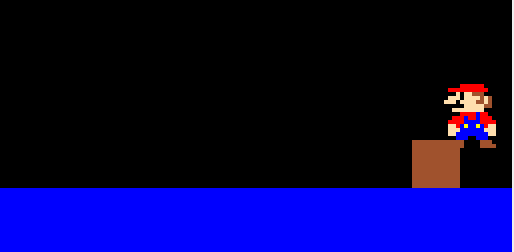
1. Under the same “tools” tab, click on “Keyboard and Display MMIO Simulator”. *\*\*Note: This is where you will be inputting the keys*
2. Press “Connect to MIPS” on both interfaces and run the program.

**Goal**

Save Mario by jumping the barrels Donkey Kong throws at him. Press the *spacebar* key to jump! If you fail to avoid the obstacle, you lose the game.

**X**

[✓](https://emojipedia.org/heavy-check-mark/)



**Section 2**

**Architectural description (Steps)**

1. First, interrupts must be enabled for an external device such as the keyboard.
2. Initialize the random seed for generating the speed of the barrel using the system’s clock.
3. Main: The program will draw the floor and Donkey Kong
   1. Floor is drawn by calling the *DrawBack* procedure that draws 8 squares of size 16. Initial coordinates must be passed: x coordinate = 0, y coordinate = 112 and size. This *DrawBox* procedure calls in to draw a horizontal line, calls to draw a dot, calls to calculate the address of the dot and get its color. It is then looped 8 times after updating the x coordinate by 16 every time.
   2. Donkey Kong is drawn using a table. Each line in this table specifies the x coordinate (x\*128), y coordinate (y\*4\*128) and the color to be saved in the address. The *DrawDK* procedure loads these parameters line by line, calculates the address (y coordinate + x coordinate + 0x10040000) and saves the color loaded into the address.
4. Print a prompt to the screen to “PRESS ENTER” and wait for user input
   1. Pass x and y to display prompt using the *OutText* procedure.
      1. OutText takes in initial coordinates, then calculates the address of these and save the white color in it. This then iterates for width and height of each character in the prompt.
   2. There is a loop that loads the “start” flag to determine if it can break out of the loop.
   3. This flag is changed in the exception handler only if the user presses the “Enter” key.
5. Print score prompt, instruction and initial Mario
   1. Pass x and y(top of screen) to display scoreprompt using the *OutText* procedure.
   2. Pass x and y(bottom of screen) to display inspmt using the *OutText* procedure.
   3. Draw resting pose for Mario using MarioTable. Like the DKTable, this table has the x coordinate, y coordinate and color value to be passed to the *DrawMario* procedure. This procedure loops through all 150 pixels on the table.
6. Moving loop: Drawing barrel and jumping Mario
   1. *DrawBarrel* procedure is called. This procedure calls DrawBox with a size of 12 and a y of 100. The x is determined by the global variable *initx* which is initialized at 25. This variable is then increased by the barrelspeed with every iteration. Likewise, the variable *endx* (with a set difference of +12 from initx), is increased by barrelspeed as well. The reasoning behind this was to avoid having the barrel reinitialize position when called again after enabledup (jumping flag) has been enabled.  
      At the end of the barrel sequence, the *RandNum* procedure is called to receive a number between 0 and 1 to determine the next barrelspeed. This prevents the game from being monotone and easily timed.

The increase of score count occurs whenever the barrelcount is exhausted. It then calls the *DrawFromInteger* procedure which strips off the digit of counter and displays onto screen through the *OutText* procedure.  
Also within the procedure, there’s an easy check of loser condition when Mario is not jumping given if the barrel’s x coordinate is greater than that of resting Mario.

* 1. Load flag set by exception handler to determine if Mario jumping sequence is enabled
     1. *If not enabled*, skip jumping loops. Reset flags and loop back to Moving loop.
     2. *If enabled*, decrease count variable that keeps track of how much left to jump and call *JumpMarioUp* procedure. This procedure calculates the y coordinate required to have Mario be drawn 3 pixels higher and calls *DrawMario* procedure. *DrawBarrel* is called after every iteration. This continues until count exhausted.  
        The count is then reinitialized, enabledup flag is set to 0, enableddown flag is set to 1 and the same sequence is executed but in the opposite direction and by calling *JumpMarioDown*.
     3. After jumping sequence is exhausted, reset flags and count. Loop back to Moving

1. Lose detection while Mario jumping
   1. Every loop in *DrawMario* procedure checks the initx and endx variables by calling the *DetermineLose* procedure
      1. If the x coordinate of the pixel to be drawn for Mario is within the initx and endx range – as well as over the 100 y coordinate -, it is considered a loss; the quitpmpt is displayed and all variables are reinitialized. It then polls the keyboard by waiting for a set end flag from the exception handler.
         1. If user input is ‘n’. Loop back to Main
         2. Any other input will be considered a ‘yes’ to the Quit question
      2. If not within range, exit the procedure and continue the drawing

**Visual representation/Flow chart** \*\* See attached, last page

**Verification plan**

The drawing of objects was pretty standard. Setting up a breakpoint after the drawing of each was sufficient to test. The following were more challenging:

1. **Test losing case**

When drawing Mario during its jump, set up a breakpoint where the check for coordinates is occurring and examine the register that serves as a loser flag (Should set to 1 when Mario within the barrel range). Execute loser procedure, where quit prompt is displayed and awaits user input. Any input other than ‘n’ should end the program.

When Mario is in resting mode, set a breakpoint during the check of barrel’s x coordinate. If it is past 112, execute Loser procedure.

1. **Test jumping feature w/ interrupt**

Set breakpoints wherever the check for the “enabledup” and “enableddown” signals are at (Exception handler, jumping loops and DrawBarrel procedure). Test keys other than spacebar, examine the enabled signal. Test jumping while Mario is still jumping, should be ineffective.

1. **Test counting feature**

Add to the score count every time a barrel successfully executes fully (barrelcount goes down to 0). Examine display when reach a 2-digit or more number, *DrawFromInteger* procedure inside *DrawBarrel* should successfully strip off each digit and pass to OutText.

**Debug/Issues**

Drawing Mario jumping and having the barrel animation required an interrupt from the keyboard. Instead of having a loop to draw the barrels over and over, I had to Draw Mario and the barrel in the same loop and dynamically increase their x and y coordinates.

Another problem I ran into listed in the architecture was initiating the DrawBarrel procedure after Mario started jumping, before the barrel ending its sequence it would revert back to its initial position. I started saving the starting x coordinate of the barrel to memory so I could access it in the same procedure and avoid reinitializing it out of time.

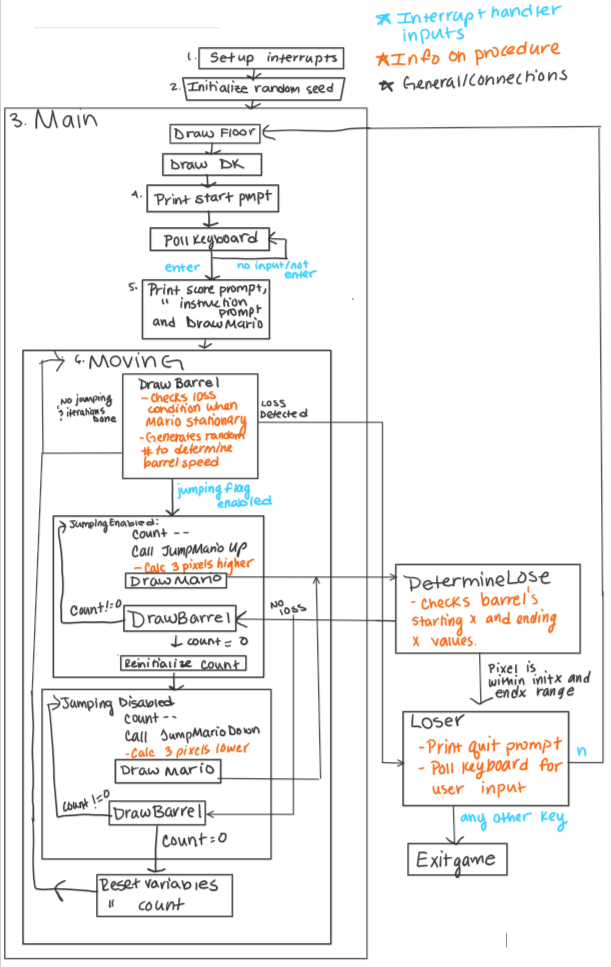
Determining whether the player lost or not was challenging. Now keeping track of the end of the box and start of the box from the variables in memory, I only had to check whether the Mario pixel lied within those bounds. It obviously wasn’t necessary to check all the pixels as done in the procedure, but making the distinction would take too long. My next problem was it not registering it when resting pose, this was because Mario wasn’t being redrawn. Therefore, in the DrawBarrel procedure, whenever the jumping flags aren’t set it checks if the barrel is past the starting point of Mario which should never be the case.

A strong challenge I faced was performance. Initially, I designed my program around a 256x256 resolution. My table for drawing objects originally listed how much to increment x by, how much to increment y by and the number of the color in the ColorTable for every pixel. I would call the DrawBox procedure with these parameters and a size of 2 per pixel. This consumed too much time, my game was basically not playable. I have to turn down the resolution to 128x128 and just expanded the units per width/height. I also redesigned the drawing tables. It now lists the calculated x address, calculated y address and the hex # for the color of pixel. The procedure only has to add the two to the base and save the color in that address. It was a risky move, but with the measurements already at hand, it was worth it. I had to change the barrels’ speed to much less or else it was impossible.

*Changes from homework submissions*

There are no longer 3 different speed settings. It made the game impossible. Instead now there’s 2 different speed settings that are determined by a random number generator. Initially, speed would change after a certain number of barrels/score count, but this would make the game easily timed and monotone. Therefore, the randomness helps to keep it challenging.

There are no sounds. I wanted to play a SNES Mario “.midi” file, but it would account for a totally separate program in itself. Adding noises to Donkey Kong throwing the barrels is doable, but it does not add that much functionality to the game so I deemed it worthless. If anything, adding multiple barrels at a time with separate speed settings would have been more practical. Needless to say, I failed in developing that.

****