CSC258 Assembly Programming Project: Doodle Jump

Due: Friday, April 9, 2021, 10:00 PM. In groups of up to 2 students.

Overview

In this project, we will implement the popular mobile game Doodle Jump (https://en.wikipedia.org/wiki/Doodle_Jump) using MIPS assembly. You may play a version of the original game at this **link** (https://poki.com/en/g/doodle-jump).

Since we don't have access to a physical computer with a MIPS processor, we will test our implementation in a simulated environment within MARS, i.e., a simulated bitmap display and a simulated keyboard input. The video below is the demonstration of a **basic** version of the game running in the simulator. As you can see in the keyboard-input window at the bottom of the screen, the player presses the key " `j` " to make the Doodler move to the left and pressed the key " `k` " move to the right, When no key is pressed, the Doodler falls straight down until it hits a platform or the bottom of the screen. If it hits a platform, the Doodler bounces up high into the air, but if the Doodler hits the bottom of the screen, the game ends. Your task in this project is to implement something similar to this demo, but better.

Note: what's in the video is just a **basic** version of the game. Implementing exactly this will NOT get you the full credit in this assignment. You will need to add some additional features to it (more details below).

1.00

Getting Started

You may work on this project in individually or in a group of two. Before you start, find a partner (there is a "Search for Teammates" post on the discussion board) or decide to work individually. **If you work in a group, create a group on MarkUs and invite your partner.**

You will create an assembly program named " `doodlejump.s` ". There is no starter code. You'll design your program from scratch. Below are a few tips that will help you get started.

Drive the Bitmap Display

To open and connect the bitmap display in MARS, click on "Tools" in the menu bar then select "Bitmap Display". In the opened window, choose the appropriate configuration values (e.g., the values shown in the video above), and click on "Connect to MIPS" to plug the display into your simulated MIPS computer.

The display is essentially an array of "units" stored in memory (starting at the chosen "base address of display" and indexed with row-major ordering (https://en.wikipedia.org/wiki/Row-_and_column-major_order)). Each unit consists of a group of pixels (as defined in "unit width/height in pixels" in the configuration). The size of the array is the total number of units in the display. For example, with the configuration in the above video demo, each unit is 8 pixels x 8 pixels and there are 32x32 = 1024 units on the display (since 256/8 = 32). Each unit is a 4-byte value which is the colour

code of the unit. For example, `0x000000` is black and `0xff0000` is red. To paint a specific unit on the display with a specific colour, you just need to store the right colour code at the right memory address (perhaps using the `sw` instruction).

Tip: Do NOT use the base location of ".data" (static data) as the "base address for display" since it may cause the display data to overlap with the static data that you define in your program and cause interesting bugs.

Tip: Google "colour picker", and you'll find a tool to help you pick any colours you like for your game. Pick a few key colors (sky, bird, pipe) and consider storing them in specific registers so that it's easy to put the colors into memory.

To help you get started, below is a short demo that paints a few units at different locations with different colours. Try to understand this demo first and make it work in MARS.

```
# Demo for painting
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8
# - Unit height in pixels: 8
# - Display width in pixels: 256
# - Display height in pixels: 256
# - Base Address for Display: 0x10008000 ($gp)
#
.data
        displayAddress: .word   0x10008000
.text
        lw $t0, displayAddress  # $t0 stores the base address for display
        li $t1, 0xff0000        # $t1 stores the red colour code
        li $t2, 0x00ff00        # $t2 stores the green colour code
        li $t3, 0x0000ff        # $t3 stores the blue colour code

        sw $t1, 0($t0)   # paint the first (top-left) unit red.
        sw $t2, 4($t0)   # paint the second unit on the first row green. Why $t0+4?
        sw $t3, 128($t0) # paint the first unit on the second row blue. Why +128?
Exit:
        li $v0, 10 # terminate the program gracefully
        syscall
```

Create animation

The most common way to create animation is to periodically repaint the whole screen. When an object is painted at a new location, it gives the illusion that the object moved. To control the rate of repaints, i.e., the number of frames per second (FPS), you want the program to sleep for a certain amount of time after each repaint. See the "System Calls" section below to see how to do that.

Tip: choose your display size and frame rate pragmatically. The simulated MIPS processor isn't super fast. If you have too many pixels on the display and too high a frame rate, the processor will have trouble keeping up with the computation. That's why in the demo video the game screen is fairly small. If you want to have a large display and fancy graphics in your game, you might consider optimizing your way of repainting the screen so that it does incremental updates instead of redrawing the whole screen; however, that will be **quite a challenge**.

Receive Keyboard Inputs

To connect the keyboard in MARS, click on "Tools" in the menu bar and select "Keyboard and Display MMIO Simulator". Then, click on the "Connect to MIPS" button on the bottom-left to plug in the simulated keyboard.

With memory-mapped I/O (MMIO), the data of the keyboard events are stored at a specific memory location. There are two values that we need to check. The first value is a 4-byte value that indicates whether there is a keystroke event -- it is 0 if and only if there is no keystroke event happening. The second value is a 4-byte value that stores the ASCII code (https://en.wikipedia.org/wiki/ASCII) of the key that has been pressed (given that there is a keystroke event). With the default keyboard configuration, the first value is stored at the memory address `0xffff0000` and the second value is stored at `0xffff0004`. Look up the ASCII code of the lowercase letter "`f`" to see what value you should look for.

System Calls

The following link has a list of the MIPS system calls that are available in the MARS simulator.

MIPS System Call Table (https://courses.missouristate.edu/KenVollmar/MARS/Help/SyscallHelp.html)

You will find some of the system calls useful (and necessary) for certain tasks such as sleeping for an interval (to control the frame rate), generating random numbers (for obstacle positions), and terminating the program. Read the doc and find what you need!

General Tips

Here some general tips for the design of your program.

1. Use memory for your variables. The few registers aren't going to be enough for allocating all the different variables that you'll need for keeping track of the state of the game. Use the "`.data`" section (static data) of your code to declare as many variables as you need.
2. Create reusable functions. Whenever you find yourself copy-and-pasting the same few lines of code over and over again, consider writing a function for it. Design the interface of your function (input arguments and return values) so that the function can be reused in a simple way.
3. Create meaningful labels. Meaningful labels for variables, functions and branch targets will make your code much easier to debug.
4. Write comments. Without proper comments, assembly programs tend to become incomprehensible quickly even for the author of the program. So, for your own interest, please write proper comments along with your code whenever possible -- it's going to save yourself a lot of time later.

Marking Scheme

This assignment is worth 9 marks, which will be awarded according to the following five milestones.

- **Milestone 1** (mark: `2/9`): the Doodlers and the platforms are properly drawn (statically) on the screen.
- **Milestone 2** (mark: `4/9`): the movement controls of the Doodler and the platforms (by the keyboard and timers) are properly implemented.
- **Milestone 3** (mark: `6/9`): a basic version of the game (similar to the one shown in the video demo above) is properly implemented.
- **Milestone 4** (mark: `8/9`): one or two of the approved additional features (listed below) are properly implemented.
- **Milestone 5** (mark: `9/9`): at least three of the approved additional features (listed below) are properly implemented.

Below is the list of approved additional features. ONLY features on this list will be considered valid additional features and given credit. If you would like to request adding a feature that is not on the list, please email the instructor. The list on this page will be updated if the requested feature is approved.

`Note:` All requests regarding additional features must be sent before **10:00 PM on March 31, 2021.** We will freeze the feature list and no longer accept new requests after this deadline.

Approved List of Additional Features (last updated: Mar 27, 2021)

1. Display the score on screen. The score should be constantly updated as the game progresses. The final score is displayed on the game-over screen.
2. Realistic physics. Make the physics of the Doodler movement more realistic by adjusting its up/down movement speed by simulating the existence of gravitational acceleration.
3. More platform types, such as moving blocks, "spring" blocks (more bouncy), and "fragile" blocks (broken when jumped upon). Different types of platform blocks are distinguished by different colours.
4. Power-ups/pick-ups. Additional power-up/pick-up objects for the Doodler to collect. The state of the game is changed upon the collection of a power-up, e.g., increase/decrease the speed, enlarge/shrink the bird size, make obstacles disappear, reset score to zero, double the score, larger platforms, shield on the Doodler, etc.
5. Player names. Allow the player to enter their name using the keyboard, and the name is displayed throughout the game as well as the game-over screen showing the final scores.
6. Two Doodlers. Have two Doodlers jumping at the same time, controlled by two different sets of keys (e.g., one controlled by " `j/k` " and one by " `d/f` ").
7. Fancier graphics. Make the graphics of game much fancier and more refined than what's shown in the video demo above. **Note:** to qualify this additional feature, the new graphics must be **significantly** better than the video demo above and it must not sacrifice the responsiveness of the gameplay. Consult with the instructor if you're not sure whether it's fancy enough.
8. Dynamic on-screen notification messages during the game such as "Amazing!", "Wow!", "Good job!". The notification should change over time.
9. Opponents / lethal creatures that can move and hurt the Doodler.
10. Shooting: the Doodler can shoot (controlled by another key) and eliminate opponents.
11. Dynamic background: dynamically changing background during the game (e.g., clouds moving horizontally). The movement must be different from that of the platforms.
12. Changing difficulty as game progresses: gradually increase the difficulty of the game (e.g., shrinking the platforms) as the game progresses.
13. Background music: add background music to the game.

Required Preamble

The code you submit (" `doodlejump.s` ") MUST include a preamble (with the format specified below) at the beginning of the file. The preamble includes information of the group members, the configuration of the bitmap display, and the features that are implemented. This is necessary information for the TA to be able to mark your submission.

Submissions without a proper preamble will NOT be marked and will receive 0 mark.

Below is an example/template of the preamble. Copy and paste it to your code and update the information accordingly.

```
###############################################################################
#
# CSC258H5S Winter 2021 Assembly Programming Project
# University of Toronto Mississauga
#
# Group members:
# - Student 1: Name, Student Number
# - Student 2 (if any): Name, Student Number
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8
# - Unit height in pixels: 8
# - Display width in pixels: 256
# - Display height in pixels: 256
# - Base Address for Display: 0x10008000 ($gp)
#
# Which milestone is reached in this submission?
# (See the assignment handout for descriptions of the milestones)
# - Milestone 1/2/3/4/5 (choose the one the applies)
#
# Which approved additional features have been implemented?
# (See the assignment handout for the list of additional features)
# 1. (fill in the feature, if any)
# 2. (fill in the feature, if any)
# 3. (fill in the feature, if any)
# ... (add more if necessary)
#
# Any additional information that the TA needs to know:
# - (write here, if any)
#
###############################################################################
```

Submission

You will submit your " `doodlejump.s` " (only this one file) by using the **web submission interface** of MarkUs. You can submit the same filename multiple times and only the latest version will be marked, so it is a good practice to submit your first version well before the deadline and then submit a newer version to overwrite when you make some more progress. It is also a good idea to backup your code after completing each milestone or additional feature, to avoid the possibility that the completed work gets broken by later work. Again, make sure your code has the required preamble as specified above.

Late submissions are penalized by 1% for every hour of lateness, rounded up, to a maximum of 24 hours. Submissions will no longer be accepted 24 hours past the deadline, except for documented unusual circumstances.

Academic Integrity

Please note that ALL submissions will be checked for plagiarism. Make sure to maintain your academic integrity carefully, and protect your own work. It is much better to take the hit on a lower assignment mark (just submit something functional, even if incomplete), than risking much worse consequences by committing an academic offence.