Spimbot Documentation

The Spimbot Documentation, Abridged

Version **1.1.1-Lab9**
Last updated **Mar 23, 2022**

Table of Contents

0 Changelog

Updates are listed in chronological order, newest updates are shown in red. We will always attempt to fix game-breaking bugs. Other bugs that get reported may get a new release or might just get a document change.

Mar 23: fixed correct version number for --version, updated location of the slow solver, fixed broken image link in section 4.1

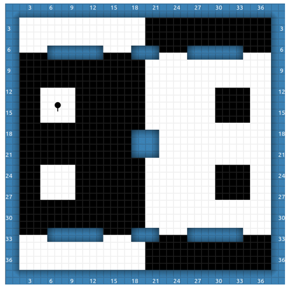
1 The Lab

Welcome to Spimbot! This lab serves as an introduction to the environment you'll be using to complete the final lab of the semester, Lab Spimbot. You'll be writing MIPS code to control a virtual robot, the SPIMBot, and will have to satisfy the requirements detailed in the Lab 9 handout to earn your grade.

We recommend you read this entire document — almost everything here will be necessary for completing Lab 9. You'll also be reusing what you learn here once you start working on Lab Spimbot at the end of the semester.

1.1 The Map

You'll see the map when you run QtSpimbot. It looks like this.



NOTE: If that's not what you see, make sure you're running QtSpimbot with the `-part1` or `-part2` flag. See Part 5, Testing for more info.

Here's what you should know about the map:

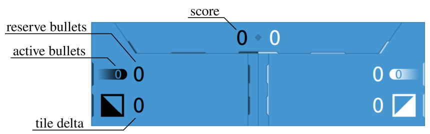
- The map is a **40x40** grid of **tiles**
  - Each tile is made of **8x8 pixels**
- **Bot 0** (black, owns white tiles) always starts in the top left corner **(Tile [7, 14] = pixel [56, 112])**
- You own half the tiles on the field to start (the white ones)
  - You can only walk on your tiles
  - You can use bullets to flip tiles you don't already own

NOTE: During Lab 9, **there is only one player**. That means your code controls **bot 0**, which starts in the top left corner, and any references to **bot 1** can be ignored.

See 1.3 Tile Types below for a summary of tile types on the map.

1.2 The HUD

The HUD is displayed directly below the map, and contains information about each player. In Lab 9, only the left side is relevant as there is no bot 1, only bot 0. The image below labels each component.



Here's what each field means:

- **Score:** a player's score, not relevant to Lab 9.
- **Reserve bullets:** the number of bullets a player has available to shoot
  - In **Lab 9 Part 1**, you have unlimited bullets and do not need to solve puzzles.
  - In **Lab 9 Part 2**, you need to earn bullets by solving puzzles.
- **Active bullets:** the number of bullets a player has on the screen
- **Tile delta (ΔT):** the difference between a player's current tile ownership and their opponent's
  - This isn't relevant to Lab 9. Don't confuse it with the number of tiles you've flipped — it's not quite the same.

To reiterate: During Lab 9, **score and tile delta are not relevant**. We'll only be looking at the objectives detailed in the **lab handout**.

1.3 Tile Types

Below is a summary of the different tile types:

Tile type	Notes
Floor 0	**Bot 0**'s tile, painted **white** on the map. **Bot 0 can walk** on these. Acts as a **wall for bot 1**.
Floor 1	**Bot 1**'s tile, painted **black** on the map. **Bot 1 can walk** on these. Acts as a **wall for bot 0**.
Wall	An immovable wall, painted **blue** on the map. **Nobody can walk** here, and bullets cannot pass through.

In short, each bot can only walk on tiles that are the **opposite** color of the bot, and nobody can walk through blue walls. A bot can take ownership of an opponent's tile by shooting it, and bullets won't stop until they hit a wall or the opponent. See **SHOOT** and **CHARGE_SHOT** in Appendix A.

You can access the map using the **GET_MAP** command. It'll return a struct that contains a 2D array with each entry corresponding to a tile on the map. The value of each entry will contain information about the corresponding tile, including the tile type. See Appendix A and Appendix B for more information!

1.4 The Objective

Your objectives for Lab 9 are detailed in the lab handout. Give it a read once you've finished with this document.

1.5 Scoring Points

Points are not relevant to Lab 9.

1.6 Resetting

Resetting is not relevant to Lab 9, as there is no adversary to shoot.

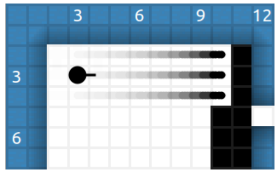
1.7 The SPIMBot

The SPIMbot **executes MIPS code** you write for it and uses memory-mapped IO (MMIO) to **interact with the world**. This section is a summary of your bot's abilities; for detailed information on **MMIO commands**, see Part 2, SPIMBot MMIO.

Bots can move around within the tiles they own (Bot 0 owns white tiles, Bot 1 owns black tiles), but they cannot move outside these tiles. They can also use MMIO sensors to see where they are on the map and to check who owns which tiles, among other things.

In addition to moving around, bots can **solve puzzles**, which will grant them **bullets** (see 2.5 Puzzle). **This is necessary for Lab 9, Part 2!** They can then **launch bullets** in any of the four **cardinal directions** using the **SHOOT** MMIO. Doing so will remove **1** bullet from the bot's ammo reserve. These bullets can be used strategically to flip enemy tiles to the bot's ownership. **You'll need to do this for both parts of Lab 9.** Bullets won't stop until they hit something, so they can flip multiple enemy tiles in a row.

Bots can also fire a **charged shot** of three bullets side-by-side using the **CHARGE_SHOT** MMIO, causing bullets to spawn at the bot's location as well as on the tiles directly besides them on either side. **You'll need to do this for both parts of Lab 9.** The bullets travel in the same direction, as shown below.



Doing this does not only consume **3** bullets, but also requires the bot to wait **10,000** cycles between starting to charge their attack and releasing it. The process of firing a charged attack is as follows:

1. Use the **CHARGE_SHOT** MMIO to start charging
   - You commit to which direction you'll fire the bullets here!
2. Wait **10,000 cycles**; don't use **SHOOT** yet!
3. Use **SHOOT** to release the charge

Take note that using **SHOOT** before 10,000 cycles have passed will just fire a single bullet and reset your progress towards a charged shot. Also, it doesn't matter what direction you use **SHOOT** with; it'll be ignored and the direction you committed to in step **1** will be used instead.

Bots can also read from **MMIO_STATUS** to see if the last command they used worked successfully. A value of **0** indicates success, and information on other possible values can be found in the starter MIPS code.

Now that you know roughly what your bot can do, let's take a closer look at all the available MMIO commands.

2 SPIMBot MMIO

SPIMBot's sensors and controls are manipulated via memory-mapped IO; that is, the IO devices are queried and controlled by reading and writing particular memory locations. All of SPIMBot's devices are mapped in the memory range **0xffff0000 - 0xffffffff**. This section will describe the available MMIO devices; for a summary of all MMIO addresses, see Appendix A.

2.1 Orientation Control

SPIMBot's orientation can be controlled in two ways:

1. By specifying an adjustment relative to the current orientation (**turn by** x degrees)
2. By specifying an absolute orientation (**point at** x degrees)

In both cases, an integer value (between **-360** and **360**) is written to **ANGLE** (0xffff0014) and then a command value is written to **ANGLE_CONTROL** (0xffff0018). If the command value is **0**, the orientation value is interpreted as a **relative** angle (the bot will **turn** by that amount). If the command value is 1, the orientation value is interpreted as an **absolute** angle (the bot will **point in** that direction). Note that writing to **ANGLE** will **not** do anything on its own; you need to write to **ANGLE_CONTROL** to actually make the bot turn.

Angles are measured in degrees, with **0** defined as facing right. **Positive angles** turn the bot **clockwise**. While it may not sound intuitive, this matches the normal Cartesian coordinates (the `+x` direction is 0°, `+y` is 90°, `-x` is 180°, and `-y` is 270°), since we consider the top-left corner to be (0,0) with `+x` and `+y` being right and down, respectively. For more details see Part 4, SPIMBot Physics.

2.2 Odometry

Your bot has sensors that tell you its **current position**. Reading from addresses **BOT_X** (0xffff0020) and **BOT_Y** (0xffff0024) will return your bot's x-coordinate and y-coordinate respectively, **in pixels** (not tiles!). Storing to these addresses, unfortunately, does nothing - no teleporting :)

2.3 Bonk (Wall Collision) Sensor

The bonk sensor **raises an interrupt** whenever your bot **runs into a wall**.

Note: Your bot's velocity is set to zero when it hits a wall, so you'll need to get it moving again.

2.4 Timer

You can do two things with the timer:

1. Check the number of cycles since the start of the game by reading from **TIMER** (0xffff001c).
2. Request a **timer interrupt** (like setting an alarm) by writing the time you want to be interrupted to **TIMER** (0xffff001c). This is great for switching between solving puzzles and moving around the map.

2.5 Puzzle

Solving puzzles is the only way to get more **bullets**, which is **necessary for Lab 9, part 2**. Each correct solution results in **1** bullet. You will be solving the Count Disjoint Regions puzzle (FillFill for short). It's a similar task to Lab 7, but we have modified the puzzle struct slightly. **Your Lab 7 solver will not work without modification for specific edge cases! You must use the provided solver instead!**

2.5.1 FillFill

The puzzle is the same puzzle as from Lab7, so feel free to go back to that lab's handout for more details. In short, your bot will be given a Puzzle struct consisting of a Canvas struct, a Line struct, and a data array. The objective is to figure out the number of disjoint regions after each line is drawn onto the canvas.

2.5.2 Puzzle Struct

The format the puzzles come in is via a struct that contains the information of Canvas and Lines. The maximum canvas size is 12×12. The maximum number of lines is 12. The struct written to the bot memory is defined as follows:

```
struct Puzzle {
    Canvas canvas;
    Lines lines;
    char data[300];
};
```

2.5.3 Requesting a Puzzle

The first step to getting a puzzle is to allocate space for it in the data segment, then write a pointer of that space to **REQUEST_PUZZLE** (0xffff00d0). However, it takes some time to generate a puzzle, so you will have to wait for a **REQUEST_PUZZLE** interrupt. When you get the **REQUEST_PUZZLE** interrupt, the puzzle struct for you to solve will be in the allocated space with the address you gave. Note that you must enable the **REQUEST_PUZZLE** interrupt or else you will never receive a puzzle.

To accept puzzle interrupts, you must turn on the mask bit specified by **REQUEST_PUZZLE_INT_MASK** (0x800). You must acknowledge the interrupt by writing a nonzero value to **REQUEST_PUZZLE_ACK** (0xffff00d8). The puzzle will then be stored in the pointer written to **REQUEST_PUZZLE**.

You can request more puzzles before solving the previous ones. But be sure to submit the solution in the same order as you requested them.

2.5.4 Submitting your Solution

After solving the puzzle, you need to submit the solution to generate a bullet. To submit your solution, simply write the pointer to your solution board to **SUBMIT_SOLUTION**. If your solution is correct, you will be rewarded with **1 bullet**! To check if you were granted a bullet for your solution, read from **MMIO_STATUS** (0xffff204c). If reading from this address yields 0, your solution was correct. If it yields anything else, your solution was not correct.

Run with the `-debug` flag, request a puzzle, and submit something to see examples of potential solutions.

2.5.5 The Slow Solver

We've given you a slow solver that you can use in your Spimbot. You can find it in **part2.s** under Lab 9 Part 2's **Lab Files To Dowload** section, along with the starter code. For all intents and purposes, it is just a solution to Lab7 and can be greatly optimized to give your bot an edge over the competition.

The arguments are the same ones as the solver given in Lab7.

If you wish to use the slow solver, you will need to allocate some space in the data segment for the solution to be stored. Then, pass the address as the second argument.

You may look at file "p2_main.s" in Lab 7 to learn how to use the slow solver.

3 Interrupts

The MIPS interrupt controller resides as part of co-processor 0. The following co-processor 0 registers (which are described in detail in section A.7 of your book) are of potential interest:

Name	Register	Explanation
Status Register	$12	This register contains the interrupt mask and interrupt enable bits.
Cause Register	$13	This register contains the exception code field and pending interrupt bits.
Exception Program Counter (EPC)	$14	This register holds the PC of the executing instruction when the exception/interrupt occurred.

3.1 Interrupt Acknowledgement

When handling an interrupt, it is important to notify the device that its interrupt has been handled, so that it can stop requesting the interrupt. This process is called "acknowledging" the interrupt. As is usually the case, interrupt acknowledgment in SPIMBot is done by writing any value to a memory-mapped I/O location.

In all cases, writing the acknowledgment addresses with any value will clear the relevant interrupt bit in the Cause register, which enables future interrupts to be detected.

Name	Interrupt Mask	Acknowledge Address
Timer	0x8000	0xffff006c
Bonk (wall collision)	0x1000	0xffff0060
Request Puzzle	0x0800	0xffff00d8

Name	Interrupt Mask	Acknowledge Address
Respawn	0x2000	0xffff00f0

3.2 Bonk

You will receive the **Bonk** interrupt if your SPIMBot runs into a wall. Your SPIMBot's velocity will also be set to zero if it runs into a wall.

3.3 Request Puzzle

You will receive the **Request Puzzle** interrupt once the requested puzzle is ready to be written into the provided memory address. You must acknowledge this interrupt for the puzzle to be written to memory!

3.4 Respawn

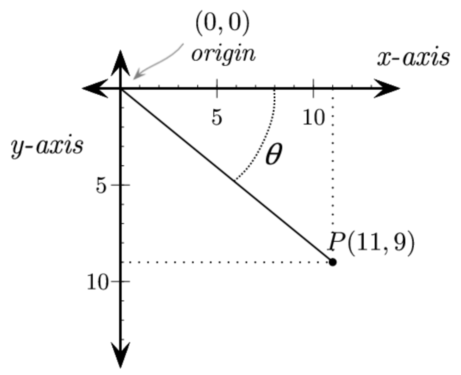
The respawn interrupt is not relevant to Lab 9, as there is no adversary to shoot.

4 SPIMBot Physics

4.1 Position and Velocity

In the SPIM Universe, positions are given in pixels. The origin is in the upper-lefthand corner of the map, i.e. `(x=0,y=0)`, while the lower-righthand corner is at `(x=320,y=320)`. Like in conventional drawings of the Cartesian plane, the **x-coordinate increases** as you go to the **right**. However, the **y-coordinate increases** as you go **down**, which might be the opposite of what you're used to. This is the convention for computer graphics, though, so we're sticking with it.

An angle of 0° is parallel to the positive x-axis. As the angle moves clockwise, it increases. When the angle is parallel to the positive y-axis (pointing **down**), it's at 90°.



When we describe the position of a SPIMBot, we refer to the coordinates of its center. The bot is just a circle with a radius of 3 pixels centered around a given point. It has a little stem pointing off in one direction to indicate which way the bot is facing.

Your bot's velocity is measured in units of pixels/10,000 cycles. This means that at maximum speed (±10), the bot moves at a speed of 1 pixel per 1000 cycles, or 1 tile (8 pixels) per 8000 cycles.

The SPIMBot has no inertia, and does not quite obey the laws of real-world physics. This means that you can rotate your bot instantly by using the **ANGLE** and **ANGLE_CONTROL** commands, and it can accelerate to a given speed instantaneously using the **VELOCITY** command.

4.2 Collisions

If your position is about to go out-of-bounds (either less 0 or greater than 320 on either axis) or cross into an impassible cell, your velocity will be set to zero and you will receive a **bonk interrupt**. Your bot will walk straight through the opponent, but it will get hit by any enemy bullet that it shares a tile with.

Note: Your position is the center of your SPIMBot! This means that your bot will partially overlap the wall before it “collides” with it. That's right, bots don't have real hitboxes when it comes to colliding with walls.

5 Running and Testing Your Code

NOTE: You **must** run QtSpimbot with `-part1` when testing your code for Lab 9 part 1, and `-part2` for Lab 9 part 2. Otherwise, you will get misleading feedback on what your grade will be.

QtSpimbot is built on top of QtSpim, so part of the interface may look familiar. To specify the MIPS file that will control your bot, use the `-bot0` argument followed by the file's name. Be sure to put any other flags before the `-bot0` flag.

Lots of useful debug information will be printed to the command line, telling you what's going on in the game. You can disable this with `-nodebug` ("no debug", not "node bug"). You can also use the `-drawcycles` flag to slow down the action and get a better look at what is going on; try a number around **1000** to slow the game down a little or **100** to slow it down a lot more.

In addition, QtSpimbot includes two arguments (`-maponly` and `-run`) to facilitate rapidly evaluating whether your program is robust under a variety of initial conditions (these options are most useful once your program is debugged).

Is your QtSpimbot instance running slowly? Try selecting the "Data" tab instead of the "Text" one.

Are you on Linux and having theming issues? Try adding `-style breeze` to your command line arguments.

5.1 Useful Command Line Arguments

Argument	Description
`-bot0 <file1.s> <file2.s> ...`	Specifies the assembly file(s) to use
`-bot1 <file1.s> <file2.s> ...`	Specifies the assembly file(s) to use for a second SPIMBot
`-part1`	Run SPIMBot under Lab 9 part 1 conditions
`-part2`	Run SPIMBot under Lab 9 part 2 conditions
`-test`	Run SPIMBot starting with 65535 bullets. Useful for testing.
`-nodebug`	Disable debug information printed to the command line
`-limit`	Change the number of cycles the game runs for. Default is 10,000,000. Set to 0 for unlimited cycles
`-randommap`	Randomly generate scenario map with the current time as the seed. Potentially affect bot start position, scenario specific positions, general randomness. Note that this overrides `-mapseed`
`-mapseed <seed>`	Randomly generate scenario map based on the given seed. Seed should be a non-negative integer. Potentially affects bot start position, scenario specific positions, general randimness. Note that this overrides `-randommap`
`-randompuzzle`	Randomly generate puzzles with the current time as the seed. Note that this oveerrides `-puzzleseed`
`-puzzleseed <seed>`	Randomly generate puzzles based on the given seed. Seed should be a non-negative integer. Note that this overrides `-randompuzzle`
`-drawcycles <num>`	Causes the map to be redrawn every num cycles. Default is 8192, and lower values slow execution down, allowing movement to be observed much better
`-largemap`	Draws a larger map (but runs a little slower)
`-smallmap`	Draws a smaller map (but runs a little faster)
`-maponly`	Doesn't pop up the QtSpim window. Most useful when combined with `-run`
`-run`	Immediately begins the execution of SPIMBot's program
`-prof_file <file>`	Specifies a file name to put gcov style execution counts for each statement. Make sure to stop the simulation before exiting, otherwise the file won't be generated
`-exit_when_done`	Automatically closes SPIMBot when contest is over
`-quiet`	Suppress extraneous error messages and warnings
`--version`	Prints the version of the binary (note the double-hyphen!) **Latest version:** 1.1.0 (Mar 16, 2022)

Tip: If you're trying to optimize your code, run with `-prof_file <file>` to dump execution counts to a file to figure out which areas of your code are being executed more frequently and could be optimized for more benefit!

Note that `-randommap` and `-mapseed` override one another, and that `-randompuzzle` and `-puzzleseed` override one another.

Appendix A

Below is a table of MMIO commands and their memory addresses. You can read or write to these addresses to get information about the game and perform in-game actions.

NOTE: commands relating to an opponent will not be relevant for Lab 9.

MMIO	Address	Acceptable Values	Read	Write
VELOCITY	0xffff0010	-10 to 10	Current velocity	Updates velocity of the SPIMBot
ANGLE	0xffff0014	-360 to 360	Current orientation	Updates angle of SPIMBot when **ANGLE_CONTROL** is written
ANGLE CONTROL	0xffff0018	0 (relative) 1 (absolute)	N/A	Updates angle to last value written to **ANGLE**
TIMER	0xffff001c	Anything	Number of elapsed cycles	Timer interrupt when elapsed cycles == write value
TIMER_ACK	0xffff006c	Anything	N/A	Acknowledge timer interrupt
BONK_ACK	0xffff0060	Anything	N/A	Acknowledge bonk interrupt
REQUEST_PUZZLE_ACK	0xffff00d8	Anything	N/A	Acknowledge request puzzle interrupt
RESPAWN_ACK	0xffff00f0	Anything	N/A	Acknowledge respawn interrupt
BOT_X	0xffff0020	N/A	Current X-coordinate, px	N/A
BOT_Y	0xffff0024	N/A	Current Y-coordinate, px	N/A
OTHER_X	0xffff00a0	N/A	Opponent's X-coordinate, tiles	N/A
OTHER_Y	0xffff00a4	N/A	Opponent's Y-coordinate, tiles	N/A
SCORES_REQUEST	0xffff1018	Valid data address	N/A	M[address] = [your score, opponent score]
REQUEST_PUZZLE	0xffff00d0	Valid data address	N/A	M[address] = new puzzle; sends Request Puzzle interrupt when ready
SUBMIT_SOLUTION	0xffff00d4	Valid data address	N/A	Submits puzzle solution at M[address]

MMIO	Address	Acceptable Values	Read	Write
GET_MAP	0xffff2008	Valid data address	N/A	Writes current **Map** struct to given memory location.
MMIO_STATUS	0xffff204c	N/A	Return status of previously used MMIO. See the debug output in your terminal for more info.	N/A
OTHER_BULLETS	0xffff200c	Valid data address	N/A	Returns list of opponent's bullets at M[address]
BOT_BULLETS	0xffff2010	Valid data address	N/A	Returns list of your bullets at M[address]
GET_AMMO	0xffff2014	N/A	Returns number of bullets that the bot has in reserve (no max)	N/A
CHARGE_SHOT	0xffff2004	Direction Enum	Returns Direction Enum representing the direction of the shot is being charged (-1 if there is no ongoing charge)	Charge a shot towards the passed direction
SHOOT	0xffff2000	Direction Enum	N/A	Fires a shot. If the bot calls **CHARGE_SHOT** prior, the direction passed into **SHOOT** is ignored and fires a shot (fully charged or not) in the direction passed to **CHARGE_SHOT**

Appendix B: Struct and Enum Definitions

Some MMIO commands, like **GET_MAP**, write a struct to an address you provide, while others read a struct from an address you provide. The contents of any aforementioned structs are shown below.

```
struct Puzzle {
    Canvas canvas;
    Lines lines;
    char data[300];
};

enum Direction {
    NORTH=0,
    EAST=1,
    SOUTH=2,
    WEST=3
};

enum Tile {
    FLOOR_0=0,
    FLOOR_1=1,
    WALL=2
}

#DEFINE MAP_SIZE 40

struct Map {
    char arr[MAP_SIZE][MAP_SIZE]; // Tile enums
};

struct bullet {
    char x_tile;
    char y_tile;
    char speed;
    char direction; // Direction enum
};

#DEFINE MAX_ACTIVE_BULLETS 15

struct active_bullets {
    int length;
    struct bullet arr[MAX_ACTIVE_BULLETS];
};
```

Appendix C: Helpful Code

You might find some of the following functions helpful in writing your SPIMbot.

```
.data
three:  .float  3.0
five:   .float  5.0
PI:     .float  3.141592
F180:   .float  180.0
.text
# -----------------------------------------------------------------------
# sb_arctan - computes the arctangent of y / x
# $a0 - x
# $a1 - y
# returns the arctangent
# -----------------------------------------------------------------------
.globl sb_arctan
sb_arctan:
    li      $v0, 0      # angle = 0;
    abs     $t0, $a0    # get absolute values
    abs     $t1, $a1
    ble     $t1, $t0, no_TURN_90
    ## if (abs(y) > abs(x)) { rotate 90 degrees }
    move    $t0, $a1    # int temp = y;
    neg     $a1, $a0    # y = -x;
    move    $a0, $t0    # x = temp;
    li      $v0, 90     # angle = 90;
no_TURN_90:
    bgez    $a0, pos_x      # skip if (x >= 0)
    ## if (x < 0)
    add     $v0, $v0, 180   # angle += 180;
pos_x:
    mtc1    $a0, $f0
    mtc1    $a1, $f1
    cvt.s.w $f0, $f0        # convert from ints to floats
    cvt.s.w $f1, $f1
    div.s   $f0, $f1, $f0   # float v = (float) y / (float) x;
    mul.s   $f1, $f0, $f0   # v^^2
    mul.s   $f2, $f1, $f0   # v^^3
    l.s     $f3, three      # load 3.0
    div.s   $f3, $f2, $f3   # v^^3/3
    sub.s   $f6, $f0, $f3   # v - v^^3/3
    mul.s   $f4, $f1, $f2   # v^^5
    l.s     $f5, five       # load 5.0
    div.s   $f5, $f4, $f5   # v^^5/5
    add.s   $f6, $f6, $f5   # value = v - v^^3/3 + v^^5/5
    l.s     $f8, PI         # load PI
    div.s   $f6, $f6, $f8   # value / PI
    l.s     $f7, F180       # load 180.0
    mul.s   $f6, $f6, $f7   # 180.0 * value / PI
    cvt.w.s $f6, $f6        # convert "delta" back to integer
    mfc1    $t0, $f6
    add     $v0, $v0, $t0   # angle += delta
    jr      $ra

# -----------------------------------------------------------------------
# euclidean_dist - computes sqrt(x^2 + y^2)
# $a0 - x
# $a1 - y
# returns the distance
# -----------------------------------------------------------------------
euclidean_dist:
    mul     $a0, $a0, $a0   # x^2
    mul     $a1, $a1, $a1   # y^2
    add     $v0, $a0, $a1   # x^2 + y^2
    mtc1    $v0, $f0
    cvt.s.w $f0, $f0        # float(x^2 + y^2)
    sqrt.s  $f0, $f0        # sqrt(x^2 + y^2)
    cvt.w.s $f0, $f0        # int(sqrt(...))
    mfc1    $v0, $f0
    jr      $ra
```