COMP2611: Computer Organization Spring 2022
Programming Project: NS-Shaft
(Deadline 11:55PM, May 15 Sun via Canvas)

1 Introduction

NS-Shaft `https://youtu.be/-SksNwLmSSE` is a platform video game created in 1900 by Nagi-P SOFT Co.,Ltd. from Japan. In this game, the player moves left and right, trying to dive deeper into the cave. You will implement a simplified NS-Shaft Game with MIPS assembly!

Figure 1 shows several snapshots of this game. The game displays a deep cave, where several platforms are placed at different positions. The platforms move upward while accelerating slowly. There are spikes at the top of the screen.

The goal is to fall from platform to platform rapidly enough to not be hit by the spikes, but slowly enough not to fall off the bottom of the screen. You can control the player by the keyboard, press key ”a” to move left and press key ”d” to move right.

The player will die if he/she jumps too fast and falls off the bottom of the screen, or he/she jumps too slow and hits the spikes at the top of the screen. The player will also die if he/she runs out of ”life.” The ”life” value is displayed on the upper left corner of the screen. The ”life” value decreases when the player touches spike platform and recovers when the player stands on a normal platform.

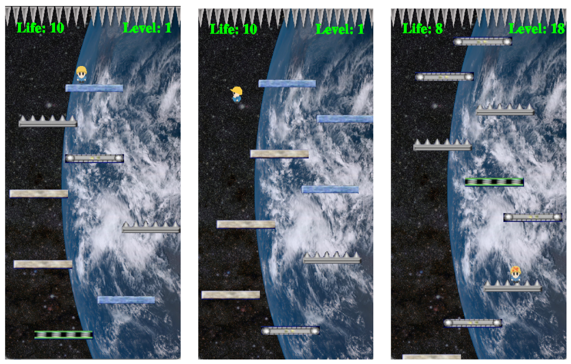

Fig. 1: Screenshots of NS-Shaft Game

There are 5 types of platform in the cave:

1. The stable normal platform : when the player stands on it, the ”life” value will recover by one until maximum.
2. The unstable flip-able platform : when the player jumps to it, the platform flips after a moment.
3. The rotating platform which keeps rolling to one direction: when the player jumps to it, the player will move left or right with a constant speed until he/she leaves this platform.
4. The elastic spring platform : When the player jumps to it, the player keeps bouncing on it vertically.
5. The spike platform which is paved with sharp spikes: when the player jumps to it, the ”life” value will decrease by one. The color of the player will turn to red for a few seconds.

The player wins the game when he/she reaches 20 layers deep in the cave. When the player wins the game, a text message ”You win!” will be displayed over the game screen, a ”game win” sound effect will be played and the game will finish. If the player fails to finish the game, a text message ”You lose!” will be displayed over the game screen, a ”game lose” sound effect will be played and the game will finish.

2 Coordinate System

The game canvas is of 300 $\times$ 600 pixels as illustrated in Figure 2. The top-left corner pixel is (0, 0) and the bottom-right corner pixel is (299, 599). The player is represented by a rectangular image of 20 $\times$ 30 pixels. The platforms are represented by rectangular images of 100 $\times$ 15 pixels. The location of each object is referenced by its *top-left* corner coordinates. When the player is running rightward on the platform, his/her *x_speed* is positive and *y_speed* is negative.

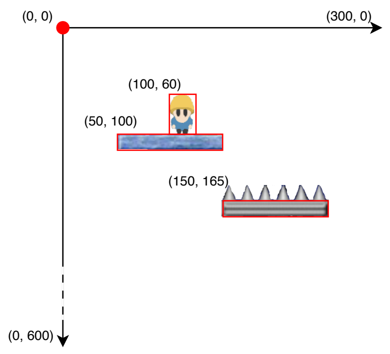

Fig. 2: The coordinate system

3 Game Objects

There are two types of game objects: player and platform.

3.1 Player

Table 1 lists the variables and attributes of the player object. We need to update the editable variables to implement the game.

The speed *player_speed* and location *player_locs* are updated constantly during the game. 1) If the player is standing on a platform, the *player_speed* is first updated according to the keyboard input. Then, the *player_speed* is further updated according to the type of the platform (e.g. the rotation speed for the rotating platform and the jumping speed for the spring platform). Finally, the location of the player *player_locs* is updated according to the final *player_speed*. 2) If the player is falling, the *player_speed* is first updated by increasing the *y_speed* by *gravity*. Then, the *player_locs* is updated according to the *player_speed*.

The appearance of the player is determined by *direction* and *hurt_status*, which can be edited via syscall. You can press ”a” or ”d” in the keyboard to let the player turn left or right, respectively.

	variable/attribute	value	description
fixed	player_id	-1	object id
	player_size	(20, 30)	the shape of player
	player_move_on_platform_speed	8	the running speed of player
	player_falling_x_speed	4	the x_speed of falling player
editable	player_locs	(x_loc, y_loc)	top-left coordinate
	player_speed	(x_speed, y_speed)	the speed of player
	life_value	0-10	the life value of player
	direction (in Syscall)	0 or 1 or 2	0: left, 1: right, 2: front
	hurt_status (in Syscall)	0 or 1	0: normal, 1: hurt

Table 1: The information of the player object.

3.2 Platform

The information of all the platforms in screen is stored in a variable *platforms*. Each platform has four attributes: (*id, x_location, y_location, type_index*). The attributes of the platforms are demonstrated in Table 2. Table 3 lists the variables and attributes of all platform objects.

type	index	image	description
normal	0		life value +1
flip-able	1		breaks after 0.5s
spring	2		player bouncing, y_speed = -6
spike	3		life value -2
left-rotation	4		player moves left, x_speed -1
right-rotation	5		player moves right, x_speed +1

Table 2: The type and index of the platform objects.

	variable/attribute	value	description
fixed	platform_size	(100, 15)	platform size
	platform_num	10	total number of platform objects
	initial_platform_data	array	info of initial platforms
editable	platforms	array	array of platform info
	current_platform_address	address	address of the current platform in *platforms*
	platform_y_speed	< 0	y_speed of all platforms
	platform_rotation_speed	1 or -1	the rolling speed of the current platform

Table 3: The information of the platform objects.

4 Game Details

4.1 Game initialization

Before the game starts, we first initialize the game via procedure *init_game*. First, the information of all the initial platforms are generated and stored into *platforms*. The player object and platform objects are created via syscall. Then, the life value, layer number and the vertical speed of platforms are initialized. Finally, the game screen is refreshed via syscall.

4.2 Collision detection: player and platform

The interaction between different game objects depends heavily on collision detection. It is required to decide whether the player hits any platform when he/she is falling. Although the images of the game objects may seem irregular, we consider two objects collide with each other when their rectangular images intersect for simplicity. The rule is simple: when the border of the game object overlaps with another object's border, they collide. You will implement the collision detection in procedure *check_platform_exists*. You can find detailed hints in the skeleton file to further figure out the logic.

4.3 Wining and losing conditions

The player must jump to more than 20 layers deep to win the game. When the player jumps to a platform with object id greater or equal to 20, he/she wins the game immediately. For the losing condition, the player will lose the game when his/her life value reduces to zero. This may because the player touches too many spines, or because he/she moves out of the screen.

5 Game Implementation

5.1 Game Loop

After the game initialization, the game will proceed in a main loop. In each game iteration of the main loop, the following steps are executed sequentially:

1. Check if the player wins or loses the game: *jal check_game_level_status*. If the game ends, process wining or losing tasks.
2. Get the current time: *jar get_time*.
3. Process the player movement: *jal player_movement*. 1) If the player is falling: *jal player_falling*. 2) If the player is on a platform, *jal process_move_input* to process the keyboard input for player running left and right.
   Note: Holding down the key generates a sequence of keystrokes, which moves the player object continuously. For MAC users, if it does not work, enable it by entering the following command in the Terminal application: *defaults write -g ApplePressAndHoldEnabled -bool false*. Replacing false by true in the command will produce the opposite effect.
4. Process the platform movement: *jal platform_movement*. 1) Destroy and generate a new platform when needed. 2) Move the platforms upward. 3) Increase the moving speed of the platforms.
5. Check the player hurt status and recover the player when time up: *jal player_hurt_recover*.
6. Check the status of the unstable platform and break the unstable platform when time up: *jal unstable_platform_break*.
7. Refresh the game screen to reflect any screen changes.
8. Take a nap: *jal have_a_nap*. Sleep for *time_sleep=100ms* as the interval between two consecutive game iterations.
9. Go to step 1.

5.2 Code Organization

The skeleton file has been partitioned into five sections:

1. Data structure.
2. Main loop and initialization.
3. Player related procedures.
4. Platform related procedures.
5. Timing and keyboard input related procedures.

All codes are sorted in this order. The player and platform related procedures implements the logic and behavior of all game objects. Therefore, most programming tasks are player or platform related procedures.

6 Programming Tasks

When you code with a high-level programming language, you always go through problem specification, algorithm design/workflow analysis, coding, debugging and documentation. Coding with low-level assembly programming language is pretty much the same.

The NS-Shaft game in MIPS assembly is challenging, but you won't start everything from scratch. The user interface is handled by modified Mars (copyright: COMP2611 teaching team). The MIPS code mainly works on the logic of the game. The programming assignment package already includes a skeleton file for you to start with.

The tasks of the programming assignment include a reading task (Task 0). You will need to read through the skeleton and grasp a big picture of the code structure. The remaining tasks (Task 1-8) are coding tasks. You will focus on implementing a few MIPS procedures with well-defined interface.

6.1 Task 0: Reading Task

Spend a few hours to read the skeleton code. You should 1. understand the data structures used in the skeleton; 2. trace the game loop 3. figure out the functionality of each procedure.

Show your understanding of the big picture of the project by drawing a flow chart (`https://en.wikipedia.org/wiki/Flowchart`) with necessary comments or explanations. Feel free to draw the flow chart withdrawing tools or by hand. Your flowchart should:

– include major procedures used in the program;
– include proper details but not too much, so that the flowchart can be fit in **one page of A4 paper**, drawn by hand or by computer;
– have good layout so that it's easy to trace and understand.

Procedure	Input	Output	Description
Task1: player_move_right			Set the horizontal speed and the direction of the player object.
Task2: check_boundary			Check if the player exceeds the horizontal boundaries.
Task3: process_landing _on_new_platform	$a0: the address of the new platform in "platforms		Take different actions when the player lands on different platforms.
Task4: check_platform_exist		$v0: 0 if no platform, 1 if platform exists	Check if the player object is standing on a platform.
Task5: update_player_life_value	$a0: 0 for reduce life value, 1 for recover life value.		Update the life value of the player.
Task6: destroy_and_create_platform			Destroy the first platform and create a new platform in "platforms."
Task7: check_game_level_status		$v0: 0: continue playing; =1: win; = 2: lose.	Check the status of the game to continue, win or lose the game.
Task8: lose			Implement the losing procedure.

Table 4: Programming tasks

6.2 Task 1-8: Programming Task

After reading the skeleton code and understanding how it works, try to understand the data structures used in the skeleton too. Complete the following MIPS procedures.

- You should not modify the skeleton code but only add your code to those procedures. Table 4 lists all the programming tasks you need to implement.
- You can see detailed instructions on how to implement each task in the skeleton file.
- For some programming tasks in the skeleton code, you need to remove the code in the answer space and write your own code.
- Pseudo instructions are allowed as long as it is supported by the Mars.

Before implementation, when you execute the skeleton code, you will find that the player can only move left and cannot fall down from a platform. The player will also exceed the left boundary when it keeps running left. To complete the game, you are encouraged to finish the tasks **in the given order** as shown in Table 4. This allows you to test your implementation after finishing each task. Many tasks has no input and output. For those tasks, you need to operate on the variables given in the data section.

Note: You are encouraged to read the whole code, but you do not need to understand every single detail of the skeleton code. You can discuss with your friends if you have difficulty in understanding it. But every single line of your code should be your own work (not something copied from your friends).

7 Syscall Services

We have implemented a group of additional syscall services to support game related functions (e.g. object movements). For your code to work, you should use the modified MARS (Mars_NS-shaft.jar) provided in the project section of our course website. Table 5 lists all the provided syscall services to implement the game.

Note that not all the new syscalls are necessary in your code, some are described here for you to understand the skeleton. Syscall code should be passed to $v0 before usage.

8 Submission

You should ***ONLY*** submit the file *comp2611_project_yourStudentID.s* with your completed code for the project. Please write down your name, student ID, and email address (as code comments) at the beginning of the file. As a good programming habit, comment your code properly and use registers wisely.

Submission is via COMP2611 Canvas. The submission deadline is 11:55PM, May 15 Sun. Try to avoid uploading in the last minute. If you upload multiple times, we will grade the latest version by default.The deadline is a hard deadline. -1 point out of 100 points will be deducted for every minute late, until 0 point is reached.

9 Grading

Your project will be graded on the basis of the functionality listed in the project description and requirements. You should ensure that your completed program can run properly in our modified MARS.

Service	Code	Parameters	Result
Create game screen	200		
Create player	201	$a0: id, $a1: x_loc, $a2: y_loc	A new player object of the given ID is created. The location of the player is set to the given x- and y- coordinates.
Create platform	202	$a0: id, $a1: x_loc, $a2: y_loc, $a3: type	A new platform object of the given ID is created. The location of the platform is set to the given x- and y- coordinates.
Create game text	203	$a0: id, $a1: x_loc, $a2: y_loc	A new text object of the given ID is created. The location of the text is set to the given x- and y- coordinates.
Update level	204	$a0: level	Update the current level.
Set player direction	205	$a0: id, $a1: direction (0: left, 1: right, 2: front)	
Refresh screen	206		
Set Location	207	$a0: id, $a1: x_loc, $a2: y_loc	
Destroy platform	208	$a0: id	
Play sound	209	$a0: sound id	
Update life value	210	$a0: life value	
Random x_location	211	$a0: last platform x_loc, $a1: screen width	$v0: new platform x_loc
Random platform type	212		$v0: type of new platform
Player hurt recover	213	$a0: id, $a1: hurt status (0: recover, 1: hurt)	

Table 5: Syscall Services