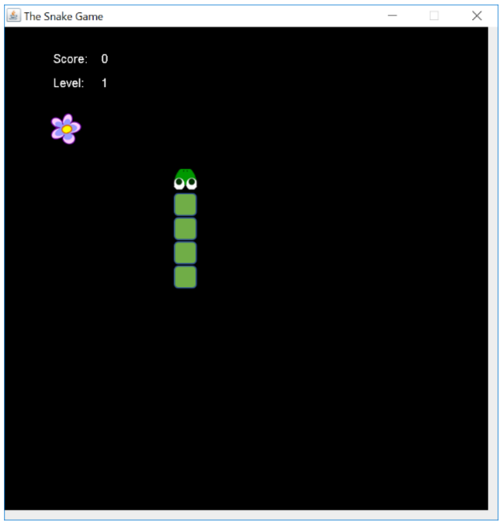
COMP2611: Computer Organization Fall 2017

Programming Project: The Snake Game

Due Date: 23:55, 30 November 2017

Introduction

In this project, you will complete *The Snake Game*. A snapshot of the game is shown below. By controlling a snake, your target is to eat a predefined number of food items, while at the same time, avoid head colliding with borders or intersecting with snake itself. Note not all food items are edible, some are poisonous. Eating edible food items lengthens the snake and leads to a next level, then eventually, to a victory. Eating a poisonous food item shortens the snake and may lead to a game loss.



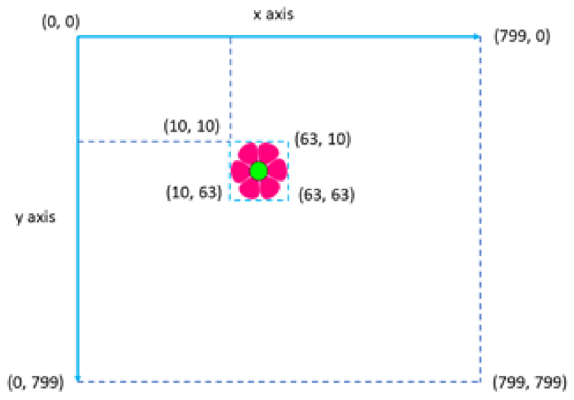


Figure 1 Screenshot and Coordinate System of the Snake Game

Game Objects

The game runs on an 800 pixel x 800 pixel square screen. The x-axis and y-axis of the coordinate system are rightward and downward (from 0) respectively, as shown in Figure 1.

Objects in the game include the snake (with a head object and a few body box objects), food items and extra boarder (for bonus part only). Every object is a square image of given size. Its location in the game screen is determined by the top-left corner's coordinates.

The sizes of objects are as follows:

Object	**Image size (x-size x y-size in pixels)**
Snake Head	40 x 40
Body Box	40 x 40
Food Item	54 x 54
Extra Border	400 x 50

Figure 2 Game Objects

Images for the game objects are listed below. To set a game object to use a certain image, set the object's "image index" to the corresponding index no. in your codes. If the image index is set to negative, the object will not be drawn on the screen.

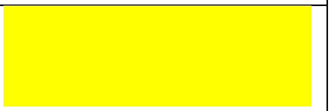
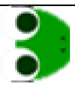
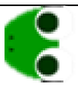
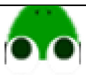
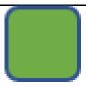
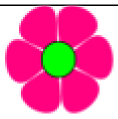
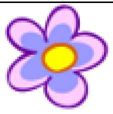
Image index	**Image**	**Description**	**Sample**
0	horizontal_border.png	Sliding border image (for bonus part only)	
1	snake_head_right.png	Snake's head moving right	
2	snake_head_down.png	Snake's head moving down	
3	snake_head_left.png	Snake's head moving left	
4	snake_head_up.png	Snake's head moving up	
5	snake_box.png	Snake's body box	
6	flower_food_one.png	Edible food image one	
7	flower_food_two.png	Edible food image two	
8	poison_food.png	Poisonous food image	

Figure 3 Images of the Game Objects

Game Initialization

The game has 2 levels. Game score is initially 0 at Level 1 and resets when moves to Level 2. Before the game starts, player is asked to set the stay time of a food item before it moves to a new position (in terms of millisecond), and a random seed for the random number generator. In level 2, the snake's speed doubles and the stay time of food item halves.

The snake's head is initially located at (x, y) = (200, 200) and is moving at a constant speed of 5 pixels (per game loop) towards the right edge of the screen. Snake's body length is initially set to 4, which is the minimum length of the snake. If the body becomes shorter than 4, the game is lost (more on this in the "Game Winning or Losing" section). The food image is randomly placed at a position that does not overlap with any part of the snake and is fully visible on the screen (all edges are within the range [0, 799]). The snake will start at (x, y) = (200, 200) with length 4 again when promoted to Level 2.

Snake Structure

The snake head object consists of a group of head-related items:

- *Object ID*: unique internal identifier for all objects in the game
- *x-coordinate*: top left position of the snake's head image in the game screen
- *y-coordinate*: top left position of the snake's head image in the game screen
- *Image index*: index of the image of the snake's head
- *Horizontal speed*: > 0 snake moves towards right edge; < 0 snake moves towards left edge; =0 no movement in horizontal direction
- *Vertical speed*: > 0 snake moves towards bottom edge; < 0 snake moves towards top edge; =0 no movement in vertical direction

Each head-related item takes 1 word (i.e. 32 bits) to store. All items are stored in array `snakeHead` as illustrated below. For example, `snakeHead[3]`stores the snake head image index.

Starting address Of snakeHead array	+4	+8	+12	+16	+20
Object ID of snake head	x-coordinate of snake head	y-coordinate of snake head	Image index of snake head	Horizontal speed of snake head	Vertical speed of snake head

Figure 4 Data Structure of Snake Head

The snake body is a structure of a group of body boxes. Each body box object consists of its Object ID, (x, y) coordinates and image index. Each body-related item also takes 1 word (i.e. 32 bits) to store. All body boxes are stored in array `snakeBody` as illustrated below. For example, `snakeBody[4]` is the object ID of the second box (i.e. box 1) in snake.

Starting address Of snakeBody array	+4	+8	+12	+16	+20	….
Object ID of box 0	Image index of body box 0	x-coordinate of body box 0	y-coordinate of body box 0	Object ID of body box 1	Image index of body box 1	…..

Figure 5 Data Structure of Snake Body

Object Movements

Food Item: There is only ONE food item in the game, although you do see different foods appears here and there in the game screen. This is been done by repeating the following steps:

- Places the food item in valid random generated location (x, y)
- Pick a random food image index (e.g. 6, 7, or 8). The corresponding food image is chosen.
- Make the food item visible for a user-defined period of time.

When generating random position, if a (x, y) coordinate leads to either 1. Overlap of the food image and any part of the snake, or 2. Partial appearance of the food image on the screen (e.g. not all image edges are within the range [0, 799]), it is not valid location to put food item and will be discarded.

Snake's Head: moves either horizontally or vertically according to the horizontal or vertical speed of the `snakeHead` data structure (Figure 4). As the snake can't move at both horizontal and vertical direction at the same time, it's straightforward that only one speed value is non-zero at a time. The next position of the head is the current position plus/minus one of the two speed values.

Snake's Body Box: In general, the movement of the snake follows its head, and each body box follows its previous body box. For example, suppose at time slot T the head is at location (x_head_T, y_head_T), this location would be the location for body box 0 at the next time slot T+1. Similarly, if body box i is at location (x_body_i_T, y_body_i_T) at time T, this would be the location of body box i+1 at the next time slot T+1.

The program uses the `snakeTargetPos` array for storing the target positions of the body parts. A target position is a position towards which a snake's body box moves. `SnakeTargetPos[i]` holds the target position of body box `i`. To be more specific, for example,

snakeTargetPos[0]	Target coordinates of body box 0 (i.e. head location)
snakeTargetPos[1]	Target coordinates of body box 1

snakeTargetPos[2]	Target coordinates of body box 2

Snake's movement is regulated by this structure and would be used in two of the procedures you will need to implement (`snakeBodyNewMovement` and `snakeOldMovement`).

The body box size is big (i.e. 40 pixels as in Figure 2), if the snake moves at 40 pixels speed, the game animation looks "bouncy". To create the smoother movement effect, the snake moves at a constant speed of 5 pixels in each `main_obj` loop iteration, as show in Figure 6.

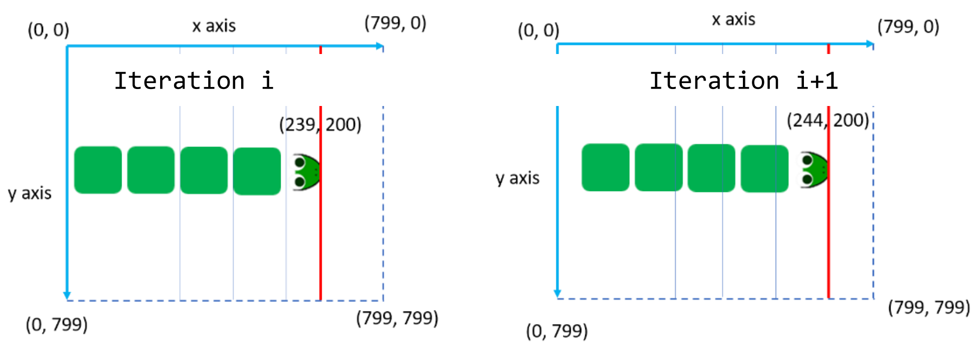


Figure 6 Snake's Movement

However we would enforce the snake body boxes move to its target position before responding to any changes. For example, the snake body box 0 is moving to the right following the snake head, when suddenly user press 'down' to change the head position (Figure 7a). Body box 0 won't change its direction to down immediately, it keeps moving to the right until reaches to the previous location of the head, i.e. target location (Figure 7b). Then body box 0 will change its direction to down to follow the head (Figure 7c).

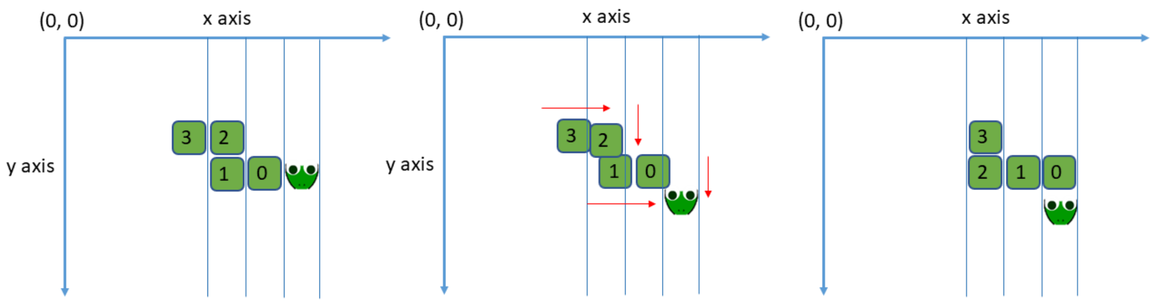


Figure 7 Movement of Snake's Body

Object Collision

Object collision can happen in two scenarios. First, a collision may occur when the snake's head hits a border and dies. `hitBorderCheck` procedure describes the scenario and provides the possible cases. It describes what parameters have to be passed and how the border collision logic works.

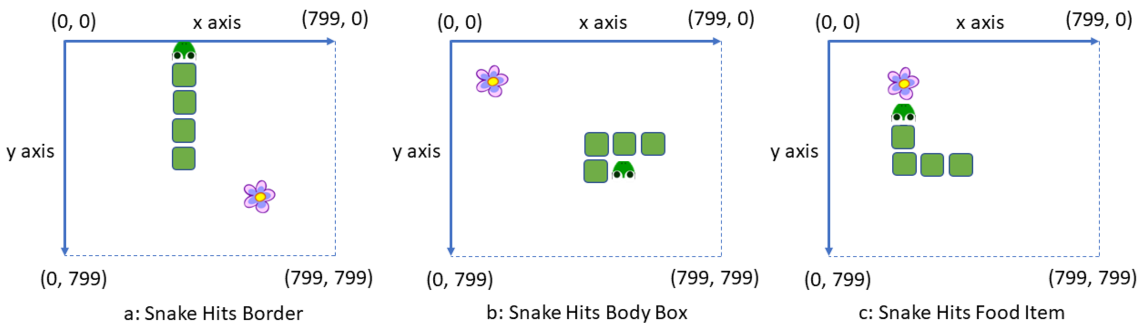


Figure 7 Object Collision

Second, the `checkObjectCollision` procedure should be used for two object-collision checking. This is a generic procedure used with square objects (`snakeHead, snakeBody, and foodState`). Refer to Figure 7 to see possible object collisions. Read the source code of the procedure for more information about the object collision logic.

Snake's Length

Snake can grow/shrink according to the eaten food item. If the snake eats a poisonous food item, it becomes shorter – loses the tail (*syscall 100*). If the snake eats an edible food item, a new body box should be displayed at the position of the current tail. You are provided with a customized syscall (read about in in *Syscall100_readme.doc*) which allows you to hide/display objects. In order to hide an object (body box), you have to use the following values: \$v0 = 100, \$a0 = 11, \$a1 = ID of object, \$a2 = -1; for displaying an object, you have to use the syscall twice with different values of \$a0 (11 and 12). The indexes of images are shown in Figure 3.

For more information about how the snake grows/shrinks, refer to the provided skeleton.

Game Winning or Losing

To win the game, two levels must be passed. Each level is completed once the snake's body length reaches a predefined value (constant stored in the `snakeBoxNo` word, currently set to be 6 for easy testing). The snake becomes longer for 1 body box every time an edible food item is eaten (i.e. head collides with food) and becomes shorter once a poisonous one is eaten. If the snake's length becomes less than the initial length (4 body boxes), the game is lost.

Other cases when the game is lost involve the head colliding with an edge of the screen or colliding with a body box of the snake. The collision functions (`hitBorderCheck` and `checkObjectCollision`) are provided for you, you will just need to use them and check a few collision cases.

Implementation (Game Loop)

All the game objects and status (score, level) are initialized at the beginning of the game. After user inputs, the game runs with a loop of the following steps:

1. Get the current time (T1) (provided for you).
2. Check for any keyboard input, which is stored using the Memory-mapped I/O scheme. If an input is available, read it and perform the action for it as follows (provided for you):

Input	**Action**
q	Terminate the game
w	Move the head upward
s	Move the head downward
a	Move the head leftward
d	Move the head rightward

3. Collision detection for the snake's head. Check for any collision with a border, a body part and the food item. If collision detected, update the status of involved objects **(to implement one part of it).**
4. Check if the current level of the game is over (provided for you).
5. Move food item if its duration has expired **(to implement).**
6. Move the snake (the head and the body boxes) **(to implement).**
7. Move the extra/sliding border **(optional bonus part).**
8. Redraw the game screen with the updated location and image of the game objects and game state information.
9. Get the current time (T2), and pause the program execution for (30 milliseconds - (T2 - T1)). Thus, the interval between two consecutive iterations of the game loop is about 30 milliseconds. By this way, we generate the 'cartoon animation' effect of the game.

Assignment Tasks

The game runs under the custom-made Mars program *Mars_4_1_withSyscall100.jar* (with copyright under COMP2611 teaching team), which supports graphical interface and sound effect. A set of custom-made syscalls is provided in custom-made Mars. User manual of the set of syscalls is provided in *Syscall100_readme.doc*.

To work with the game objects, you must use the data structures (MIPS word arrays) defined at the beginning of the code:

- Snake movement `speed` (1 word),
- `snakeHead` (6 words)
- `snakeBoxNo` (1 word)
- `snakeBody` (4 words per body box)
- `snakeTargetPos` (2 words per body box)
- `foodState` (4 words).

You can also use `snakeHeadSize`, `snakeBoxSize` and `foodPicSize` structures, but they are just constant values described in this document.

For bonus part, there are `bonusID` (1 word), `bonusXCoord` (1 word) and `bonusDirection` (1 word). Read the code comments for detailed description.

Read the skeleton code *snake_game_skeleton.asm* carefully and implement the following procedures. Your code should be in the MIPS assembly language. You may also use pseudo instructions in your program.

Procedure	**Task**
`placeFoodItem`	Placing the food item image on the screen. You have to generate a random number for image index in the range [6, 8]. In addition, you must place the image at a position that matches food placement requirements specified in the previous sections.
`shortenSnakesBody`	Make to disappear the last body box of the snake's body, update the snake's length and check if the length is < 4. If yes, jump to the lost game state. To make a body box disappear, use the provided *syscall 100* and set the image index of the body box to a negative value (e.g., -1).
`stretchSnakesBody`	Place a new body box at the current position of the snake's tail and update the

	`snakeTargetPos` structure accordingly. Use the provided *syscall 100* for displaying a new body box at the position of the current tail.
`snakeNewMovement`	Snake has reached its targets (values in `snakeTargetPos`). Update `snakeTargetPos` and move the snake by the current speed.
`snakeOldMovement`	Snake has not reached its targets yet, so move the snake towards the values stored by the `snakeBodyPos` structure.
`reinitializeSnake`	Reset the coordinates of the snake's head to the initial values (200, 200), reset the initial head image index, and reset the speed to the initial speed.
`check_head_food_hit(label)`	Implement checking if the head collides/hits the food image. If the head collides with the food item, call one of the above described procedures: either `stretchSnakesBody` or `shortenSnakesBody`. In order to choose the procedure, use the result of the `getFoodPoints` procedure.

(Bonus, optional)
 	
`addExtraBorder` (display an extra border at level 2)
 	
`checkExtraBorderCollision`
 	
`checkIfNewFoodItemOverlapSlidingBorder`
 	
`moveSlidingBorder`
	The bonus part involves implementing and using the three procedures shown on the left. `addExtraBorder` uses the `bonusID` value and syscall_100 to place a new border (extra order or sliding border) on the screen at level 2. `checkExtraBorderBollision` checks if an object collides with the extra border. `checkIfNewFoodItemOverlapSlidingBorder` is specialized procedure for checking if the position of the food item on the y-axis is within the range of the sliding border. The food item must be placed either fully above the sliding border or fully below the border. `moveSlidingBorder` procedure moves the border from left edge of the screen to the right edge and backwards.
	
	An additional task requires you to insert some

	extra code in the original code. The places where to insert are enclosed in the following way:
	`############`
	`#  BONUS PART`
	`###############`
	
	`########################`
	`# BONUS PART ENDS HERE`
	`########################`

Do not modify the given skeleton code (e.g. change certain registers). You should only add your code under the assigned tasks.

Bonus: Adding and Controlling an Extra Border

The bonus part adds a new feature to the game. It displays an extra sliding border of the size 400 x 50, which keeps moving from the right edge of the screen to the left and bounces back. Read the code segments with the "BONUS PART" comments for more details.

There is 10% bonus mark (i.e., 10 marks if the full project mark is 100). Since the project counts for 15% of COMP2611 score, if you successfully finish the bonus part, you will get additional 1.5 points in your total score.

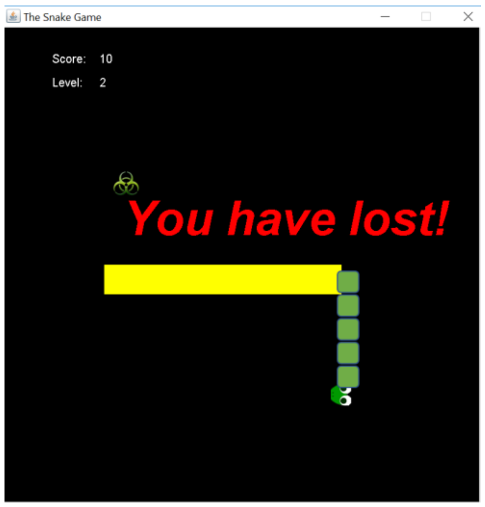


Figure 9 Screenshot of the Bonus Part

Submission

You should submit a single MIPS file with your completed codes for the project using the CASS (https://course.cse.ust.hk/cass). **No late submission is allowed.** Please avoid to upload your file in the last minute. The submitted file name must be exactly *snake_<Your student ID>.asm*. The CASS user manual is in this link http://cssystem.cse.ust.hk/UGuides/cass/index.html. Multiple submissions to CASS are allowed. However we'll only mark the latest version before deadline.

At the beginning of the file, please write down your name, student ID, email address, and lab section in the following format:

#Name:

#ID:

#Email:

#Lab Section:

Grading

Your project will be graded on the basis of the functionality listed in the game requirements, rather than the line-by-line implementation of MIPS code. Therefore, you should make sure that your submitted codes can be executed properly in the modified Mars program. 0 points will be given to any submission which fails to execute.

Inline code comments are not graded but are highly recommended. You may be invited to explain your codes in a face-to-face session (e.g., organized during this Fall 2017 examination period after the project due date).

Contacts

If you have any questions related to the project, please send an email to Mr. Justinas LINGYS at jlingys@connect.ust.hk. Or post the question on COMP2611 Fall 17 Facebook. Your email's layout should follow the following pattern:

To: jlingys@connect.ust.hk

Subject: COMP_2611_FALL_17_PA

If your email does not follow the provided pattern, it may be ignored by the TA and you may never receive a reply to your email or it may take a long period of time to receive one.