COMP2611 Computer Organization Fall 2022

Programming Project: The Gold Miner

Submission deadline: 11:55PM, Nov 30 via Canvas

1 Introduction

In this project, you are going to implement the Gold Miner game with MIPS assembly language. Figure 1 shows a snapshot of the COMP2611 Gold Miner game. You, as the gold miner, operate a hook to snatch up golds and gems. Your boss gives you a "reasonable" quota to meet and a "more than enough" deadline to deliver at each level.

Fig. 1: The snapshot of the Gold Miner

The gold miner works in a mine that is full of gold, gems, and rocks. The player mines the minerals using a hook. The hook is attached to a winch and swings in a pendulum. The player needs to skillfully choose the right angle to shoot the hook and aim at the desired mineral. When the hook is attached to a mineral, the gold miner reels them in. The different sizes and mass of the minerals lead to different rewinding speeds — the lighter the hooked object, the faster the

winch rewinds. For example, rocks are the heaviest, so it takes a long time to haul in. The player can throw some dynamites in the hook's pointing direction to destroy the minerals.

When the hooked mineral reaches the winch, it is sold immediately to debit the company's balance. Once the balance exceeds the quota, the gold miner proceeds to the next level, or the player wins the game if it is the last level. However, if the timer is up and the gold miner fails to meet the quota, they will be fired, and the player loses the game.

The COMP2611 Gold Miner game is implemented by a modified Mars emulator (Java-based) and MIPS. The video and audio effects, simple algorithms which control the game object, and other miscellaneous tasks are done in the Mars emulator. On the other hand, the main game logic and your tasks in the programming assignment are implemented in MIPS.

2 Game coordinate system

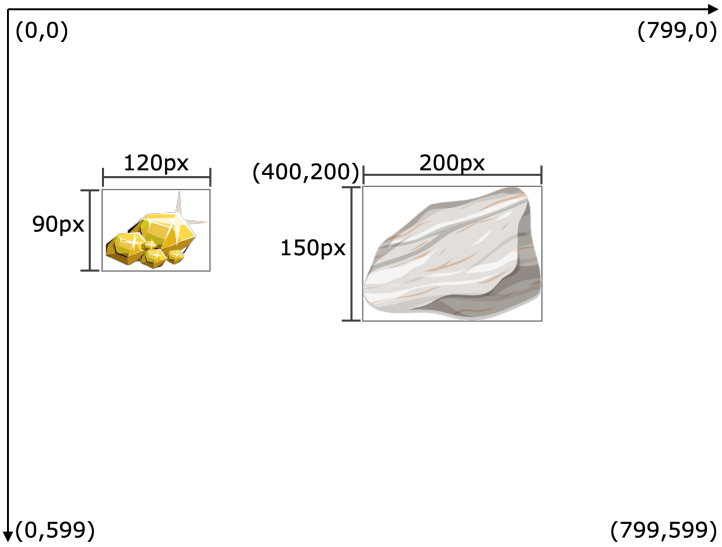

Fig. 2: The coordinate system of the game and rectangular objects.

The game screen is of 800 × 600 pixels as illustrated in Figure 2. The top-left is the origin of the coordinates, denoted as (0, 0); Likewise, the bottom-right is (799, 599). The value of x-axis increases from the left to the right and the value of y-axis increases from the top to the bottom. This follows Java Swing coordinate system, which is used to support GUI in the game.

For simplicity, all game objects are with images of rectangular shape. The top left coordinate of each object is used to represent its location. For example, the

rock in Figure 2 is at position (400, 200). Its width is 200px and height 150px. Notice that the size here is enlarged for demonstration and is not the same as in the game.

3 Game objects

There are five types of game objects: hook, dynamite, gold, gem, and rock. Every object has attributes as listed below. You can manipulate the attributes through appropriate syscall services (details in Section 2).

- Location: the top-left (x, y) coordinate which indicates the object's current location. The next location of the game object depends on its current location, moving direction, and moving speed.
- Speed: the integer variable for the game object's moving speed in pixels per iteration.
- Direction: the integer variable for the game object's moving direction in degrees. Let 0 denotes the direction that the object is naturally placed in gravity and being perpendicular to the ground, or in traditionally 270° of the Cartesian coordinate system. Adding positive degrees to current direction means moving in anti-clockwise, and negative degrees clockwise.
- Price: the integer variable indicating the market value of a game object. For example, golds and gems are worth $500 and $2000, respectively. Rocks are worthless.

Object	ID	Width	Height	Speed	Price	Initial location
Hook	1	22	32	12	0	(389, 100)
Gold	[2, 47]	60	45	6	500	Random
Gem	[2, 47]	30	18	9	1500	Random
Rock	[2, 47]	100	75	3	0	Random
Dynamite	[60, 89]	30	45	15	0	(385, 80)

Table 1: The game objects' attributes.

4 Game details

4.1 Game level initialization

All levels are randomly generated by allocating different numbers of minerals to the mine. The number of gems, golds, and rocks in each level are $2 \times n$, $3 \times n$, and $4 \times n$ respectively, where n is the level. Likewise, there are $1 \times n$ dynamites at the level n, i.e., 1 in level 1. The subsequent levels are more challenging than the first level in terms of higher quotas and tighter time limits. Still, there are also more minerals to showcase the player's skills.

The initial location of minerals is randomized at each level. They are randomly distributed to a grid system to simplify the game level design and avoid overlaps.

The 800×600 pixels game screen has two parts: 1) the top 800×100 region is reserved for the user interface and the winch, and 2) the bottom 800×500 region is the cave burying minerals. The cave is further divided into 8×6 grids, which have six rows and eight columns, with 25px margins on the left and right. Each grid has the size of 100×75, which is equivalent to the size of the largest mineral – the rock. Two grids are forbidden because they are close to the initial hook's location, so there are 46 grids for allocation. The grid system is maintained in the Mars simulator, and is reset whenever setting the game level is called. You can acquire a new assigned location with the corresponding syscall services (details in Section 2). You can also implement your level design logic to make the game more fun and challenging to play.

4.2 Winch, hook, and dynamite

The hook demonstrates three states: **Swinging**, **Shooting**, and **Rewinding**.

Swinging: The winch located at (400, 80) is the center that connects the hook with a rope (20px long). The hook's direction is initialized at 0 degrees. The desired movement of the hook is to swing in a pendulum, having the direction change in 3 degrees per iteration. The range of movement should be limited from -85 degrees to 85 degrees. Moving in clockwise requires updating the current direction with negative degrees.

Shooting: The player can press the space bar to shoot the hook in its pointing direction. This also stops the hook from swinging. Once it fires, the player must wait for the winch to rewind before shooting it again. The rope connecting the winch and the hook becomes infinitely long to reach the boundaries.

Rewinding: The hook starts rewinding until it goes beyond the boundaries or hits any mineral. In both cases, the winch hauls the hook back in its shooting speed (if it hits nothing) or the mineral's speed (if it hits a mineral). The rewinding stops when the hook reaches the winch. Then, the hook is reset to its original position and speed and starts swinging again. At the same time, any hooked mineral is sold immediately and transfer to the company's balance.

The player can throw a stick of dynamite to destroy anything caught in the hook by pressing the "D" key. The dynamites are thrown from the winch towards the direction where the hook points to. Dynamites can be thrown at any time regardless of the hook's shooting status. The dynamites are not replenishable until the next level.

4.3 Collision detection: hook and mineral

The interaction between different game objects depends heavily on collision detection. It is required to decide whether the hook hits any minerals, the hook reaches the boundary or the winch, and the dynamites touch any minerals. Although the images of the game objects seem irregular, we consider two game

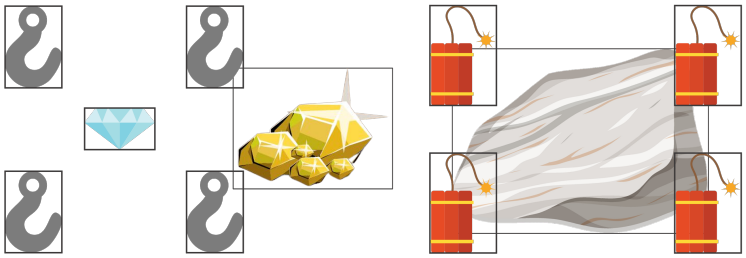

Fig. 3: Three examples of collision detection: 1) the gem collides with nothing; 2) the gold collides with two hooks but no dynamites; 3) the rock collides with all four dynamites.

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. Figure 3 shows three examples of collision detection. Note that the rock is considered to collide with all four dynamites, even if the top-left dynamite seems separated in human eyes.

Collision detection is fundamental to many vision-related tasks. A simple approach is to consider the coordinates of the four corners. Take the rock as an example; we can check whether at least one corner of the rock is located in the dynamite image region (i.e., the two objects intersect). Given the location (i.e., coordinate of the top-left corner) and the dimensions of the two objects, the calculation is straightforward.

Another approach is to consider the inverse, which lists all the conditions where collision must not occur. For example, a simple check is implemented to see whether the hook goes beyond the boundaries. There are three conditions: the hook's x-location ≥ 850, the hook's x-location ≤ -50, and the hook's y-location ≥ 650. Any condition that holds will trigger the "rewinding" state.

4.4 Wining and Losing Conditions

The game has three levels. The player passes a level when their balance meets with the quota. The player wins the game when they passes all levels. The player loses, and the game terminates immediately when the player fails to meet the quota within the given time.

Many of you must be good game players. We hope you can enjoy the game made with MIPS after you get all tasks done!

5 Game Implementation

5.1 Data Structure

The game-related data are saved in quite a few variables and simple 1D arrays. All data structures used are static. Check the data segment at the very beginning of the skeleton code to know more details.

5.2 Game Loop

The game is initialized with the `game_level_init` function which specify the parameters for the first game level. Then, the `init_game` function creates the specified number of game objects and places them in game canvas.

After that, the game begins looping over the `main_loop` function. Each iteration follows a collection of procedures, e.g., checks winning and losing conditions, updates the game status, checks collisions among game objects, gets the keyboard input, moves the game objects, etc. Towards the end of each game iteration, the screen is refreshed to reflect the changes in the game.

6 Tasks

When you code with 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 Gold Miner game in MIPS assembly is challenging, but you won't start everything from scratch. The fancy 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.

'Divide-and-conquer' strategy is used in the skeleton. The skeleton code is well organized with sub-tasks handled by different MIPS procedures. 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 1-6) 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). Feel free to draw the flow chart with drawing tools or by hand.

Your flowchart should

- include enough details, e.g. major procedures should be included in your flow chart.
- have good layout so that it's easy to trace and understand.

6.2 Task 1 to 6 Programming Tasks

Once you build up a big picture of the project, you can zoom in to the following coding tasks. Table 2 lists all the programming tasks you need to implement. You should not modify the skeleton code except the TODO sections, unless you are well aware of the risks brought by the modification. Try to follow good conventions, e.g. comment your code properly and use registers wisely.

You can discuss with your friends if you encounter difficulty in the project, or actively seek help from the teaching team. But every single line of your code should be your own work (not something copied from your friends). Do not show your code to others. Do not upload your code to GitHub or Cloud unless you set the access right to be private.

Task	Description
Task1: `move_hook_swing`	This procedure moves the hook by changing its direction for one game iteration. To swing in a pendulum, the hook should stay between -85 to 85 degrees. Input: - Output: -
Task2: `check_intersection`	This procedure checks whether the two input rectangles are intersected. Notice that the coordinates are passed through the stack. Input: recA((x1,y1), (x2,y2)), recB((x3, y3), (x4, y4)) Output: $v0=1 if the two rectangles intersect, $v0=0 otherwise
Task3: `check_hook_hit`	Check whether the hook collides with a specified mineral. This procedure pushes the coordinates of two game objects into the stack, and then calls the procedure check_intersection. Input: $a0 = mineral_index Output: $v0=1 if the two objects intersect, $v0=0 otherwise
Task4: `update_mineral_at_winch`	This procedure manages the operations after the mineral has reached the winch. They include updating level balance and destroying the mineral. Input: $a0 = mineral_id Output: -
Task5: `check_dynamite_hit`	Check whether the dynamite collides with any mineral. This procedure loops over the minerals to find any intersected minerals. It should also destroy the intersected mineral before jumping back to return address. Input: $a0 = dynamite_id Output: $v0=1 if the two objects intersect, $v0=0 otherwise
Task6: `check_game_status`	Check whether the game should continue, has been won, or has been lost. Input: - Output: $v0 = 0 if not end, 1 if win, and 2 if lose

Table 2: Programming task description

7 Evaluation

Your project will be evaluated with a comprehensive grading scheme. Please read the following sections carefully to avoid unnecessary mark deduction.

7.1 Submission

You must ***ONLY*** submit one file: `comp2611_project_yourStudentID.zip`.

The zip file should include a picture/pdf of the flow chart, and your completed MIPS code for the project `comp2611_project_yourStudentID.asm`. Please write down your name, student ID, and email address at the beginning of the files.

Submit your file via Canvas. The deadline is a hard deadline. Try to avoid uploading in the last minute. If you upload multiple times, we will grade the latest version by default.

7.2 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.

For suspected plagiarism cases, we will invite you to a Q&A session and explain your code. If you fail to do so, your project score will be 0 and an additional 10% will be deducted from your course overall score.

7.3 Self-proposed MIPS project

If you're excited in MIPS programming and wish to further challenge yourself, you can implement additional functionalities in the Gold Miner game. The more sophisticated, and of course, more interesting version of the game includes different power-ups, a store that sells dynamites, movable winch, and various obstacles that put extra pressure on players.

You're also welcomed to create a brand-new version of the Gold Miner or any other game from scratch with your imagination.

Email course instructor (lixin@ust.hk) for expression of interest and your plan by Nov 8 (Monday). For those who propose to work on the extension of the game, all or some of the original tasks in the programming assignment can be replaced. If you're proposing a brand-new version of the Gold Miner or any other game, you don't need to work on the original programming assignment. NOTE: You have to get prior approval before you start.

You will be invited to give a 10-min demonstration to show your work during study break.

8 Syscall Services

A modified MARS (`NewMars.jar`) with additional set of syscall services is needed to support the game (e.g. fancy UI, music, etc.). Table 3 and Table 4 list all the provides 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.

The GUI part of this game is implemented in modified MARS, but the game logic, e.g. whether the dynamite hits a mineral, is determined by MIPS code.

Service	Code	Parameters	Result
Create the game screen	100	$a0 = base address of title's string $a1 = width $a2 = height	Create a game instance.
Refresh the game screen	101		Draw all objects on the game screen.
Game sound control	102	$a0 = sound ID: 0 for background music, 1 for winning sound effect, 2 for losing sound effect, 3 for emit sound effect, 4 for hit sound effect $a1 = control: 0 for play once, 1 for play in loop indefinitely, 2 for stop	Control the sound effect associated with the sound ID.
Set game level information	103	$a0 = info type: 0 for game level, 1 for level timer, 2 for level quota, 3 for level balance, 4 for available dynamites $a1 = the value	Set the game level information.
Get game level object location	104		$v0 = x-coordinate of the randomly assigned location in mineral grid system $v1 = y-coordinate of the randomly assigned location in mineral grid system

Table 3: Syscall Services 100 - 104

Create a game object	105	$a0 = ID of the object $a1 = object type: 0 for hook, 1 for gold, 2 for gem, 3 for rock, 4 for dynamite $a2 = x-coordinate $a3 = y-coordinate	Create a game object and set its location accordingly.
Create a text	106	$a0 = ID of the object $a1 = x-coordinate $a2 = y-coordinate $a3 = the display string	A new text of the given ID and type is created.
Destroy game object	107	$a0 = ID of the object	Destroy the game object in java memory
Get object's location	108	$a0 = ID of the object	$v0 = x-coordinate $v1 = y-coordinate
Set object's location	109	$a0 = ID of the object $a1 = x-coordinate $a2 = y-coordinate	The object's location is set to the corresponding parameter.
Update object's location	110	$a0 = ID of the object	The object's location is updated according to the current location, direction, and speed.
Get object's speed	111	$a0 = ID of the object	$v0 = the object's speed
Set object's speed	112	$a0 = ID of the object $a1 = the object's speed	The object's speed is set to the parameter.
Get object's direction	113	$a0 = ID of the object	$v0 = the object's direction
Set object's direction	114	$a0 = ID of the object $a1 = the object's direction	The object's direction is set to the parameter.
Update the hook's direction	115	$a0 = ID of the hook $a1 = the delta degree	The hook's location is updated according to the current location, delta degree, and radius from winch.
Get object's price	116	$a0 = ID of the object	$v0 = the object's price
Set object's price	117	$a0 = ID of the object $a1 = the object's price	The price of the object is set to the parameter.
Get object's isHooked status	118	$a0 = ID of the object	$v0 = 1 for isHooked, 0 for not isHooked.
Toggle object's isHooked status	119	$a0 = ID of the object	The object's isHooked status is negated.
Get the hook's isShoot status	120	$a0 = ID of the hook	$v0 = 1 for isShoot, 0 for not isShoot.
Toggle the hook's isShoot status	121	$a0 = ID of the hook	The hook's isShoot status is negated.
Get the hook's isClockwise status	122	$a0 = ID of the hook	$v0 = 1 for isClockwise, 0 for not isClockwise.
Toggle the hook's isClockwise status	123	$a0 = ID of the hook	The hook's isClockwise status is negated.

Table 4: Syscall Services 105 - 123