# COMP2611: Computer Organization

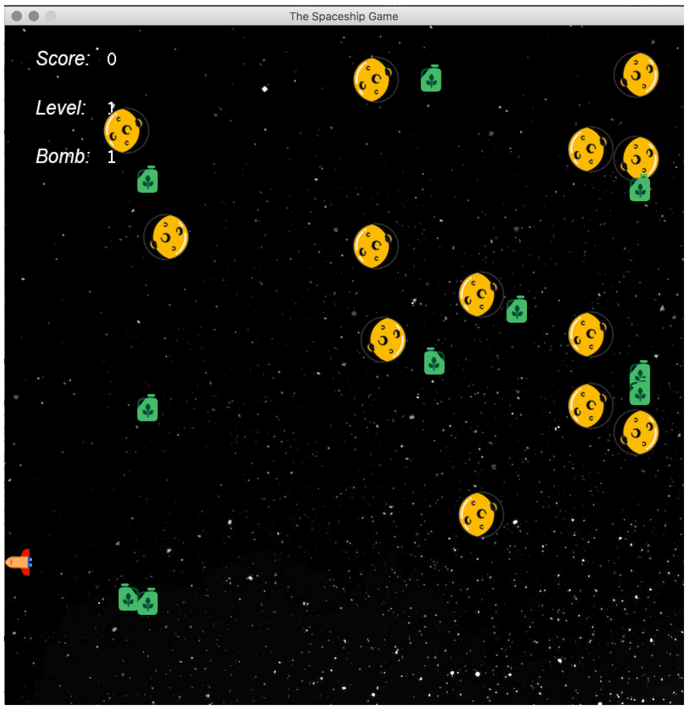
# Spring 2017

# Programming Project: The Spaceship Game

# Due Date: 9 May, 2017, 23:59

## Introduction

In this project, you will complete a game called *The Spaceship Game*. A snapshot of the game is shown below. By controlling the spaceship, your target is to 1) drop bomb to destroy all aerolites, and 2) collect all fuels. Completion of both tasks promotes you to a higher level. After passing the 3rd level, you win! However, if the spaceship collides with aerolites, you will lose this game.



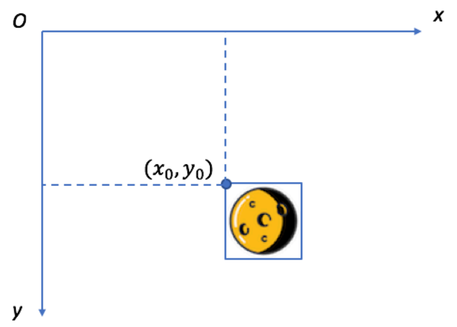
## Game Objects

The game is running on an 800-pixel square screen. The x-axis and y-axis of the coordinate system are rightward and downward (from 0), respectively. Each game object: spaceship, aerolite or fuel, is presented as a square image. The location of an object is given with its top-left corner coordinate. The lengths of objects are listed as follows:

| Object | Image size (in pixel) | Status | Hit Point |
|---|---|---|---|
| Spaceship | 32 x 32 | | |
| Aerolite | 60 x 60 | Set to 0 if removed from screen. Set to 1 if displayed on screen | (Basic) 10 points for collision (Bonus) 10 points for complete collision, 5 points for partial collision |
| Fuel | 32 x 32 | | |
| Bomb | 16 x16 | | |

The game screen is illustrated below. The bottom-right pixel of the game screen is at (799, 799). We refer aerolites and fuels as **target objects**.

In terms of object status, more specifically, the bomb status 1 indicates it is available for ejecting, value 0 indicates it is not available. The Fuel status 1 indicates it is available for collection, value 0 indicates it is already collected. The Aerolite status 1 indicates it is not destroyed, value 0 indicates it is already destroyed.



## Game Initialization

The game has 3 levels. Game score is initially 0 at Level 1 and carries along all 3 levels. Player sets the number of fuels (up to 10) in initialization. The aerolite number is 3 more than the fuel number. For the following levels, the number of aerolites and fuels are increased by 3 (refer to procedure `main_next_level`).

The spaceship initially locates at (x, y) = (384, 384), which is the center of the screen, and moves rightward at the speed of 8. At the beginning of each level, the x-coordinate of target object (aerolite or fuel) is zero and the y-coordinate is randomly set between 0 and 800 (minus whose size).

## Object Movements

**Target objects**: Move horizontally only. When it reaches the screen boundary, it is bounced backward. The moving speed of each target object is randomly set between 1 and 10.

**The spaceship**: Moves either horizontally or vertically by user control (keys *a, d, w, s*), and stops at the boundary, at the speed of 8.

**The bomb:** ejected from the center of the spaceship at the speed of 16, with the same direction as the spaceship.

## Collision Detection

Check of intersection (or collision detection) is based on the rectangles of those object images, and is processed by procedure `IsIntersected`.

**The spaceship** is destroyed when intersecting with aerolite(s) and game is over. When the spaceship intersects with fuel, the fuel is removed from screen and game continues. There is only one (recyclable) bomb available. Once a bomb is released and still flying, user can't drop another bomb. The bomb disappears when it crosses out of the boundary, and is ready to be used again.

**The bomb** may collide (intersect) with aerolites or fuels. Intersection with a fuel object is ignored, both bomb and fuel fly along their desired pathway. When the bomb collides with aerolites, two levels of collision (partial or complete) are defined (details given in the latter section). You need to distinguish them to earn bonus points.

- Partial collision: 5 points are added to the game score, and the aerolite still flies in the screen.
- Complete collision: 10 points are added to the game score, and the aerolite disappears in the screen.

The bomb explodes (or disappears) after hitting an aerolite. It's then ready to be used again.

*Note that though aerolites have circular shapes, we detect collision between their image rectangles.*

## Game Winning or Losing

Each level of the game requires the player to destroy all aerolites and collect all fuels. The game has 3 levels. If the spaceship runs into aerolites, game is over. The game score indicates the total points you get from this game. Press 'q' whenever you want to quit the game.

## Implementation

All the game objects and status (score, level, bomb availability flag) are initialized at the beginning of the game. When initialization parameters (fuels number, random seed) are ready, the game runs with a loop of the following steps:

1. Get the current time (T1).
2. Remove any destroyed aerolites and collected fuels from the game screen.
3. Check for any keyboard input, which is stored using the Memory-mapped I/O scheme (see the section Hints for details). If an input is available, read it and perform the action for it as follows:

| **Input** | **Action** |
|---|---|
| q | Terminate the game |
| 1 (number) | Drop the bomb (when it is available) |
| w | Move the spaceship upward |
| s | Move the spaceship downward |
| a | Move the spaceship leftward |
| d | Move the spaceship rightward |

4. Collision detection for the spaceship. Check for any collision with aerolites and fuels. If collision detected, update the status of involved objects.
5. Collision detection for the bomb. Check for any collision with aerolites. If collision detected, update the status of involved objects and the game score.
6. Update the image for partially collided aerolite.
7. Check whether the game ends. If it is won or lost, display the corresponding message "You won!" or “You lost!”, and then terminate the game. Or you reach next level, then re-initialize the game and jump to Step 9.
8. Move the spaceship.
9. Move aerolites and fuels.
10. Move the bomb.

11. Redraw the game screen with the updated location and image of the game objects and game state information.

12. 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.

## 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*.

Read the skeleton code *spaceship.s* carefully and implement the following procedures. Your code should be in MIPS assembly language.

| **Procedure** | **Task** |
|---|---|
| `initGame` | Initialization of a game. Assign initial location, image index, speed, status to 1) the spaceship and the bomb, 2) aerolites and fuels (in loop). Procedure `randnum` is provided to generate random numbers. |
| `processInput` | Step 3. Responses to keyboard pressing. Including ‘1’, ‘q’, ‘a’, ‘w’, ‘s’, ‘d’.<br>(note it’s number 1, but not ‘l’ for ‘like’). |
| `IsIntersected` | Helper function, which checks whether two rectangles intersect each other. |
| `collisionDetectionSpaceship` | Step 4. Use procedure `isIntersected` to implement following actions for the spaceship: 1) lose the game if intersected with aerolites, and 2) collect fuels if intersected. |
| `collisionDetectionBomb` | Step 5. Use procedure `isIntersected` to implement the action for the bomb: damage the aerolite if intersected. |

| | |
|---|---|
| `moveSpaceship` | Step 8. Move the spaceship according to keyboard input (hint: refer to procedure `moveAeroliteFuel`). |
| `moveBomb` | Step 10. Move the bomb ejected from the spaceship (hint: refer to procedure `moveAeroliteFuel`). |
| **(Bonus, optional)**<br><br>`collisionDetectionBomb`<br>(modified, able to different partial and complete collision)<br><br>`updateDamagedImages` | Step 6. Add partial collision detection among bomb and aerolites in procedure `collisionDetectionBomb`,. When it's the first time that an aerolite is hit by bomb, if it's partial collision, update the aerolite image from *aerolite_*.jpg* to *aerolite_d_*.jpg* in procedure `updateDamagedImages`**.** If it's a complete collision, the aerolite is destroyed. The bomb disappears and is ready to be re-used. If it's the second time an aerolite is hit by bomb, it's always destroyed. |

To work with the game objects, you must use the data structures (of MIPS word array) defined at the beginning of the code: **spaceship (6 words)**, **aerolites (5 words)**, **fuels (5 words)** and **bombs (6 words)**. Read the code comments for detailed description.

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

The game images are provided and are listed below. To set a game object to use a certain image, set the object's property "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.

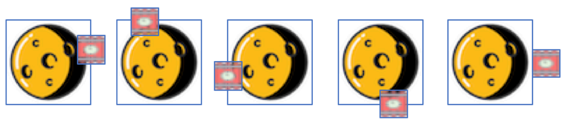
| **Image index** | **Image** | **Description** |
|---|---|---|
| 0 | background.jpg | Game screen background |
| 1 | spaceship_right.png | Spaceship moving right |
| 2 | spaceship_left.png | Spaceship moving left |
| 3 | aerolite_right.png | Aerolite moving right |

| | | |
|---|---|---|
| 4 | aerolite_left.png | Aerolite moving left |
| 5 | aerolite_d_right.png | Damaged aerolite moving right |
| 6 | aerolite_d_left.png | Damaged aerolite moving left |
| 7 | fuel_right.png | Fuel moving right |
| 8 | fuel_left.png | Fuel moving left |
| 9 | bomb.png | Bomb image |
| 10 | spaceship_up.png | Spaceship moving up |
| 11 | spaceship_down.png | Spaceship moving down |

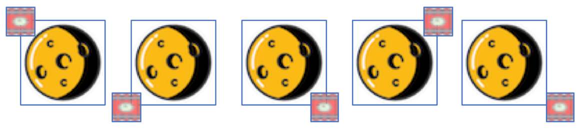
## Bonus: Partial/Complete Collision Detection among Bomb and Aerolite

There is 20% bonus mark (e.g., 20 marks if the full project mark is 100) for implementing *partial* collision detection. Complete collision among bomb and aerolite happens when one **full** edge of bomb rectangle touches or goes into the aerolite rectangle. All other cases are partial collision. Below are some examples of complete and partial collision.

Complete Collision



Partial Collision



## Hints

The keyboard inputs for playing the game are stored using the Memory-mapped I/O scheme. The procedure `getInput` in the file *spaceship.s* is provided to check and read any input stored using this scheme.

For any two rectangles, say A and B, they do not intersect (or touch) each other if and only if one of the following conditions holds:

- A's largest x-coordinate is smaller than B's smallest x-coordinate
- A's smallest x-coordinate is larger than B's largest x-coordinate
- A's largest y-coordinate is smaller than B's smallest y-coordinate
- A's smallest y-coordinate is larger than B's largest y-coordinate

The functions `randnum` is provided for generating a random number.

Read the code comments of all these provided procedures for how to use them. Use of those procedures is optional.

## Submission

You should submit the file *spaceship.s* 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 *spaceship_<Your student ID>.s*. 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. Therefore, you should make sure that your submitted codes can be executed properly in the modified Mars program.

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 Spring 2017 examination period after the project due date).