CSC258 Project *Breakout* Report

Fuyang (Scott) Cui

November 29, 2022

Contents

1 Memory Layout **2**
1.1 Immutable Data . . . 2
1.2 Mutable Data . . . 2
1.3 The Stack, Registers, and 4
1.4 The Bitmap Display . . . 4
1.5 Screenshots of the Memory . . . 4

2 The Static Scene **5**

3 Collision Detections **6**
3.1 The `is_collide` Function . . . 6
3.2 The Collision Handlers . . . 7

4 Features **8**
4.1 Easy Features . . . 8
4.2 Hard Features . . . 9

5 How to play? **10**
5.1 Normal Mode . . . 11
5.2 Automatic Mode . . . 11
5.3 God Mode . . . 12

IMPORTANT: If the game is extremely laggy, please restart the MARS.

For unknown reasons, MARS starts laggying after few hours of coding and playing.

1 Memory Layout

1.1 Immutable Data

Part of the memory of my breakout consists of a bunch of immutable data. In my breakout, immutable data are read-only: my breakout never writes to them. We can only change these immutable variables in the source code, and the effects of the changes will reflect in my breakout once the program reruns.

Besides the given `ADDR_DSPL` and `ADDR_KBRD`, I reserved some more memory blocks for the immutable data in my breakout:

1. `WALL_ATTRIBUTES`.

   This variable represents the attributes of the basic building block of my breakout: the walls. I allocated 2 words for this variable and each word stores an integer.

   - `WALL_ATTRIBUTES + 0` represents the color of the wall.
   - `WALL_ATTRIBUTES + 4` represents the thickness of the wall.

2. `PADDLE_ATTRIBUTES`.

   Similarly, this variable represents the attributes of the paddle in my breakout.

   - `PADDLE_ATTRIBUTES + 0` represents the color of the paddle.
   - `PADDLE_ATTRIBUTES + 4` represents the length of the paddle.
   - `PADDLE_ATTRIBUTES + 8` represents the thickness of the paddle.

3. `BALL_ATTRIBUTES`.

   This variable represents the attributes of the ball in my breakout.

   - `BALL_ATTRIBUTES + 0` represents the color of the ball.
   - `BALL_ATTRIBUTES + 4` represents the radius of the ball.

1.2 Mutable Data

Most of the `.data` section is occupied by mutable data. Mutable data in my breakout are data that may be read and overwritten. Some mutable data may change before executing the instructions, and the effects of those changes will reflect to my breakout once the program reruns.

1. `BRICK_ATTRIBUTES`

   This variable represents the attributes of the bricks.

   - `BRICK_ATTRIBUTES + 0` represents the thickness of the bricks.
   - `BRICK_ATTRIBUTES + 4` represents the number of bricks per row.
   - `BRICK_ATTRIBUTES + 8` represents the space between rows.
   - `BRICK_ATTRIBUTES + 12` represents the space between bricks in a row.
   - `BRICK_ATTRIBUTES + 16` to `BRICK_ATTRIBUTES + 40` represents the colors of rows of bricks from top to bottom. There are 7 colors in total.

2. `GAME_STATUS`

   This is an integer signifying the status of the game. `GAME_STATUS == 0` means the game is freezed (paused), and `GAME_STATUS == 1` represents the game is running.

   If the game is paused, then the game loop only listens for keyboard inputs. Therefore, no collision detections, objects moving, and screen drawing happen during the freeze.

3. `PLAYER_STATUS`

   This is an integer that represents the number of "hearts" the player has. If the heart value drops to 0, then the player dies.

4. `WALL_AABB`

   This variable represents the collision boxes of the walls.

   - `WALL_AABB + 0` represents the lower bound of the top wall.
   - `WALL_AABB + 4` represents the right bound of the left wall.
   - `WALL_AABB + 8` represents the left bound of the right wall.

   If the ball hit any bound, the collision happens and the ball changes its direction(s). Detailed description of how the ball changes its direction will be illustrated in section 3.

5. `BALL_AABB`

   This variable represents the collision box of the ball. This variable is used for detecting collisions and moving the ball.

   - `BALL_AABB + 0` represents the x position of the upper left corner of the ball. I will denote it as x_0 of the ball.
   - `BALL_AABB + 4` represents the y position of the upper left corner of the ball. I will denote it as y_0 of the ball.
   - `BALL_AABB + 8` represents the x position of the upper right corner of the ball. I will denote it as x_1 of the ball.
   - `BALL_AABB + 8` represents the y position of the lower left corner of the ball. I will denote it as y_1 of the ball.

6. `BALL_DIRECTION`

   This variable represents the direction of the ball.

   - `BALL_DIRECTION + 0` represents x direction of the ball. A positive value represents a leftward direction. A negative value represents a rightward direction.
   - `BALL_DIRECTION + 4` represents y direction of the ball. A positive value represents a downward direction. A negative value represents a upward direction.

   This variable will be mutated every time the ball changes its direction. The `move_ball` function utilizes this variable and move the ball in the corresponding direction.

7. `PADDLE_AABB`

   This variable represents the collision box of the paddle. This variable is used for detecting collisions and moving the paddle.

   - `PADDLE_AABB + 0` represents the x position of the upper left corner of the paddle. I will denote it as x_0 of the paddle.
   - `PADDLE_AABB + 4` represents the y position of the upper left corner of the paddle. I will denote it as y_0 of the paddle.
   - `PADDLE_AABB + 8` represents the x position of the upper right corner of the paddle. I will denote it as x_1 of the paddle.
   - `PADDLE_AABB + 8` represents the y position of the lower left corner of the paddle. I will denote it as y_1 of the paddle.

8. `BRICKS_DATA`

   This variable represents the data of all bricks. Its capacity depends on the number of bricks we have. I implemented 10 levels and each level has a different number of bricks. Therefore, the design choice is that I allocate the space needed for the maximum number of bricks (1750 integers).

   Each brick can be fully characterized in my breakout using 5 integers. The first integer specifies the remaining health of the brick. If the health drops to 0, then the brick disappears. The health is decremented when the ball hit the brick. The 4 integers specify the collision box of the brick as usual.

9. `BRICK_SOUND_PITCH_OFFSET`

   This variable represents the current pitch offset of the sound generated by colliding bricks.

10. `MOVING_MODE`

    This variable represents the movement mode of the ball.

    - `MOVING_MODE = 0` represents that the ball will move in a discrete fashion. The player can fully control the ball and move the ball in any direction.
    - `MOVING_MODE = 1` (default) represents that the ball will move in a continuous fashion with directions specified by `BALL_DIRECTION`. This mode is used in a normal game setting.

1.3 The Stack, Registers, and ...

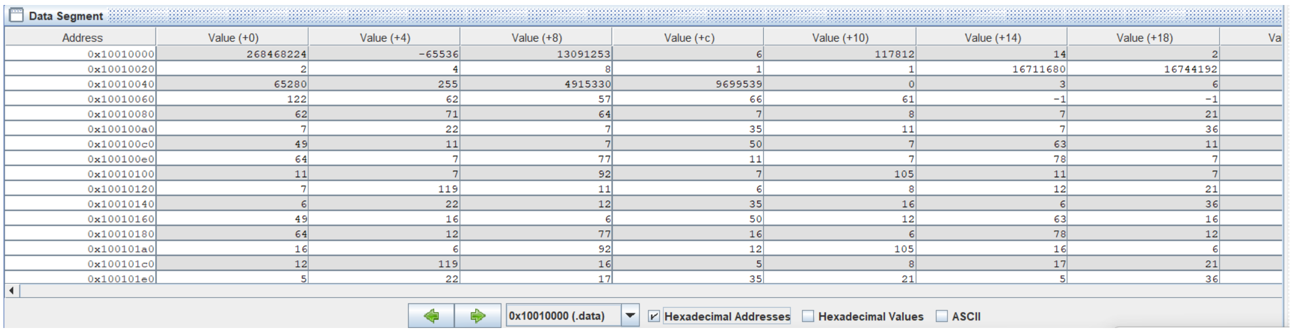
The stack and registers have no special usages. I use them as how Prof. Badr taught us to do.

1.4 The Bitmap Display

I set the unit width and height for my breakout to 4. Also, the display width is 512 and the display height is 256. Then, I constructed a coordinate system. The coordinate system starts from the left corner (the origin) and ends at the right corner $(128, 0)$ and the lower left corner $(64, 0)$. All variables related to thickness, length, and ... are measured in units.

The `draw_row`, `draw_block_unit`, and `coordinate_to_display` functions collaborate together to draw the entire scene out.

1.5 Screenshots of the Memory



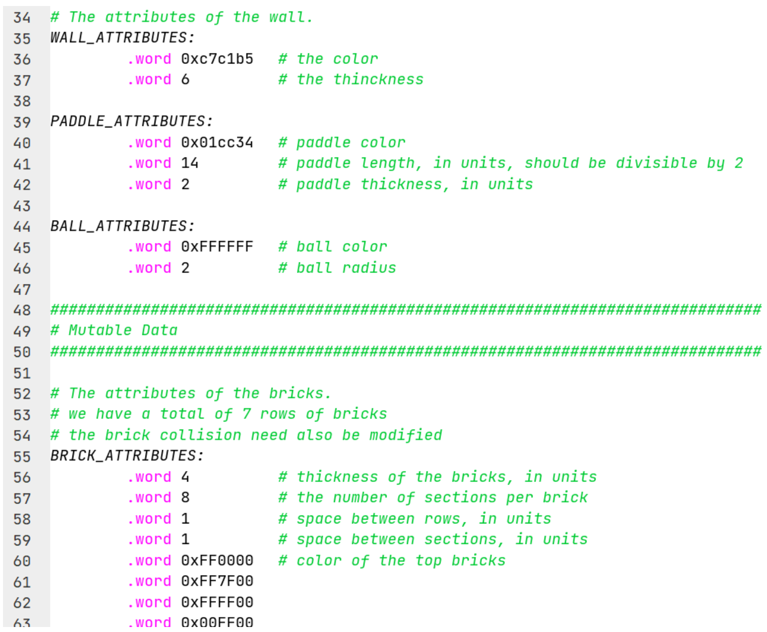
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Va
0x10010000	268468224	-65536	13091253	6	117812	14	2	
0x10010020	2	4	8	1	1	16711680	16744192	
0x10010040	65280	255	4915330	9699539	0	3	6	
0x10010060	122	62	57	66	61	-1	-1	
0x10010080	62	71	64	7	8	7	21	
0x100100a0	7	22	7	35	11	7	36	
0x100100c0	49	11	7	50	7	63	11	
0x100100e0	64	7	77	11	7	78	7	
0x10010100	11	7	92	7	105	11	7	
0x10010120	7	119	11	6	8	12	21	
0x10010140	6	22	12	35	16	6	36	
0x10010160	49	16	6	50	12	63	16	
0x10010180	64	12	77	16	6	78	12	
0x100101a0	16	6	92	12	105	16	6	
0x100101c0	12	119	16	5	8	17	21	
0x100101e0	5	22	17	35	21	5	36	

0x10010000 (.data) [✓ Hexadecimal Addresses] Hexadecimal Values ASCII

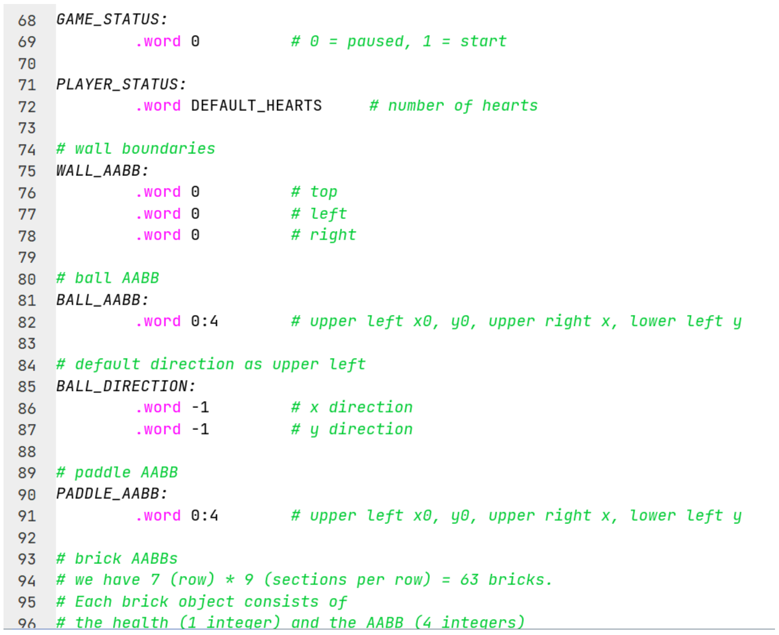
Figure 1: A screenshot of the memory of `.data` section.

As we can observe from Figure 1, data are laid out in the order I introduced.



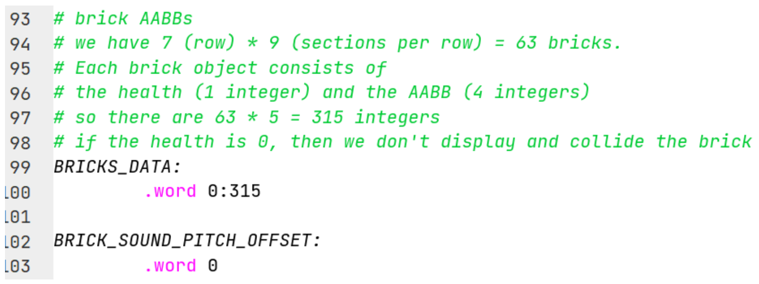
```
# The attributes of the wall.
WALL_ATTRIBUTES:
        .word 0xc7c1b5   # the color
        .word 6          # the thinckness

PADDLE_ATTRIBUTES:
        .word 0x01cc34   # paddle color
        .word 14         # paddle length, in units, should be divisible by 2
        .word 2          # paddle thickness, in units

BALL_ATTRIBUTES:
        .word 0xFFFFFF   # ball color
        .word 2          # ball radius

############################################################################
# Mutable Data
############################################################################

# The attributes of the bricks.
# we have a total of 7 rows of bricks
# the brick collision need also be modified
BRICK_ATTRIBUTES:
        .word 4          # thickness of the bricks, in units
        .word 8          # the number of sections per brick
        .word 1          # space between rows, in units
        .word 1          # space between sections, in units
        .word 0xFF0000   # color of the top bricks
        .word 0xFF7F00
        .word 0xFFFF00
        .word 0x00FF00
```

Figure 2: A screenshot of the `.data` section.



```
GAME_STATUS:
        .word 0          # 0 = paused, 1 = start

PLAYER_STATUS:
        .word DEFAULT_HEARTS     # number of hearts

# wall boundaries
WALL_AABB:
        .word 0          # top
        .word 0          # left
        .word 0          # right

# ball AABB
BALL_AABB:
        .word 0:4        # upper left x0, y0, upper right x, lower left y

# default direction as upper left
BALL_DIRECTION:
        .word -1         # x direction
        .word -1         # y direction

# paddle AABB
PADDLE_AABB:
        .word 0:4        # upper left x0, y0, upper right x, lower left y

# brick AABBs
# we have 7 (row) * 9 (sections per row) = 63 bricks.
# Each brick object consists of
# the health (1 integer) and the AABB (4 integers)
```

Figure 3: A screenshot of the `.data` section.



```
# brick AABBs
# we have 7 (row) * 9 (sections per row) = 63 bricks.
# Each brick object consists of
# the health (1 integer) and the AABB (4 integers)
# so there are 63 * 5 = 315 integers
# if the health is 0, then we don't display and collide the brick
BRICKS_DATA:
        .word 0:315

BRICK_SOUND_PITCH_OFFSET:
        .word 0
```

Figure 4: A screenshot of the `.data` section.

2 The Static Scene

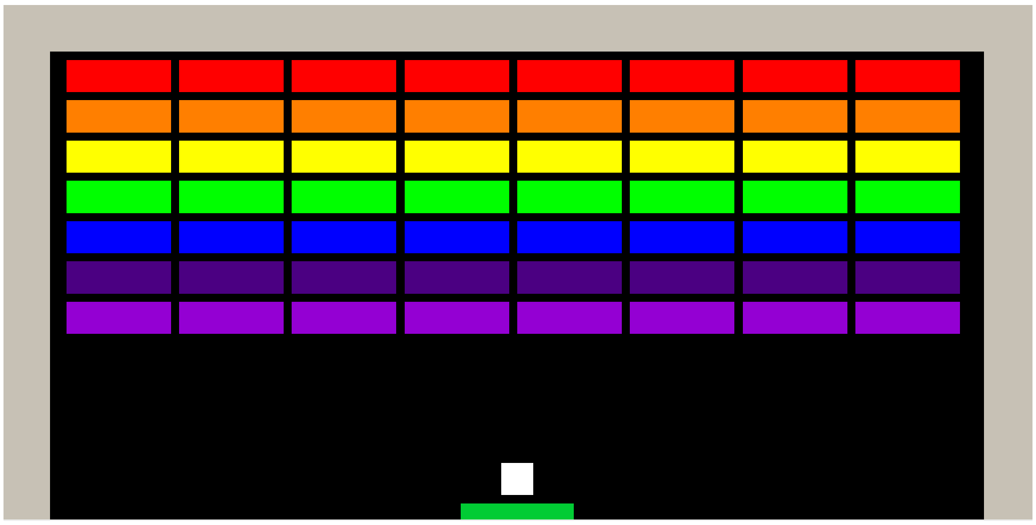


Figure 5: A screenshot of a static scene of my breakout.

3 Collision Detections

3.1 The `is_collide` Function

This function takes two parameters: The address of the first collision box and the address of the second collision box. Collision boxes are specified in section 1.2. This function returns 2 integers: `$v0` specifies if there is a collision between the two collision boxes, and `$v1` specifies the type of the collision if exists, or `-1` otherwise.

This function divides collisions into 4 types. For simplicity, let's assume the first parameter is the ball and the second parameter is a brick.

1. Type 1: If the ball hit the brick on a horizontal side, the function returns a type 1 collision.

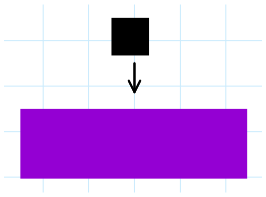


Figure 6: A illustration of a Type 1 collision.

2. Type 2: If the ball hit the brick on a vertical side, the function returns a type 2 collision.

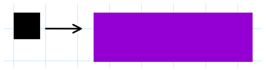


Figure 7: A illustration of a Type 2 collision.

3. Type 3: If the ball hit the brick from a contrary direction on a corner, the function returns a type 3 collision.

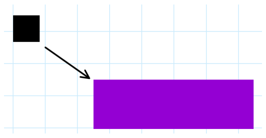


Figure 8: A illustration of a Type 3 collision.

4. Type 4: If the ball hit the brick from the same direction on a corner, the function returns a type 4 collision.

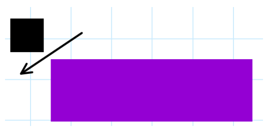


Figure 9: A illustration of a Type 4 collision.

3.2 The Collision Handlers

Different types of collisions are handled differently.

1. For a Type 1 collision, the handler flip the vertical direction of the ball.

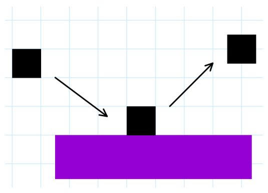


Figure 10: A illustration of a Type 1 collision handler.

2. For a Type 2 collision, the handler flip the horizontal direction of the ball.

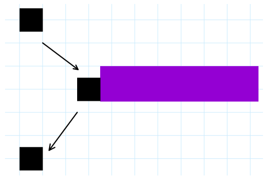


Figure 11: A illustration of a Type 2 collision handler.

3. For a Type 3 collision, the handler flip both the vertical and horizontal directions of the ball.

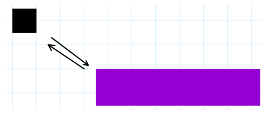


Figure 12: A illustration of a Type 3 collision handler.

4. For a Type 4 collision, the handler doesn't flip any direction.

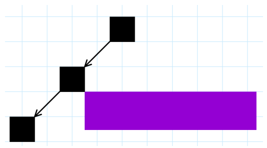


Figure 13: A illustration of a Type 4 collision handler.

These illustrations demonstrate my implementation of how will the ball change directions when it collides.

4 Features

I implemented easy feature 1, 4, and 5 and hard feature 3 and 4 in the handout.

4.1 Easy Features

1. Support "multiple lives" (at least 3) so that the player can attempt to make all the bricks disappear multiple times. The state of previous attempts (i.e., broken bricks) should be retained for subsequent attempts.

   To implement this feature, I added a new mutable variable: `PLAYER_STATUS`. A detailed description of this variable can be viewed in 1.2 Mutable Data.

   Then, I modified the function `move_ball` which is used to move the ball with directions specified by `BALL_DIRECTION`. If the $y1$ of the ball hits the void, then the function decrements `PLAYER_STATUS`, sleeps for 2 seconds, and then redraws the paddle and the ball at their initial positions.

   If the `PLAYER_STATUS` was already 1, then the function will quit the game.

   Thus, in this way, the previous attempts are retained and the player can play multiple times. We can set the default value of `PLAYER_STATUS` to $n \in \mathbb{N}$ to give the player n opportunities.

4. Add sound effects for different conditions like colliding with the wall, brick, and paddle and for winning and game over.

   I added some random sound effects for colliding with the wall and the paddle. Then, I added a variable `BRICK_SOUND_PITCH_OFFSET` to record the current offset of the pitch of the sound effect for colliding with a brick. The offset will start from 0 and increment to 24. In this way, a pitch-increasing sequence of piano sounds will be played when the ball collides bricks.

5. Allow the user to pause the game by pressing the keyboard key `p`.

To implement this feature, I added a new mutable variable `GAME_STATUS`. A detailed description of this variable can be viewed in 1.2 Mutable Data.

If `GAME_STATUS` is 0, then my breakout only processes the keyboard inputs. Therefore, nothing moves if the game is paused. If `GAME_STATUS` is 1, then the game proceeds as normal.

The keyboard event for the key `p` flips the `GAME_STATUS`, so we can use `p` to pause/continue the game.

4.2 Hard Features

3. Require bricks be hit by the ball multiple times before breaking. Demonstrate the players progress toward breaking a brick in some way (e.g., different colours).

   In `BRICKS_DATA` variable (detailed description in 1.2 Mutable Data), each brick is associated with a health value. A collision will cause the health value of the collided brick decrement. Once the health value hit 0, the brick is considered broken, the brick won't be drawn again, and the collision box of it is considered invalid.

   Thus, I modified the collision detection loops and the function to redraw the bricks to support the functionality.

   I defined the top row of bricks has the highest initial health value (7) and the bottom row of bricks has the lowest initial health value (1). I associated the health values with the colors of the brick. Low-health bricks are displayed in bottom-row color and high-health bricks are displayed in top-row color.

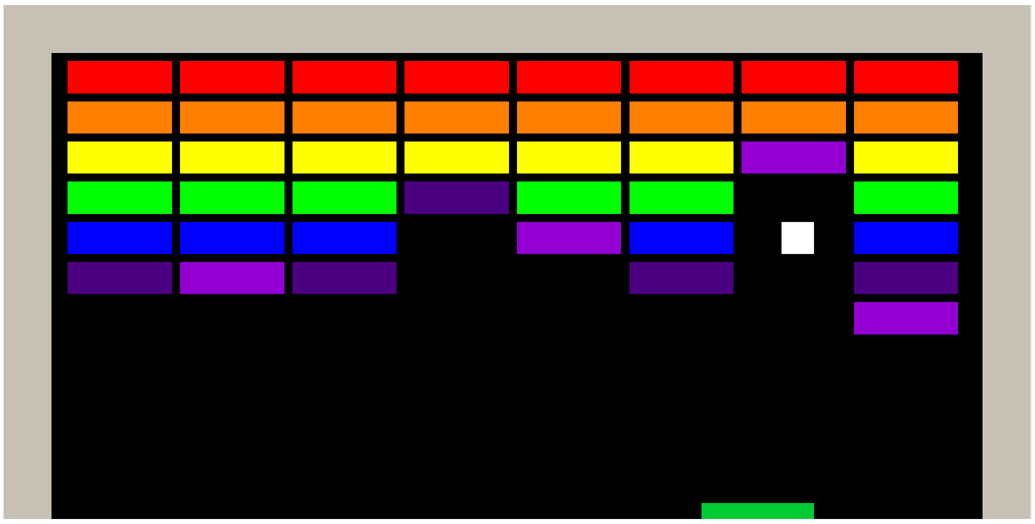


Figure 14: A illustration of the hard feature.

4. Create a second level with a different layout of bricks.

   With the help of `BRICK_ATTRIBUTES`, I can create different layouts easily.

   I define layout 1 to be 1 brick per row, layout 2 to be 2 bricks per row, and ....

   I added some keyboard listeners for keys `1`, `2`, ... Each listener pauses the current game, reset the display, update the brick attributes, reset the player's status, and finally reinitialize to the new layout. Therefore, I can switch to different levels anytime I want.

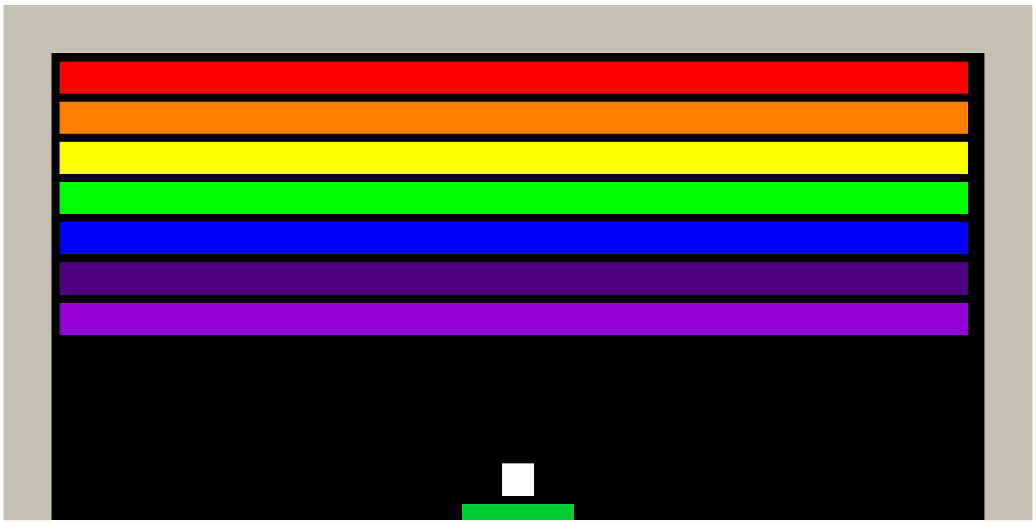


Figure 15: A illustration of the level 1 layout.

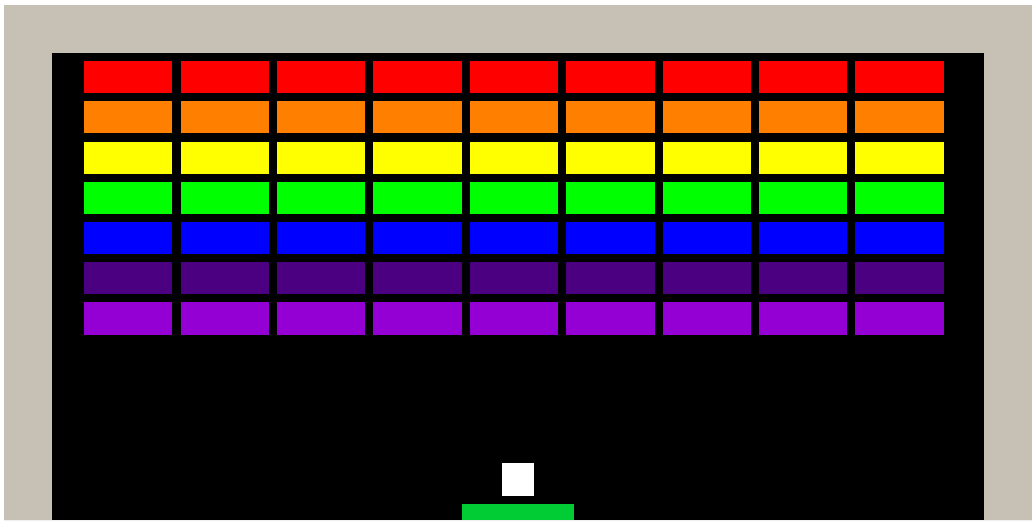


Figure 16: A illustration of the level 9 layout.

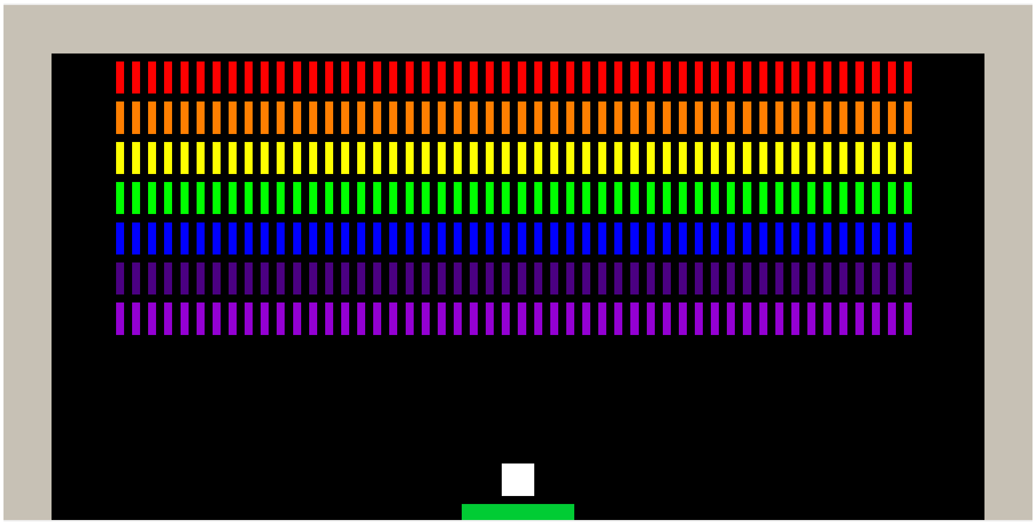


Figure 17: A illustration of the level mysterious layout.

5 How to play?

IMPORTANT: If the game is extremely laggy, please restart the MARS.

For unknown reasons, MARS starts laggying after few hours of coding and playing.

There are 3 modes in total to play.

Before playing in any mode, please set the bitmap display as in the preamble.

- Unit Width in Pixels = 4
- Unit Height in Pixels = 4
- Display Width in Pixels = 512
- Display Height in Pixels = 256
- Base address for display = 0x10008000 ($gp)

5.1 Normal Mode

To play the Normal Mode, please set the equivalences to the followings:

- `.eqv AUTO_MODE 0`
- `.eqv SLEEP 32` (if you feel very hard to play with, feel free to sleep longer)
- `.eqv DEFAULT_HEARTS 3` (if you feel very hard to play with, feel free to add more hearts)
- `.eqv DEFAULT_MOVING_MODE 1`

Operations:

- `s`: Starts the game.
- `p`: Pauses/continues the game.
- `q`: Quits the game.
- `a`: Moves the paddle leftward.
- `d`: Moves the paddle rightward.
- `1-9`: Changes level number 1 - 9.
- `0`: Changes to a mostly unplayable level.

Notes:

- After changing the level, you need to press `s` to start the game.
- You can change to different levels anytime you want.
- The game will quit automatically if you lose.
- After a failure, if you have at least one more heart, you will respawn after 2 seconds. Then, you need to press `s` to resume the game.

5.2 Automatic Mode

In this mode, the paddle follows the ball automatically.

To play the automatic mode, please set the equivalences to the followings:

- `.eqv AUTO_MODE 1`
- `.eqv SLEEP 0` (basically, any sleep values you want: the smaller, the faster)
- `.eqv DEFAULT_HEARTS 1` (any integer greater than or equal to 1)
- `.eqv DEFAULT_MOVING_MODE 1`

5.3 God Mode

In the God Mode, you can use keys to fully control the movement of the ball.

To play the god mode, please set the equivalences to the followings:

- `.eqv AUTO_MODE 1`
- `.eqv SLEEP 0` (basically, any sleep values you want: the smaller, the faster)
- `.eqv DEFAULT_HEARTS 1` (any integer greater than or equal to 1)
- `.eqv DEFAULT_MOVING_MODE 0`

Operations (in addition to operations in the Normal Mode):

- `i`: Moves the ball upward 1 unit.
- `j`: Moves the ball leftward 1 unit.
- `k`: Moves the ball downward 1 unit.
- `l`: Moves the ball rightward 1 unit.
- `u`: Moves the ball up-leftward 1 unit.
- `o`: Moves the ball up-rightward 1 unit.
- `m`: Moves the ball down-leftward 1 unit.
- `.`: Moves the ball down-rightward 1 unit.
- `,`: Resets the directions of the ball to no direction. ($(0, 0)$ for x and y directions)
- `n`: Flips the moving mode (see 1.2 Mutable Data).