GUI based game: Racket

Nikita Melentjevs, Julian Michelsen

January 25, 2019

1 requirements

Create a Gui based game using racket, accepting user needs and requirements, visually pleasing functional, bug free final product.
refining of the problem and posing possible solutions.

- Create a character for the player to control?
- Create a level scape for the player to navigate?
- Create some way to level up and increase difficulty?
- Create enemies of some sort/obstacles?
- Create an initial menu screen for the player to choose options and such.

Possible Solutions:

- portal game
- shooter game
- geometry dash

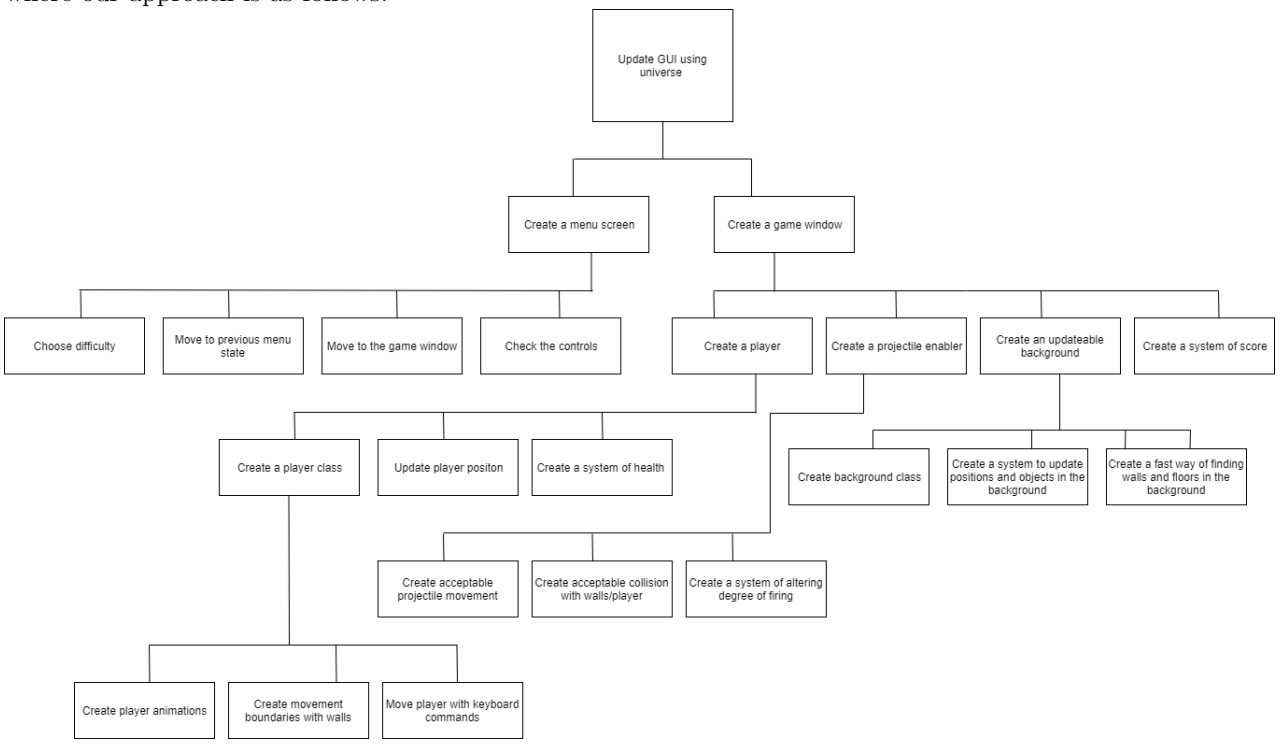
2 Design

Our game is an open-loop system as it relies on the input of the user to create any sort of desired output:

- Our input system is the keyboard where keyboard commands are entered, as well as the cursor and cursor clicks on the initial menu screen.
- The controller is the update function, where time is taken into consideration and the user inputs are taken into consideration.
- The response is the movement of the player on the GUI.

2.1 Top Down Approach

Our game is being created with top down design:
where our approach is as follows:



2.2 Use Cases

- Create movement through keyboard clicks
- Fire a projectile through keyboard clicks
- Create a background that looks reasonable
- Create enemies to avoid
- Create a system of health
- Create a system of score
- Create animations for the player
- Create an acceptable leveling system

2.3 UML Class Diagram

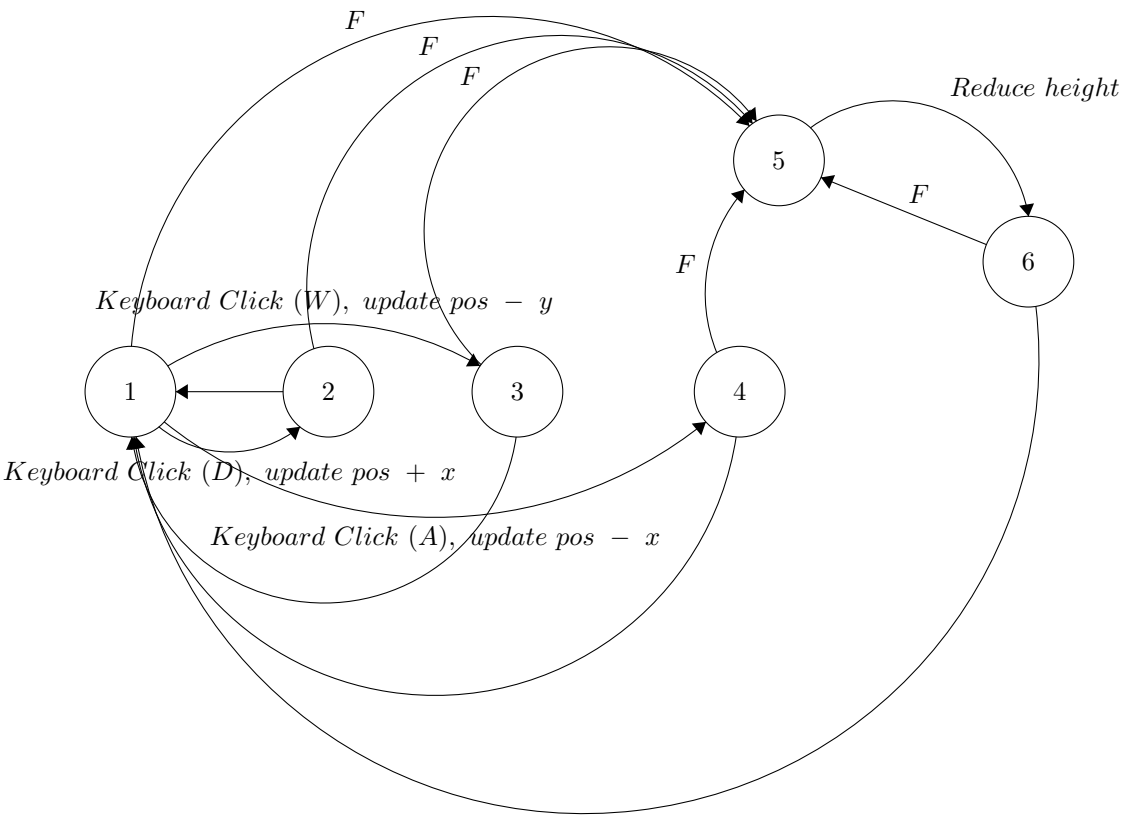
Write your subsection text here.

2.4 Different Data Structures

Lists, Hash tables, Sets

2.5 FSM's

Movement through Keyboard Click: where F represents a check-falling function
Blank line represents time passing.

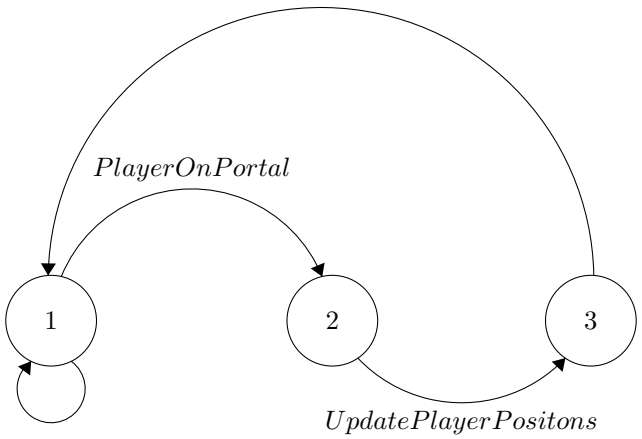


Code Built from this FSM

```
(cond
 [(not (equal? 0 animation))
   (if (= direction 0)
      (if (= (send background check-walls (+ x 1) (+ y 16)) 0)
         (set! x (+ x 1)) #t)
      (if (= (send background check-walls (- x 1) (+ y 16)) 0)
         (set! x (- x 1)) #t))]
```

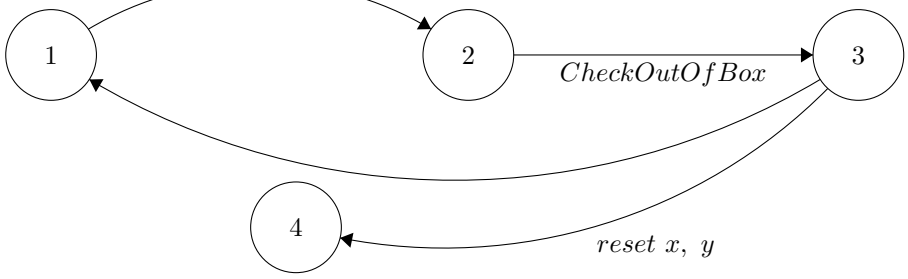
```
      )]
  (if (and (and (not (equal? animation "jA"))
           (not (equal? animation "jD")))
       (= (send background check-bases x (+ y 16)) 0))
       (block
       (set! y (+ y 2))
       (set! falling 1) #t)
       (set! falling 0))
  )
```

Portal Relocation:



Projectile updating after being fired:

$x = x + 3,\ y = y((tan(piAngle/180) * 3 * (-1))$



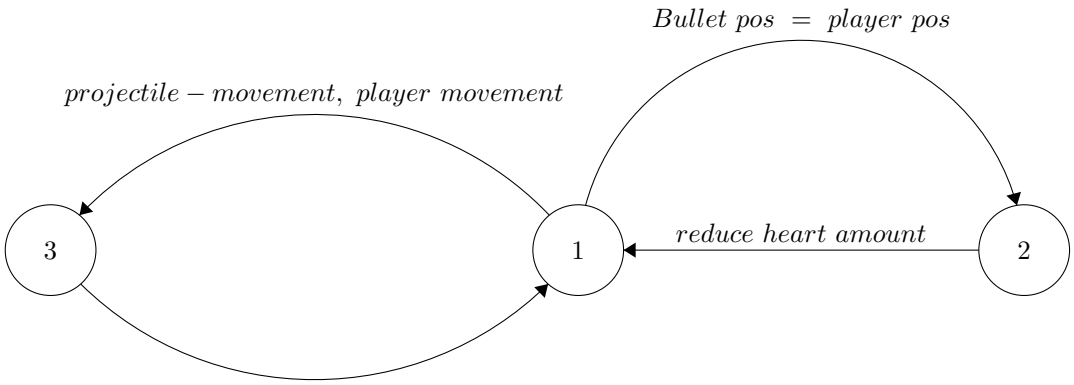
This FSM was created by reverse engineering the function that we had already created:

```
(define (update-projectile angle x y direction)
   (define s1 +)
   (define s2 -)
```

```
    (cond
     [(and (>= angle -90) (<= angle 90)) (set! s2 +)]
    )
    (if (= direction 1) (set! s1 -) #t)
    (set! y (s2 y (* (* (tan (/ (* pi angle) 180)) 3) -1)))
    (set! x (s1 x 3))
    (list x y)
  )
```

2.6 EFSM's

Life reduction EFSM with movement and projectile movement:



2.7 Human Interface

The color background that we have decided to use makes it very easy to distinguish between the walls and the background. The bullet color and the enemy color was chosen to be red as it is the most commonly used color for dangerous objects, so that the player instinctively understands that they should avoid the bullets without being told to.

The star in the interface is also a universal symbol for an object that should be collected. Otherwise player may not know whether or not to move towards it. The hearts are placed at the top of the screen to be easily visible by the player.

3 Implementation

3.1 Pre-conditions and Post-conditions

Length Function:

Precondition: $[x : list \wedge y = 0]$

Postcondition: $[y = z \wedge (x(z) \in x \wedge x(z+1) \leftrightarrow \prime())]$

We use : to denote type of a variable.

Check Portal Touching Function:

Precondition: $[b - portal \neq \prime() \wedge length(\text{bx b-portal}) \geq 2 \wedge length(\text{by b-portal}) \geq 2 \wedge x \neq \prime() \wedge y \neq \prime()]$

Postcondition: $[y = z \wedge (x(z) \in x \wedge x(z+1) \leftrightarrow \prime())]$

Where we use (attribute class) to denote class attributes for specific instantiations of a class

3.2 Side Effects

The lifebarchange funcion not only changes the amount of hearts the player currently has, but also the bullet class field has-hit, this is an example of where side effects can be very useful, when trigger-checks are updated so that they cannot be triggered again. The change function alters almost all the classes in the program, as well as our hash-table that is used for the movement and portal gun rotation.
There are problems that come along with the change function have multiple side effects, such as the difficulty in locating problems in the code and what classes the issue is being created in.

4 Testing and Maintaining

4.1 Test Cases and Assertions

- **test cases for player movement**:

  Pressing a while not in air and on platform that is > 10 pixels away from either a drop or wall to the left of the players character will decrease the players x position by 10 pixels while y stays the same.

$x = x_0 - 10 \wedge y = y_0$

Pressing d while not in air and on platform that is > 10 pixels away from either a drop or wall to the right of the players character will increase the players x position by 10 pixels while y stays the same.

$x = x_0 + 10 \wedge y = y_0$

We then gather our test cases and assertions for **keyboard press a d**

Test Cases:

- player-x = player-x + 10
- player-y = player-y
- player-x = player-x - 10

Assertions:

```
(test-begin
 (cond
   [(and (= (send background check-walls (+ x 10) (+ y 16)
       0))
        (and (not (= (send background check-bases (+ x 10)
            (+ y 16) 0)))
           (equal? animation "mD")))
             (check-equal? x (+ x0 10))
             (check-equal? y x0)]
   [(and (= (send background check-walls (- x 10) (+ y 16)
       0))
        (and (not (= (send background check-bases (- x 10)
            (+ y 16) 0)))
           (equal? animation "mA")))
             (check-equal? x (- x0 10))
             (check-equal? y x0)]
   )
)
```

- **test cases for teleporting through portals**:

if the player x position and y position match that of either a blue/orange portal then its position is then set to the appropriate x and y position of the other portal.

$blue \rightarrow orange$
$orange \rightarrow blue$

$x = O(x_r + B(x_l - x_0)) \wedge y = O((y_t - y_b)/2)$

when going through B portal

$x = B(x_r + O(x_l - x_0)) \wedge y = B((y_t - y_b)/2)$

when going through O portal

Here we use $O(y_t)$ and $O(y_b)$ to represent the top and bottom of the y positions of the Orange portal.
we also use $O(x_r)$ and $O(x_l)$ to represent the right and left x values of the Orange portal, we use the same logic for the Blue portal.

We then gather our test cases and assertions for **portal teleporting**.

Test Cases:

- player-x = orange-portal-right-x + player-x - blue-portal-right-x
- player-y = orange-portal-top-y - 10
- player-y = orange-portal-bottom-y + 10
- player-x = orange-portal-right-x + 30
- player-y = orange-portal-top-y + player-y - blue-portal-top-y
- player-x = orange-portal-right-x - 30
- player-y = orange-portal-bottom-y + player-y - blue-portal-top-y

Assertions:

```
(test-begin
 (cond
   [(equal? (send b-portal get-direction) "d")
      (check-equal? player-x
                (+ o-portal-r-x (- player-x0 b-portal-r-x)))
      (check-equal? player-y (- o-portal-y-t 10))]
   [(equal? (send b-portal get-direction) "u")
      (check-equal? player-x
                (+ o-portal-r-x (- player-x0 b-portal-r-x)))
      (check-equal? player-y (+ o-portal-y-b 10))]
   [(equal? (send b-portal get-direction) "l")
      (check-equal? player-x (+ o-portal-r-x 30))
      (check-equal? player-y
                (+ o-portal-t-y
                  (- player-y0 b-portal-t-y)))]
   [(equal? (send b-portal get-direction) "r")
      (check-equal? player-x (- o-portal-r-x 30))
      (check-equal? player-y
```

```
                    (+ o-portal-t-y
                      (- player-y0 b-portal-t-y)))]
      )
    )
```

And likewise for the blue portal and its assertions and test cases.

- **test cases enemy movement on platforms**:

  When an enemy touches the end of a base platform it moves in the opposite direction

  $\Longleftrightarrow$

  When there is no platform to $x + (3 * speed)$ of the enemy's x position then it's direction is reversed.

  we then gather our test cases and assertions for **enemy movement**.

  Test Cases:

  - direction = (direction + 1) mod 2
  - x = x + 1.5*speed
  - x = x - 1.5*speed
  - y = y + 2

  Assertions:

```
(test-begin
 (cond
   [(and (and
      (not (= (send background check-bases x (+ y 16)) 0))
      (not (= (send background check-bases (+ x (* 1.5
          speed)) (+ y 16)) 0)))
      (= direction 0))
         (check-equal? direction (modulo (+ 1 direction0)
             2))
         (check-equal? x (+ x0 (* 1.5 speed)))]
   [(and (and
      (not (= (send background check-bases x (+ y 16)) 0))
      (not (= (send background check-bases (- x (* 1.5
          speed)) (+ y 16)) 0)))
      (= direction 1))
         (check-equal? direction (modulo (+ 1 direction0)
             2))
         (check-equal? x (- x0 (* 1.5 speed)))]
   [(= (send background check-bases x (+ y 16)) 0)
```

```
            (check-equal? y (+ y0 2))
            (check-equal? falling 0)]
        )
    )
```

4.2 Specification vs Implementation

4.3 User Feedback

survey-link here: https://www.surveymonkey.co.uk/r/FDXDWKN

Data collection and analysis below:

Controls	Look	Enjoyability	Design and Smoothness
2.5/5	4/5	3/5	3/5

Ratings are an average of all surveys taken / 5

Amount of surveys taken: 15

4.4 Amendments from the data

Controls:

Controls needed to be improved, we made adjustments to it by first adding a section on the menu page labelled controls, so that users could understand how the controls worked before actually entering the game, we also spaced our the keys that needed to be pressed so that both hands could be used.

Look:

The look of the game was our best part so we decided to focus on other sections which were not as strong.

Enjoyability:

Enjoyability is always a difficult aspect of the game to increase, but by looking at which difficulty users preferred, we could gather that the easy mode was the most enjoyable, meaning that we should tweak the other difficulties to be more like the easy mode. Thus we lowered the movement speed of enemies in the 2 other difficulties as well as increasing the time to get the stars.

Design and Smoothness:

To fix issues with lag we implemented binary searches instead of linear searches

anywhere we could, this took down the lag by a small margin, but not a noticeable amount.

4.5 Updated User Feedback

Controls	Look	Enjoyability	Design and Smoothness
3.5/5	4/5	3.5/5	3.5/5

Amount of surveys taken: 12

As we can clearly see, our adjustments did improve the ratings that we got, but they still were not that high unfortunately.

5 Conclusion

We are happy with our end result, however we would have liked to add some additional features, and reduce the amount of bugs with the portal, however this was taking too much time to complete. Thus we decided to move on with other features of the program.