CS3370 – Nature of Programming Languages

Topic:

Simulating 2D Role-Playing Game (RPG) With Ruby

Group 4			
Group Members	**No.**	**Name**	**ID**
	1	Ta Duc Duy	1624634
	2	Truong Thanh Hung	1624817
	3	Dao Nguyen Khoi	1624717
Lecturer	AProf. Han Dinh Nguyen		

- Ha Noi, August 20th, 2024 –

Table of Contents

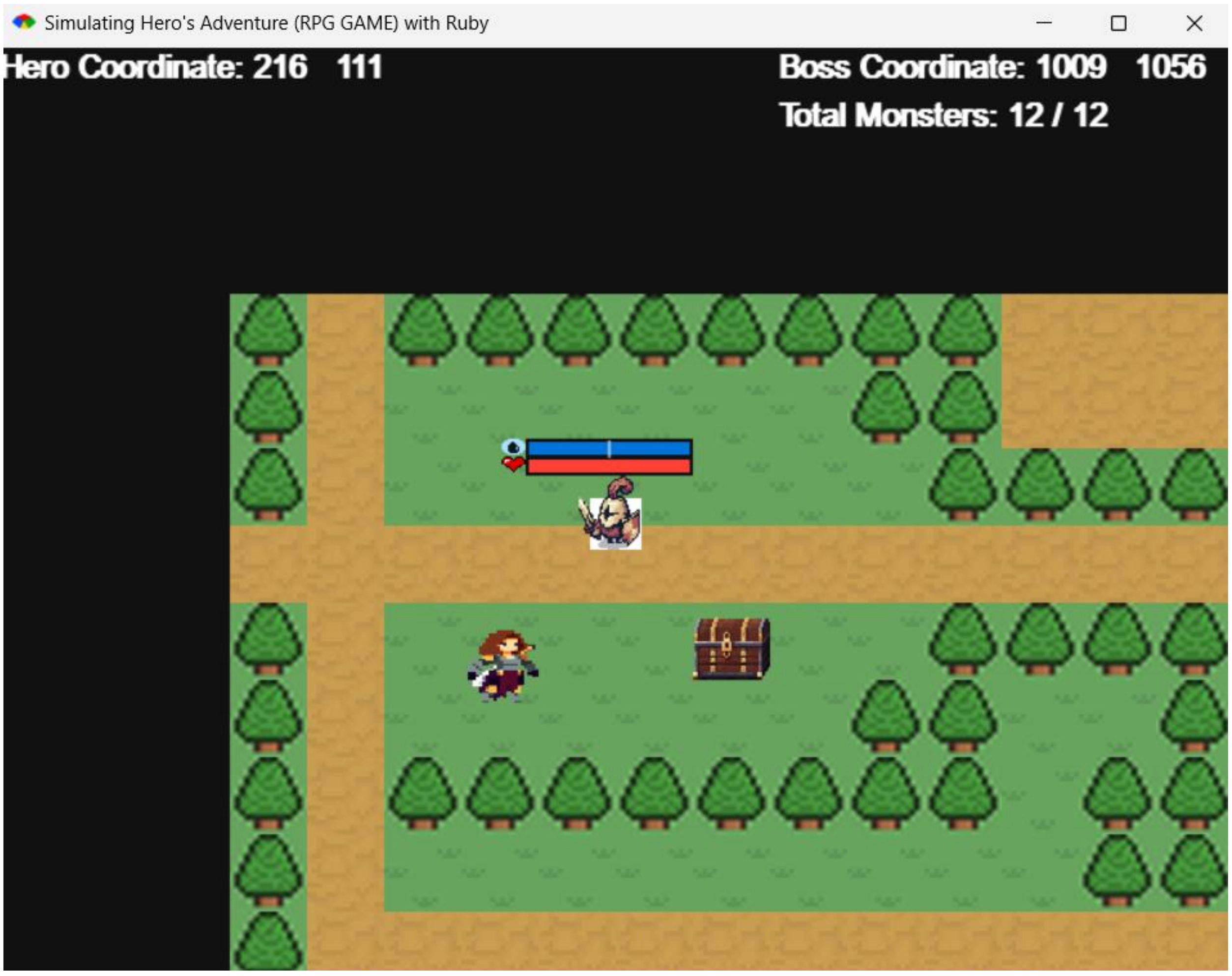
1 Contribution Page

Member	Task	Detailed Task
Trương Thanh Hùng 1624817	Design **Statistic** for characters	1. Design HP, ATK, DEF, SPEED, ...
	Design **Game Combat**	1. Define **when** game combat occurs
		2. Define **actions** that should happen in combat
	Design **Attack Behaviour** between Character	1. Design **HP bar** and **MP bar** for player, HP bar for monsters
		2. Design **hitbox** and **attackbox** for player and monster
		3. Design some **special skills** for player
	Design **Policy of When Dead, When Alive** for Character	1. Check if monsters are dead then throw them out of map
		2. Check if player is dead then clear the window and set die screen
	Design **Interaction** between **Player** and **Map**	1. Check what kind of tile player is standing on and then apply the coressponding effect of that type of tile
	Design **Animation section**	1. Design **walk animation** for Player, NPCs, Monsters
Tạ Đức Duy 1624634	Design and implement **Map section**	1. Design and implement class of **Tile** (wall, grass, ground, water, fire, tree)
		2. Design class **Map**: - Map is comprised of many Tiles - Using 2D-array to store position of Tiles in the map
	Design and implement functions of **Checking Collision** between objects	1. Check collision between characters (Player, Monsters, NPCs) and solid areas in the map (wall, tree)
		2. Check collision between characters (Player, Monsters, NPCs) and boundary of the map
		3. Check collision between characters (Player, Monsters, NPCs) and items in the map
		4. Check collision between one character and other characters
	Design and implement **World and Camera** for the game	1. Display just a part of entire map with related objects (entire map is called World) that fits the player's screen (the player's screen is called Camera)
	Design and implement **Monster section**	1. Design super class for Monster and subclasses that inherit from the super class
		2. Design functions of random move
		3. Design functions of pursuing move: - Design and implement A star algorithm
Đào Nguyên Khôi 1624717	Design **Item, Inventory section**	1. Design an inventory containing several items
		2. Define **user's action** to open inventory, pick an item from inventory, close inventory
	Design and implement **NPC section**	1. Player can talk to the NPC
		2. NPC gives Player useful items when talking to Player

2 Introduction

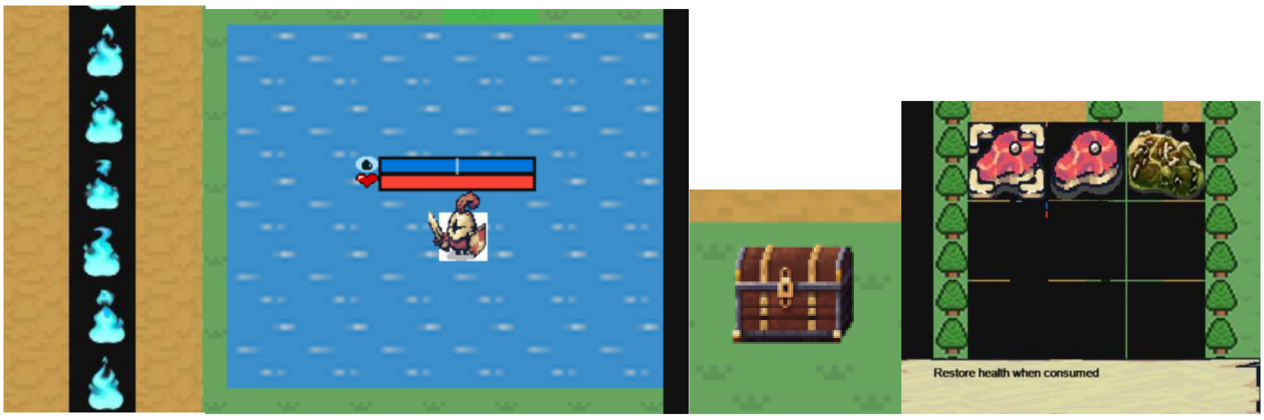
2.1 Name and Description.

Project name: “*Simulating 2D - Role Playing Game (PRG) with Ruby*”.

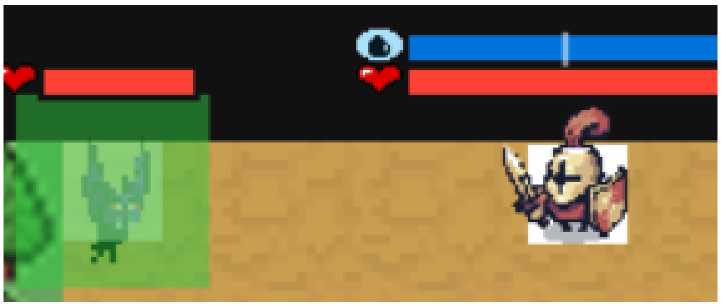

Simulating Hero's Adventure (RPG GAME) with Ruby
Hero Coordinate: 216 111
Boss Coordinate: 1009 1056
Total Monsters: 12 / 12

Description:

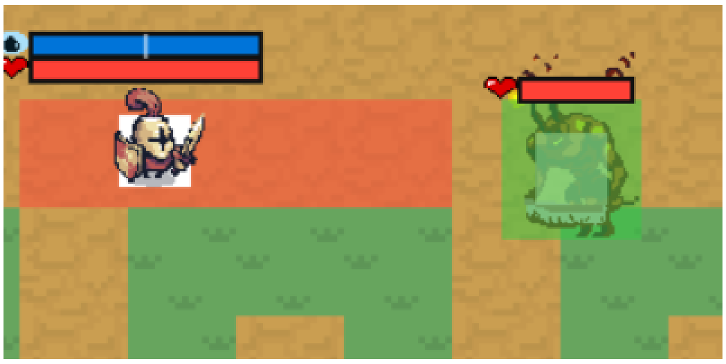
- An RPG (Role Playing Game) where A player fights against monsters on the map, he/she has to defeat all the monsters in the map in order to win the game or he/she runs out of HP and loses the game.
- There are special terrains (tiles) that might help (water) or hurt (fire) the player, lying randomly on the map. Also, there are items that might help the player or hurt the player.



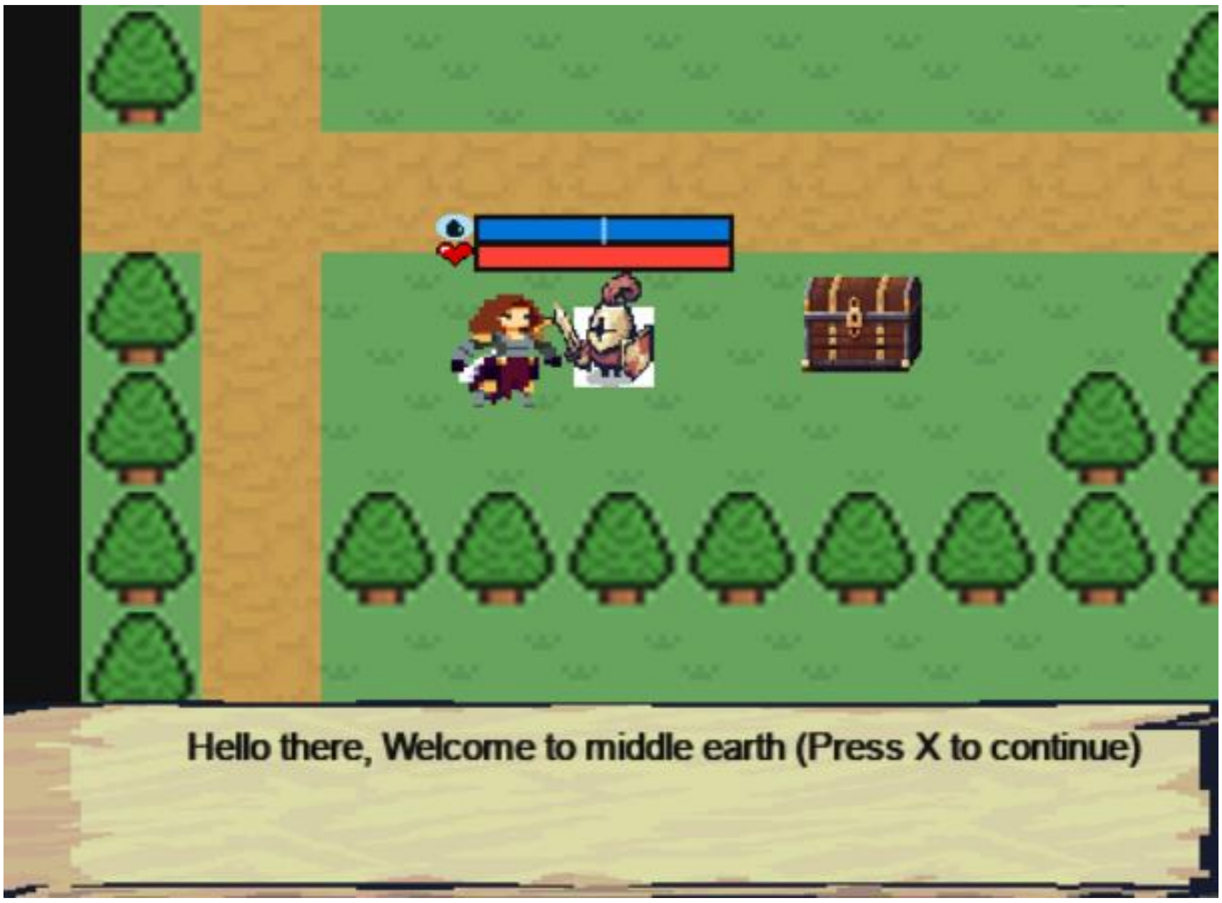
- The monster in the map will dive into two main monsters:
  - Random Monsters: Monsters that move randomly and when the player in its range it will attack the player.



  - Perusing Monsters: Monsters that move towards the player and will attack the player when in its range.



- There are NPC that will give the player Items that help the player.



Hello there, Welcome to middle earth (Press X to continue)

.

2.2 Functional Requirements

2.2.1 Map Requirements

A map is a two-dimensional array of tiles. Each tile represents a special area on the map. In particular, there are six types of tiles.

- **Grass:** This area is a regular area, and it allows the hero, and monsters to move into.
- **Ground:** This area is a regular area, and it allows the hero, and monsters to move into.
- **Wall:** This area does not allow the hero and monsters to move into.
- **Tree:** This area does not allow the hero and monsters to move into.
- **Water:** Hero recovers a portion of lost health when moving into this area while monsters are not affected.
- **Fire:** Hero loses health when moving into this area while monsters are not affected.

2.2.2 Monster Requirements

Design a system of monsters so that the monsters are able to move randomly, pursue player. In particular, there are 2 types of monsters.

- **Regular Monster:** This monster moves randomly on the map.
- **Target Monster:** This monster pursues the hero.

2.2.3 NPC Requirements

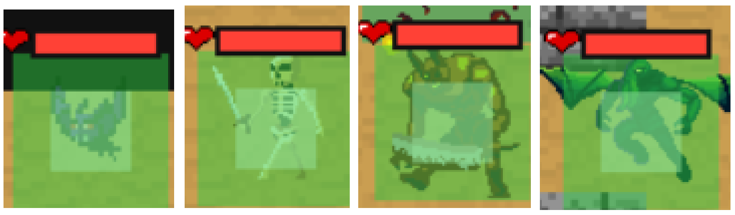
- The game allows player to interact with NPC within a certain range.
- The NPC must give useful instructions of how the user should play the game.

2.2.4 Item Requirements

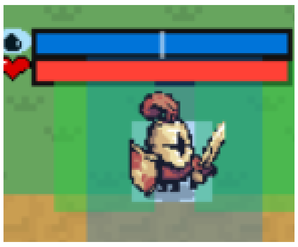
- The game allows the player to gather items across the map and save them in player's inventory and allows the player to consume the item effectively. Item's effect reflects to player's attributes.

2.2.5 Animation Requirements.

- For each character (including player and monster) has its own attack box (green box) and hit box (light green box).
- Each character in the game include monsters and player have it own animation for walk, idle, attack, hurt and dead.



For the player he/she will have a special attack box (blue box) for using special skills.



- These boxes must move along with the player and monster in order to check if they are hit.
- These boxes must disappear with the character when it died.

2.3 Programming Language Selection Reasons and Criteria.

We choose Ruby as the programming language for this project because there is a library that meets our requirements for developing this game. That library called "Ruby2d". (https://www.ruby2d.com). Some amazing feature of Ruby2d is listed, as below.

- Use "loop" which is in "Ruby2d" that helps to loop the game main, the commands in the side the "loop" will run continuously 60 times in a second (depending on the hardware of the computer).
- "Rectangle" that will create a rectangle on the screen with its x, and y position and its width and height. That will help to deal with creating a hit-box and attack-box.
- "sprite" will help us with animation, it will create an object that prints many pictures in a period of time that the coder sets.
- "text" will help us to print the text on the screen.
- "sound" will help us to set the background music.
- "key:", "key_down:", "key_held:" that will help us to deal with the input of the user, when user presses key on the keyboard.
- "image" will hold an image to print out on the screen.

2.4 Concerning Problems.

2.4.1 Structure of Entity

In our game, there are many objects (or entities) such as monsters, items, obstacles, etc. But what makes up this object? Well, all objects in our game have the `Rectangle` structure, including the following attributes.

```
Structure Entity
{
    int x
    int y
    int width
    int height
}
```

The coordinate of an entity is determined by the coordinate of the top-left corner `(x, y)`.`width, height` represent for the dimensions of the entity. The following figure gives you a sense of what an entity looks like.

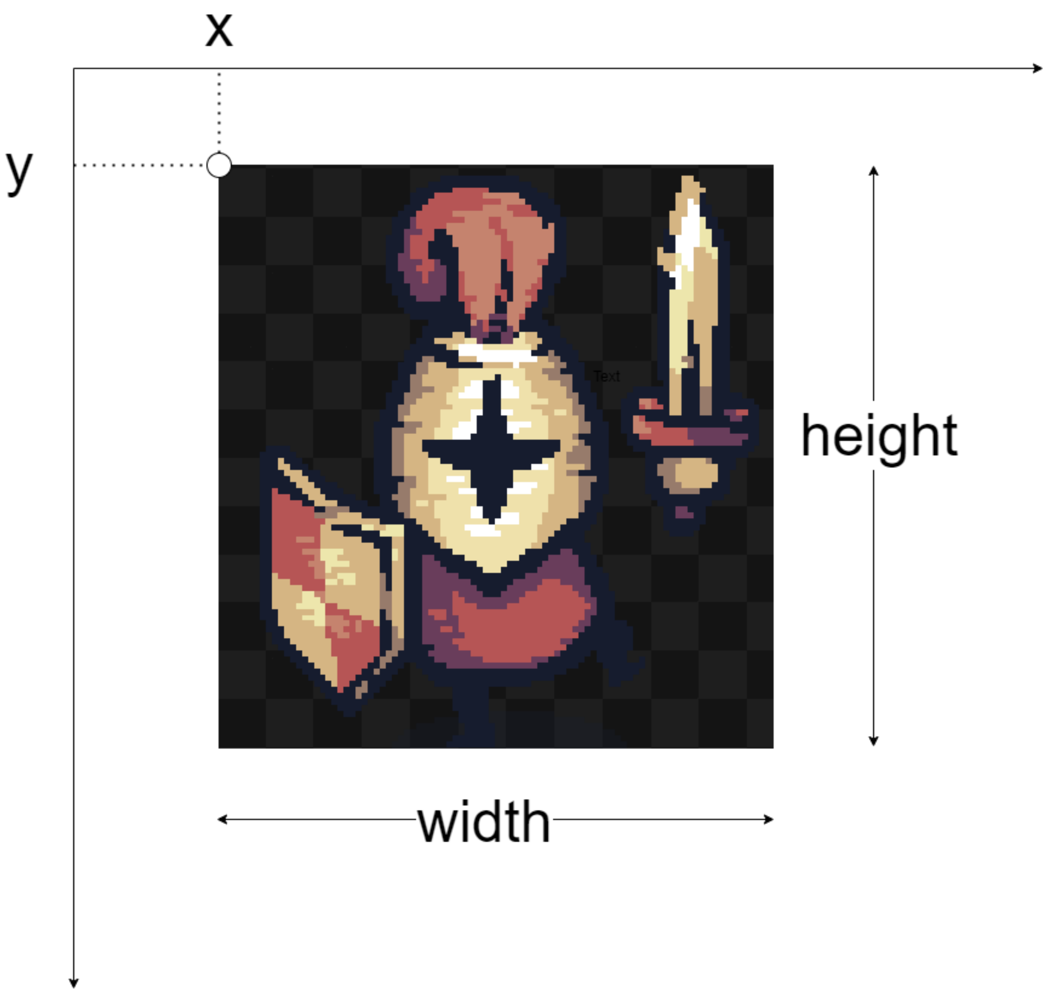


Figure 1. Structure of an Entity

2.4.2 Game Loop (The Core of 2D Game)

How does an object look moveable? Well, the mechanism that makes an object moveable is printing many pictures in a small amount of time. For example, printing 60 different frames (pictures) per second will give you a feeling of movement. This process is given by the following figure.

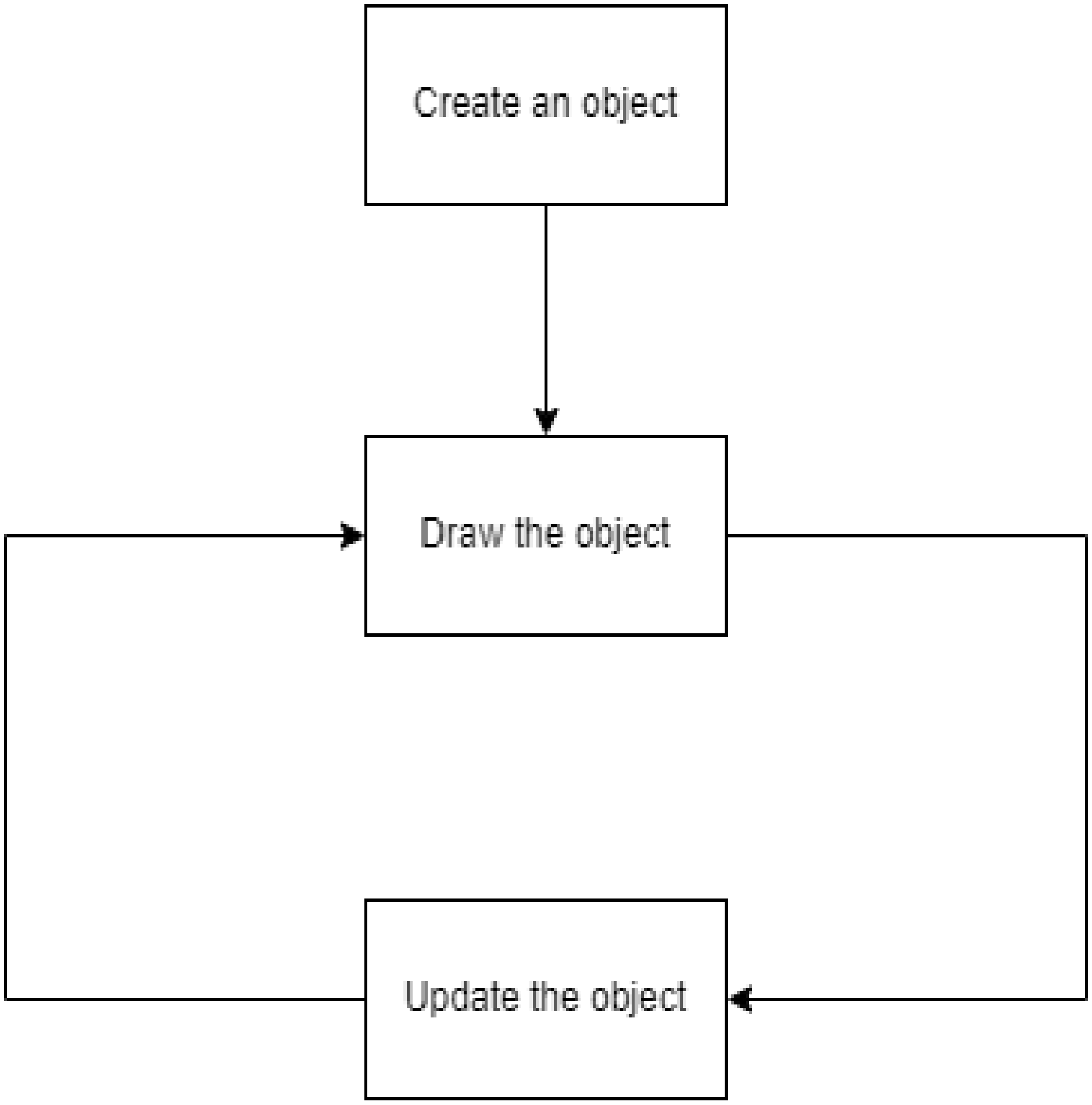


Figure 2. Game Loop

2.4.3 Collision Detection

In our game, we usually concert *when two entities collide with each other?* For example, to determine when a combat occurs, we need to know when the hero and monster collide to each other. To determine picking item on the map, we need to know when the hero collides items.

So, checking the collision between two entities is very important. The idea of checking collision between two entities is determining when two rectangles are intersected with each other in nature. The algorithm to check collision between two entities is shown as the following.

Algorithm 1: Checking Collision Between Two Entities

```
1  procedure checkCollision(entity1, entity2)
2      if $entity1.x + entity1.width \geq entity2.x$ &&
3        $entity1.x \leq entity2.x + entity2.width$ &&
4        $entity1.y + entity1.height \geq entity2.y$ &&
5        $entity1.y \leq entity2.y + entity2.height$ then
6        return true
7      return false
```

Figure 3. Collision Detection Algorithm

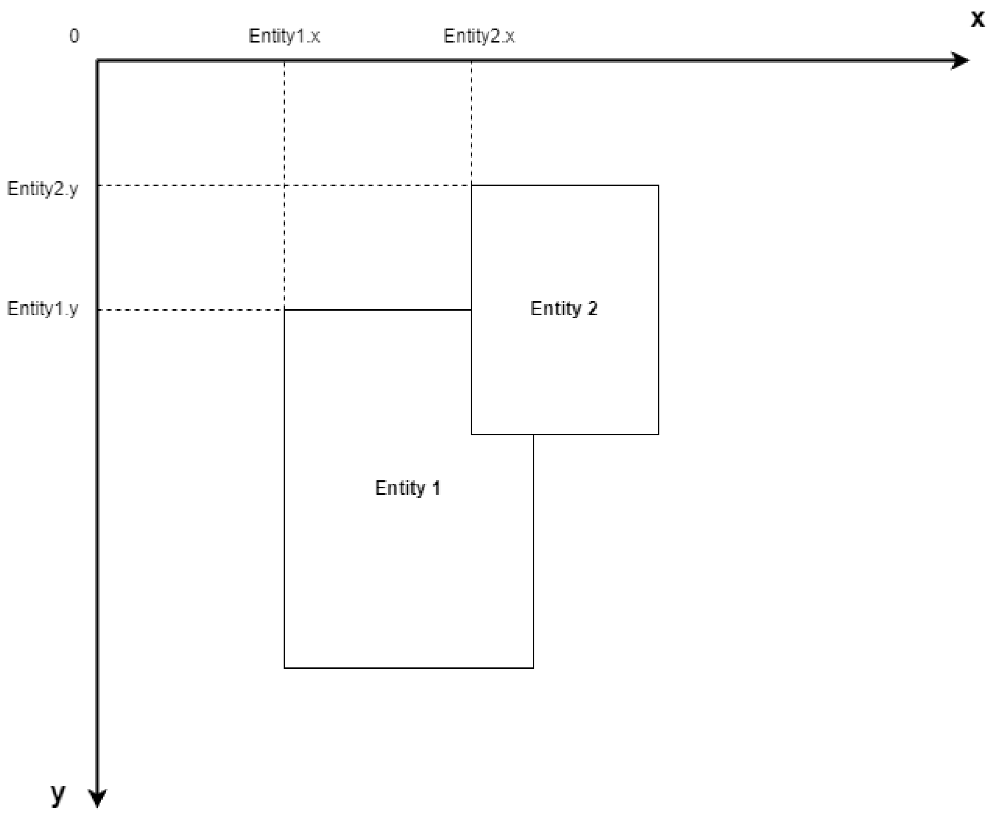


Figure 4. When two entities collide to each other

2.4.4 Finding the Shortest Path

2.4.4.1 Representing the map as a graph data structure

In our game, the map is a two-dimensional array of Rectangles (or you can imagine the map looks like a grid of rectangles), and each Rectangle represents a special area (e.g. wall, fire, tree, water, etc). The following shows the structure of the map.

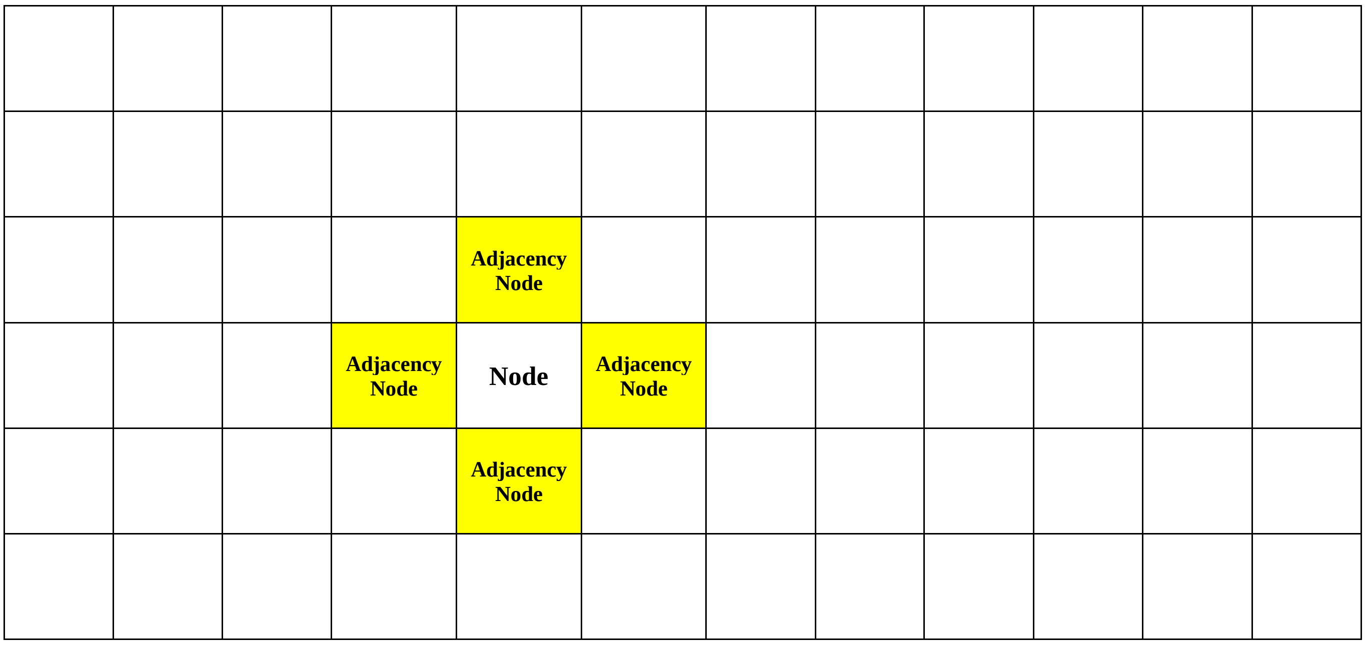


Figure 5. Structure of Map

To represent this map under the view of a graph, we can do the following thing.

- Each rectangle is considered as a node
- Two rectangles having a common side are considered two adjacent nodes. In this case, it is easy to see that one node will have exactly four adjacent nodes.

2.4.4.2 A* algorithm

In this part, we will show how A* algorithm is applied to find the shortest path between two points in a two-dimensional grid. Before we do that, we need to introduce the following definitions.

- Structure of Node
- G cost, H cost, F cost of a Node

2.4.4.3 Structure of Node

```
Structure Node
{
    int row
    int col
    boolean solid
    boolean open
    boolean checked

    Node parent

    int gCost
    int hCost
    int fCost
}
```

2.4.4.4 G-cost, H-cost, F-cost of a Node

- The **G-cost** of a node X is the distance from the starting node to the node X. Mathematically, the G-cost of X can be calculated using the following formula:

$$gCost(X) = |X.row - startingNode.row| + |X.col - startingNode.col|$$

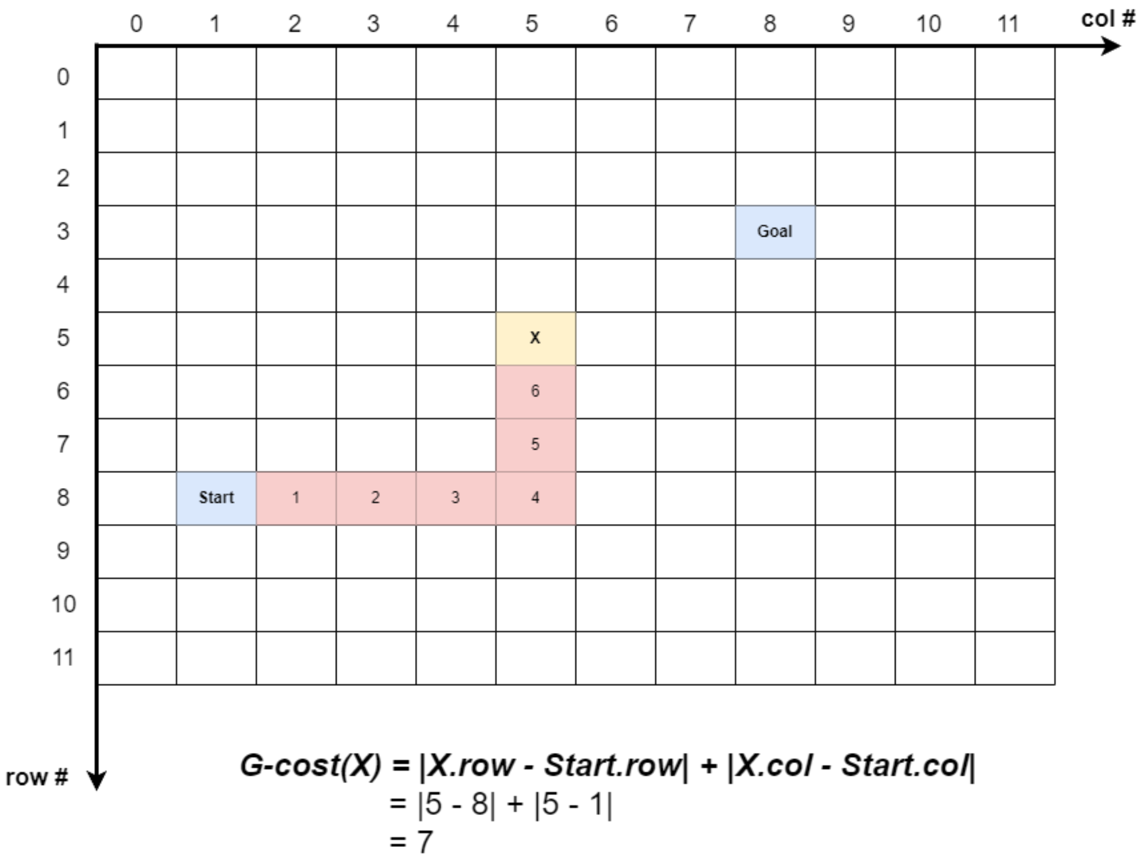


Figure 6. The distance from the starting node to node X, or F-Cost of X, is 7, including steps (8, 2), (8,3), (8,4), (8,5), (7,5), (6,5), (5,5)

- The **H-cost** of node X is the distance from node X to the goal node. Mathematically, the H-cost of X can be calculated using the following formula:

$$hCost(X) = |X.row - goalNode.row| + |X.col - goalNode.col|$$

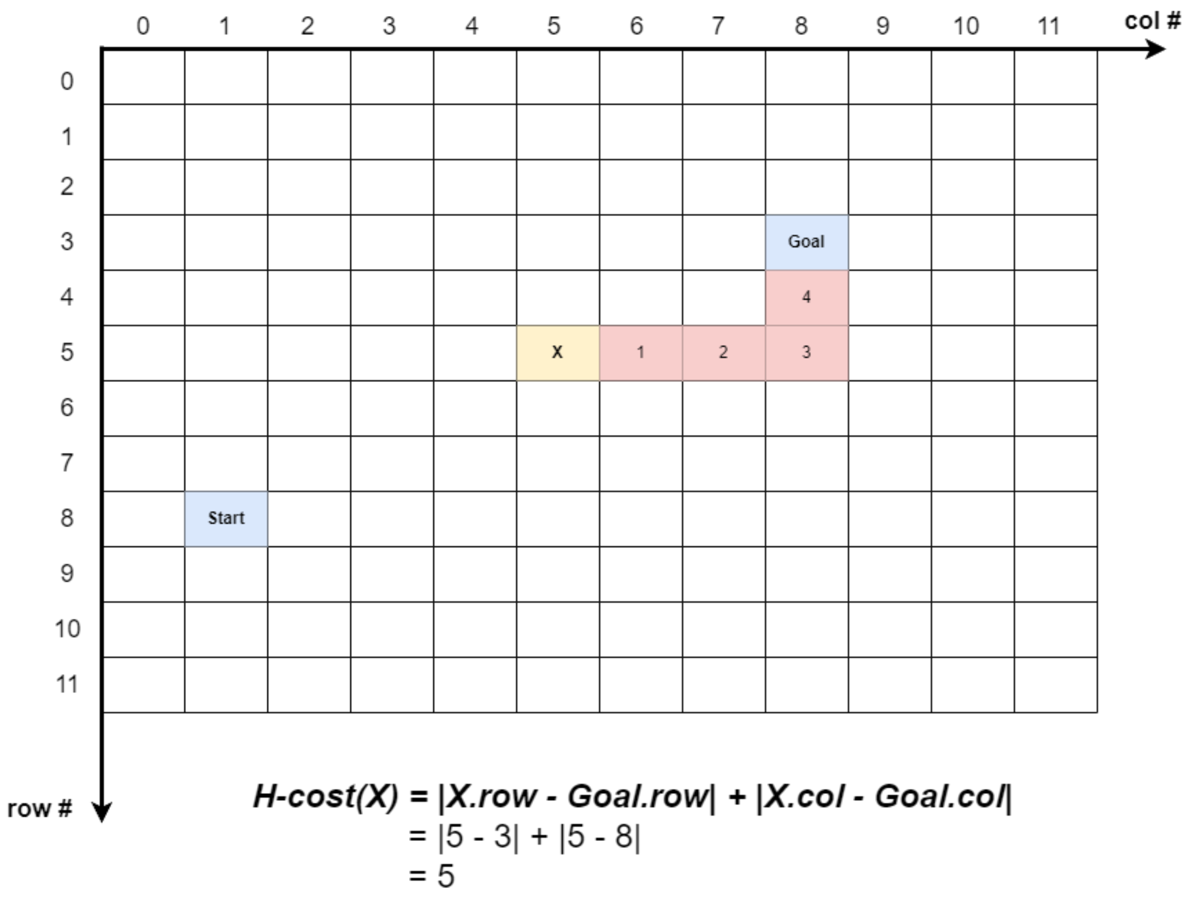


Figure 7. The distance from node X to the goal node, or H-Cost of X, is 5, including steps (5,6), (5,7), (5,8), (4,8), (3,8)

- The F-cost of node X is the sum of the H-cost of X and the G-cost of X, which is mathematically represented by the following formula.

$$fCost(X) = gCost(X) + hCost(X)$$

2.4.4.5 Pseudocode of A* algorithm

Algorithm 2: A* Algorithm in Finding The Shortest Path Between the Two Points in the Grid

```
1  Procedure: A-Star-Find-The-Shortest-Path(nodes: [][] Node, startNode: Node, goalNode: Node)
2      // 0. Initialization
3      openList = Ø
4      currentNode = startNode
5      found = false
6
7
8      // 1. Calculate costs for each node
9      for i = 0 to nodes.MaxRow – 1 do
10         for j = 0 to nodes.MaxCol – 1 do
```

```
            // 1. Calculate G-cost
            $nodes[i][j].gCost = |nodes[i][j].row - startNode.row| + |nodes[i][j].col - startNode.col|$
            // 2. Calculate H-cost
            $nodes[i][j].hCost = |nodes[i][j].row - goalNode.row| + |nodes[i][j].col - goalNode.col|$
            // 3. Calculate F-cost
            $nodes[i][j].fCost = nodes[i][j].gCost + nodes[i][j].hCost$


// 2. Search the shortest path
$currentNode.checked = true$
$openList.add(currentNode)$

while $found = false$ && $openList.size > 0$ do
    $currentNode.checked = true$
    $openList.remove(currentNode)$
    for each $u \in Adj[currentNode]$ do                    // Push adjacents of currentN ode to openList
        if $u.open = false$ && $u.checked = false$ && $u.solid = false$ then
            $u.open = true$
            $u.parent = currentNode$
            $openList.add(u)$
    $bestNodeIndex = -1$
    $bestNodefCost = \infty$
    for $i = 0$ to $openList.size - 1$ do                                   //Find the best node
        if $openList[i].fCost < bestNodefCost$ then           // Check if this node's F-Cost is bette
            $bestNodeIndex = i$
            $bestNodefCost = openList[i].fCost$
        else if $openList[i].fCost = bestNodefCost$ then      // If F-cost is equal, check the G-cost
            if $openList[i].gCost < openList[bestNodeIndex].gCost$ then
                $bestNodeIndex = i$

    if $bestNodeIndex \neq -1$ then
        $currentNode = openList[bestNodeIndex]$        // The best node is used for the next iteration
        if $currentNode = goalNode$ then
            $found = true$


//3. Track back the path
```

```
    Path = $\emptyset$
    **if** $found$ = true **then**
        currentNode = goalNode
        **while** currenNode $\neq$ startNode **do**
            Path.InsertAtFront(currentNode)
            currentNode = currentNode.parent
    **return** Path
```

3 Methodology.

3.1 Major components.

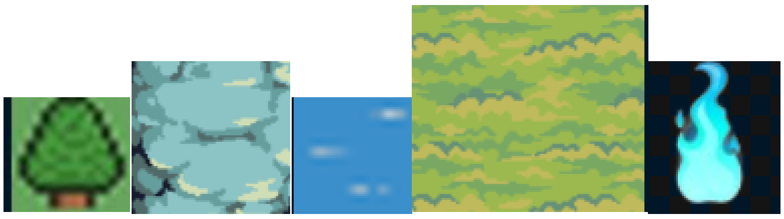
3.1.1 Player.

- Player will be put in the map, moving around freely and interact with tile on the way.
- He/she can attack and defeat monster, he/she has to defeat all the monsters in the map in order to win the game.
- He has normal attack (which cannot break stance of the monster) and special attack (can break stance of the monster).

3.1.2 Monsters.

- Monsters will be added into the map by an array of monsters.
- They will move freely by random move or pursue move depend on the kind of monster and the situation they are in.
- Monster can attack the player.
- They can be killed.

3.1.3 Map.

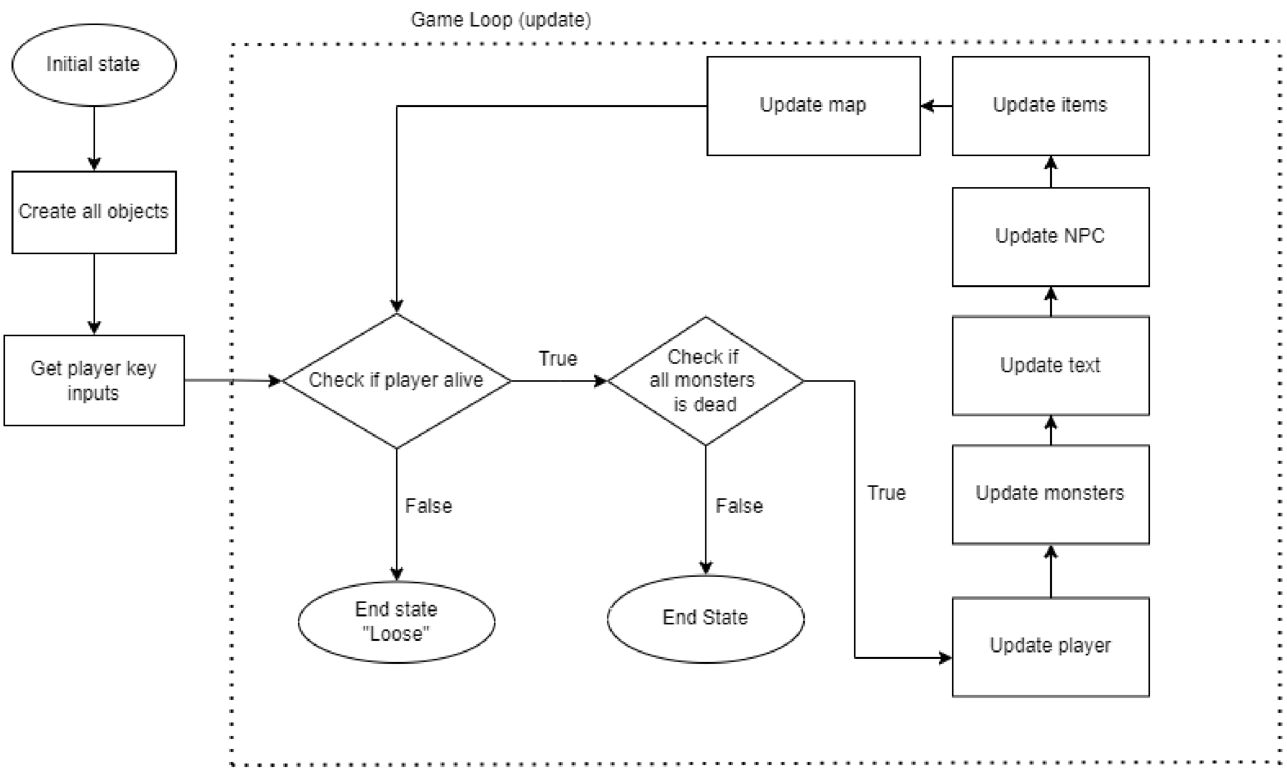


- Map are built up by different tiles (tree, wall, water, land, fire, …).
- Each tile has it own different interaction with player.

3.1.4 Items.

- There are different items in the map serve different purposes (heal or hurt player).

3.2 Process of Software.

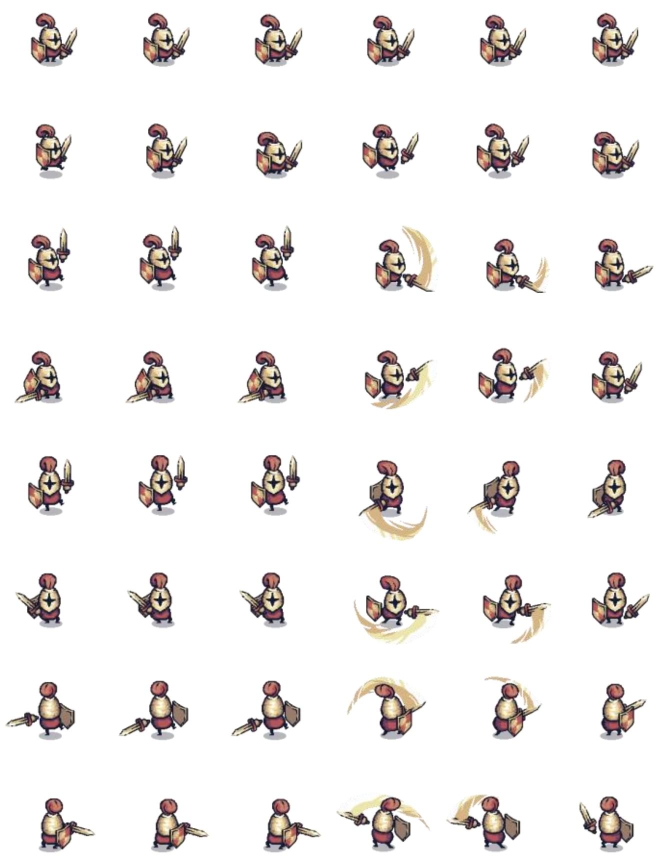


- First the software will create all the objects we need of the game such as monsters, player, items, tiles of the map and the map itself.
- Then the software continuously looking for player's key input.
- In the game loop:
    - The game will first check if player is alive and did all monsters die? In order to continue or end game with the loose or win text display and break the loop.
    - If the game continue it will update the player (its position, animation, health, mana).
    - Then it will update monsters (their position, animation, health).

- Then it will update the text on the screen.
- Then it updates the NPC (animation).
- Then it updates the items (be used or not).
- Finally, it updates the map (draw the map according to the player, and will affect the player corresponding with the tile player is standing on).

3.3 Using Language Features in Implementation.

- Using Ruby2D Features, use the Sprite class from Ruby2D to visually represent the Player, Items, Chests, and NPC on the screen. The animation is achieved by printing each frame by frame. For example, the player in idle animation will print the first line in a continuously loop, and of course the sprite help us to do so, it provide the position and width, height where exactly to cut out the image of each frame and then group it into an animation.



```
animations: {
  count: [
    {
      x: 0, y: 0,
      width: 35, height: 41,
      time: 300
    },
    {
      x: 26, y: 46,
      width: 35, height: 38,
      time: 400
    },
    {
      x: 65, y: 10,
      width: 32, height: 41,
      time: 500
    },
    {
      x: 10, y: 99,
      width: 32, height: 38,
      time: 600
    },
    {
      x: 74, y: 80,
      width: 32, height: 38,
      time: 700
    }
  ]
```

- The update block and key Handling provided by Ruby2D runs continuously, checking for key presses to move the player and for interactions with chests, items, and NPC. This demonstrates how Ruby2D simplifies managing game state and player inputs in real-time.
- The “rectangle” class be used to create hit-box and attack-box for player and monster. It includes the position of the rectangle, its width and height. Here a quick example to use the rectangle class:

```
Rectangle.new(
  x: 125, y: 250,
  width: 200, height: 150,
  color: 'teal',
  z: 20
)
```

Moreover, using 2 rectangles stack on top of each other, we created health bar and magic bar.

- The "sound" class be used to set the background sound for the game. By running the code:

```
song = Music.new('song.mp3')

# Play the music
song.play
```

4 Result and Discussion.

4.1 Result.

- We obtain a good game, where player can attack monster and monster can attack player. By using many rectangle shapes to check if they intersect or not, we can tell player's hit-box is in attack-box of monster and vice versa. That is how we check for attacking. The same go with checking for obstacle.
- The monster and player have its health store in the health bar which is created by 2 rectangles stack on top of each other. The one on top is the current health of object, the under one is base health of object, the one on top will change accordingly to the health of object in real time, it will decrease width according to the object hp. The same go with magic bar.
- Player can interact with the world, can move freely and cannot move when interact with an obstacle. Monsters also can move freely but will ignore the effect of tiles on it (except obstacle).
- We have items that can be store in the inventory, to be used.

4.2 Discussion.

- As we can see that Ruby2d did help us a lot in creating object and set interaction between object easily. However, ruby2d is not a good platform for performance:

  - When it creates an object, even if the object is not draw on screen the object is still exist, it still used the memory of the system and continuously calculating like another normal object that being draw on the screen.
  - We cannot remove the object when already create it, ruby2d do not have any method to remove the created object, which made it hard to remove a dead monster (we have to throw it far away in the map to make sure it won't appear on the screen again) which lead to the bad performance.
  - Moreover, ruby2d is not an open source code so we cannot interfere in order to update the use of it.
- Inconclusion, Ruby is pretty great to create a game with the library ruby2d but that mean we have to exchange for performance. Ruby is not a great language to create a game that required a lot of performance but it is great to create a normal game or small game that not require a lot of performance.

5 Reference.

- Ruby2D. (n.d.). Ruby 2D - Simple 2D Graphics for Ruby. Retrieved from https://www.ruby2d.com

6 Appendix.

- Game Loop (code snippet):

```
update do
  if (!player.healthBar.isDead? && !isAllMonsterDead(monsters))
    #1. Update Player
    player.updatePlayer(monsters, map, npcs, items)

    #2. Update Monsters
    for i in 0..(monsters.length - 1)
      monsters[i].updateMonster(player, map, items, npcs, monsters)
    end

    #3. Update Texts
    text.text = "Hero Coordinate: #{player.worldX}   #{player.worldY} "
    text1.text = "Boss Coordinate: #{monsters[0].worldX}   #{monsters[0].worldY}"
    currentNumberOfMonsters = 0
    for i in 0..(monsters.length - 1)
      if(monsters[i].exist == true)
        currentNumberOfMonsters = currentNumberOfMonsters + 1
      end
    end
    text2.text = "Total Monsters: #{currentNumberOfMonsters} / #{monsters.length }"

    #4. Update NPCs
    current_interacting_npc = -1
    for i in 0..(npcs.length - 1)
      npcs[i].updateNPC(player, map, i)
      if player.talktoNpc != -1
        current_interacting_npc = player.talktoNpc
      end
    end
    # Restore the interaction state after processing all items
    player.talktoNpc = current_interacting_npc
    #5. Update Items in map and preserve player.interacting
    current_interacting_chest = -1
    for i in 0..(items.length - 1)
      items[i].updateChest(player, i)
      if player.interacting != -1
        current_interacting_chest = player.interacting
      end
    end
    # Restore the interaction state after processing all items
    player.interacting = current_interacting_chest
    #6. Update Map
    map.updateMap(player)
  else
    case player.healthBar.isDead?
    when true
      if !@isSwitched
        Window.clear
        @isSwitched = true
        text_Loose.add
      end
    when false
      if !@isSwitched
        Window.clear
        @isSwitched = true
        text_Win.add
      end
    end
  end
End
```

- Player attack-box and hit-box (code snippet, same go for monsters):

```
    #10. Hit box
    @hitBox = Rectangle.new(
      x: @x + @solidArea.x,
      y: @y + @solidArea.y,            # Position
      width: 32, height: 32,  # Size
      opacity: 0
    )
    #14. Attack boxes
    @attackBoxRight = Rectangle.new(
      x: 360+48-15, y: 264-48+40,
      width: 40 , height: 50+10,
      opacity: 0
    )

    @attackBoxLeft = Rectangle.new(
      x: 360-48 + 15 + 10, y: 264-48+40,
   width: 40, height: 50+10,
   opacity: 0
  )

  @attackBoxSpecial = Rectangle.new(
   x: 360-10, y: 264-10,
   width: 70, height: 80,
   opacity: 0
  )

```

- Player attacking method (code snippet):

```
 def attackInBox(monsters)
  if @canAttack
   case @facing
   when 'right'
    @canAttack = false
    @image.play(animation: :attackSideFirst) do
     monsters.each do |monster|
      if CCHECK.intersect(@attackBoxRight.x,@attackBoxRight.y,@attackBoxRight.width,@attackBoxRight.height,
       monster.hitBox.x,monster.hitBox.y,monster.hitBox.width,monster.hitBox.height)
       monster.beAttacked(@attack)
      end
     end
     @image.play(animation: :attackSideSecond) do
      @canAttack = true
      self.stop
     end
    end
   when 'left'
    @canAttack = false
    @image.play(animation: :attackSideFirst, flip: :horizontal) do
     monsters.each do |monster|
      if CCHECK.intersect(@attackBoxLeft.x,@attackBoxLeft.y,@attackBoxLeft.width,@attackBoxLeft.height,
       monster.hitBox.x,monster.hitBox.y,monster.hitBox.width,monster.hitBox.height)
       monster.beAttacked(@attack)
      end
     end
     @image.play(animation: :attackSideSecond, flip: :horizontal) do
      @canAttack = true
      self.stop()
     end
    end
   end
  end
```

```ruby
    end

    def attackSpecial(monsters)
      if @magicBar.canUseSkill?
        @image.play(animation: :attackSpecial) do
          monsters.each do |monster|
            if
CCHECK.intersect(@attackBoxSpecial.x,@attackBoxSpecial.y,@attackBoxSpecial.width,@attackBoxSpecial.height,
              monster.hitBox.x,monster.hitBox.y,monster.hitBox.width,monster.hitBox.height)
              monster.beAttacked(@attack*2.5)
              if monster.exist
                monster.canmove = false
                monster.image.play(animation: :hurt) do
                  monster.canmove = true
                end
              end
            end
          end
          @magicBar.useSpecialskill
          self.stop()
        end
      end
    end

    def beAttacked(ammounts)
      @healthBar.hp -= ammounts
    end

```

- update player code:

```ruby
    def updatePlayer(monsters, map, npcs, items)

      #1. Update Health bar
      self.healthBar.update()
      #2. Update Magic bar
      self.magicBar.update()
      #3. Move
      self.move(monsters, map, npcs, items)

    end

```

- Update map:

```ruby
      def camera(player)
          for i in 0..CP::MAX_WORLD_ROWS-1
              for j in 0..CP::MAX_WORLD_COLS-1

                  # World Coordinate of tile[i][j]
```

```
                worldX = j * CP::TILE_SIZE
                worldY = i * CP::TILE_SIZE

                # Screen Coordinate of tile[i][j] should be
                screenX = worldX - player.worldX + player.x
                screenY = worldY - player.worldY + player.y

                #World Coordinate of Camera
                cameraWorldX = player.worldX - player.x
                cameraWorldY = player.worldY - player.y




                # Rendering game by removing unnessary images (we keep images in camera's scope, and remove otherwise)
                if(CCHECK.intersect(cameraWorldX, cameraWorldY, CP::SCREEN_WIDTH, CP::SCREEN_HEIGHT,
                                    worldX, worldY, CP::TILE_SIZE, CP::TILE_SIZE) == true)  #Notice we want the dimension of camera is exactly same as our window
                    @tileSet[i][j].image.x = screenX
                    @tileSet[i][j].image.y = screenY
                    @tileSet[i][j].image.add

                    #Check what type is standing on and apply effect
                    if CCHECK.intersect(player.hitBox.x, player.hitBox.y, player.hitBox.width, player.hitBox.height,
                        screenX, screenY, CP::TILE_SIZE, CP::TILE_SIZE)
                        if @tileSet[i][j].is_a?(Fire) || @tileSet[i][j].is_a?(Water)
                            case @tileSet[i][j].is_a?(Fire)
                            when true
                                player.beAttacked(1)
                            when false
                                player.beAttacked(-1)
                                player.magicBar.mp += 0.9
                            end
                        end
                    end
                else
                    @tileSet[i][j].image.remove
                end
            end
        end
    end

    #
    def updateMap(player)
        self.camera(player)
    end
```