KAUNAS UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATICS

T120B166 Development of Computer Games and Interactive Applications

Mineworld

Group, Name and Surname:
Rokas Jurgelionis IFF-0/2
Matas Vaitkevičius IFF-0/3

Date: *2023.02.21*

Kaunas, 2023

Tables of Contents

Tables of Images

Table of Tables/functions

Work Distribution Table:

Name/Surname	***Description of game development part***
Rokas Jurgelionis	*World generation, core mechanics*
Matas Vaitkevičius	*Boss design, UI, Character movement, code refactoring*

Description of Your Game

Mineworld is a fast paced game of minesweeper inspired by roguelike and boss rush genres. The player controls a character that has to carefully traverse a minesweeper grid searching for grand loot. In order to win, the player has to defeat an entire array of powerful bosses that guard legendary treasures.

Aforementioned bosses may inhabit unique biomes located in every cardinal direction. For example, a forest biome would be located northwards from the starting location. Its unique boss would have the ability to leap around the grid, trying to land on the player character. In order to damage this boss, the player would have to lure the boss to land on a mine instead. Likewise, a desert biome could present a challenge via sandstorms, that cover up the tiles and provide an ever-shifting environment.

Upon defeating a boss, they would drop an item that could change the way the players interact with the game. They could get the ability to disarm mines or maybe even lay new ones.

The game will be developed using Godot engine and will primarily be focused towards PC use. Gamepad support could be implemented as well

Laboratory work #1

List of tasks (main functionality of your project)

1. Generate a minesweeper grid
2. Create controls for a player character
3. Create a death screen
4. Create a level selection menu

Solution

Task #1. *Generate a minesweeper grid*

In order to begin creating anything for our game, a game grid is first required. We are striving to achieve the feel of the original “Windows minesweeper” game, so we will be using the original logic behind minesweeper. We do this by first making all of the tile objects and then setting a specified amount of tiles to be bombs. Once we have this, we calculate how many bombs are around each tile and we hide all of the tiles. Now that we are in the game, we can click on each tile, where a controller can check if it was pressed and reveal surrounding tiles.

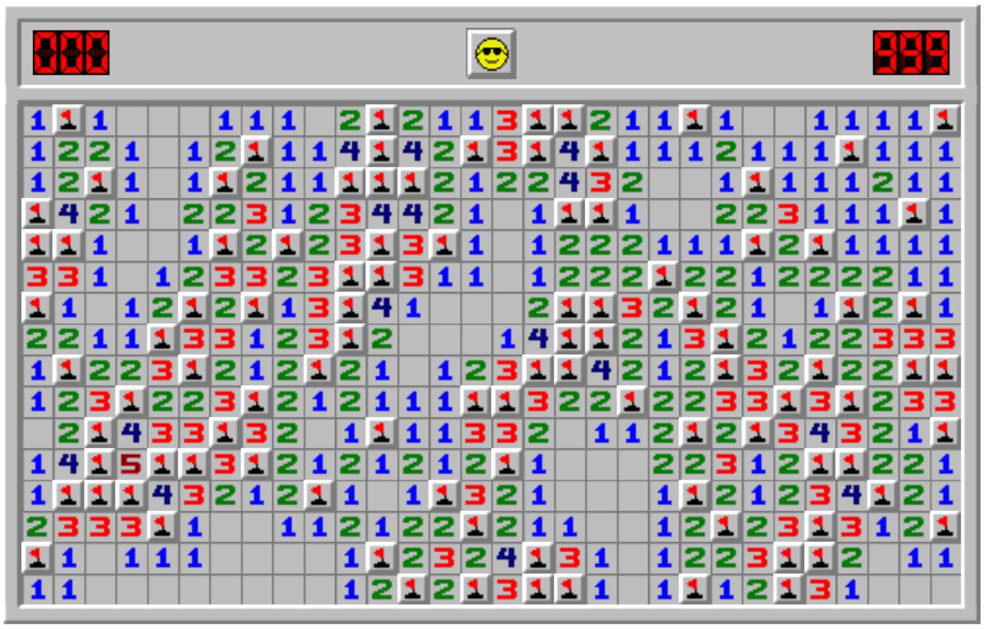
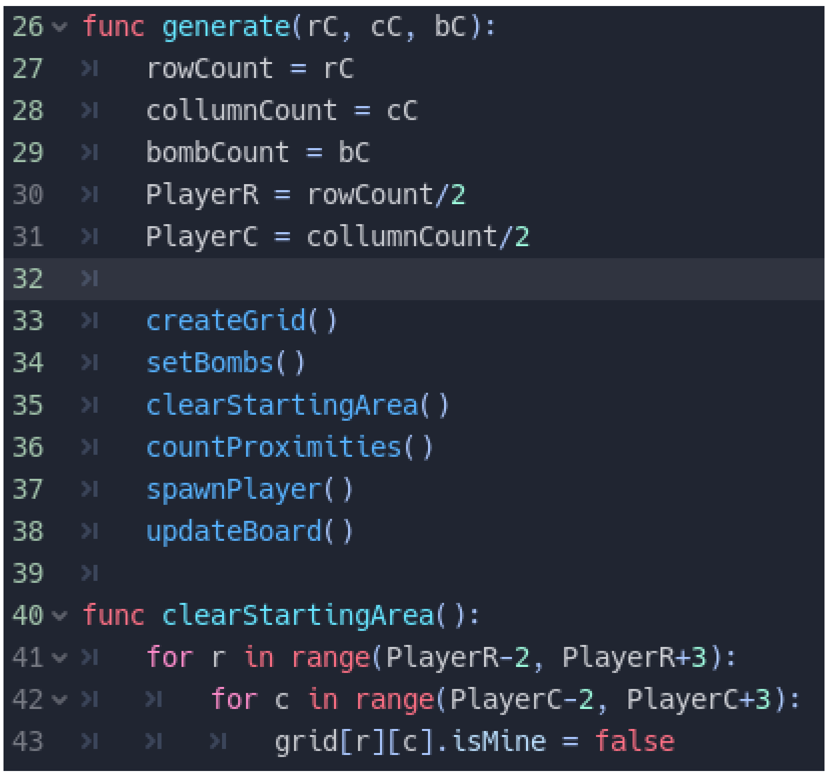


Figure 1. Original minesweeper



```
func generate(rC, cC, bC):
	rowCount = rC
	collumnCount = cC
	bombCount = bC
	PlayerR = rowCount/2
	PlayerC = collumnCount/2
	
	createGrid()
	setBombs()
	clearStartingArea()
	countProximities()
	spawnPlayer()
	updateBoard()
	
func clearStartingArea():
	for r in range(PlayerR-2, PlayerR+3):
		for c in range(PlayerC-2, PlayerC+3):
			grid[r][c].isMine = false
```

Table 1. Setting up the grid

Task #2. *Create controls for the player character*

Now that we have the basic minesweeper game, we can begin implementing our main feature - a player character on the grid that can move around clear mines. We do this by creating an object controlled with arrow keys which is also followed by the camera. Once we detect that a player moves, we check the tile the player moved onto and reveal it and the surrounding tiles.

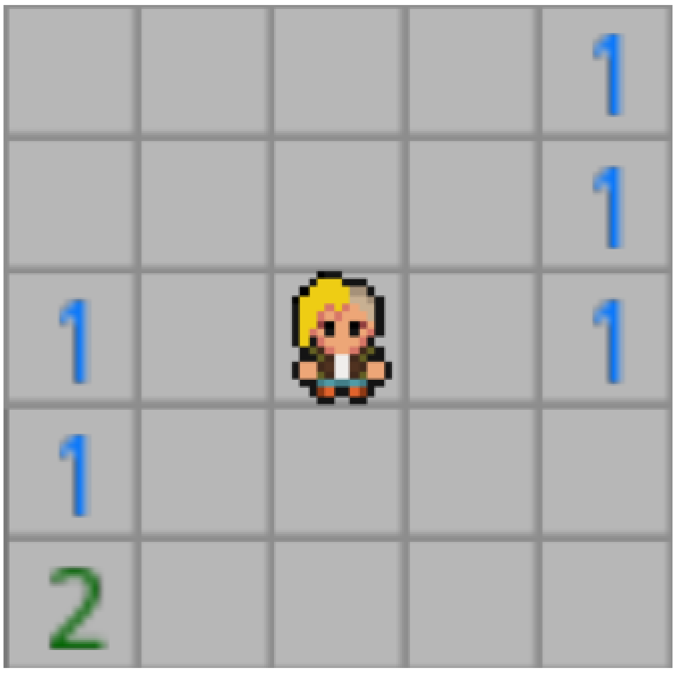
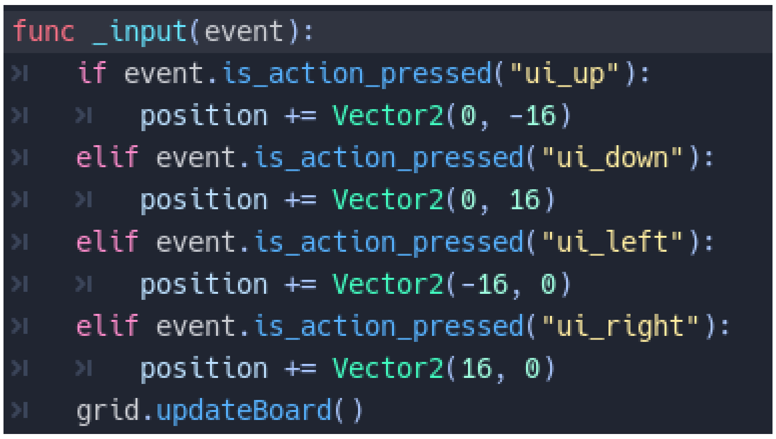


Figure 2. Player sprite



```
func _input(event):
	if event.is_action_pressed("ui_up"):
		position += Vector2(0, -16)
	elif event.is_action_pressed("ui_down"):
		position += Vector2(0, 16)
	elif event.is_action_pressed("ui_left"):
		position += Vector2(-16, 0)
	elif event.is_action_pressed("ui_right"):
		position += Vector2(16, 0)
	grid.updateBoard()
```

Table 2. Player movement

Task #3. *Create a death screen*

Of course, in order to have a game, we need a loss condition, which in this case will be the player moving onto a mine. Upon entering the tile, we display a "Game Over" message and disable player's movement.

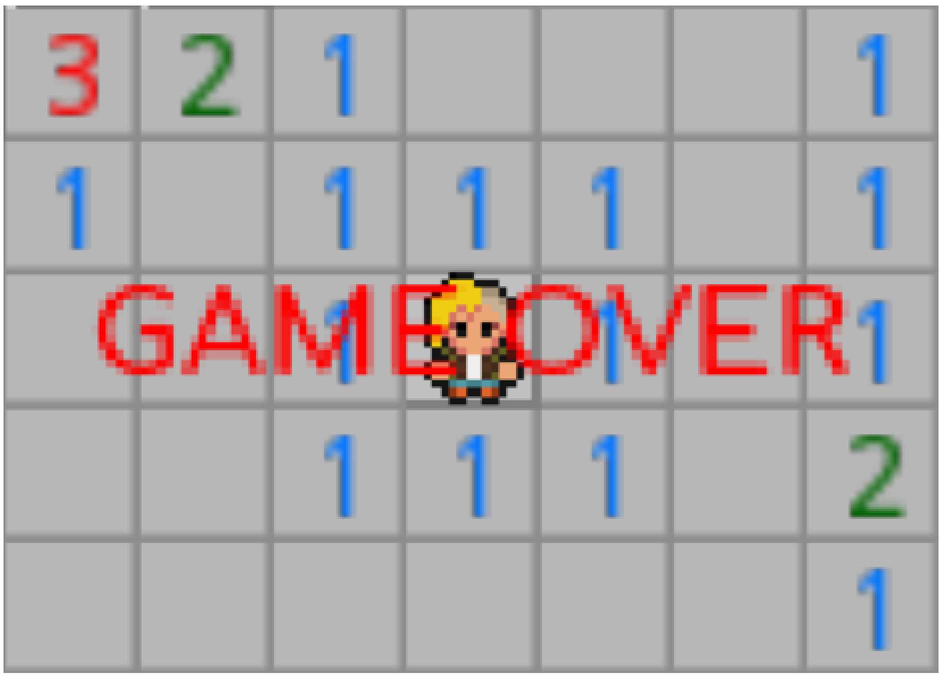
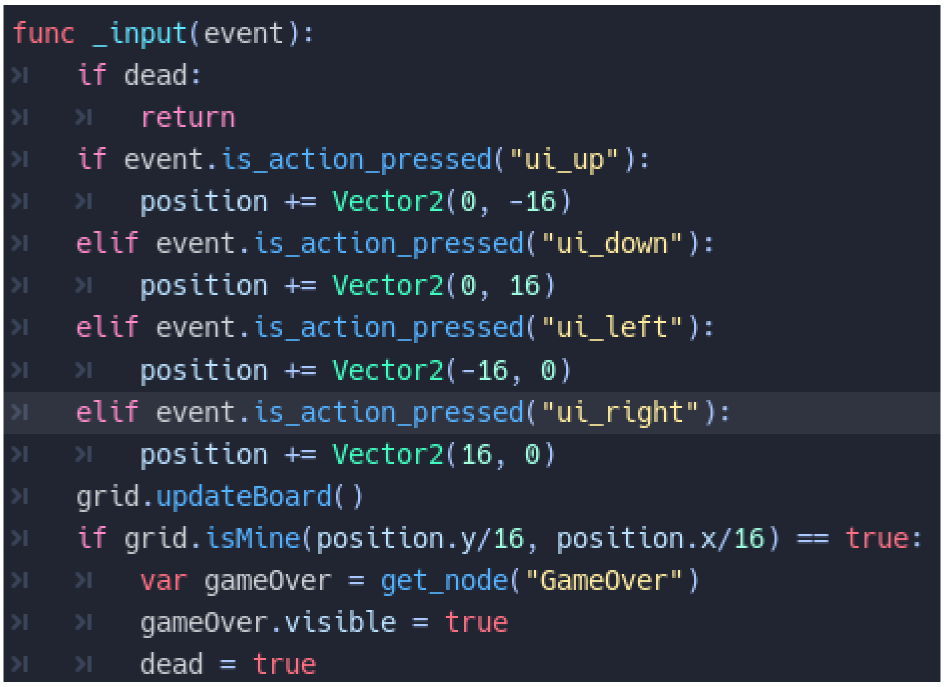


Figure 3. Game over screen



```
func _input(event):
	if dead:
		return
	if event.is_action_pressed("ui_up"):
		position += Vector2(0, -16)
	elif event.is_action_pressed("ui_down"):
		position += Vector2(0, 16)
	elif event.is_action_pressed("ui_left"):
		position += Vector2(-16, 0)
	elif event.is_action_pressed("ui_right"):
		position += Vector2(16, 0)
	grid.updateBoard()
	if grid.isMine(position.y/16, position.x/16) == true:
		var gameOver = get_node("GameOver")
		gameOver.visible = true
		dead = true
```

Table 3. Modified player movement

Task #4. *Create a level selection menu*

Currently, our game has 3 levels (Easy, Medium and Hard) with different parameters. In order to achieve this, we created 2 different scenes, where one was the grid scene that we worked on in the previous tasks and the other one was the menu scene. Within the menu scene, we created 3 controllers with labels that act as buttons which detect a click. These clicks are then sent to the LevelSelect script that loads the next scene with different parameters that describe the size of the grid and the amount of bombs in it.

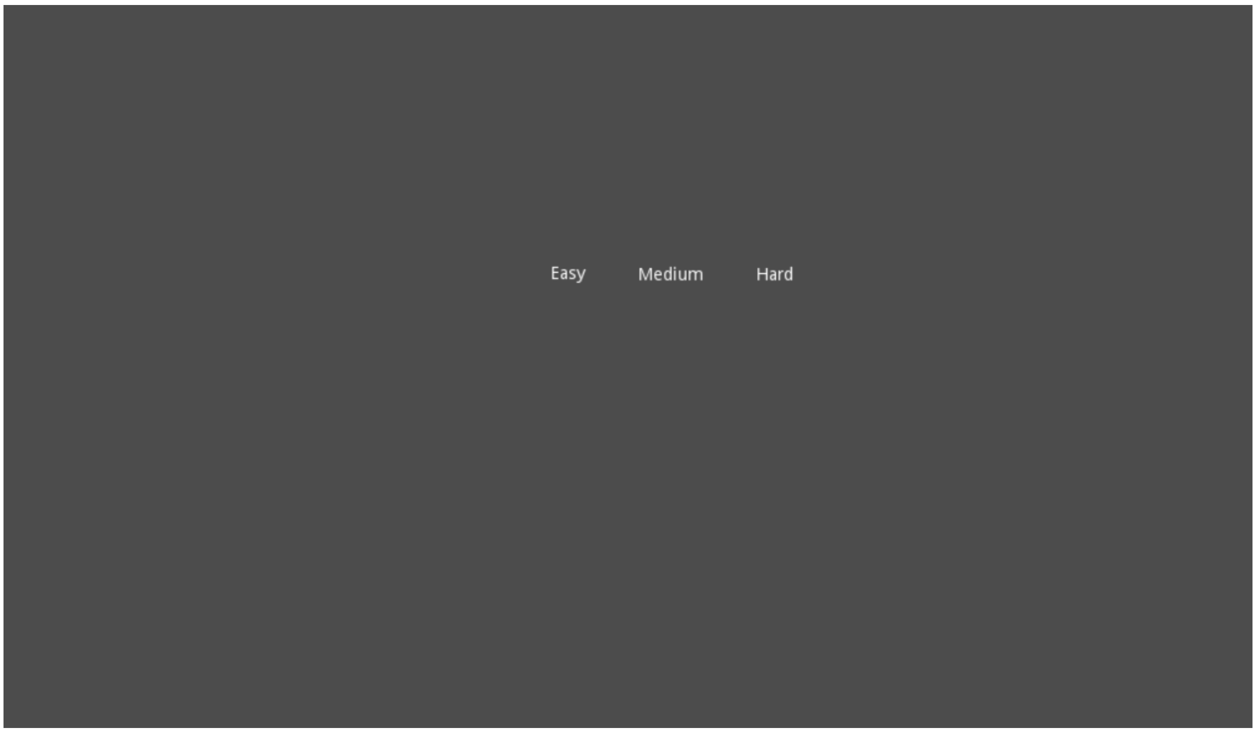
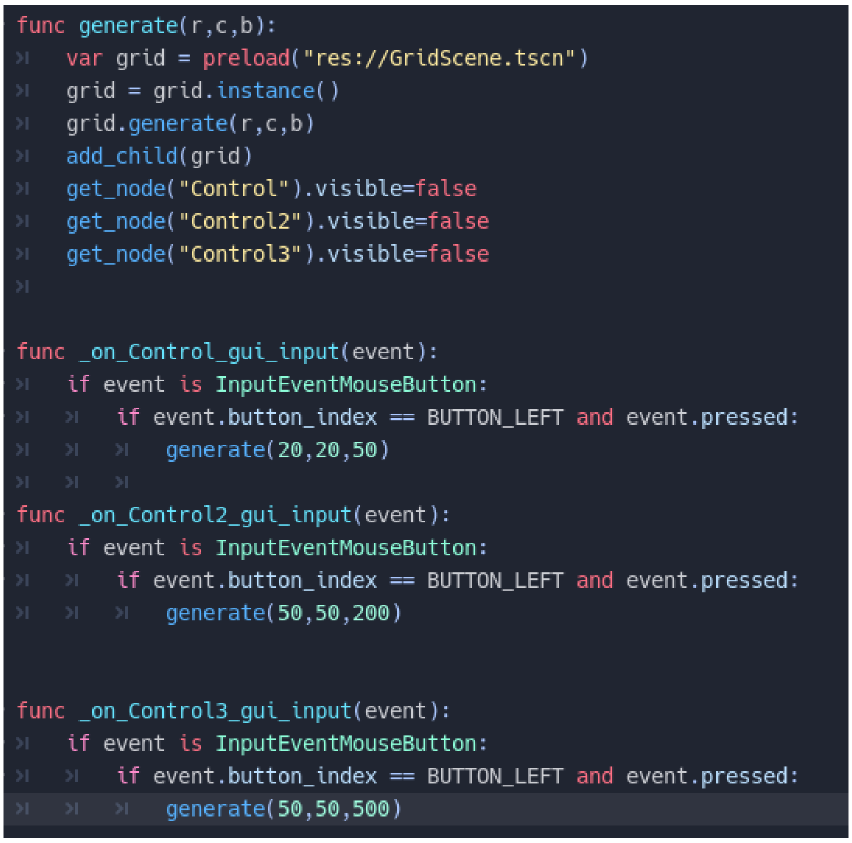


Figure 4. Level selection menu



```
func generate(r,c,b):
	var grid = preload("res://GridScene.tscn")
	grid = grid.instance()
	grid.generate(r,c,b)
	add_child(grid)
	get_node("Control").visible=false
	get_node("Control2").visible=false
	get_node("Control3").visible=false


func _on_Control_gui_input(event):
	if event is InputEventMouseButton:
		if event.button_index == BUTTON_LEFT and event.pressed:
			generate(20,20,50)

func _on_Control2_gui_input(event):
	if event is InputEventMouseButton:
		if event.button_index == BUTTON_LEFT and event.pressed:
			generate(50,50,200)


func _on_Control3_gui_input(event):
	if event is InputEventMouseButton:
		if event.button_index == BUTTON_LEFT and event.pressed:
			generate(50,50,500)
```

Table 4. Level selection script

Laboratory work #2

List of tasks

1. Basic biomes
2. Dynamic map generation
3. Side-scrolling boss
4. Movement queue and animation

Solution

Task #1. *Basic biomes*

To add more variety and strategy to the game, we decided to implement a biome system, where different biomes have different amounts of mines, incentivizing the player to search for easier biomes to make progress. The biomes are generated by first creating an array of random integers and then expanding the array and later smoothing it down (the full algorithm explanation can be found in the cuberite biome generation explanation website [1]).

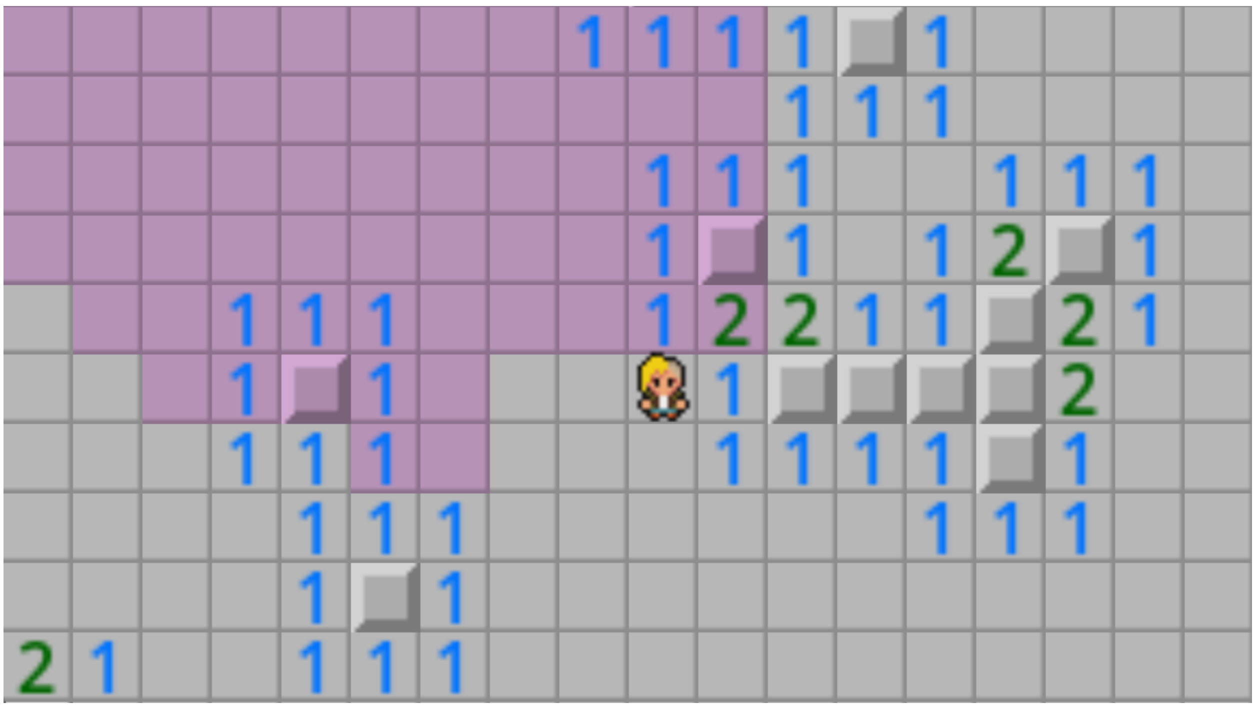
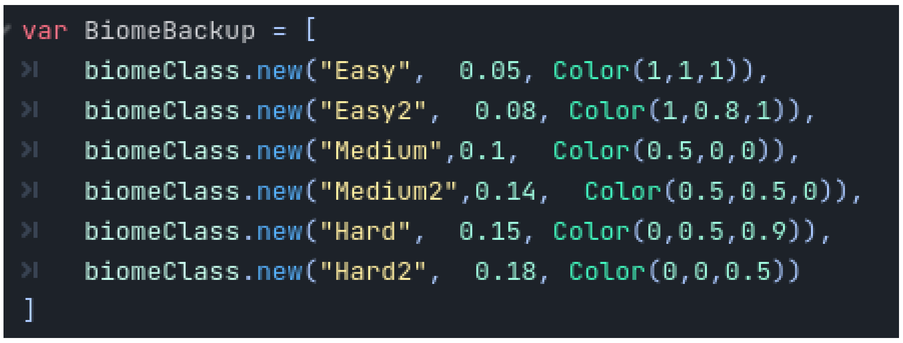


Figure 5. Player standing in front of two easy biomes

•



```
var BiomeBackup = [
	biomeClass.new("Easy",  0.05, Color(1,1,1)),
	biomeClass.new("Easy2",  0.08, Color(1,0.8,1)),
	biomeClass.new("Medium",0.1,  Color(0.5,0,0)),
	biomeClass.new("Medium2",0.14,  Color(0.5,0.5,0)),
	biomeClass.new("Hard",  0.15, Color(0,0.5,0.9)),
	biomeClass.new("Hard2",  0.18, Color(0,0,0.5))
]
```

Table 5. Various biomes

Task #2. *Dynamic map generation*

Before this implementation, we were only able to generate grids of sizes 150x150 tiles, which was not enough for the game. This was because we generated all of our tiles at once, so for this task we implemented a dynamic generation system, which would load the map as the player would walk near unloaded tiles. With this, we are now able to generate grids of 1500x1500 tiles, which will be more than enough for us moving on

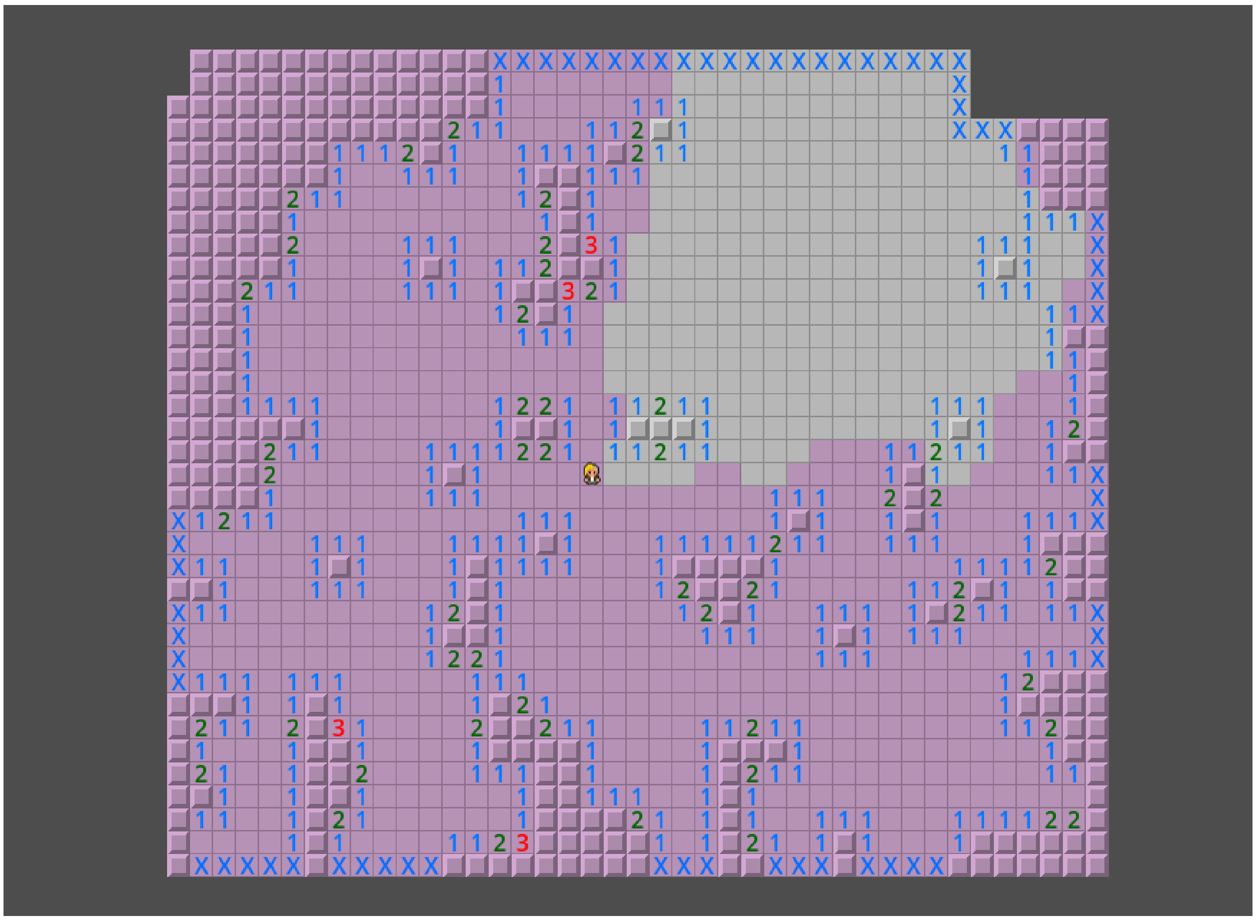
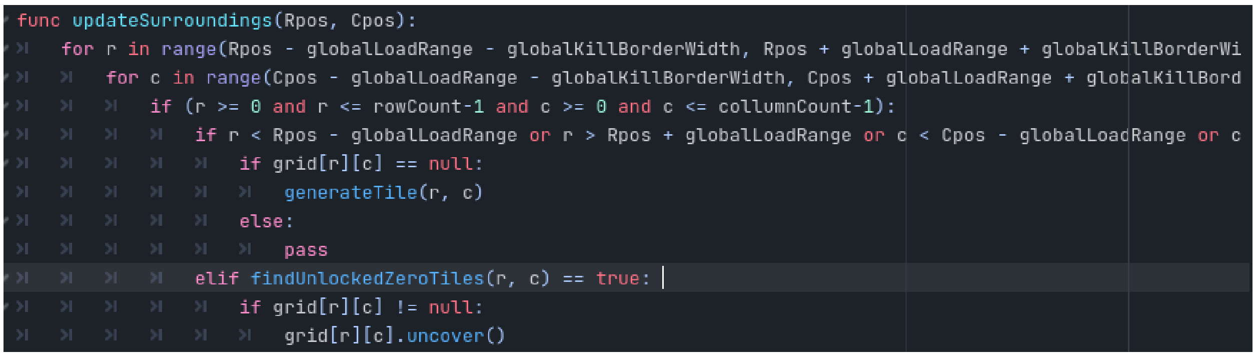


Figure 6. Partially loaded map



```
func updateSurroundings(Rpos, Cpos):
    for r in range(Rpos - globalLoadRange - globalKillBorderWidth, Rpos + globalLoadRange + globalKillBorderWi
        for c in range(Cpos - globalLoadRange - globalKillBorderWidth, Cpos + globalLoadRange + globalKillBord
            if (r >= 0 and r <= rowCount-1 and c >= 0 and c <= collumnCount-1):
                if r < Rpos - globalLoadRange or r > Rpos + globalLoadRange or c < Cpos - globalLoadRange or c
                    if grid[r][c] == null:
                        generateTile(r, c)
                    else:
                        pass
                elif findUnlockedZeroTiles(r, c) == true:
                    if grid[r][c] != null:
                        grid[r][c].uncover()
```

Table 6. Dynamic loading

Task #3. *Side-scrolling boss*

Because it isn't particularly interesting to aimlessly walk around, we added a boss that would act as a wall that moved across the screen, where if it is touched, the player loses. The boss adjusts his speed so that the further the player is away from it, the faster it will move, which doesn't allow the player to get away from it as easily and keeps it as a more constant threat.

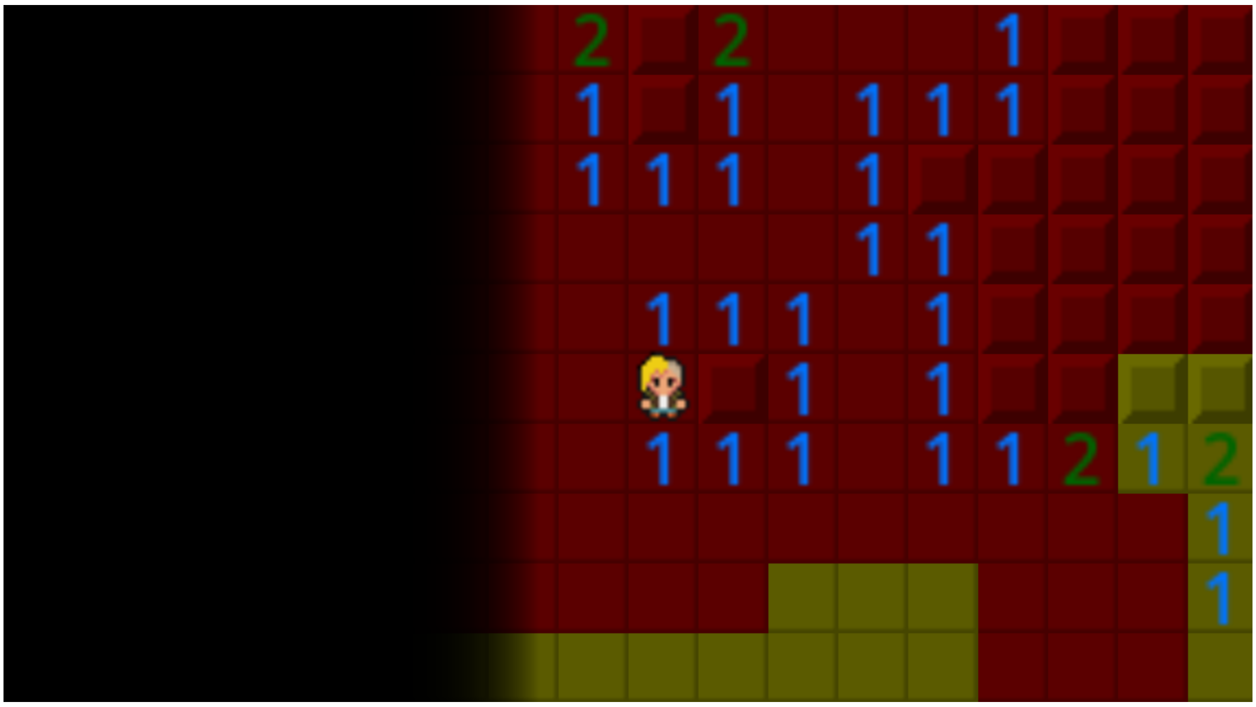
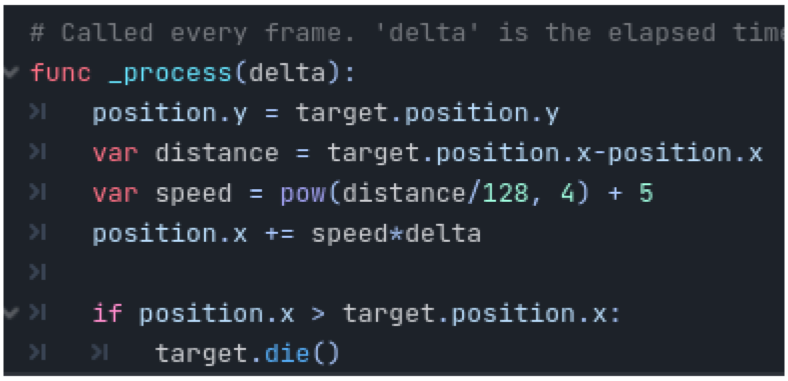


Figure 7. A creeping shadow constantly covering up the grid



```
# Called every frame. 'delta' is the elapsed time
func _process(delta):
	position.y = target.position.y
	var distance = target.position.x-position.x
	var speed = pow(distance/128, 4) + 5
	position.x += speed*delta
	
	if position.x > target.position.x:
		target.die()
```

Table 7. Moving the boss closer to the player

Task #4. *Movement queue and animation*

To improve player experience we added 2 more minor additions - an input queue that allows the game to remember inputs and an animation of the player character. The input queue was created to allow the player to better control the character, as before the addition, every input made during movement was not taken into account, making it frustrating to constantly have to wait and time your press to move quickly. Now, the player is able to press the next button to continue smoothly moving.

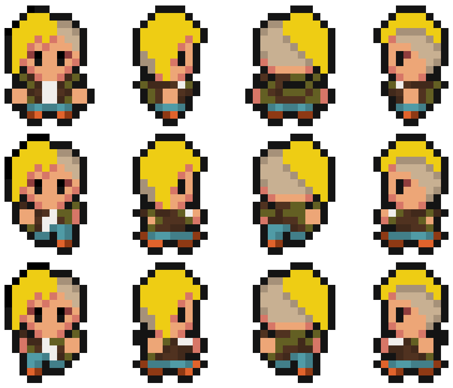
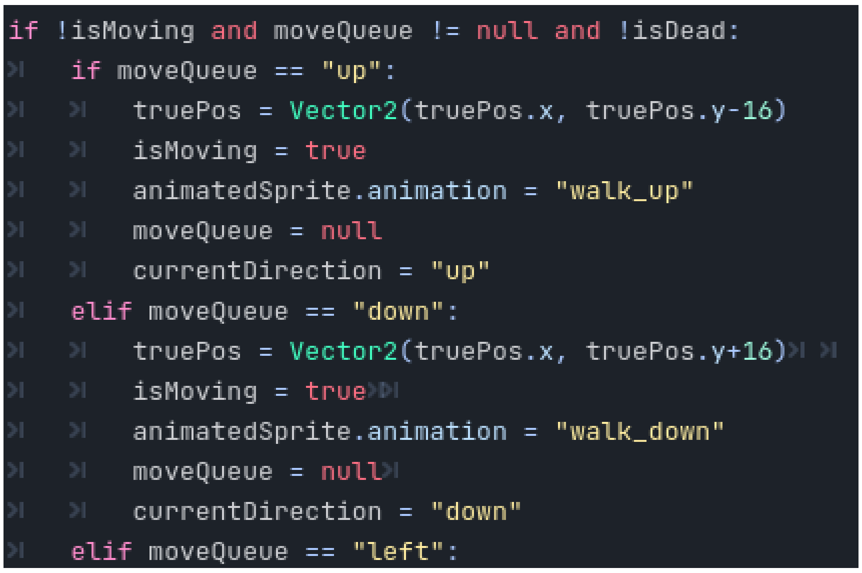


Figure 8. Character spritesheet



```
if !isMoving and moveQueue != null and !isDead:
	if moveQueue == "up":
		truePos = Vector2(truePos.x, truePos.y-16)
		isMoving = true
		animatedSprite.animation = "walk_up"
		moveQueue = null
		currentDirection = "up"
	elif moveQueue == "down":
		truePos = Vector2(truePos.x, truePos.y+16)
		isMoving = true
		animatedSprite.animation = "walk_down"
		moveQueue = null
		currentDirection = "down"
	elif moveQueue == "left":
```

Table 8. Character move queue

Laboratory work #3

List of tasks

1. Menu rework
2. Lives system
3. “Frog” boss
4. More biome mechanics
5. “Sniper” boss

Solution

Task #1. *Menu rework*

In order to make better first impressions, there was an attempt to improve the main menu. After all, it is the first thing a player would see whenever they launch a game. The menu currently allows the player to choose from three types of games, each featuring a unique boss.

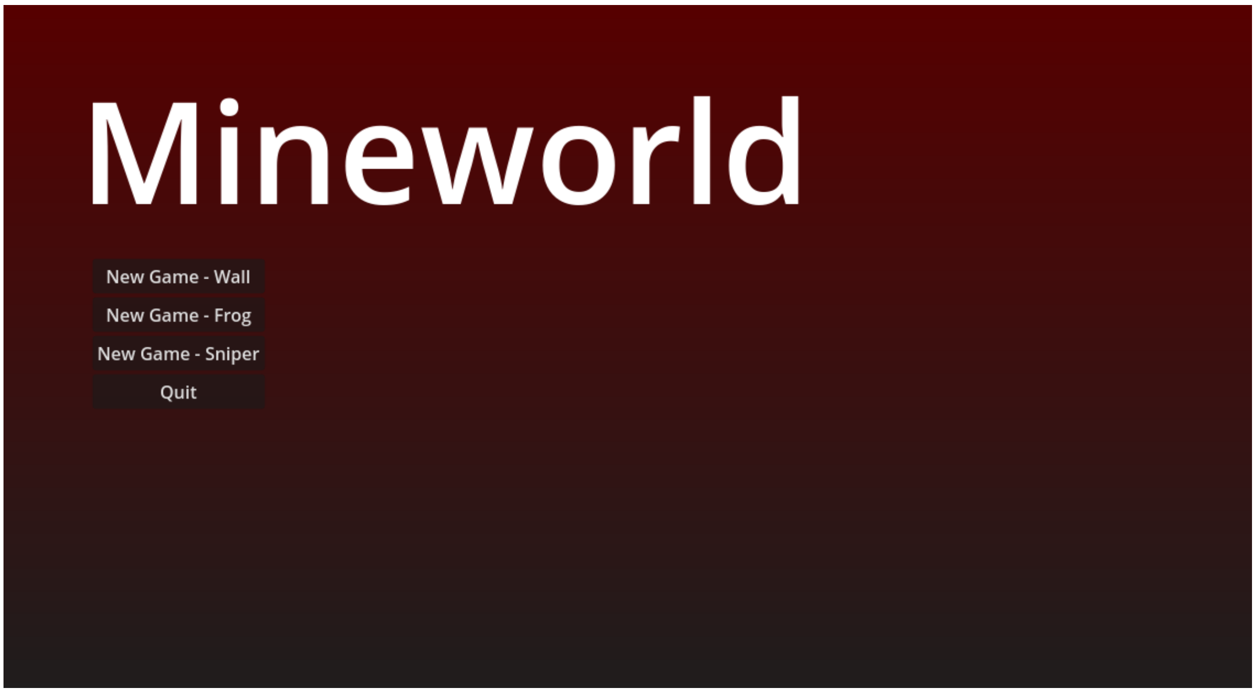
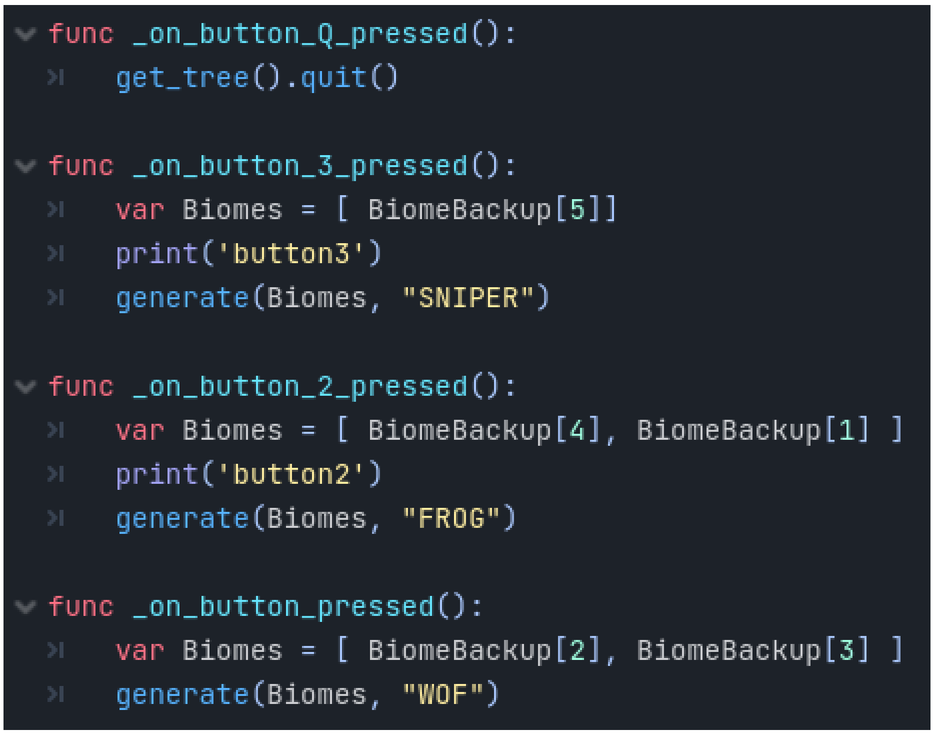


Figure 9. Main menu



```
func _on_button_Q_pressed():
    get_tree().quit()

func _on_button_3_pressed():
    var Biomes = [ BiomeBackup[5]]
    print('button3')
    generate(Biomes, "SNIPER")

func _on_button_2_pressed():
    var Biomes = [ BiomeBackup[4], BiomeBackup[1] ]
    print('button2')
    generate(Biomes, "FROG")

func _on_button_pressed():
    var Biomes = [ BiomeBackup[2], BiomeBackup[3] ]
    generate(Biomes, "WOF")
```

Table 9. Menu selection code

Task #2. *Lives system*

Minesweeper tends to have a single life, however, our game aims to provide longer playthroughs and so we found it needlessly frustrating to lose for a simple mistake if you were playing for a long time. This is why we implemented a lives system, which would allow the player to step on 3 mines before dying. Stepping on a mine would also make an explosion that would clear surrounding tiles, meaning that confident players can use this mechanic to clear tiles faster, while careful players can use it as a safety net.

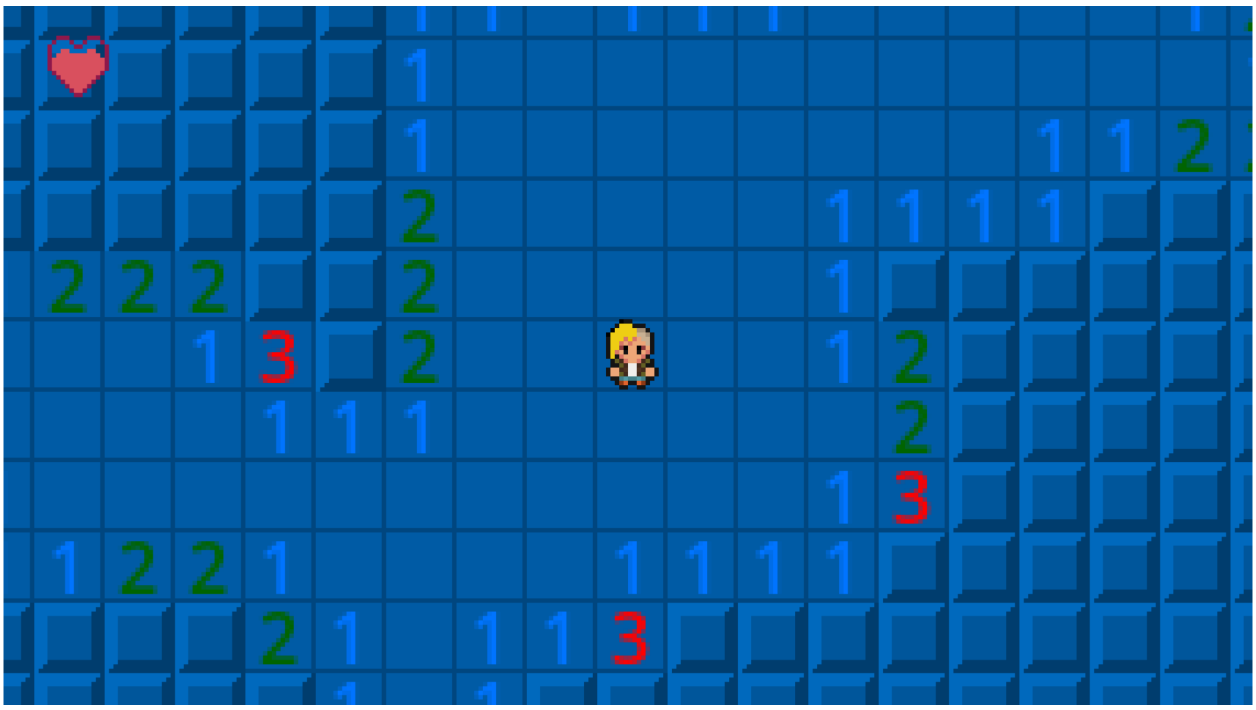
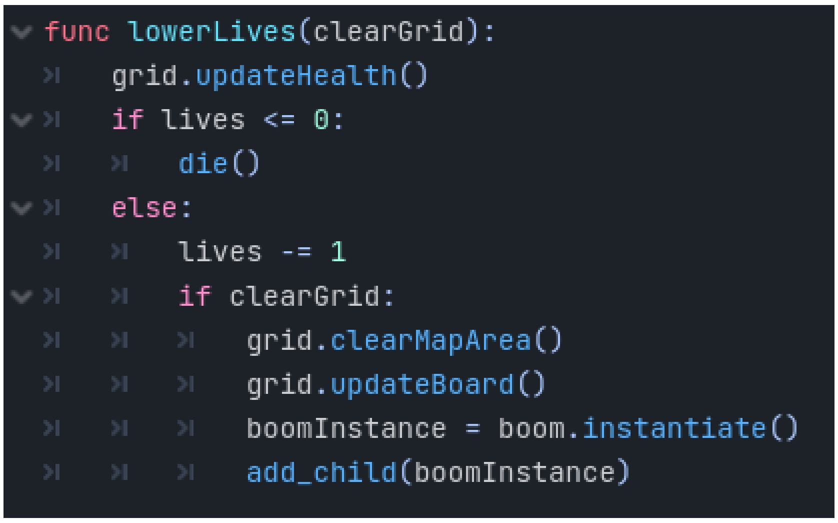


Figure 10. Lives shown in the top left of the screen



```
func lowerLives(clearGrid):
	grid.updateHealth()
	if lives <= 0:
		die()
	else:
		lives -= 1
		if clearGrid:
			grid.clearMapArea()
			grid.updateBoard()
			boomInstance = boom.instantiate()
			add_child(boomInstance)
```

Table 10. Life calculation

Task #3. *"Frog" boss*

To include more variation in gameplay, a second boss was added. "Frog" boss jumps around the map, trying to catch the player. If the frog lands on a player, they take damage. However, if the frog lands on a mine, the frog takes damage itself. After taking sufficient damage, frog is defeated.

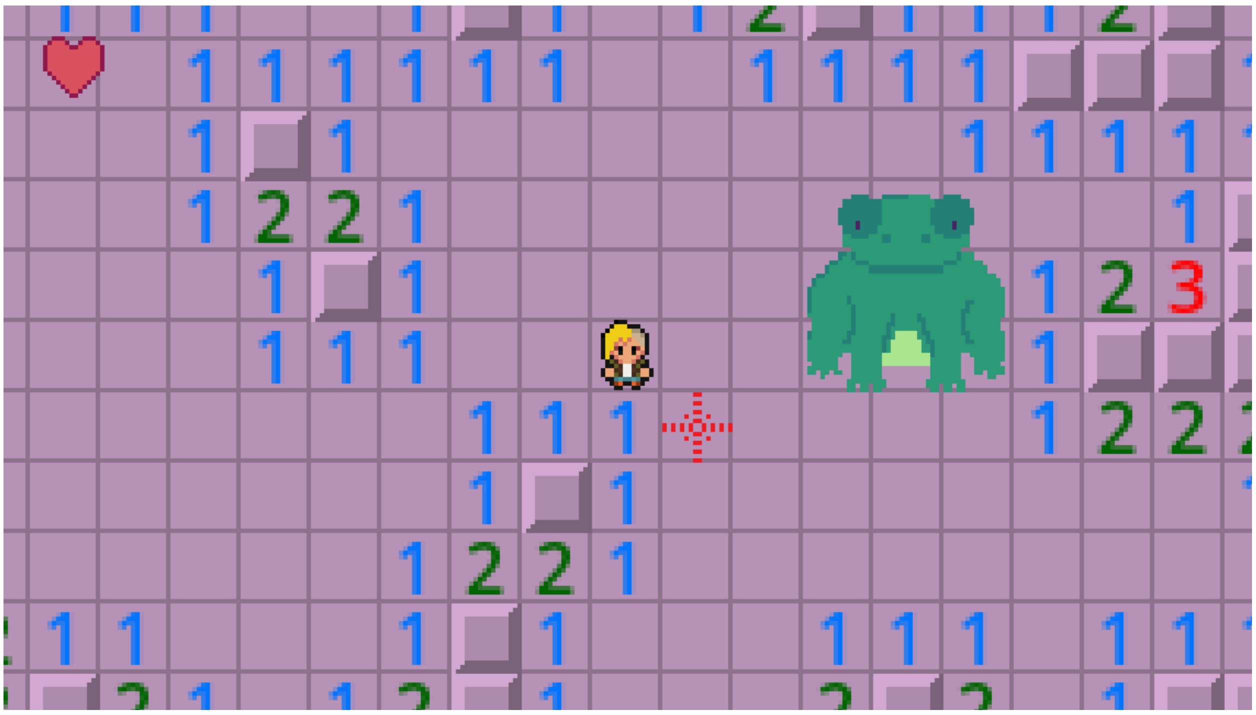
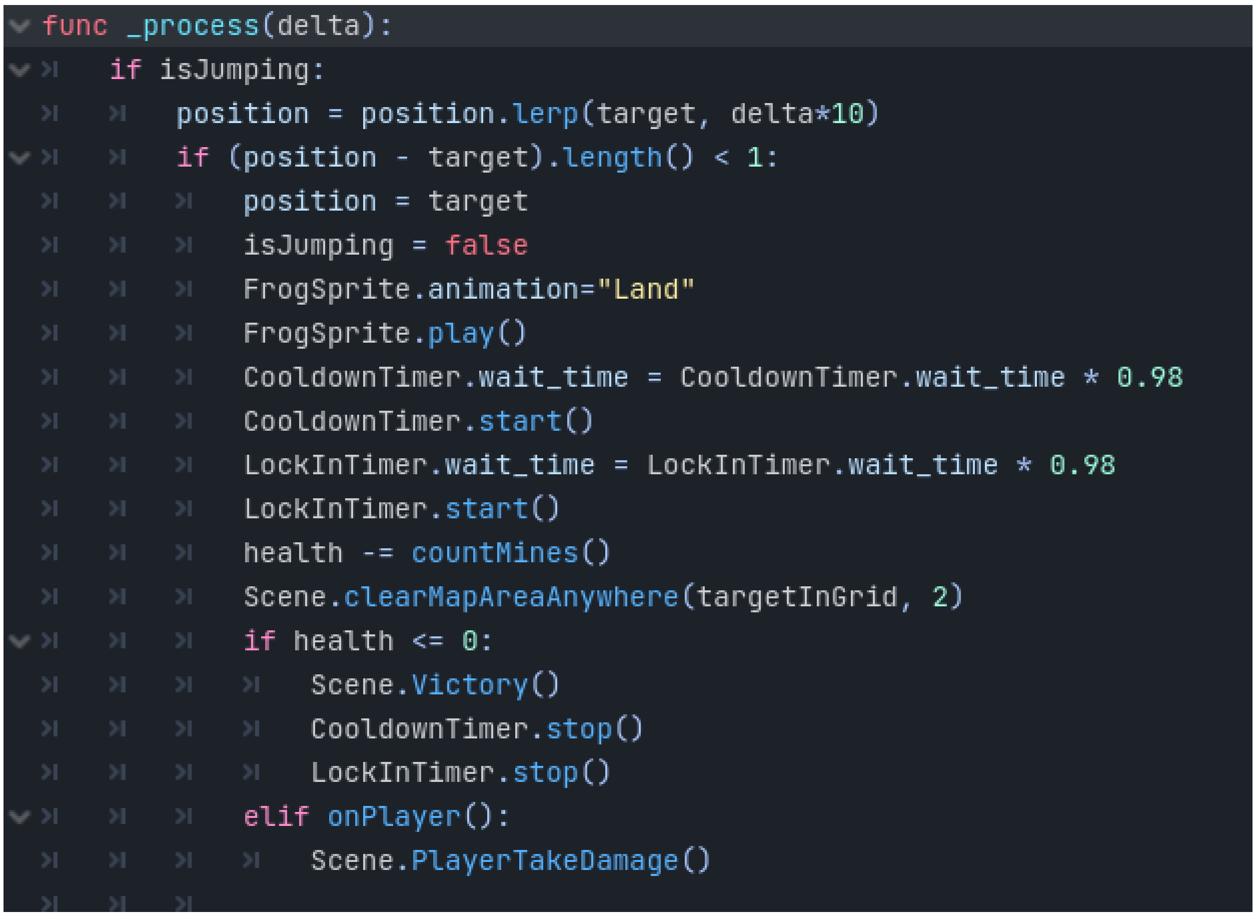


Figure 11. Frog



```
func _process(delta):
	if isJumping:
		position = position.lerp(target, delta*10)
		if (position - target).length() < 1:
			position = target
			isJumping = false
			FrogSprite.animation="Land"
			FrogSprite.play()
			CooldownTimer.wait_time = CooldownTimer.wait_time * 0.98
			CooldownTimer.start()
			LockInTimer.wait_time = LockInTimer.wait_time * 0.98
			LockInTimer.start()
			health -= countMines()
			Scene.clearMapAreaAnywhere(targetInGrid, 2)
			if health <= 0:
				Scene.Victory()
				CooldownTimer.stop()
				LockInTimer.stop()
			elif onPlayer():
				Scene.PlayerTakeDamage()
```

Table 11. Frog Progress

Task #4. *More biome mechanics*

To add more variety in gameplay, 2 more mechanics were created that could be applied to biomes. The first was a fog that would appear when entering a foggy biome - the fog would then slowly engulf the player and severely reduce visibility, forcing the player to better remember its surroundings. The second mechanic were walls, which would simply block the player's movement.

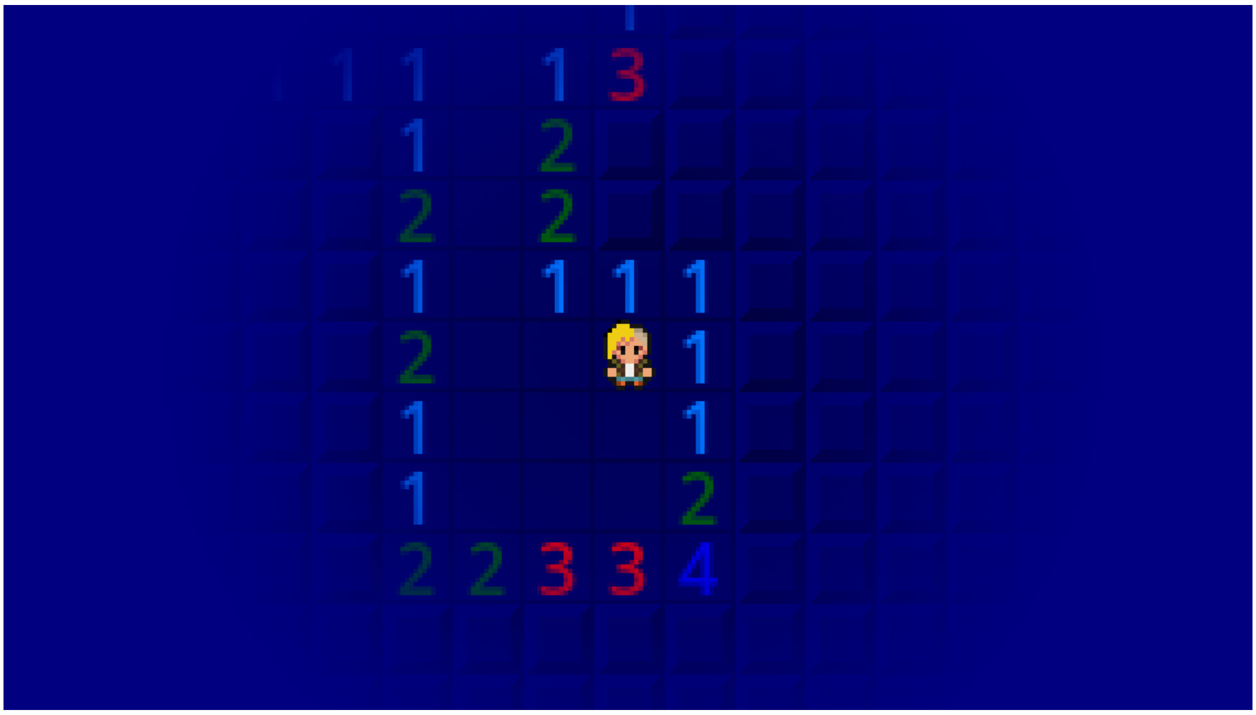
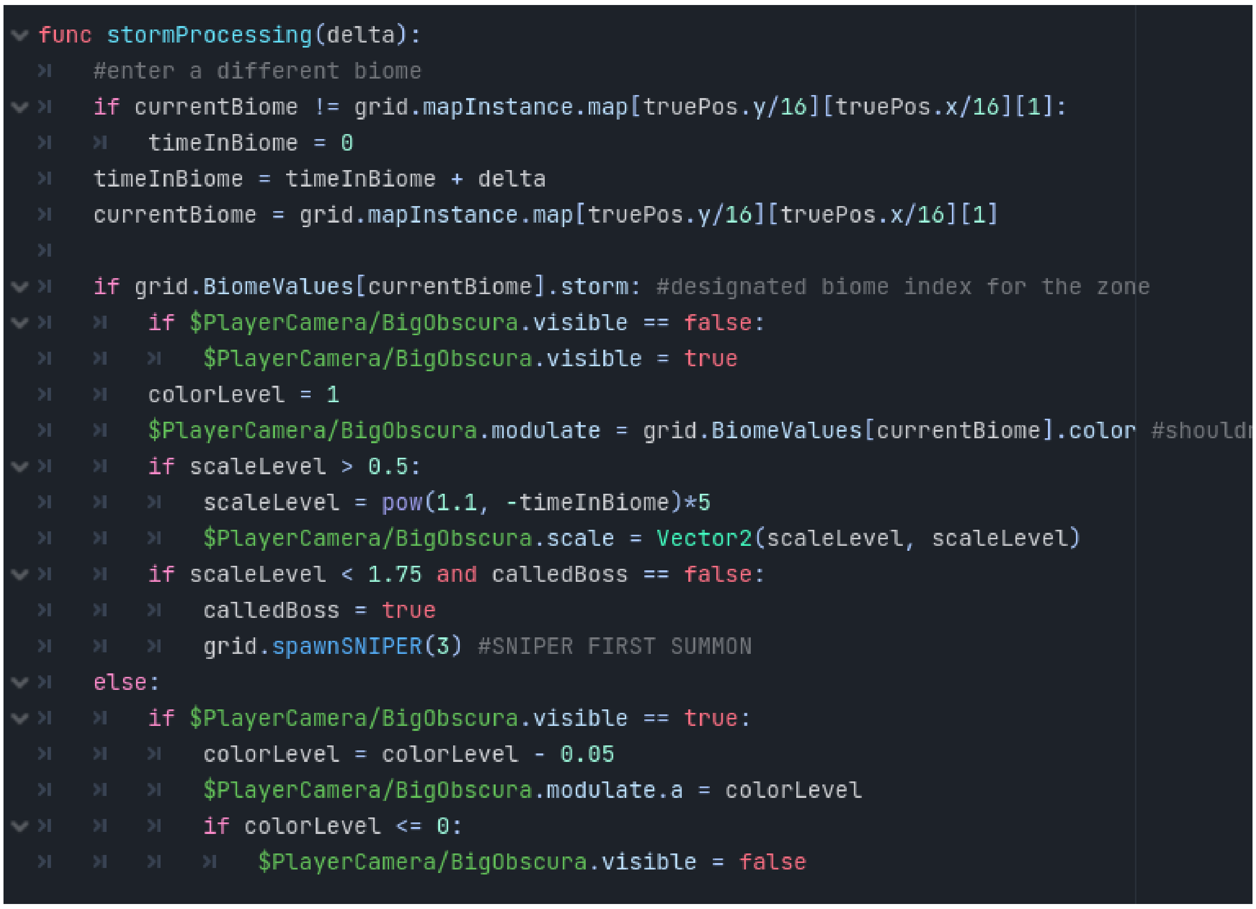


Figure 12. Fog effect



```
func stormProcessing(delta):
    #enter a different biome
    if currentBiome != grid.mapInstance.map[truePos.y/16][truePos.x/16][1]:
        timeInBiome = 0
    timeInBiome = timeInBiome + delta
    currentBiome = grid.mapInstance.map[truePos.y/16][truePos.x/16][1]

    if grid.BiomeValues[currentBiome].storm: #designated biome index for the zone
        if $PlayerCamera/BigObscura.visible == false:
            $PlayerCamera/BigObscura.visible = true
        colorLevel = 1
        $PlayerCamera/BigObscura.modulate = grid.BiomeValues[currentBiome].color #shouldn
        if scaleLevel > 0.5:
            scaleLevel = pow(1.1, -timeInBiome)*5
            $PlayerCamera/BigObscura.scale = Vector2(scaleLevel, scaleLevel)
        if scaleLevel < 1.75 and calledBoss == false:
            calledBoss = true
            grid.spawnSNIPER(3) #SNIPER FIRST SUMMON
    else:
        if $PlayerCamera/BigObscura.visible == true:
            colorLevel = colorLevel - 0.05
            $PlayerCamera/BigObscura.modulate.a = colorLevel
            if colorLevel <= 0:
                $PlayerCamera/BigObscura.visible = false
```

Table 12. Fog colour code

Task #5. *“Sniper” boss*

For another boss, a “sniper” idea was chosen. If a player was in a foggy biome long enough, a sniper would appear and would begin to target the player. Every set amount of time, the sniper shoots a bullet that the player has to avoid or take damage. The only way to defeat the sniper is to track him by the laser he creates and touch him, at which point he will teleport to a new location. Repeating this 2 more times kills the sniper.

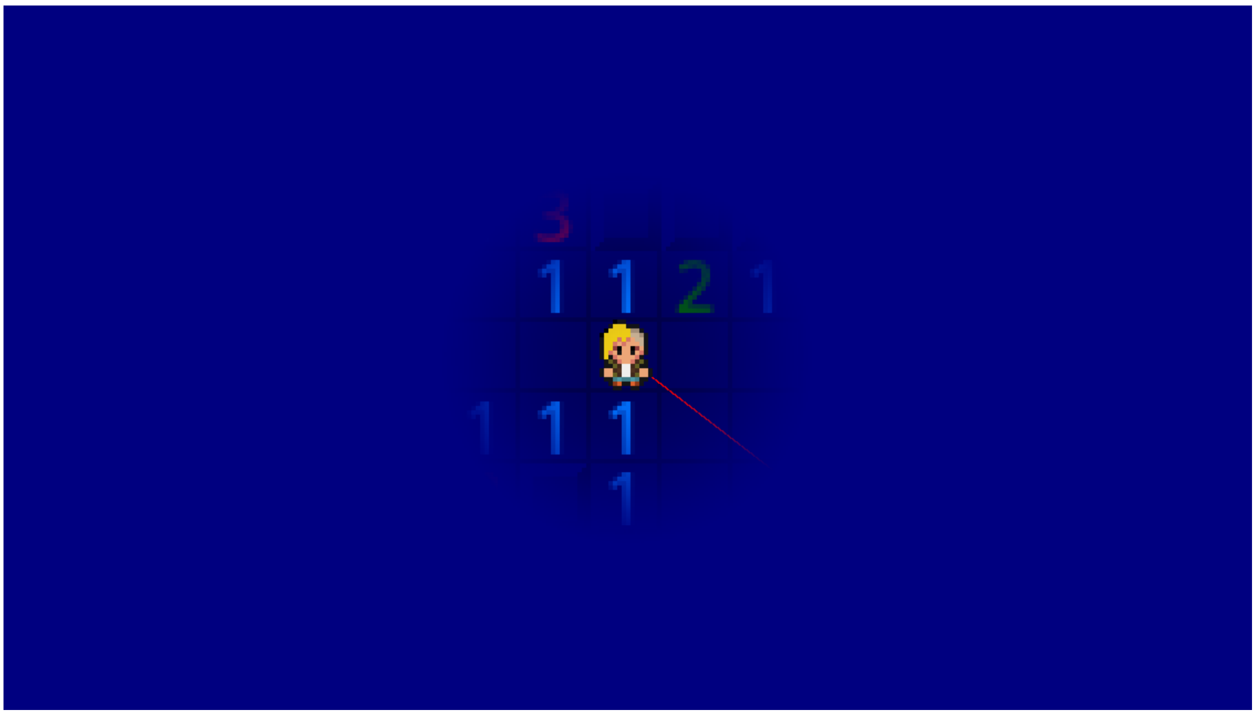
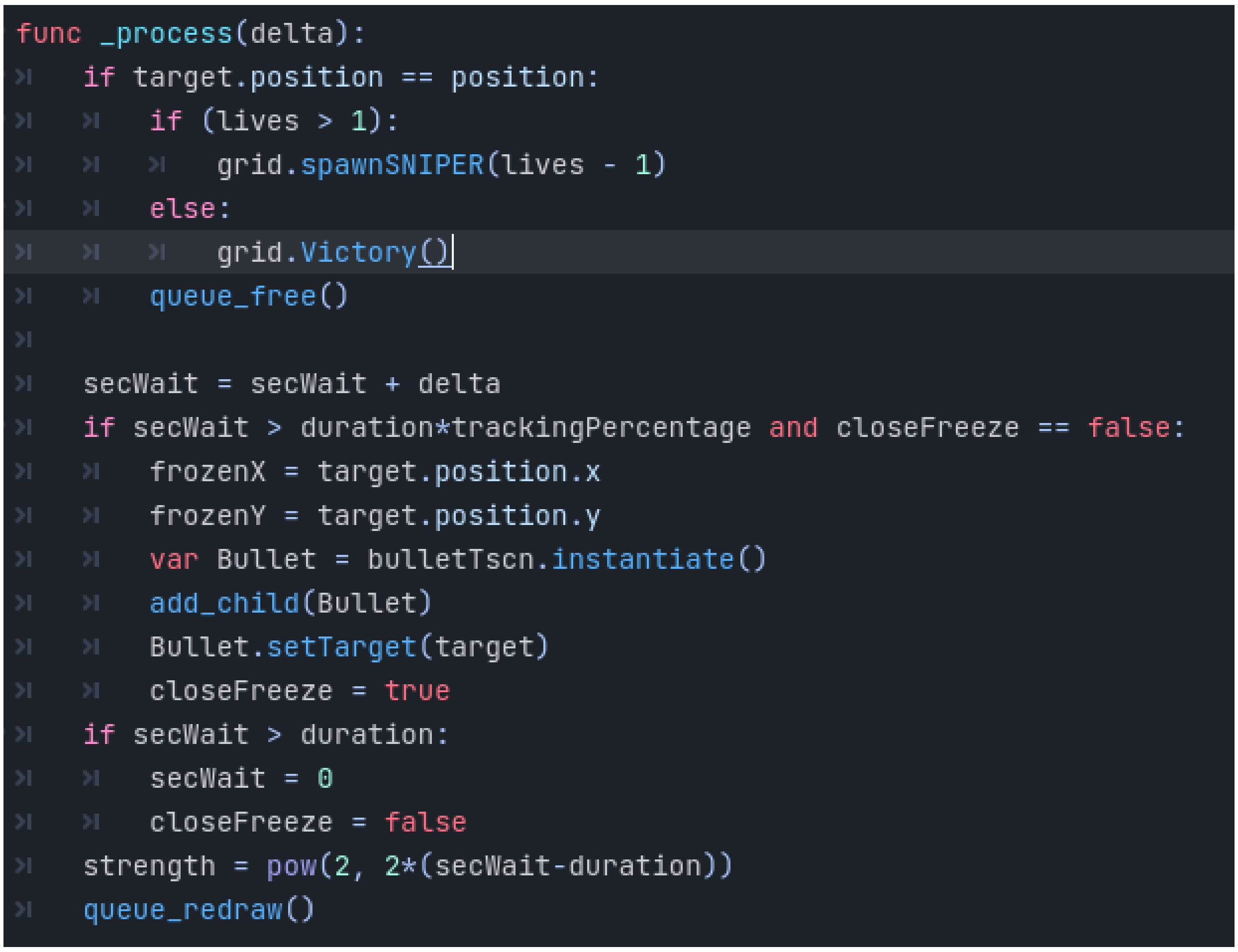


Figure 13. Sniper’s laser in fog

In the case of using functions, the description of each main function should be completed with the source code FRAGMENTS (the functions should be indexed in a separate table of contents);



```
func _process(delta):
	if target.position == position:
		if (lives > 1):
			grid.spawnSNIPER(lives - 1)
		else:
			grid.Victory()
		queue_free()

	secWait = secWait + delta
	if secWait > duration*trackingPercentage and closeFreeze == false:
		frozenX = target.position.x
		frozenY = target.position.y
		var Bullet = bulletTscn.instantiate()
		add_child(Bullet)
		Bullet.setTarget(target)
		closeFreeze = true
	if secWait > duration:
		secWait = 0
		closeFreeze = false
	strength = pow(2, 2*(secWait-duration))
	queue_redraw()
```

Table 13. Main sniper behaviour

User's manual (for the Individual work defence)

How to play? *Aenean eu quam gravida, laoreet nisl eu, sagittis quam. Donec sit amet nunc nisi. Sed vel ipsum metus. Nullam accumsan vestibulum ex. Aenean eu quam gravida, laoreet nisl eu, sagittis quam. Donec sit amet nunc nisi. Sed vel ipsum metus. Nullam accumsan vestibulum ex.*

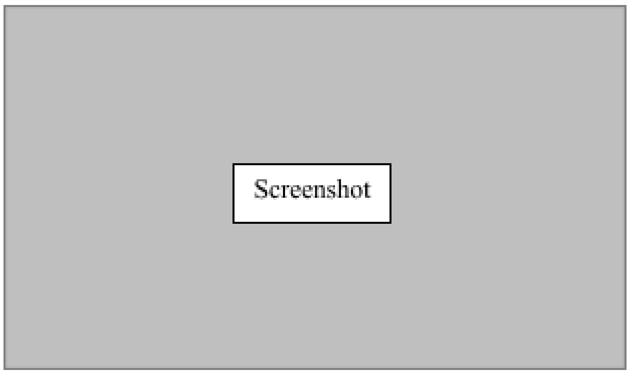


Figure 10. Screenshot #5

Nunc vel enim vel magna interdum dapibus id nec nisl. Suspendisse elit augue, accumsan tempor erat sed, gravida suscipit urna. Duis blandit lacus et finibus finibus. Mauris pretium pharetra orci dictum luctus. Nullam commodo magna a tincidunt malesuada.

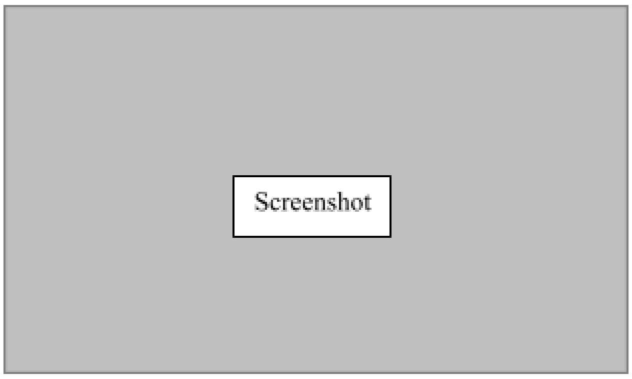


Figure 11. Screenshot #5

Sed sollicitudin justo erat, viverra luctus mi consequat non. Sed ut condimentum libero. Duis rutrum lacus ante, vitae feugiat ex faucibus at. Maecenas pulvinar et augue sed commodo.

Descriptions of the rules of the game. Nunc quis condimentum lacus. Quisque felis neque, ullamcorper vel posuere eget, blandit non neque. Nam in varius erat. Duis molestie sit amet eros vel rhoncus. Nunc quis condimentum lacus. Quisque felis neque, ullamcorper vel posuere eget, blandit non neque. Nam in varius erat. Duis molestie sit amet eros vel rhoncus.

Descriptions of the controls / keys. Donec et lorem vitae ligula bibendum faucibus. Suspendisse interdum quis augue sed luctus. Curabitur ac diam augue. In hac habitasse platea dictumst. Curabitur maximus maximus tortor. Nunc quis condimentum lacus. Quisque felis neque, ullamcorper vel posuere eget, blandit non neque. Nam in varius erat. Duis molestie sit amet eros vel rhoncus.

Literature list

1. Biome generation algorithms. *http://cuberite.xoft.cz/docs/Generator.html*
2. Source #2. *Url*
3. ...
4. Source #N. *Url*

ANNEX

All source code is contained in this part.