Game/Project name: Island Escape

By: Jack Chang

Original game plot/idea:

You (the player) are the world's most intelligent criminal, and you have been put onto an island in the middle of who knows where, with the world's top security. Your job, of course, is to escape. You will have to figure out a way to get through many different things to escape from the island, including different robots that will be trying to capture you, collect certain items to get through certain parts, disable certain things so that you can get through certain parts, and so on. This is a turn-based game on a 2d grid of tiles/squares; after you make an actio, whatever enemies there are (all the robots) all make a move (or possibly more) as well, trying to prevent you from reaching the exit. A lot of tiles on the grid will have a motion detector on it, meaning that if you are on it, the robots will know. But, some tiles won't have a motion detector, meaning the robots will not know you are there if you go to that square. You have also stolen one of their GPS tracker things so you get the motion detector as well, with the same rules; you only know the locations of robots that are on squares with a motion detector.

Different screens of the game:

- Title screen
- Level select
- Settings
- Each level
- Level win screen
- Level loss/game over screen
- Intro/story screen
- Escape/game win screen

Classes (OOP) and their functions:

- Button
    - __init__(self, x0, y0, x1, y1, clickEvent, regularFill, hoverFill, textColor, outline, text, font, fontSize, bold) → initializes a Button object where (x0, y0) is the top left of the button, and (x1, y1) is the right bottom of the button, clickEvent is a string as to what to do when this button is clicked, and uses the rest of the parameters to draw the button and make the text
    - getBounds(self) → returns a tuple (x0, y0, x1, y1) where (x0, y0) is the top left of the button, and (x1, y1) is the right bottom of the button
- GridObject
    - __init__(self, row, col) → initializes a GridObject object at (row, col) on the playing grid of tiles
- Robot → extends GridObject
    - __init__(self, row, col, chaseType) → initializes a Robot object at (row, col), and chaseType is a string, which determines how the Robot behaves
-

Things to be stored in the app object and what data type they are (all initialized at the start):

Core components:

- width (the width of the screen) → int
- height (the height of the screen) → int
- mode (the current screen) → string
- musicOn → boolean
- soundEffOn → boolean
- levelsCompleted → set
- buttons → list of Buttons

In level:

- player → GridObject
- robots → list of Robots
- grid → 2d list
- rows → int
- cols → int
- exit → GridObject
- leftMargin → int
- rightMargin → int
- topMargin → int
- bottomMargin → int
- visualizeBFS (a testing boolean that is True when I want to see the BFS path) → boolean
- BFSPath (for testing purposes to store the BFS path if I ever want to look at it) → list

App draw functions:

- drawBackground(canvas, app)
    - Draws whatever the background is
- drawGrid(canvas, app)
    - Draws the playing grid
- drawPlayer(canvas, app)
    - Draws the player
- drawExit(canvas, app)
    - Draws the exit
- drawRobots(canvas, app)
    - Draws the hunters
- drawButtons(canvas, app)
    - Draws all buttons

Other functions:

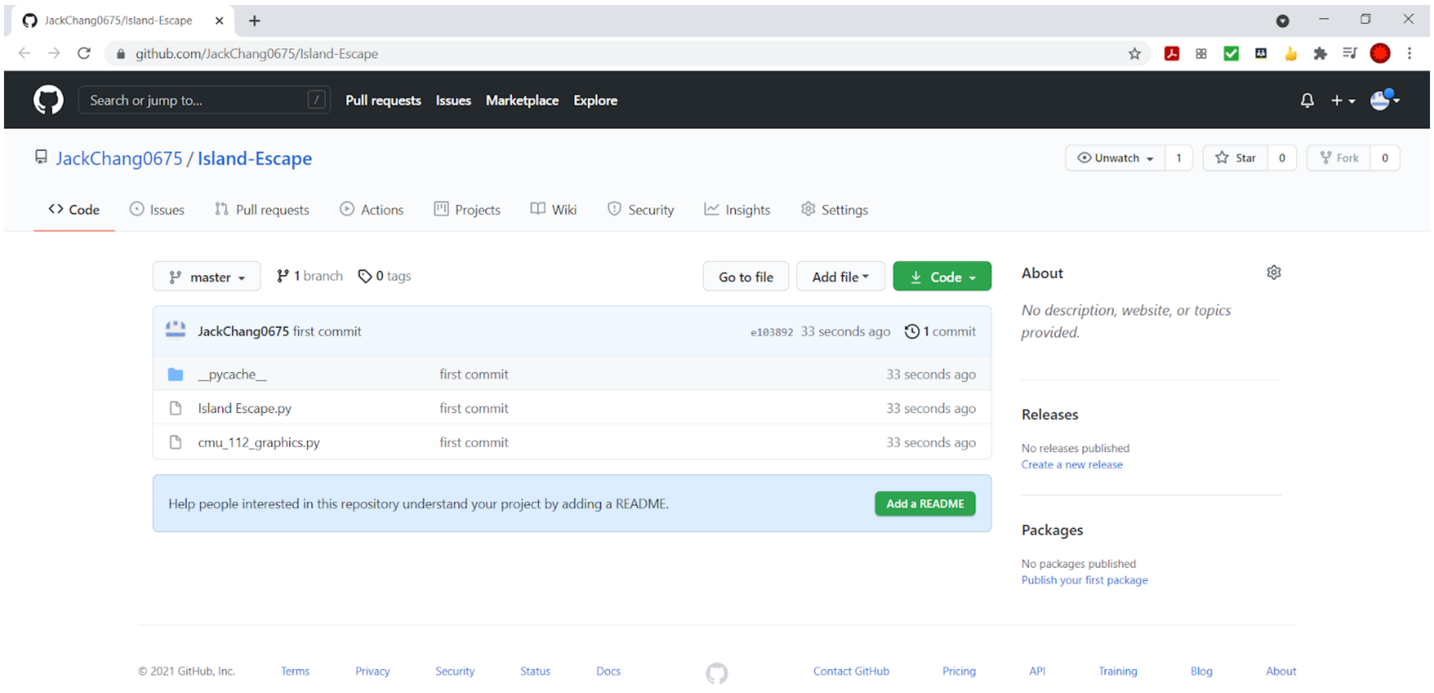
- switchScreen(app, screenName)
- getShortestPath(app, object1, object2)
- checkForButtonPress(app, clickX, clickY)
- insideButton(x0, y0, x1, y1, clickX, clickY)
- checkForButtonHover(app, mouseX, mouseY)
- isValidPos(app, row, col)
- getCellBounds(app, row, col)
- moveRobots(app)

Algorithmic Plan:

So far, it seems the trickiest part of the project is creating an algorithm for the robots to use to chase the player or prevent them from reaching the exit. To accomplish this, I will be using a breadth-first search (BFS) algorithm so that the robots can detect the shortest path between themselves and the player, and make one move on that path (one move closer to the player) on their moves. I will write a function, getShortestDistance(app, object1, object2), that gives me a tuple in the form of (distance, path) where distance is the shortest distance between object1 and object2, and path is a list representing all the points on that shortest path (each in the form of a tuple (row, col)).

Backup Plan:

I will be periodically uploading my code to Github (at least once a day) to back up my code. Below is a screenshot of my first push to Github.



JackChang0675/Island-Escape

github.com/JackChang0675/Island-Escape

JackChang0675 / Island-Escape

JackChang0675 first commit		e103892 33 seconds ago 1 commit
__pycache__	first commit	33 seconds ago
Island Escape.py	first commit	33 seconds ago
cmu_112_graphics.py	first commit	33 seconds ago

Help people interested in this repository understand your project by adding a README.

About

No description, website, or topics provided.

Releases

No releases published

Packages

No packages published

© 2021 GitHub, Inc.

<u>TP2 Update (only things added):</u>

<u>Things to be stored in the app object and what data type they are (all initialized at the start):</u>

- buttonsActivated → list of GridObjects
- itemsOnGround → list of GridObjects
- pickups → list of strings
- loseLevel (stores the level that was just lost so it's easy to go back) → string
- winLevel (stores the level that was just won so it's easy to go back) → string

<u>Other functions:</u>

- loadGameOver(app)
- checkForWin(app)
- loadWin(app)
- generateLevel(app)
- isValidGrid(app, row, col)
- all the functions that initialize the buttons for each screen and the levels themselves

<u>Algorithmic Plan v2:</u>

The trickiest part of the project this time around was generating a random level that is possible to win. I originally tried to do so using Kruskal's algorithm, but it proved to be difficult, since all the online examples I found had nodes and edges as two separate things, but for my project they are the same thing (walls are a node as well, not something in between nodes). I found the same problem when trying to use Prim's algorithm, so I ended up not using either algorithm to generate my maze. I'm sure there's some way to do it using those, but I couldn't find it/figure it out.

Trying to figure out a way to do it myself, I first tried using a Dijkstra + random weights to generate a random guaranteed path from the player's starting position to the exit, then add random walls around that. Basically, I used a dictionary to add random weights between each of the tiles (so moving from a tile to another isn't always the same cost of 1), then ran a Dijkstra on it to get the shortest path based on those weights. After completing this, I found a few problems with it. First of all, the path was almost always skewed towards the middle (meaning it always passed through somewhere close to the center of the board), though there were the occasional paths that actually looped around a bit and didn't do that. Second of all, it was possible to completely cut off parts of the grid since the obstacles were randomly generated. Even though it is guaranteed that the player can make it to the exit, other parts of the board could be completely cut off by obstacles, which I didn't like.

I then tried another method: looping through each spot on the grid, then using rng to decide whether or not to add an obstacle at that tile, but then only adding that obstacle if it still keeps all the empty squares connected. To do this, created a helper function called isValidGrid that ran a BFS from every node to check if all the empty squares are all part of one connected component. If so, then add that obstacle, but if not, then don't add it, and continue. This gave me completely randomly generated obstacles, and it was guaranteed that all squares were reachable from all other squares.

After generating this, I came across yet another problem: in my randomly generated level, I place the robot in a random place, but it is possible that the robot would directly block the only path between the player and the exit, so that would make the level impossible since the robot would always catch the player. To counter this, I added guaranteed isolated obstacles, which are obstacles that don't have any other obstacles adjacent to them. This solves the problem because the player can always circle around these and juke out the robot if needed, so that guarantees that the player can always beat the level.

TP3 Update (only things added):

Classes (OOP) and their functions:

- Sound(path, channel)
- Mine(self, row, col, color) → extends GridObject
- BomberRobot(self, row, col, interval) → extends GridObject
- Item(self, row, col, name) → extends GridObject
- GridButton(self, app, row, col, wallRow, wallCol, color) → extends GridObject
- Teleporter(self, x0, y0, x1, y1, color)

Things to be stored in the app object and what data type they are (all initialized at the start):

- Sprites/Images: robotImage, robotSprite, bomberRobotImage, bomberRobotSprite, playerImage, playerSprite, shieldImage, shieldSprite
- Music/SFX: curSong, clickSound, explosionSound, deathSound, marioCoinSound, shieldBreakSound, powerUpSound, shieldHitSound, gridButtonPressedSound
- music → boolean
- sfx → boolean
- gridButtons → list of GridButtons
- bomberRobots → list of BomberRobots
- teleporters → list of Teleporters
- justTeleported → boolean
- mines → list of Mines
- shieldTicks → int
- shieldColor → string
- konamiSequence → list
- konamiIndex → int
- konamiUnlocked → boolean

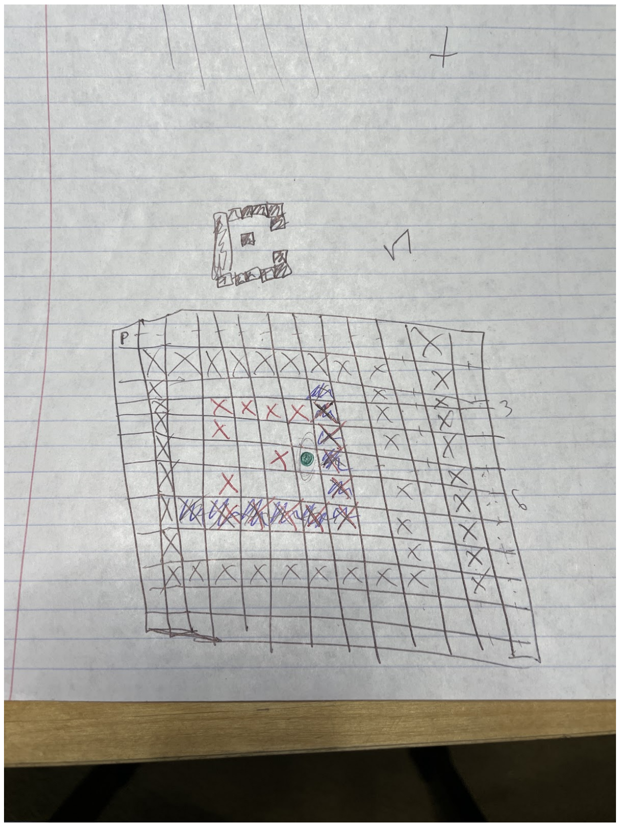
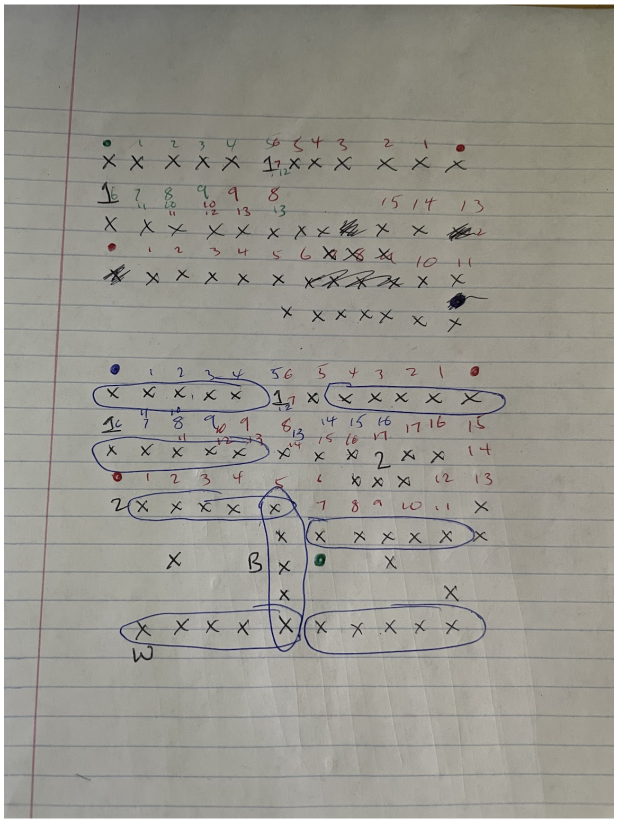
Other functions:

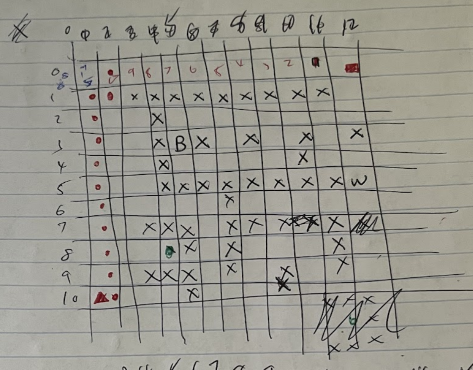
- checkGridButtonPresses(app)
- checkTeleportation(app)
- checkMines(app)
- checkItems(app)

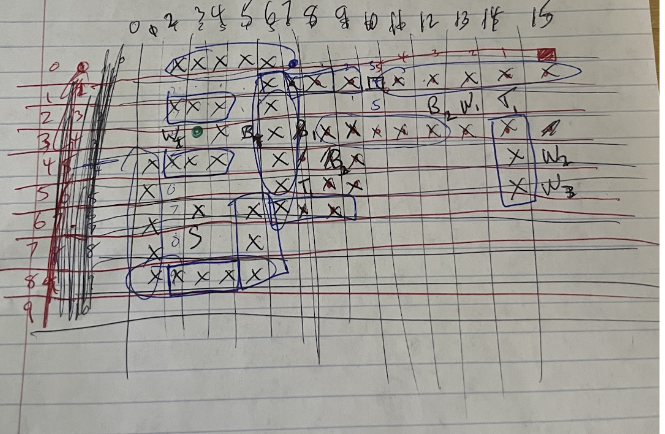
- scaleRobotImage(app)
- scaleBomberRobotImage(app)
- scalePlayerImage(app)
- scaleShieldImage(app)
- animateTeleporters(app)
- animateMines(app)
- animateShield(app)
- toggleMusic(app)
- toggleSFX(app)
- unlockAllLevels(app)

<u>Algorithmic Plan v3:</u>

A lot of stuff and special features were added here. Not going to get into detail because there isn't much of an actual "algorithm" behind these. I did spend a lot of time designing the levels as well (at least the later ones); they took like over half an hour each just design and test. Below are images from my design process.


B
W


W4
B4
B1
B2
W1
T1
B3
T
S
W2
W3

Citations/Credits:

All code was written by me.

Sprites/Art:

Player Sprite: http://www.clker.com/cliparts/e/i/4/q/8/a/man.svg

Robot Sprite: https://image.pngaaa.com/607/1244607-middle.png

Bomber Robot Sprite:
https://www.clipartmax.com/png/small/0-9222_download-robots-clipart.png

Shield Sprite:
https://www.clipartmax.com/png/middle/353-3534160_winged-shield-clip-art-shield-clipart-free.png

Music/Songs:

Menu Screens Music: “Homestuck” by Mark Hadley
Level 1 Music: “Explore” by Homestuck
Level 2 Music: “Showtime” by Homestuck
Level 3 Music: “Harleboss” by Mark Hadley
Level 4 Music: “Bronze Rebel” by Yan Rodriguez
Level 5 Music: “CH3CK TH3 M3T4D4T4” by James Roach
Level 6 Music: “MeGaLoVania” by Toby Fox and Joren de Bruin