A Level Project

Name: Tony Lu

Centre: Dulwich College

Candidate Number:

Contents

[Analysis 3](#_Toc106828798)

[Introduction 3](#_Toc106828799)

[Why is the project needed? 3](#_Toc106828800)

[Computational Suitability 3](#_Toc106828801)

[Research 3](#_Toc106828802)

[Stakeholders 4](#_Toc106828803)

[Hardware Requirements 4](#_Toc106828804)

[Software Requirements 4](#_Toc106828805)

[Limitations 7](#_Toc106828806)

[Success Criteria 7](#_Toc106828807)

[Usability 7](#_Toc106828808)

[Design 8](#_Toc106828809)

[Game loop 8](#_Toc106828810)

[Interface 8](#_Toc106828811)

[Variables and Data Structures 8](#_Toc106828812)

[Functionality 10](#_Toc106828813)

[Test Plan 23](#_Toc106828814)

[Implementation 24](#_Toc106828815)

[Iteration 1 24](#_Toc106828816)

[Requirements being developed 24](#_Toc106828817)

[Errors 24](#_Toc106828818)

[Conclusion 24](#_Toc106828819)

[Iteration 2 24](#_Toc106828820)

[Requirements being developed: Generating random matrix and A star pathfinding algorithm inside the matrix 24](#_Toc106828821)

[Errors 24](#_Toc106828822)

[Conclusion 24](#_Toc106828823)

[Iteration 3 24](#_Toc106828824)

[Requirements being developed 24](#_Toc106828825)

[Errors 24](#_Toc106828826)

[Conclusion 24](#_Toc106828827)

[Testing 25](#_Toc106828828)

[Evaluation 26](#_Toc106828829)

# Analysis

## Introduction

What is your game about?

Pathfinding in random generated matrix map, the goal is to find the cheapest way to construct a road from one point to another.

## Why is the project needed?

I have made a few simple games a while ago, and I have a few friends (about five) that wish to play a game made by me. So I will be making this game to entertain them and will interview them for their ideas and suggestions.

There is not a game similar to this that exist in the market. Which is a very simple maths type game that stretches people’s computational thinking skills. Also, nowadays most games have a microtransactions element to them or are too complex, so I would like to fill in the gap in the market, creating a simple at the same time fun game for the players.

## Computational Suitability

Why is suited to being played on a computer?

The controls are simple and can be done simply by mouse, also some save files will be created and to be stored in the computer’s hard drive.

Also the situation in a real life situation as building roads and other infrastructures can be quite costing and takes quite a long time, for example years and months. A normal person wouldn’t be able to participate in such an activity in real life.

As for the part for the solution by a computer, the problem of finding an optimal path in a matrix is non-tractable, as there are too many ways a path can be built, but a heuristic can be used to reduce the complexity of the algorithm, thus making it suitable for computational methods.

## Research

What type of games have you looked at and drawn inspiration from e.g. Pacman, platform games, side scrolling games etc.

There isn’t a game similar to the game being created but the genre of the game is resource management or logic. I have looked at strategy games such as minesweeper and dwarf fortress.

Minesweeper is a logic game, which has a very simple user interface, which involves clicking on map tiles to discover whether it is an empty tile, or a bomb, the objective being avoiding the bomb while the empty tiles hint how many bombs are around it. Nor the graphics or action make the game interesting (in contrary to most modern video games), making it a great example for an arcade / simple game.

In game screenshot:



Dwarf fortress is developed in 2002, which is a construction and civilization simulation game with a pixel style with texted-based graphics. For every new save, a new fantasy world map is generated, for each tile of the map, generated values for geology, hydrogeology and biogeography combine to compute information of the tile, thus the actions available for such tile and the interaction of the civilization with the tile.

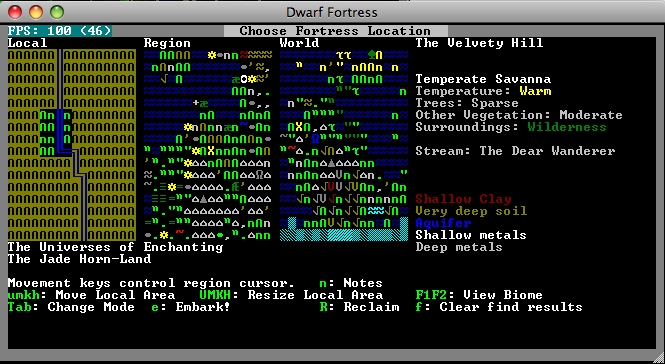
The process involves procedurally generated basic elements like elevation, rainfall, mineral distribution, drainage and temperature. For example, a high-rainfall and low-drainage area would make a swamp.

Also there are many other features such as combat and random event, but they are not of interest of this project, as this project’s purpose is just a road construction simulator game.

In game screenshot:

Dwarf fortress map section


There is a road building feature in dwarf fortress which allows a fast arrival of external traffic, this has inspired the creation of this project. This means that the essential features of the project would be to have an algorithm to procedurally generate terrain and to have construction functions. Also, the game would have a similar UI design to that of dwarf fortress.



The UI design of Dwarf Fortress

## Stakeholders

Who would be interested in playing your game?

Mostly people who enjoy a relaxing game that involves calculations, there are not many maths games in the market, there are not any games quite similar to this. Some wished to have online multiplayer and save files so these requirements are added. People who enjoy games such as minesweeper and sudoku would enjoy this simple game.

My friends have found the concepts to be quite interesting and would be willing to give it a try once it comes out. People who only have access to computers with low processing powers would be interested in playing the game as well.

Who can help you design the game?

My project mentor, Dr. Storey who can help me with the technical aspects of the project, my classmates which can peer review my project, give me advice on design and finally my other friends which can also give me suggestions on how to improve the design but they have less technical knowledge on game developing and programming.

Internet resources can also be quite helpful as I would be able to find similar examples for the designing of the game.

# Hardware Requirements

Program is probably able to run on 2 GB of RAM and a basic processor (1 GHz). There isn’t much complex UIs or too many sprites being loaded in the game, so the graphics requirement is set very low.

A mouse is needed and a keyboard is not necessary. As most of the methods will only require clicking on tiles or buttons, similar to mine sweeper.

A screen with a low resolution such as 1280 x 720 is needed, to display the full pygame surface for the gameplay.

A disk space of around 10 MB is needed to store the program and save files, the save files will be stored as json or text file, which means that they would not require too much space, unless many games are played.

# Software Requirements

Python installed in the computer, because the program is written in Python language. When the program is run in VS code, pygame should be installed by “pip install pygame”.

VS code is chosen because the IDE is quite easy to use and has a visualization feature to see how the game runs making it suitable for game testing and developing. Developing in VS code is quite convenient as it has quite a smart autocomplete and syntaxing function, including the syntax of new modules that have been imported such as pygame. Also it highlights sections of code according to their function and most importantly whether the loop they are in is ran or not (in VS code code which is not accessed will be highlighted using a grey-blue colour) helping with visualizing the code while debugging. Finally, as the project is expected to contain at least 1000 lines of code, finding a specific section of code for debugging or modifying can be quite difficult using other IDEs, but with VS code, it allows searching in the file by ctrl + f, allowing quick navigation across the file finding specific sections of code.

The pygame library is used, it is a simple library designed for game programming, allowing user interactions with the program. Python as a object oriented programming language is needed as classes and objects need to be implemented, for example nodes of my A star algorithm and the display of tiles and buttons on the screen. A procedural programming approach will struggle to implement this game, as its code will not be reusable for the game elements, each button and each other sprite will have to be programmed from the very basics, as libraries would not be available for a procedural language, over-complicating the project massively.

The program would work on all operating systems such as linux, windows and mac as long as python 3 is installed. Python is a compiled high level language, so it is independent to the operating system.

The program must be loaded within 5 seconds, because it would be uninteresting to have the users wait for too long.

# Features

|  |  |  |
| --- | --- | --- |
| Index | Description | Justification |
|  | Has a black screen of 1280 x 720 | The game has many user interface elements, this screen size is required. It is quite a standard size for a game, which wouldn’t require a screen with higher than average resolution, suitable for the low-spec computers. |
|  | A minimalist user interface and clean, pixel art style buttons | The game’s content matches the style of simple UI and as the screen size is limited (1) and pygame library is used, simple buttons are more suitable to reduce the requirement of processing power. |
|  | A “map” needs to be generated randomly each time the program is booted up. | The game should be interesting for the player, having a different map loaded up every time the game is booted up, which should be an essential part of the game. |
|  | Function which can generate an all-zero square matrix. | The matrix generated will be used to store the outputs of the terrain-generation algorithm, this is to support the implementation of (3). |
|  | Three 2D arrays are needed to store the values of the different attributes of a tile | The cost of traversal of each tile and the colour of each tile are computed using multiple attributes, thus 2D arrays will have to be used, an array of tuples isn’t a good option because it would be harder to implement. The arrays will allow a similar implementation to that of drawf fortress, where each tile has a few different attributes. |
|  | Having an algorithm to calculate the traversal price of each tile. | This is essential for the game to function and this is the cost which is used by the user and the pathfinding algorithm, it has been suggested by my peers that Dikjstra’s or A star algorithm should be used. |
|  | Having an algorithm to compute the colour of each tile according to the matrices in requirement 4 | Using the arrays completed in (5). This allows the game to have more interesting visuals for the tile map and allows for a map created similar to that of dwarf fortress. |
|  | A pathfinding algorithm | The algorithm is used for finding a good path for the matrix, thus allowing calculating the score. |
|  | Method to construct roads which update the UI | This is the main gameplay element, it should also update the cost of the path as player constructs roads. |
|  | Method to deconstruct roads which update the UI | This is to reverse the tiles selected by method in requirement 10, this allows the path to be corrected after the user decides to change, it should update the UI and update the cost of path by the player. |
|  | Have a timer for one game | This is to help the players track the amount of time used in one game, it also is used to calculate the score. |
|  | Have a submit button | This button is used to submit the path created by the player, which will check for validity, if the path is tangled or connects. |
|  | Have a score calculation algorithm | This will calculate the score that the player has achieved after the button in requirement 13 is clicked on. |
|  | A high score table for the users (offline) | This allows the players to see their previous high scores, making the game more interesting and competitive, which can be displayed on the playing screen. |
|  | Custom fonts and simplistic user interface | As can be concluded from previous requirements, so the game is not too taxin on the computer. |
|  | There should be a menu | The menu allows enter play, scoreboard and instructions. |
|  | There should be a written set of instructions | The game can be quite disorienting to start, so an instruction screen can be quite helpful for new players. |
|  | There could be a feature to save a score | The score can be stored to be later put in a ranking in the save file. |
|  | There could be a feature to encrypt the save file | The save file would be visible to the player, to not allow unwanted modification, the file should be encrypted when saved, and then decrypted when the file needs to be loaded. |
|  | There might be a multiplayer gamemode | By using the python sockets, it is possible to implement competition between two players online |
|  | There could be a pathfinding visualisation option for the matrix | This allows the player to understand the process of the algorithm finding a way through the matrix. |
|  | There could be sound effects | Sound effects make the game more interesting and more attractive to players. Satisfying sound can be played when a button is clicked to encourage more gameplay. |
|  | Won’t have online accounts | There isn’t a dedicated server for the game nor a database, so there couldn’t bbe any online accounts as they need to be stored in a database. |
|  | Won’t have 3D graphics | The game is designed for all computers including the very low spec ones, the 3D graphics contradict with this requirement, making the graphics requirement much higher as more objects need to be rendered for a 3D display. |
|  | Won’t have complex UI | The game should be quite straightforward and easily explained in the instructions, so the UI shouldn’t be too complex, which can be quite confusing to some players. |
|  | Won’t have music | There won’t be any music because the programmer doesn’t know about creating video game music and the copyright, designs and patents acts must be followed, so it is preferable that the game doesn’t have any music. |
|  | Won’t have a loading screen | Although most games have loading screens for the start, there is hardly a need for that in this game, as the scenes are supposed to load up quite fast. |
|  | There could be a feature to have custom player names | This can be used to distinguish between the players on the same device, which is suggested by my friends. |
|  | There could be a feature to see more rankings than that displayed in the playing screen | This is suggested by my peers as they are interested in seeing the score history to compete with each other. |
|  | There could be a feature to modify the difficulty of the game | This can be used to make the game more interesting for the players once they have familiarised with the algorithm and the game (suggested by my friend in the interview). |
|  | There could be a loading screen/ transition screen | This can make the game smoother making the user experience more enjoyable, this is inspired by other simple games on the market. |
|  | There could be a difficulty selection | This can make the game more interesting, suggested by Dr. Storey. |
|  | There could be some ASCII type graphics | Allows a more aesthetically pleasing interface, which is inspired by Dwarf Fortress. |

# Limitations

No dedicated server to host the game, which would have allowed features such as online accounts. Also the game is to be ran on basically all computers including the low-spec ones, so there won’t be 3D graphics and neither there will be complex UI. Visual studio code seems to an error with playing audios using mixer, which will cause the game to not have any audios as wished.

# Success Criteria

1. Has a screen size of 1280 x 720, because the game has quite a lot of buttons and UI elements to show, so this size is quite reasonable
2. Displays a matrix of colors which is randomly generated each time the game boots up, this act as the map for the player
3. Game has a menu with play button, ranking button and instructions, this allows the player to understand what to do with the game and allow them to access their save files
4. Game allows the selection and deselections using mouse, this is the operation of which gameplay consists of
5. Game displays the best cost of the route calculated by a pathfinding algorithm and the current cost of the route which player has picked.
6. While playing, a short list of ranking is displayed
7. A submit button to get the player’s score, this is the way to end the current game

# Usability

To see if the software works, the output of the current cost can be checked manually using a calculator, also the path of the optimum cost should be output in the console during the testing phase, this allows the checking of sum of cost and see if the path is connected. The score calculation algorithm when programmed should also be checked manually on at least 3 runs.

All the buttons in all menus should be interactable by using left click with the player’s mouse hovering over them, which all of them must be checked when they are instantiated.

For beta testing, as the game is quite simple, the instructions can be explained clearly by text. So the text screen will be displayed if a button in the menu is clicked. The instructions are as follows:

After clicking start, the timer will start. Within the least time possible,

attempt to find the least cost path from the top left square to the right bottom square.

The darker the square, the more coins are needed to build a road on the square.

Do not attempt to build backwards (to the left or up directions), as the path needs to be 17 tiles long.

Attempt to create a path with lower price or equal to that of the A star algorithm, which rewards more points.

Click submit to check your score, the path needs to be 17 tiles long,

continuous and contains top left and bottom right tiles.

Close the window after finishing, the score is saved.

This is written on the “instructions.txt file” inside the game folder.

# Design

## Set up

The game generates three matrices using the procedural generation algorithm.

The three matrices are passed through a algorithm which computes the cost of traversal through each tile and returns as another matrix (denoted costMatrix or priceMatrix).

All the gameTiles should be assorted into a group and assigned their x and y coordinates and labelled using a tuple for future references.

All the variables that will determine which loop is running should be defined.

Classes and functions defined.

## Game loop

It is the loop that should be developed first, as this is the most important section of the project.

The game loop draws a 1280 x 720 screen with 60 FPS.

It should draw the black background first.

Then the gameObjects will be drawn on the screen.

The texts are blit onto the screen, also the texts that are to be displayed over the button rectangles.

Then the texts are updated if there is a change (for example the costs).

The inputs of the players are taken being click and mouse position

Game is set on 60 fps.

## Menu loop

The buttons and title should be displayed on screen.

The menu loop stops when the play button is clicked, which will then skip to the game loop.

Another internal loop to display the instructions should initiate if button for checking instruction is clicked.

Another internal loop to display the ranking should initiate if button for checking ranking is clicked.

Game is set on 60 fps

# Interface

## Menu

### First iteration of the menu

Diagram

Description automatically generated

The screen contain all necessary elements but doesn’t have a colour palette yet.

#### Feedback 1

Rating of the menu screen?

7/10

Which color scheme?

Same as the game screen.

What to improve?

The three buttons should be placed in a line, which makes instructions and highscores buttons more catchy for the player and will be more aesthetically pleasing.

### Second iteration of the menu

Graphical user interface, table

Description automatically generated

This way the menu screen is more compact and more similar to those of modern games such as Minecraft. The background will be black (0,0,0), the buttons will have a pink-purple colour and the texts will have a pixel-styled font and a white colour.

When the play button is clicked, the game screen opens up.

## First iteration of the game user interface



### Feedback

Rating of the menu screen?

4/10

Which color scheme?

Pink/purple, and dark themed.

|  |  |  |
| --- | --- | --- |
|  |  |  |

What to improve?

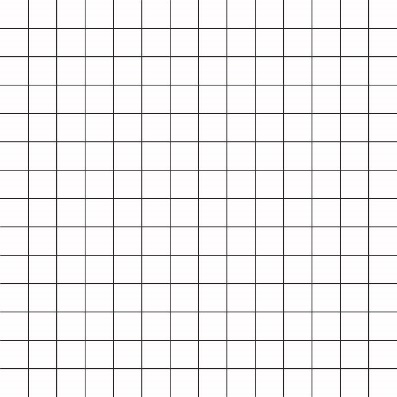
There is no need for visualization of the algorithm, as most players won’t have the technical knowledge of the pathfinding algorithm; there is also no need for a player name, as the online multiplayer feature will not be developed; the volume control won’t be needed due to the limitations of visual studio code; some space around the map can be fitted to avoid cramping and chaotic visuals during gameplay.

## Second iteration of the game user interface

Text

Description automatically generated with low confidence

The elements get more spacing, allowing a less cramped UI design. And the essential features are decided according to feedback. The map would be a square filled with tiles of different colours which represent their cost, something like the image below (but with coloured tiles):



## Variables and Data Structures

List all classes and methods clearly.

Library Generator methods: generateInitBiome(), diamondSquare()

Class gameTile()

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Reason** |
| color | Tuple | This parameter determines the color of the tile shown on screen |
| width | Int | This parameter determines the width of the square tile (how big the square tile is shown on screen) |
| x | Int | This parameter determines the x coordinate of the square tile |
| y | Int | This parameter determines the y coordinate of the square tile |
| a | Int | This parameter is the identifier for a tile alongside b in the group myTile |
| b | Int | This parameters is the identifier for a tile alongside a in the group myTile, a tile can be referred to as group[b][a] |
| cost | Int | It is the cost of traversal of this gameTile, used for calculating the current cost of the path created by the player. |
| selected | Bool | This parameter indicates the state of the square tile of if it was selected by the player, it is set to false for default value. |

|  |  |  |
| --- | --- | --- |
| **Method** | **Attributes** | **Reason** |
| updateWhenSelected | / | This function is applied when a tile is clicked on by the user, if the gameTile was not selected, it is turned into another colour, and the function returns selected.  If the gameTile was already selected, the gameTile’s colour is recovered to original colour and the function returns deselected, it will change to a specific colour if selected and will return to its original colour if deselected. |

Class tile(): class used for the A star pathfinding algorithm.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Reason** |
| parent | tile object | This is the tile to be traversed previous to the tile in the algorithm. |
| position | Tuple | This is the position of the tile inside the matrix, the position allows the determination of neighbouring tile objects. |
| g | Int | The traversal cost of the tile object. This is quoted from the cost calculation algorithm. |
| h | Int | The heuristic cost of the tile object to reach the end. This is calculated using the mean times the straight line Pythagorean distance from the tile to the destination. |
| f | int | The sum of g plus h, value used for the A star algorithm to decide which tile to traverse. |

Class button(): button objects used for the user interface for the user to input, which runs a function when clicked, it will inherit the pygame.sprite class, it will display different colours depending on its stage (normal, hovered and clicked).

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Reason** |
| x | int | This is the on screen x coordinate of the button sprite that will be drawn. |
| y | int | This is the on screen y coordinate of the button sprite that will be drawn. |
| width | int | This is the x length of the button on screen. |
| height | int | This is the y length of the button on screen. |
| buttonText | str | This is the text that would be displayed on top of the rectangle of the button. |

|  |  |  |
| --- | --- | --- |
| Method | Attribute | Reason |
| Process() | / | This will run during the course of which the button is drawn on screen, which will constantly check for a need of a colour change, and change the colour of the button whenever it is needed. |

Other methods (outside of class)

|  |  |  |
| --- | --- | --- |
| Method | Attribute | Reason |
| terrainGeneration() | To be decided | This method will fill in a matrix with patterned numbers similar to the height of a terrain. This is the function responsible for creating the map and for the varied colours of the tiles. |
| returnPath() | current\_node | This method will return the reversed path in the A star algorithm. |
| returnMean() | costMatrix | This method will calculate the mean cost of the whole matrix, as an element to calculate the heuristic of the A star algorithm. |
| Astar() | costMatrix, start, end | This method should return a solution which is a path connecting start tile and end tile in the costMatrix, this path according to the method should be quite fast but maybe not the lowest cost solution as a heuristic is used. |
| CheckContinuity() | myList | This is used to check if the path submitted by the player is continuous. |
| CalculateScore() | Seconds, nowCost, desiredCost | This will return a score depending on the three attributes giving the player the score they have achieved. |
| Mainloop() | / | This is the main game loop which will be ran about 60 times a second (depending on the setting) |

## Functionality

Describe the game loop and any global variables or constants

For each method:

* Explain what it does and how which requirement it fulfils
* Write out pseudo code
* Draw a flowchart

Algorithms:

### Terrain generation algorithm

Cited from: <https://chschulte.github.io/teaching/theses/TRITA-ICT-EX-2015:72.pdf>

There are many terrain generation algorithms, there are three viable options:

Diamond-square algorithm, basically it deforms large line segments into smaller ones, making the terrain appear more natural. A heightmap is input into the algorithm, which has a few line segments to be broken by the algorithm. The major feature of this algorithm is the ability to generate terrain which can be merged to create a bigger terrain (which is not needed for this project), the algorithm’s major advantage is that it saves memory and has a constant space complexity.

Perlin Noise, The principle of this function is by combining different wave functions with different periods and amplitudes. There are two components to this algorithm, a noise function and an interpolation function. The noise function maps an input to an output, where the output has no correlation to th output. Then the interpolation function smoothes out the outputs. Its advantage is that it is quite fast and only takes three parameters. However the terrain generated would have similar features across the board.

Diagram

Description automatically generated with medium confidence

For a better varied tile map, diamond-square algorithm is chosen for this project. Also diamond-square is easier to implement and less complicated to program.

The terrain generation (diamond square).

The algorithm is explained using this diagram:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

The four corner values (grey) are generated randomly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Diamond step: The value in the middle of the four values (yellow) is the mean of the four corners + a random number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Square step: the red squares take value from the average of its diamond, for example, the top red square will take the average of its left grey square, right grey square and the yellow square.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Repeat again the diamond step on the blue squares.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Repeat again the square step on the green squares. The algorithm is complete

In the generator.py file (which we can treat it as a library), there are two methods. Both are required to fulfill the need for a generating a random map every time the game runs.

The diamond square algorithm is used.

The flowchart implementation:

Diagram

Description automatically generated

The first method, generateInitBiome() takes in a matrix and generates four random numbers for four corners.

procedure generateInitBiome(mt, width, rangeOfValue)

#mt is the matrix that needs to be modified

#width is the width of the square matrix

#rangeOfValue is the intended range of the randomly generated values

mt[0][0] = randomInt(0, rangeOfValue)

mt[0][width-1] = randomInt (0, rangeOfValue)

mt[width-1][0] = randomInt (0, rangeOfValue)

mt[width-1][width-1] = randomInt (0, rangeOfValue)

endprocedure

And the second method, diamondSquare() takes the matrix, firstly takes its mid point and set its value to the mean of four corners plus a randomly generated integer.

Then

procedure diamondSquare(mt, width, rangeOfValue, distance, step\_size)

while step\_size > 1

for i in range(width)

for j in range(width)

mt[i][j] = average of the square corners which are step\_size apart + random number

next j

next i

for i in range(width)

for j in range(width)

mt[i][j] = average of the diamond corners which are step\_size width + random number

next j

next i

step\_size = step\_size/2

endwhile

endprocedure

This pseudocode implementation seems to be very simple, however, in pseudocode it doesn’t account the iteration errors that can be produced at the corner/side values in both steps, so in python implementation multiple nested loops which account for the errors would have to be used. Also the algorithm has to operate on a square matrix which have the side length of 2^n+1.

### Pathfinding algorithm

Pathfinding algorithm is a very important component to the game, as it provides a computer-generated solution to building the path in the matrix, making it much more interesting for the player and also provides a basis for the calculation of score.

For the pathfinding algorithm to calculate the least-cost route from one point to another, we will use A\* algorithm. It is one of the most complete pathfinding algorithms and also one of the most used in computer science. Unless the graph is pre-processed or if the memory is limited, A\* is often the best solution to the problems. Its heuristic will be calculated as:

Average of all the tiles inside the rectangle which the starting and final tile form, times their diagonal distance.

Diagram

Description automatically generated

The algorithm approximately looks like this:

class tile()

private parent

private position

procedure new(myParent, myPosition)

g = 0

h = 0

f = 0

endprocedure

endclass

function aStar(costMatrix, start, end)

startTile = tile(“None”, start)

endTile = tile(“None”, end)

oList = []

cList = []

oList.append(startTile)

while len(oList) > 0

currentTile = oList[0]

currentIndex = 0

for (t,i) in oList

currentTile = t

currentIndex = i

endfor

oList.pop(currentIndex)

cList.append(currentTile)

if currentTile = endTile then

path = []

current = currentTile

while current != “None”

path.append(current.position)

current = current.parent

endwhile

return reversed path #this will be implemented

endif

children = []

for new\_position in adjacent tiles #this will have to be implemented as a list

tile\_position = one adjacent tile #implemented as a tuple

new\_tile = tile(currentTile, tile\_position)

children.append(new\_tile)

endfor

for child in children

for closed in cList

if child = closed

pass

endif

next closed

child.g = currentTile.g + child.cost

child.h = heuristic #this will be implemented as a method

child.f = child.g + child.h

for open in oList

if child = open

pass

endif

next open

next child

endwhile

## Test Plan

Fill in the table below with at least 30 tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Description** | **Input** | **Expected Outcome** |
|  | An all zero square matrix is generated | Run the code | If a print request is done, the all zero square matrix should be printed with desire width |
|  | The all zero square matrix should have its corners with value filled in using random numbers within a specified range. | Run the code | If a print request is done, a square matrix should be output with the corner values filled in and within range |
|  | The square matrix should have its values filled in using the diamond-square algorithm, forming a patterned square matrix with values | Run the code | If a print request is done, the square matrix should be output with the desired pattern |
|  | Screen appears and is 1280 by 720 pixels | Run the code | Screen appears on the display |
|  | Three matrices are generated using the terrain generation algorithm and printed | Run the code | Three matrices should be output consecutively for each time the program is ran |
|  | Each element of each matrix will go through a function which will merge all three matrices into one with each element the product from function of the elements at the same position of the three matrices. This will be checked manually after it being printed | Run the code | One matrix should be output, being the priceMatrix. |
|  | The A star algorithm is ran on the priceMatrix and prints out an optimal path for the priceMatrix everytime the program is ran | Run the code | A list of tuples (coordinates of tiles) should be output in the terminal everytime the program is ran, each of the elements should uniquely refer to a tile coordinate and the list should have a length of (width\*2-1) |
|  | A black screen of 1280 x 720 | Run the code | A black screen of 1280 x 720 should be opened on the monitor as the programme is running |
|  | The tiles are drawn in position relative to that of their positions in the priceMatrix and their colour should be a result of a function which takes input from the previous three matrices | Run the code | On the black screen such a matrix should be drawn with various colours |
|  | When clicked on one tile, the tile should turn another colour | User clicking on tiles | The tile should turn a distinct colour as they are clicked on, such as black or orange which should be set globally. This means that the tile is selected for pathing. |
|  | When tile which is modified by test 10 is clicked again, the tile should return to its original colour | User clicking on tiles which are affected by test 10 | The tile should return to its original colour, this means that the tile is deselected from the path list. |
|  | The submit button when clicked, the display score screen should be displayed | User clicking on the submit button in the game screen after building a path | As the player has built a continuous and none revolving path from top left corner to the bottom right corner and click on the submit button, a score should be calculated and displayed. |
|  | When starting the program, a menu should be displayed | Run the code | A menu with title and other programmed buttons should be displayed right after the programme is ran. |
|  | Clicking on the instructions or the ranking buttons | Clicking the buttons in the menu | For example, when the show instructions button is clicked, the instructions menu should open in the same window with a return button allowing the user to return to the menu screen. Each button should start their corresponding screen. |
|  | When clicking on the return buttons on the bottom of the menus mentioned in 14 | Clicking on the return buttons that are on the bottom corner of the screens mentioned in 14 | The menu is closed and the main menu is opened (in implementation the main menu is drawn over the said menus). |
|  |  |  |  |

# Implementation

## Iteration 1

### Requirements being developed

|  |  |
| --- | --- |
| A “map” needs to be generated randomly each time the program is booted up. | The game should be interesting for the player, having a different map loaded up every time the game is booted up. |
| Function which can generate an all-zero square matrix. | The matrix generated will be used to store the outputs of the terrain-generation algorithm. |
| Three 2D arrays are needed to store the values of the different attributes of a tile | The cost of traversal of each tile and the colour of each tile are computed using multiple attributes, thus 2D arrays will have to be used, an array of tuples isn’t a good option because it would be harder to implement. |
| Having an algorithm to calculate the traversal price of each tile. | This is essential for the game to function and this is the cost which is used by the user and the pathfinding algorithm. |

All of these are programmed in “generator.py” file, as it serves as a library for the terrain generation functions of the game, making the programme more elegant.

#### All-zero 2D arrays

So the all-zero square matrix will serve as basic for all of the upcoming processes, thus it will be coded first.

seaLevelMatrix = [[0 for x in range(mat\_width)] for y in range(mat\_width)]

This is a much more compact way to define such a matrix, instead of writing the 0s out.

When testing for the functionality:

print(seaLevelMatrix)

which prints out

A picture containing black, electronics, closeup

Description automatically generated

As desired, the matrix will then need four randomly generated values for corners. The representation of such four corners would be: [0][0], [0][width - 1], [width - 1][0], [width - 1][width - 1], as the matrix is a square matrix with a width of constant width.

#### Generator function for the 2D arrays

So, the function, generateInitBiome can be defined, which takes three parameters, mt, the matrix, width, the width of the matrix and rangeOfValue, range which the random number is taken from.

def generateInitBiome(mt, width, rangeOfValue):

    mt[0][0] = random.randint(0,rangeOfValue)

    mt[0][width-1] = random.randint(0,rangeOfValue)

    mt[width-1][0] = random.randint(0,rangeOfValue)

    mt[width-1][width-1] = random.randint(0,rangeOfValue)

This function is tested:

generateInitBiome(seaLevelMatrix, 9, 10)

outputs:

A picture containing black, electronics, closeup

Description automatically generated

This specific function will not require further modification.

Now the terrain generation algorithm needs to be programmed. Basically directly translating the pseudocode program which in python would be:

First, the random module will allow the creation of randomly generated number, which will be useful for

import random

Then the main body of the code will be:

def diamondSquare(mt, width, rangeOfValue, distance, step\_size):

    while step\_size > 1:

        #this is the diamond step

        for i in range(width):

            for j in range(width):

                r = random.randint(0,rangeOfValue)

                if i % distance == 0 and j % distance == 0 and i > 0 and j > 0 and i < width - 1 and j < width -1:

                  mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/4 + r)

        #this is the square step

        for i in range(width):

            for j in range(width):

                k = random.randint(0,rangeOfValue)

                mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/4 + k)

        step\_size = step\_size / 2

        distance = int(step\_size / 2)

Where distance is initially set as the int(step\_size/2), step\_size as the width – 1.

##### Error 1.1

When running the function on seaLevelMatrix, as expected, an error would be output as when the algorithm runs on the non-central values on the first iteration, because the function is trying to access a value which is out of bound of the matrix.



Now all the cases for list index out of range errors need to be included and accounted for in the operations, where I will use a series nested try and except, as using the specific ranges of tiles will be too complex and would not save more time for programming.

def diamondSquare(mt, width, rangeOfValue, distance, step\_size):

    #mt is the matrix that needs to be modified

    #width is the width of the square matrix

    #rangeOfValue is the intended range of the randomly generated values

    #distance should always be half of step\_size

    #step\_size helps with checking how many times the procedure has iterated and serves as a stopping point when it equals 1

    while step\_size > 1:

        #this is the diamond step

        for i in range(width):

            for j in range(width):

                r = random.randint(0,rangeOfValue)

                if i % distance == 0 and j % distance == 0 and i > 0 and j > 0 and i < width - 1 and j < width -1:

                    mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/4 + r)

                #code below lists all possible situations where a value does not have 4 corner values

                else:

                    try:

                        mt[i][j] = round((mt[i+distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/3 + r)

                    except IndexError:

                        try:

                            mt[i][j] = round((mt[i-distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/3 + r)

                        except IndexError:

                            try:

                                mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i+distance][j-distance])/3 + r)

                            except IndexError:

                                try:

                                    mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i-distance][j-distance])/3 + r)

                                except IndexError:

                                    try:

                                        mt[i][j] = round((mt[i-distance][j-distance] + mt[i+distance][j-distance])/2 + r)

                                    except IndexError:

                                        try:

                                            mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance])/2 + r)

                                        except IndexError:

                                            try:

                                                mt[i][j] = round((mt[i-distance][j+distance] + mt[i-distance][j-distance])/2 + r)

                                            except IndexError:

                                                try:

                                                    mt[i][j] = round((mt[i+distance][j+distance] + mt[i+distance][j-distance])/2 + r)

                                                except IndexError:

                                                    try:

                                                        mt[i][j] = round((mt[i+distance][j+distance] + r))

                                                    except IndexError:

                                                        try:

                                                            mt[i][j] = round((mt[i+distance][j-distance] + r))

                                                        except IndexError:

                                                            try:

                                                                mt[i][j] = round((mt[i-distance][j+distance] + r))

                                                            except IndexError:

                                                                try:

                                                                    mt[i][j] = round((mt[i-distance][j-distance] + r))

                                                                except IndexError:

                                                                    pass

        #this is the square step

        for i in range(width):

            for j in range(width):

                k = random.randint(0,rangeOfValue)

                try:

                    mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/4 + k)

                except IndexError:

                    try:

                        mt[i][j] = round((mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/3 + k)

                    except IndexError:

                        try:

                            mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/3 + k)

                        except IndexError:

                            try:

                                mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i+distance][j])/3 + k)

                            except IndexError:

                                try:

                                    mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j])/3+k)

                                except IndexError:

                                    try:

                                        mt[i][j] = round((mt[i][j+distance]+mt[i+distance][j])/2+k)

                                    except IndexError:

                                        try:

                                            mt[i][j] = round((mt[i][j-distance]+mt[i-distance][j])/2+k)

                                        except IndexError:

                                            try:

                                                mt[i][j] = round((mt[i][j+distance]+mt[i-distance][j])/2+k)

                                            except IndexError:

                                                try:

                                                    mt[i][j] = round((mt[i][j-distance]+mt[i+distance][j])/2+k)

                                                except:

                                                    pass

        step\_size = step\_size / 2

        distance = int(step\_size / 2)

#### Testing

|  |  |  |  |
| --- | --- | --- | --- |
|  | An all zero square matrix is generated | Run the code | If a print request is done, the all zero square matrix should be printed with desire width |
|  | The all zero square matrix should have its corners with value filled in using random numbers within a specified range. | Run the code | If a print request is done, a square matrix should be output with the corner values filled in and within range |
|  | The square matrix should have its values filled in using the diamond-square algorithm, forming a patterned square matrix with values | Run the code | If a print request is done, the square matrix should be output with the desired pattern |

And testing the function on seaLevelMatrix and print seaLevelMatrix should complete the requirement.

diamondSquare(seaLevelMatrix, 17, 10, 8, 16)

print(seaLevelMatrix)

Which outputs:

A screen shot of a computer

Description automatically generated with low confidence

This is good but the code can be modified to print out the results in a more presentable fashion, so instead of print, the nested for loops are used:

for x in seaLevelMatrix:

    for i in x:

        print(i, end = " ")

    print()

Which outputs:

A picture containing text, keyboard, electronics

Description automatically generated

This perfectly satisfies the requirement, concluding the end of the first iteration.

### Improvements to be done

The matrix generated might need to be modified in a certain way according to the cost calculation algorithm, which might include changing the rangeOfValue. The testing codes will have to be moved to the main.py file to keep the code elegant for the game file, the functions for setup can stay in the generator.py file.

Three of the procedurally generated matrices will need to be defined, which will be used to calculate the colour and the cost matrix for further operations.

## Iteration 2

### Requirements being developed

|  |  |
| --- | --- |
| A “map” needs to be generated randomly each time the program is booted up. | The game should be interesting for the player, having a different map loaded up every time the game is booted up. |
| Three 2D arrays are needed to store the values of the different attributes of a tile | The cost of traversal of each tile and the colour of each tile are computed using multiple attributes, thus 2D arrays will have to be used, an array of tuples isn’t a good option because it would be harder to implement. |
| Has a black screen of 1280 x 720 | The game has many user interface elements, this screen size is required. It is quite a standard size for a game, which wouldn’t require a screen with higher than average resolution, suitable for the low-spec computers. |

I have decided to implement the map feature by using the object-oriented approach with python. The map is constructed using separate tiles, which needs to be implemented one by one, the OOP approach is perfect for this matter.

All the programming is done in a separate python file named main.py, which is the file that needs to be run to play the game.

Pygame is imported then initiated, as it is the essential module for this game.

import pygame

pygame.init()

#### The gameTile Class

First, a class named gameTile is declared, this will serve as the blueprint for each tile of the map.

#defining class for a tile that is draw on screen

class gameTile(pygame.sprite.Sprite):

Quoting from variables and data structures, the gameTile object, the following attributes need to be declared for the gameTile class:

    def \_\_init\_\_(self, color, width, x, y, a, b, cost, selected=False):

I will make use of the sprite super class from the pygame library, as the tiles will need to be displayed on the screen and interact with the player.

    def \_\_init\_\_(self, color, width, x, y, a, b, cost):

        #a is the horizontal index and b is the vertical index

        super().\_\_init\_\_()

Now we set the sprite of the tile as a simple square with a variable color, which its position is at (x, y) of the screen.

        self.image = pygame.Surface([width,width])

        self.color = color

        self.image.fill(color)

        self.rect = self.image.get\_rect()

        self.rect.x = x

        self.rect.y = y

Now the identification attributes need to be declared, (a, b) will serve as a unique identifier for a tile, which will be used for later purposes such as the A star algorithm then the cost which will require an algorithm to calculate according the the matrices determined in iteration 1.

        self.a = a

        self.b = b

        self.cost = cost

#### Gameloop and background

Firstly, the variables screen size (as a tuple), the screen and the clock need to be defined for the program.

(win\_width, win\_height) = (1280, 720)

screen = pygame.display.set\_mode((win\_width, win\_height))

clock = pygame.time.Clock()

Now ,define the function which contains the loop

def mainloop():

The loop will be implemented as a while true loop which will be terminated by using the quit() function from the pygame library.

Inside the loop, the program needs to constantly check for the case which the user closes the program window, which the program needs to quit, mean while, the background of the game which is black (0, 0, 0) is filled over the whole window, where for 60 frames per second, the display window is updated for each frame.

def mainloop():

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

        screen.fill((0,0,0))

        pygame.display.flip()

        clock.tick(60)

Now that the basic loop is defined, we will test it by running the code after writing mainloop() at the end of the program, which means that the program will run the mainloop() function which we have just defined.

A picture containing application

Description automatically generated

A 1280 x 720 pygame window is opened which is filled in with black and this window can be closed using the cross button on the top right corner without outputting any errors, which concludes the success of the mainloop() function for now.

#### Matrices

Now we will work on the three matrices that will be used for the generation of the costs and colour values of the tile map. For their generation, the generator.py file’s functions will be needed. We will use the file as a library. So we eliminate all of the lines that were used for testing, which will leave the file as:

import random

def diamondSquare(mt, width, rangeOfValue, distance, step\_size):

    #mt is the matrix that needs to be modified

    #width is the width of the square matrix

    #rangeOfValue is the intended range of the randomly generated values

    #distance should always be half of step\_size

    #step\_size helps with checking how many times the procedure has iterated and serves as a stopping point when it equals 1

    while step\_size > 1:

        #this is the diamond step

        for i in range(width):

            for j in range(width):

                r = random.randint(0,rangeOfValue)

                if i % distance == 0 and j % distance == 0 and i > 0 and j > 0 and i < width - 1 and j < width -1:

                    mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/4 + r)

                #code below lists all possible situations where a value does not have 4 corner values

                else:

                    try:

                        mt[i][j] = round((mt[i+distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/3 + r)

                    except IndexError:

                        try:

                            mt[i][j] = round((mt[i-distance][j+distance] + mt[i-distance][j-distance] + mt[i+distance][j-distance])/3 + r)

                        except IndexError:

                            try:

                                mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i+distance][j-distance])/3 + r)

                            except IndexError:

                                try:

                                    mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance] + mt[i-distance][j-distance])/3 + r)

                                except IndexError:

                                    try:

                                        mt[i][j] = round((mt[i-distance][j-distance] + mt[i+distance][j-distance])/2 + r)

                                    except IndexError:

                                        try:

                                            mt[i][j] = round((mt[i-distance][j+distance] + mt[i+distance][j+distance])/2 + r)

                                        except IndexError:

                                            try:

                                                mt[i][j] = round((mt[i-distance][j+distance] + mt[i-distance][j-distance])/2 + r)

                                            except IndexError:

                                                try:

                                                    mt[i][j] = round((mt[i+distance][j+distance] + mt[i+distance][j-distance])/2 + r)

                                                except IndexError:

                                                    try:

                                                        mt[i][j] = round((mt[i+distance][j+distance] + r))

                                                    except IndexError:

                                                        try:

                                                            mt[i][j] = round((mt[i+distance][j-distance] + r))

                                                        except IndexError:

                                                            try:

                                                                mt[i][j] = round((mt[i-distance][j+distance] + r))

                                                            except IndexError:

                                                                try:

                                                                    mt[i][j] = round((mt[i-distance][j-distance] + r))

                                                                except IndexError:

                                                                    pass

        #this is the square step

        for i in range(width):

            for j in range(width):

                k = random.randint(0,rangeOfValue)

                try:

                    mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/4 + k)

                except IndexError:

                    try:

                        mt[i][j] = round((mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/3 + k)

                    except IndexError:

                        try:

                            mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j]+mt[i+distance][j])/3 + k)

                        except IndexError:

                            try:

                                mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i+distance][j])/3 + k)

                            except IndexError:

                                try:

                                    mt[i][j] = round((mt[i][j+distance]+mt[i][j-distance]+mt[i-distance][j])/3+k)

                                except IndexError:

                                    try:

                                        mt[i][j] = round((mt[i][j+distance]+mt[i+distance][j])/2+k)

                                    except IndexError:

                                        try:

                                            mt[i][j] = round((mt[i][j-distance]+mt[i-distance][j])/2+k)

                                        except IndexError:

                                            try:

                                                mt[i][j] = round((mt[i][j+distance]+mt[i-distance][j])/2+k)

                                            except IndexError:

                                                try:

                                                    mt[i][j] = round((mt[i][j-distance]+mt[i+distance][j])/2+k)

                                                except:

                                                    pass

        step\_size = step\_size / 2

        distance = int(step\_size / 2)

Now, in the main.py file, two sprite groups need to be defined,

#tile\_group contains all the tiles as they have special interactions with the user

tile\_group = pygame.sprite.Group()

#all\_sprite\_group contain all the sprites to be included in the main game screen

all\_sprite\_group = pygame.sprite.Group()

The three 2D matrices will be named seaLevelMatrix, soilQuality and drainage. Each of these 2D matrices will have mat\_width number of integers in mat\_width number of arrays contained within.

So mat\_width is defined beforehand, as explained previously in the terrain-generation algorithm, the width has to be 1 + 2n where n >= 2, I will use 9 as an example:

mat\_width = 9

Then, the other parameters for the generator function will need to be defined:

#diamond-square use these variables

mat\_width = 9

step\_size = mat\_width - 1

distance = int(step\_size / 2)

rangeOfValue = 20

For each matrix, firstly an all-zero version needs to be defined:

seaLevelMatrix = [[0 for x in range(mat\_width)] for y in range(mat\_width)]

Then, the corner values will be generated using the function imported from generator.py:

generator.generateInitBiome(seaLevelMatrix, mat\_width, 16)

And finally, the terrain generation algorithm needs to be applied on the matrix:

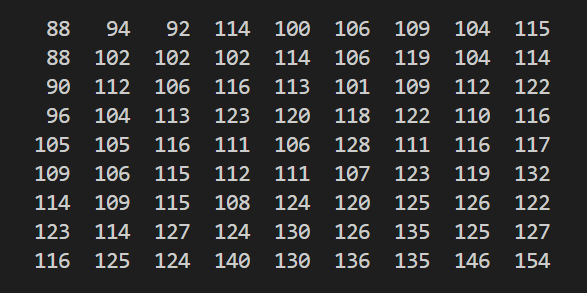
generator.diamondSquare(seaLevelMatrix, mat\_width, rangeOfValue, distance, step\_size)

Now the matrix needs to be output to test for:

print('\n'.join([''.join(['{:5}'.format(item) for item in row])

      for row in seaLevelMatrix]))

Which outputs:



Now, we just have to repeat the same code for the other two matrices:

#generate the matrices of the three factors on cost

#define as all-zero

#generate corner values

#generate all values

seaLevelMatrix = [[0 for x in range(mat\_width)] for y in range(mat\_width)]

print

generator.generateInitBiome(seaLevelMatrix, mat\_width, 16)

generator.diamondSquare(seaLevelMatrix, mat\_width, rangeOfValue, distance, step\_size)

soilQuality = [[0 for x in range(mat\_width)] for y in range(mat\_width)]

generator.generateInitBiome(soilQuality, mat\_width, 16)

generator.diamondSquare(soilQuality, mat\_width, rangeOfValue, distance, step\_size)

drainage = [[0 for x in range(mat\_width)] for y in range(mat\_width)]

generator.generateInitBiome(drainage, mat\_width, 16)

generator.diamondSquare(drainage, mat\_width, rangeOfValue, distance, step\_size)

This will complete the requirements for having three separate matrices to store the number values of tiles. This can also be implemented using a 2D array of tuples with three values each, but for the purpose of this program, this would make the referencing too complex, to reference a value inside the tuple of the matrix, it would look like: seaLevelMatrix[0][0][0].

### Improvements to be done

## Iteration 3

### Requirements being developed:

|  |  |
| --- | --- |
| A pathfinding algorithm | The algorithm is used for finding a good path for the matrix, thus allowing calculating the score. |

This is my first attempt at implementing the A star algorithm:

class tile():

    def \_\_init\_\_(self, parent=None, position=None):

        self.parent = parent

        self.position = position

        self.g = 0

        self.h = 0

        self.f = 0

def aStar(costMatrix, start, end):

    startTile = tile(None, start)

    startTile.g = startTile.h = startTile.f = costMatrix[start[0]][start[1]]

    endTile = tile(None, end)

    endTile.g = endTile.h = endTile.f = 0

    openList = []

    closedList = []

    openList.append(startTile)

    while len(openList) > 0:

        currentTile = openList[0]

        currentIndex = 0

        for ind, it in enumerate(openList):

            if it.f < currentTile.f:

                currentTile = it

                currentIndex = ind

        openList.pop(currentIndex)

        closedList.append(currentTile)

        if currentTile == endTile:

            path = []

            current = currentTile

            while current is not None:

                path.append(current.position)

                current = current.parent

            return path[::-1]

        children = []

        for newPosition in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:

            tilePosition = (currentTile.position[0] + newPosition[0], currentTile.position[1] + newPosition[1])

            if tilePosition[0] > (mat\_width - 1) or tilePosition[0] < 0 or tilePosition[1] > (mat\_width -1) or tilePosition[1] < 0:

                continue

            if costMatrix[tilePosition[0]][tilePosition[1]] != 0:

                continue

            newTile = tile(currentTile, tilePosition)

            children.append(newTile)

    for child in children:

        for closedChild in closedList:

            if child == closedChild:

                continue

        child.g = currentTile.g + costMatrix[start[0]][start[1]]

        child.h = sqrt((child.position[0] - endTile.position[0]) \*\* 2) + ((child.position[1] - endTile.position[1]) \*\* 2)\*500

        child.f = child.g + child.h

        for openTile in openList:

            if child == openTile and child.g >= openTile.g:

                continue

        openList.append(child)

#### Error 3.1

In this code there are several issues, namely: the g, h and f of the startTile, which they shouldn’t be 0; python generates imaginary part for the sqrt() function, so we will have to take the real part of the result.

def aStar(costMatrix, start, end):

    startTile = tile(None, start)

    startTile.g = costMatrix[start[0]][start[1]]

    endTile = tile(None, end)

    endTile.g = endTile.h = endTile.f = 0

    startTile.h = sqrt((endTile.position[0] - startTile.position[0]) \*\* 2) + ((endTile.position[1] - startTile.position[1]) \*\* 2)\*500

startTile.f = startTile.g + startTile.h.real

This is the fix for the first suggested issue.

        child.h = sqrt((child.position[0] - endTile.position[0]) \*\* 2) + ((child.position[1] - endTile.position[1]) \*\* 2)\*500

        child.f = child.g + child.h.real

This will operate on a matrix randomly generated, its cost is calculated as such:

def calculateCost(SL, SQ, D):

    cost = round(1\*SL + 1.5\*SQ + 2\*D)

    return cost

#### Error 3.2

However there is an issue which the path returned always gives None.

example of erroneous output
Example of an erroneous output for the shortest path.

I have inserted print statements as an attempt to investigate the error and debug the algorithm.

Upon close inspection, there has been an indentation error.

        for child in children:

            for closedChild in closedList:

                if child == closedChild:

                    continue

            child.g = currentTile.g + costMatrix[start[0]][start[1]]

            child.h = sqrt((child.position[0] - endTile.position[0]) \*\* 2) + ((child.position[1] - endTile.position[1]) \*\* 2)\*500

            child.f = child.g + child.h.real

            print(child.f)

            for openTile in openList:

                if child == openTile and child.g >= openTile.g:

                    continue

            openList.append(child)

#### Error 3.3

However, changing the indentation still didn’t fix the issue.

It seems that this part of the algorithm is not accessed, which I have inspected the code, it seems that an erroneous statement has been inserted under while len(openList > 0):

            if costMatrix[tilePosition[0]][tilePosition[1]] != 0:

                continue

Now the problem is that the algorithm outputs this when popping things off the list:

\_\_main\_\_.tile object at 0x00000219AE6CF520>, <\_\_main\_\_.tile object at

0x00000219AE6CF640>, <\_\_main\_\_.tile object at 0x00000219AE6CF6A0>, <\_\_main\_\_.tile object at 0x00000219AE6CB7C0>, <\_\_main\_\_.tile object at 0x00000219AE6CBAC0>, <\_\_main\_\_.tile object at 0x00000219AE6CF940>, <\_\_main\_\_.tile object at 0x00000219AE6CFA60>, <\_\_main\_\_.tile object at 0x00000219AE6CFAC0>, <\_\_main\_\_.tile object at 0x00000219AE6C5E20>, <\_\_main\_\_.tile object at 0x00000219AE6CFCA0>, <\_\_main\_\_.tile obj

Which means that the closed list seem to contain only the memory addresses. This seems to be a problem related to the storage method of python, so I modified the code for a bit, giving it a stopping condition as a count, which makes the program end at a reasonable point.

Also with a bit of research I have found a python module for priority lists called heapq, which is quite convenient for my project, as a star algorithm requires the use of a priority list.

#here starts the pathfinding algorithm

#create tile class for the nodes purposed for the A star algorithm

class tile():

    #the \_\_init\_\_ function should take in the parent node and the position of the current node

    def \_\_init\_\_(self, parent=None, position=None):

        #assign the variables

        self.parent = parent

        self.position = position

        #g is the actual traversal cost of the node

        self.g = 0

        #h is the heuristic output of the node

        self.h = 0

        #f is the sum of g and h, the value which the algorithm will use

        self.f = 0

    #allows assigning and comparison between tiles

    #allows comparison between tiles

    def \_\_eq\_\_(self, other):

        return self.position == other.position

    def \_\_lt\_\_(self, other):

        return self.f < other.f

    def \_\_gt\_\_(self, other):

        return self.f > other.f

def returnPath(current\_node):

    path = []

    current = current\_node

    while current is not None:

        path.append(current.position)

        current = current.parent

    return path[::-1]

def aStar(costMatrix, start, end):

    #define starting tile and the ending tile with their f, g, and h

    startTile = tile(None, tuple(start))

    startTile.g = costMatrix[start[0]][start[1]]

    endTile = tile(None, tuple(end))

    endTile.g = endTile.h = endTile.f = 0

    startTile.h = 0

    startTile.f = startTile.g + startTile.h

    #end of defining start and end

    openList = []

    closedList = []

    heapq.heapify(openList)

    heapq.heappush(openList, startTile)

    #a stopping condition

    count = 0

    maxCount = (len(costMatrix)\*len(costMatrix[0]) // 2)

    #add startTile to the open list

    openList.append(startTile)

    adjacent = [(0, -1), (0, 1), (-1, 0), (1, 0)]

    while len(openList) > 0 and count <= maxCount:

        count += 0

        if count > maxCount:

            warn("stop")

            return returnPath

        currentTile = heapq.heappop(openList)

        closedList.append(currentTile)

        if currentTile == endTile:

            return returnPath(currentTile)

        #define children of the current tile

        children = []

        for newPosition in [(0, -1), (0, 1), (-1, 0), (1, 0)]:

            tilePosition = (currentTile.position[0] + newPosition[0], currentTile.position[1] + newPosition[1])

            if tilePosition[0] > (mat\_width - 1) or tilePosition[0] < 0 or tilePosition[1] > (mat\_width -1) or tilePosition[1] < 0:

                continue

            newTile = tile(currentTile, tuple(tilePosition))

            children.append(newTile)

        for child in children:

            if len([closedChild for closedChild in closedList if closedChild == child]) > 0:

                continue

            child.g = currentTile.g + costMatrix[child.position[1]][child.position[0]]

            child.h = sqrt((child.position[0] - endTile.position[0]) \*\* 2) + ((child.position[1] - endTile.position[1]) \*\* 2)\*400

            child.f = child.g + child.h.real

            if len([openTile for openTile in openList if child.position == openTile.position and child.g > openTile.g]) > 0:

                continue

            heapq.heappush(openList, child)

    warn("no solution")

    return None

Thanks to the heap modules, now the program is much more elegant and easier to understand.

I was able to achieve the expected results, concluding the success of my implementation of the A star algorithm. To optimize the algorithm, I will be needing a better heuristic, which I have been using just the estimate of the mean calculated by me, I will now be defining the function to find the exact value of all the mean of the costMatrix.

def returnMean(costMatrix):

    sum = 0

    for i in range(len(costMatrix)-1):

        for j in range(len(costMatrix[0])-1):

            sum += costMatrix[i][j]

    mean = sum//(len(costMatrix)\*len(costMatrix[0]))

    return mean

And I will implement it into the heuristic function.

child.h = sqrt((child.position[0] - endTile.position[0]) \*\* 2) + ((child.position[1] - endTile.position[1]) \*\* 2)\*mean

According to peer interview, the original calculateCost function is too boring and unrealistic as the squares have quite similar costs with no large differences.

def calculateCost(SL, SQ, D):

    cost = round(8\*SL + 1.5\*SQ + 2\*D)

    return cost

Then some other attributes in the gameTile class has to be added such as its position and its cost and the selected state.

    def \_\_init\_\_(self, color, width, x, y, a, b, cost, selected=False):

        super().\_\_init\_\_()

    # Create a sprite and fill it with colour

        self.image = pygame.Surface([width,width])

        self.color = color

        self.image.fill(color)

        self.rect = self.image.get\_rect()

        self.rect.x = x

        self.rect.y = y

        self.a = a

        self.b = b

        self.cost = cost

        self.selected = selected

Now, as we have the selected attribute (which is set to False originally until it is updated), we can have a function for the gameTile class for when the tile is selected for road construction.

    def updateWhenSelected(self):

        if self.selected == False:

            self.selected = True

            self.image.fill((0,0,0))

        else:

            self.selected = False

            self.image.fill(self.color)

### Improvements to be done

### Conclusion

## Iteration 4

### Requirements being developed

Basic display of the map.

### Improvements to be done

### Conclusion

## Iteration 5

### Requirements being developed

I added in custom font for the displays of texts, I have found that the pixelized fonts to fit the theme of the game quite well, python doesn’t process the relative directories because of the /s, so sys has to be imported to form the directory for the font.

normalFont = pygame.font.Font(os.path.join("A-level-Project", "game", 'normalFont.ttf'), 16)

Inside the game loop, two statements can be added to blit the text onto the screen.

while running:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            running = False

    all\_sprite\_group.update()

    screen.fill((0,0,0))

    all\_sprite\_group.draw(screen)

    #text elements

    myName = normalFont.render('Made by Tony Lu', False, (255, 240, 230))

    screen.blit(myName,(0,0))

    clock.tick(60)

    pygame.display.update()

pygame.quit()

Now I would like to add the text to display the target cost which was calculated by the a star algorithm. So I added a variable for the text.

lCost = "Target cost: " + str(totalCost)

Then inside the game loop

    targetCost = normalFont.render(lCost, False, (255, 240, 230))

Then displaying it

    screen.blit(targetCost,(810, 360))

Also the colors of the gameTile sprites have been adjusted to be more calm, previously each of the RGB values are equal to each attribute values.

color = ((256-1.03\*drainage[i][j]), (256-1.05\*soilQuality[i][j]), (256-0.8\*seaLevelMatrix[i][j]))

A picture containing bathroom, tiled, tub, tile

Description automatically generated

Then according to the peer interview, I obtained the feedback that the colours lack vibrancy in appearance, making the game quite uninteresting, and also there might be too many tiles making the game too complex for the general public.

But I have spotted that the numbers that are output by the calculateCost algorithm to be too high, so I modified the calculateCost algorithm to be:

def calculateCost(SL, SQ, D):

    cost = round((0.1\*(SL\*\*2) + 15\*SQ + 20\*D)/100)-40

    return cost

As the requirements have specified, the darkness of the colour should reflect how much such tile costs, hence the colour calculation statement is also modfied:

color = (256-1\*drainage[i][j]), (256-0.75\*soilQuality[i][j]), (256-(0.005\*(seaLevelMatrix[i][j]\*\*2)))

The functions will have to be programmed now, the following iteration will document how the following objectives are completed:

1. Function for road construction and its UI updates.
2. Add scoring system in relation to how close the user got to the perfect solution
3. Cancel function for constructed road

Requirements 1 and 3 are tackled by establishing a new attribute for the gameTile class, allowing the gameTile to switch state of selected or deselected.

### Errors

### Conclusion

# Testing

After finishing with the development, the tests included in the test plan are carried out on the program.

# Evaluation

There are many features that can be added in the game still, for example, audio when clicking on a button or on a tile, which can be easily implemented, but is drawn back by the erroneous interaction between visual studio code and the pygame library.