Computer Science Project

Cameron Mitchell

Documentation

2019-2020

Contents

1.0: Analysis

1.1.1: Problem Identification

In today's world many teenagers play FPS or "First-person shooter" games and not enough play more strategy games which may be more beneficial to their development and introduce them to more dynamic skills like management. To solve this problem, I will create a game for entertainment purposes that will attempt to create a basic simulation of building and running a town or city. These management skills are highly used in the real world as it requires the user to maintain a lot of different priorities about what to build, where to build it etc. which can be used in a management or more senior job in a company.

Due to this, there is a need in the market for a game which can help develop these management skills, especially as financial management is scarce in secondary schools across the country. The reason I would make it educational would be to allow the user to want to constantly play the game and therefore constantly improve their management skills. There is a genre called "City-Building" games which are very similar to the game that I would like to create these, however, are propriety games whereas mine would be open-source so that people could add their own content to the game, improving their experience, therefore keeping them on the game for longer and increasing the benefits of playing the game such as the management skills. As well as this, the game would therefore be free and allow more users to access the benefits of playing the game.

1.1.2: Use of Computational Methods

The use of a computer would allow the user to do things they wouldn't be able to without one, creating and building a town or city in real life would take a lot of money and/or time whereas on a computer it can be almost instant so would allow the user to gain the benefits of the simulation without having to do it in real life. Computational methods are also much better than other methods because it can quickly and effectively respond to the user input and generate an output, something that is necessary for a game like the one I will create as the player's view of the screen will constantly need to be updated.

As my solution will also rely on graphics being generated throughout the course of the game computational methods are needed because they can be generated quickly and can calculate where a graphic need to be placed and rendered rapidly. As computational methods are quick the user will also be able to have an experience which is uninterrupted as the graphics can be loaded and rendered during other user actions and are needed as calculations are faster than that a human can do are required and in real-time.

The use of non-computational methods is simply not feasible for my target market (12-18 years), this is due to the amount of time and calculations someone would have to do in order to create maybe a paper model of a city and do it that way.

Another reason would be that people of this age range don't have full-time jobs and therefore don't have enough disposable income to buy a huge amount of land, hire the staff and get the materials to build the infrastructure to support a growing town or city in real life. So the benefits of using a computational method would be much better than other non-computational methods due to the speed, low-cost and ease of using a computer. As more stakeholders have a computer to run the solution on the stakeholders would also be much happier with a computer-developed solution.

I will use abstraction in my project, this will be used to simplify the complex nature of the real world into a playable state. Examples of this will include the simplification of areas of land into a small grid-based layout with the no change in incline or terrain throughout the grid compared to the natural variations in real life. This would be far too complex to program in and not necessary for the function of the actual simulation so I will model the land as flat and the same size. I will also display the land as two-dimensional as an abstraction from our three-dimensional reality as this would be hard to program and wouldn't be required for the simulation as the two-dimensional display is enough to suggest to the user what each element on the screen is.

I will use decomposition in my project, this will be used to break much larger tasks such as the drawing of text and buttons on the screen into much smaller tasks and problems like the position of the drawing, font of the text and colour of the text and buttons. This will be carried out throughout the project and will allow the complex problems to be decomposed into many simpler tasks. An example of this being the decomposition of my success criteria into many smaller tasks within each sprints design stage.

1.2: Stakeholders

The stakeholders for this project will be 12-18-year olds. This is the chosen age range because the solution aims to develop the skills such as management and strategy in this age range, alongside other qualifications they may be doing, benefitting them. Older Children and teenagers require a fast-paced, interesting game as they have busy lifestyles and get bored more easily than other age ranges, so my solution will include an intuitive UI so that the user doesn't have to read lots of instructions so that the attention of the user will be focused on the game and will be interesting for them. The controls will include a series of buttons and will mostly use the mouse so that the controls will be easy and simplify playing of the game for the users. The game will also have to be entertaining so that the user doesn't get bored and stop playing the game, as to reduce the benefits of playing it. The game will include a lot of colourful graphics and images to keep the user's interest and keep them playing the game.

Furthermore, the game will sustain the user's interest as the game will increase in difficulty as the game progresses due to the user having to manage more and more as the game progresses. This increase in difficulty will mean that older users get a more challenging and interesting experience while the younger and more novice

players will get a beginner and easier version of the game at the beginning to learn the basics before getting more complex.

The game will not just be for the 12-18 age range and can be played by people outside of this age range, but I will conduct an opportunity sample of potential users to see what type of game they would like to see developed so that the benefits I am trying to create with the solution are maximised.

1.3.1: Research

I researched possible existing solutions to this problem and found three solutions which I'll use to give me an insight into how I could create my solution and what works for my age range and stakeholders by using three examples from a wider target market and not just my specific age range.

1.3.1.1: SimCity (2013)

The first game I researched was SimCity, which is a single/multiplayer city-building game developed by Maxis and published by Electronic Arts.

In this game the user starts off with a blank map and a road leading off to a highway, where the "people" of the city come from to move into their new "houses". The player then builds roads on this map and highlights various areas around these roads to tell the game what type of building the user wants to build there as the user expands the town the game gets harder with new services needing to be built to sustain a population while the houses "upgrade" to a house that can support a larger population. This game has a system whereby the user clicks on a building they want to place and a faint version of the building appears to help the user see where the building will be built and how it will look once it has been placed, it also features a modular system whereby the user can customise different services to expand its capabilities. These ideas seem too complex for my age range but could be added at a later stage of development if the game seems too easy. The game is also 3D, something which would be very hard to implement and isn't necessary for the solution I am creating so I would not include this feature in my solution, even though it would make the game look better, but would restrict the user base to people with good hardware so it wouldn't be ideal as staying 2D would allow the user to play on most devices as opposed to only a PC. The game doesn't have a definitive end and only ends if the user loses all of their money or until the user ends the game, this is a feature I will probably use as it would mean that the user can experiment with the game, again increasing time spent on the game, increasing its benefits.

SimCity Features

Key:
This indicates features I will adapt
This indicates features I could adapt
This indicates features I won't adapt

A 3D view of the buildings means that the user can easily see the various buildings in the city and looks aesthetically pleasing. But it isn't necessary and would take a lot of development time as well as not being necessary for the solution. This is not a feature I will adapt.

Buildings are of differing "heights" to represent their varying levels. I think I would adapt the idea of buildings having levels, but it would display this on the side of the tile with the house on for simplicity. This would allow the user to see the development and progress in their game and keep them interested for longer.

Varying road lengths is something that I will adapt so that the game is customisable but won't be dynamic like curves as the game will be 2D based and square tile based. Customisability will make sure each game is different and will retain interest in the game, even after multiple sessions.

Colourful Aesthetic Design to make the user sustain interest in the game, this is something that I will probably adapt for my solution.

Quests from your "people" gives the user a challenge and a different game each time, this would help with adding difficulty and variety but a little too complicated so probably something I wouldn't use in my solution and isn't necessary for my solution.

Allowing the user to have multiple cities and regions would be a feature which could be added in the future but wouldn't be essential. I could save each world in a txt file or something else and allow the user to save and edit each, which would mean the user could experiment within the game environment and save their

Allows the user to control time so the simulation speeds up, this is a feature that I probably won't include as it would take a lot of development and isn't really required for the solution to work as intended.

Displays city specifications to the user in a simple, clean user interface that allows the user to see at a glance how the city is doing with population, finances and demand for certain buildings. This is a feature that I will use as it will be vital for the user to see their progress in the game and what they can build based off the finances.

Gives an option for the user to remove buildings once placed, this is probably something that would be useful for my solution as it would allow for the user to make mistakes and learn.

1.3.1.2: Cities Skylines

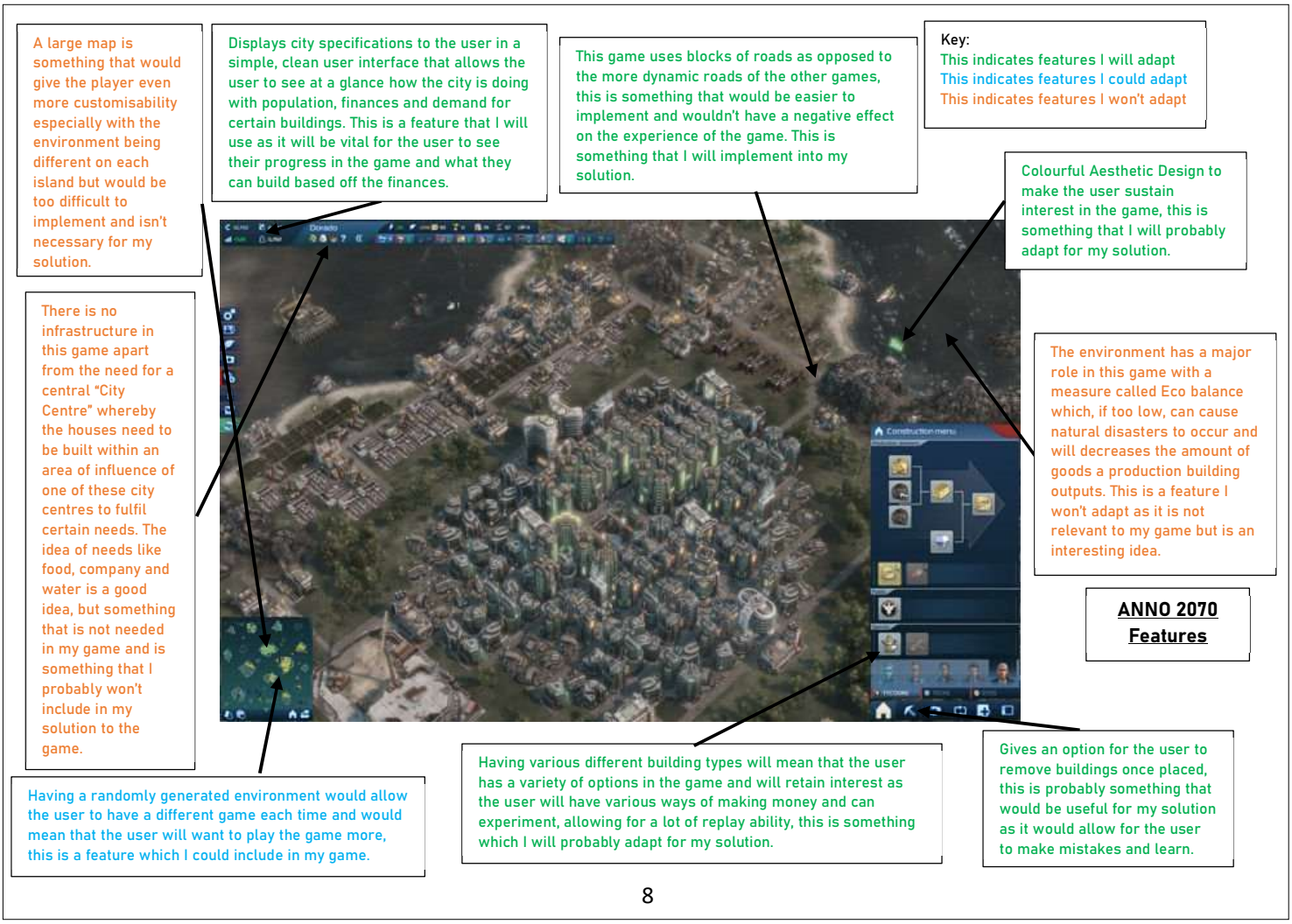
The second game I researched was Cities Skylines, which is a single player city-building game developed by Colossal Order and published by Paradox Interactive.

In this game the user starts off with an empty map, like SimCity but the difference is that you can expand the map once you reach certain population milestones. The user, again, starts off by building roads and then zoning areas for various types of buildings around those roads. The user also gets to place services, but this time gets to place infrastructure, increasing the games complexity and difficulty – something that I could add at a later development stage. Unlike SimCity, you cannot modularly build buildings, but you can build multiple sizes of a type of building, so you could build a hospital and clinic for example. This idea would be ideal for my project and is something I will probably implement as it adds more variety to the game and means that the game is less complex than a modular system yet you can get better and more expansive networks of public services once the need is there. This game is also 3D, but has a much simpler aesthetic design, if I was going to make the solution 3D it would be done like this as it is much simpler than the complex designs of SimCity. However, I will do my solution in a top-down 2D viewpoint to simplify the design and make it easy for the user to see where each building is. This game also rewards users for reaching certain population milestones, with money and unlocks of various larger public services, this seems like a feature I could add at a later stage of development as it would give the user an incentive to carry on playing.

1.3.1.3: Anno 2070

The third and final game I researched was Anno 2070, which is a single/multiplayer game developed by Related Designs and Blue Byte and published by Ubisoft.

This game is completely different to the previous two games as it focuses on a futuristic setting and a series of islands rather than a small section of a map. The aim of the game is to build homes for people who have different classes, starting at worker and then progressing to executives, an interesting idea, but as I want to create a wealth based system to avoid the class complexity I will make my system so it is a series of wealth levels to add a feeling of progression and complexity to the game without the over-complex class system. The game also adds a unique choice at the start of each game as to whether you want to be the industrial tycoons or the environmentally friendly ecos, this adds another layer of complexity but something that I will avoid as I would need to add more than double the amount of graphics for a not that big gain in experience or quality for the user. It is a game that has combat, something that I won't adapt for my game as it is not really part of the game's core function and wouldn't be included in this version of the game but could be added in the far future due to its open-source nature.



1.3.1.4: Survey

To understand the need and wants of the target market and stakeholders I will first carry out some primary data collection in the form of a survey. This would give me a much clearer insight into the problem and what I need to do to make a successful solution. The survey will be conducted on students aged 12-18 that regularly play video games and will involve both genders. My survey will include questions on how often the play strategy games and how often they play other types of games. It will also include questions based on the features and functions needed from a solution of the kind that I will be developing.

Results:

I conducted the research on 15 students aged 12-18 including 4 girls and 11 boys.

How often do you play video games each week?

0-3	3-7	7-12	12+
0	6	7	2

What type of game do you play the most?

FPS	Puzzle	Strategy	Educational	MMORPG
9	1	3	0	2

Would you prefer a social class-based game or a wealth-based game?

Social Class-based	Wealth-based
4	11

Would you prefer a randomly generated map or a pre-generated map?

Randomly generated	Pre-generated
7	8

Would you rather use a mouse or keyboard to play the game?

Mouse	Keyboard
9	6

What do you play video games on?

Mobile	PC
2	13

Would you prefer a block-based or dynamic-based road placement system?

Block-based	Dynamic-based
10	5

Would you want to be able to view your city statistics?

Yes	No
13	2

Would you want to be able to save your progress?

Yes	No
9	6

Would you want the map to be expandable?

Yes	No
10	5

1.3.1.5: Research Conclusion

From my research, including a survey and pre-existing solutions, I have considered what features are necessary for my solution as well as features that I don't need to include. I will be developing my solution for the PC platform as this is what my target market and stakeholders use to play games like the solution that I am going to develop, based on my survey. As well as the platform being best suited to this type of game as the user can easily control and manage a city simulation using primarily a mouse and a keyboard.

From SimCity, I will adapt many different and varied ideas for my project while retaining a 2D GUI and not a 3D one. The first feature I will adapt will be the placement and management of various public services such as health, water, sewage and power. This would allow a massive variety to the game and would add difficulty as the user has to manage multiple services at once. Another feature I will adapt will be varying road lengths but will use blocks instead of a dynamic-curvy road structure to simplify design and development. The colourful aesthetic design I will also adapt as this will keep the user interested and make them play the game for longer, it would also attract a wider number of stakeholders as the game would look better than other games.

From Cities Skylines, I will also adapt some features. The ability to remove buildings through a demolition tool will also be something I will include in my solution as it allows the user to learn to play the game effectively and learn from their mistakes, allowing them to improve. I will use a similar city specification tab from this game to allow the user to see their progress and is functional as it will allow the user to see how much money they must build roads, buildings etc.

From Anno 2070, I will include a similar clean UI to allow the user to easily see their statistics and choose a type of building to build quickly and more effectively manage their city.

From my survey I have found out that the desired user uses a PC rather than a mobile device which means that I need to develop my code so that it is most optimised for a mouse/keyboard control rather than a touch-based control. I have also found out that the user would rather have a mouse-based control rather than a keyboard control so I will need to make a lot of clickable buttons that the user can interact with using a mouse when playing the game. I also have learnt that most users play FPS games with a minority playing the type of game I am looking to develop, so this means that the game will have to be fast-paced to allow for the user to transition from the fast-paced environment of an FPS game to a strategy city-building game. I will also make my game wealth-based, as in the various levels of the residential buildings in the game will be based off a 3-tier "wealth" system.

I will also make my game a pre-generated map as that's what the majority want but will attempt to also make it randomly generated if I have the development resources to make the game more enjoyable to more users. I am including other features that users would want as well, such as: block-based road placement, ability to view city statistics, ability to save progress and then potentially expandable maps.

1.3.2: My Solution

1.3.2.1: Essential Features

Overall, I will develop a 2D city-building and management simulation whereby the user builds, upgrades, maintains and earns taxes from a variety of different buildings that the user may build, including residential buildings and public services. The purpose of my solution will be to allow stakeholders aged primarily from 12-18 to develop management and strategy skills that may become useful to them in managing time for their studies etc.

The game will manage difficulty by allowing each residential building to be "upgraded", that is increased in demands it needs from public service buildings and water. This would allow the user to have a relatively easy game at the beginning and then for it to naturally get harder as the necessary investments in public buildings and such increases, increasing the difficulty of the game. The game will only end when the user quits or the user runs out of money, to simulate real life consequences.

A pre-generated map will allow the game to progress at the right rate without being too hard, from scarcity of resources etc. A procedurally generated map may be able to be implemented but it depends on the development resources that are available during development.

A variety of public service types will be added to the game including fire, police, health and education. This will add more things to the game to manage which will increase the difficulty and make the game more interesting, retaining user interest.

I will also include a system whereby the user can place roads in multiple arrangements to allow the user to customise their game and make the game be able to be replayed multiple times with a different experience. This will be achieved through using the grid-based system to allow the user to place a road in any arrangement they would like.

A small box on the UI will also show the user the statistics of the game, such as population and money. This will allow the user to easily track their progress and the population metric will effectively be the game "score" as the larger population they can support, the harder the game is and so the user gets a higher "score".

The user should be able to save their progress of the game to resume it if they can no longer play at the time. This will be accomplished using a text-file and the contents of each grid will be saved here as well as the city's statistics. This should mean the user plays for longer as they can reload various states of their city and correct their mistakes.

A menu screen will allow the user to start a new game or load the latest game save. There will be no help-guide and the user will have to learn how to play the game on their own, adding to the difficulty and gives the user more freedom to learn, something I am trying to encourage in the game.

The game will be programmed for the PC platform and using mostly a mouse as this is what most stakeholders use and would therefore be the most beneficial.

Summary:

- The difficulty will be scaled with the level and amount of residential buildings and the public services they will therefore need provided, a population mechanic along with a wealth mechanic
- The "score" will be determined from the user's population size
- A pre-generated map will be used unless there is enough time to create a generator for a procedurally generated map
- A variety of different buildings will need to be implemented to increase difficulty
- Roads will need to be able to be built to connect buildings together
- The program will use a mouse to control most things in the game
- A colourful aesthetic will be implemented to appeal to the user and make sure the distinction between different types of buildings is much easier to see
- A menu will be used to save and start games
- The state of games will be saved for the user to carry on their progress
- The user will be able to see their city's statistics within the UI in real-time

1.3.2.2: Limitations

The major limitation to my project will be time management, as my project contains many different features and different buildings, I will need to judiciously use my time so that I do not run out of time. There are many features that I could include if I manage my time correctly, such as a procedurally generated map and more building types like industry. I will prioritise the features that I need to include in my essential features section and add these other, less-important, features later if I have the development time.

The second limitation of my project is that it won't include all the features that a real city-building simulator does, such as infrastructure of pipes and electricity cables so will not be totally realistic to real city-management and will be less developed than some other existing solutions such as Cities: skylines. The non-expandability of the map will also mean that the user is limited to how much they can develop a city and therefore bring down replay ability for the user.

Furthermore, it would be hard to implement a very colourful design as it would be hard to include graphics that include more than one colour as importing the pictures would be a lot harder than just filling in that box on the grid. The 2D nature would also limit this and would mean that the user would be more likely to go with an existing solution.

The actual development of the game would also be a challenge as I will be using python and mostly the library "PyGame" as it will be easy to create a clock to refresh the UI, something that would be extremely hard to implement in something like C++. As I am not that familiar with creating various things in Pygame I will need to use existing examples and research how other solutions have accomplished things like saving a grid into a text document and a main screen to load a game and start a new game from.

1.3.2.3: Requirements

User Hardware and Software

Mouse, Keyboard, Monitor

The user will require these basic peripherals in order to play the game, the mouse will be the main way the user will control the program. This will include clicking on buttons, tiles and exiting the game, but the keyboard may be used for shortcuts or to exit the program. The monitor is also required to allow the user to see the map of the game and where they need to click to place a building etc.

CPU – 1Ghz speed

My solution will not require a very fast CPU as the game will only be a 2D game which doesn't require any advanced logic and or lighting so should be relatively simple for the CPU to render. It will also not require a GPU for this reason, if I were to create a 3D game it would be required.

RAM – 1GB

Not much RAM will be required to play the game as the program will store states of tiles as arrays so the RAM will not need to store vast amounts of data at one time. The user should have more RAM than this basic requirement, however, as other processes will be running simultaneously with this program.

Hard Drive – Less than 100 MB needed

Less than 100MB will be needed for all my game files to be on a user's computer, but the user will need more than this to run the device anyway, with windows being much larger than this by default so storage shouldn't really be a problem.

Windows Operating System with Python

I may export the project as an executable, but the most likely eventuality is that I export the project as a python files, this means that the user will need to have python installed on their device. Windows is also needed as I will be referencing libraries that contain windows-specific code.

Developer Hardware

Mouse, Keyboard, Monitor

I will need to have a mouse as it will allow me to use the various features of an IDE as I need to be able to click on various buttons to allow me to carry out functions such as running the program and testing the solution. I will also need a keyboard to document my project and type the code to be used in the program. A monitor is required for me to see the IDE and code I have typed as well as test the solution for errors, it will need to be of at least 1024 x 768 to allow me to view a decent amount of code at once.

CPU – 1.8Ghz Speed

I will need a CPU faster than the user's as I will need to much more quickly run the code for testing and load up the IDE and various examples that I need in order to effectively create my solution.

RAM – 1GB

A minimum amount of RAM is needed to load the IDE, OS and program at once so that I can test the program and code the project. Also, I would be able to load up a browser such as Chrome to allow me to research solutions to various errors I might find from testing.

Hard Disk – 10GB+

I would need a substantial amount of storage space in order to download and install an IDE, OS and the libraries that I need in order to effectively create and test a solution.

Developer Software

The language I will use to develop my solution will be python as I am experienced with writing programs in the language and along with libraries such as pygame it allows me to create my solution with a lot more ease than with, say C++. I will be using the visual studio Integrated development environment to create my solution as it gives me access to many useful features. I chose this IDE as it is free, as to reduce the costs of the solution and it allows me to add different libraries if I need them in my solution at a later stage.

The reasons I will use an IDE:

- Visual studio scans the code you type and can identify errors such as non-existent variables and can automatically indent your code to allow the programmer to code effectively as it is automatic.
- Visual studio also makes I easy to test my code, such as adding breakpoints into my code to allow me to test a specific part of my code for errors.
- Visual studio has a built-in library downloader and installer for python so this would mean I can add many various libraries to my project without having the need to keep downloading them off websites and other sources.

1.3.2.4: Success Criteria

1. Game environment – creates a window in which a map is visible and contains a grid of squares so that the user can eventually place things inside this grid, the grid will be made of white squares.
2. Detect grid – creates a way for the program to know what grid has been pressed on or whether they haven't pressed on any to allow the user to place objects in a grid of their choosing.
3. Create button – creates a button for the user to choose an object to place to allow the user to place an object of their choosing.
4. Place Object – once the user has clicked on the button and on a grid tile the tile will turn black to test the placement is working.
5. Exit Button – create a button to allow the user to quit the game.
6. Create multiple buttons – create buttons that allow the user to select what colour they want to place; this will make sure the user can switch between the various types of buildings that they want to place.
7. Create statistics UI – Add a box to allow the user to see their "money" and "population", this will allow the user to see what types of buildings they can build.
8. Assign wealth – Assign a wealth to each button and set the money to some figure greater than this and don't allow the user to place that object if they don't have enough money and display this in the UI.
9. Assign population – Assign a population to placed objects and then display this in the UI.
10. Create Utilities – Create/Assign buttons for each of the utilities in the game, the amount of these required will be determined off the population size.

11. Timed Game – Create a clock which refreshes money every half-second so that the game seems fluid.
12. Maintenance Costs and Taxes – Create a system whereby the utilities reduce the money variable by a set amount per utility per 15 secs, and each residential building gives you a set amount of money per 15 secs with the intention to act as a sort of tax and that you have to keep the money stable.
13. Check Road – Checks the placed object is next to a road and if not, then the user is unable to place the desired object.
14. Assign proper aesthetic – Create a system whereby empty tiles are green to represent grass and residential and utility buildings have different colours based on their purpose to allow the user to more easily see what services they have and what they need.
15. Add wealth-system – Create 3 types of residential housing, $, $$, $$$ to illustrate the various levels of wealth of each house. The amount of services required will increase as you go from $ to $$$ as to increase the difficulty. Have these assigned different colours or graphics.
16. Save Game – Add an ability for the user to save their game to resume it another time through saving the grid, money and population in a text file.
17. Default Map – Add a default/new-game grid, money and population to load when a new game is started.
18. Main Menu – Create a start menu that comes up when you load the program that will allow the user to select load save game and start new game.

2.0: 1st Sprint

2.1: Design

I my first sprint I will create the foundations of my game, I will start by laying out the initial design of the window that the program will run in. I will then create a way of telling which part of the grid layout the user has clicked on in order to place objects where the user desires later. After I have tested my initial development and can select and place an object, I can later adapt this for other objects. The aim of the first sprint is to setup the initial grid layout and be able to place an object, shown by changing the colour of a tile in the grid. I will base my first sprint off the first 4 success criteria and split it up into smaller parts:

1. Setup Game Environment:
    a. Create a window of a suitable size
    b. Create grid layout of the "map"
2. Detect what part of the grid:
    a. Detect position of user input in the window
    b. Detect which grid the user has clicked on
3. Create Button:
    a. Create a button that the user can click
4. Place Object:
    a. Allow the user to change the colour of a tile of the grid they click on after clicking on the button – "Placing an object"

I have chosen to break down the sprint into these smaller sub-tasks within my success criteria and will program them in order as each next task is dependent on the completion of the previous task. I cannot allow the user to change the colour of a tile in a grid without creating the grid first. I will first plan out my program structure using algorithms to make sure that I know what parts I need to code for each task and to make sure my program runs efficiently.

2.1.1: Game Environment

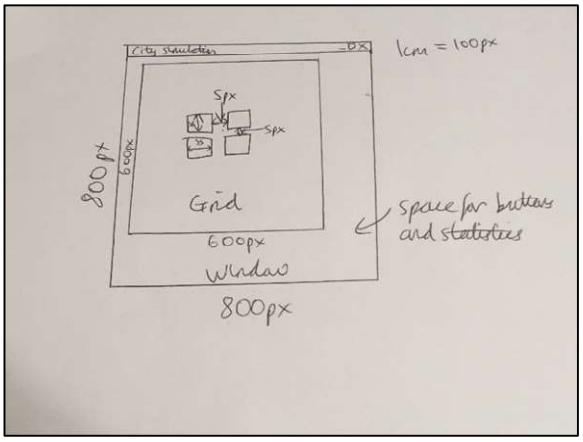
To develop the game, I will be using Python and some python libraries including pygame. This will allow me to crate the UI for the game and add many of the features such as buttons and a grid that I need for my solution. The IDE I will be using will be Visual Studio as there is a community free edition and has an inbuilt library downloader for python.

a. I will first use a function provided in the pygame library I will create a window that will appear once the program is run. It will be an 800 x 800-pixel sized window that should be sufficient to contain the buttons and grid layout for the game and the small size would allow it to be run on most modern devices, allowing the stakeholders to use the program on more monitor sizes, making it more accessible. The background colour of the program will be black, to allow the grid to be easily visible atop it.

```
Screen = pygame.window
WINDOWSIZE = (800,  800)
WINDOWTITLE = “Title”
```

Window size represents the width and height of the window and the title represents title of the window; the window is then initialised. Screen represents the actual window that pygame will run in.

b. There is also a function within pygame for me to create a grid, I can then apply a 2D array to this to provide a way of storing the status of each tile within the grid to be edited and data to be drawn from when saving etc. This will be a 10x10 grid so that the user will have 100 tiles to place objects on, which should be enough to play the game optimally. This can be increased with later developments but will be this size for now. The grid will be white in colour and it will be done this way to make it easy for the user to see the grid in contrast to the black background.



Having a 600 x 600 Grid would allow me each tile to have 55 pixels each as I would leave a 5-pixel margin between each of the tiles in the grid. Allowing the user to see each tile individually much more easily, due to the black background and means the user is much less likely to mis click on a tile that they didn't intend to.

```
Land = []
For i in range 1 to 10:
        Land.append([])
        For i in range 1 to 10:
                Land[row].append(0)
```

2.1.2: Detect Grid

a. In order to detect the user's, input I will again use the pygame library to detect the user's click. I can use the “pygame.mouse.get_pressed” command to get the status of the button when it's clicked and then run a specific function based on if it's clicked or not.

```
If user does event:
    If event == MOUSE PRESSED:
        print(“Hello”)
```

Using some function that will output “Hello” once the user has pressed on the screen, will allow me to see whether the user has clicked and run functions based on this.

b. I would then set a range of coordinates for each grid tile and then if the user clicks in one of these ranges, that grid tile would be the one to have its value changed in the array.

```
WIDTH = 55
HEIGHT = 55
MARGIN = 5
done = False
While not done:
    If event == exit:
        done = True
    Else:
        If event == MOUSE PRESSED:
            pos = getLocation()
            x,y = pos[0], pos[1]
            column = x / (width + margin)
            row = y / (height + margin)
            Land[row][column] = 1
Clock = pygame.clock(60)
Screen.update
```

Having each tile as a 55 x 55 pixel square with a 5 px margin between each allows the tile that the user has clicked on to be accurately estimated using the column and row values with the x and y coordinates for the cursor position when it is pressed. The element at that predicted tile will be changed to a 1 for the tile to be updated later. The done variable means that only if the user chooses to exit the window does the code stop executing. The clock variable means that the window will refresh x times per second. I have set this to 60 as this will be enough.

2.1.3: Create Button

a. For the user to select what kind of "object" to place in a grid tile they will be a button that will be depressed once the user has clicked on it. It will then become not depressed after the object has been placed.

```
WHITE = (255,255,255)
RED = (255, 0, 0)
GRAY = (128,128,128)

WIDTH = 55
HEIGHT = 55
MARGIN = 5
CreateButton(height = 150, width = 150, WHITE, "Button"):
done = False
While not done:
    If event == exit:
        done = True
    Else:
        If event == MOUSE PRESSED:
            pos = getLocation()
            x,y = pos[0], pos[1]
            if (5 <= x <= 150) and (605 <= 710):
                if depressed = True:
                    pressed = 0
                    depressed = False
                else:
                    pressed = 1
                    depressed = True
            column = x / (width + margin)
            row = y / (height + margin)
            Land[row][column] = 1
Clock = pygame.clock(60)
Screen.update
```

This algorithm means that if the button is pressed it will become depressed and pressed will be set to 1 so that if the user then clicks on the grid the element that represents the clicked tile will turn to 1 if the button is pressed. If the button is not pressed and the user clicks on the grid nothing will occur as pressed = 0. If the button is then pressed on again it will become not depressed and pressed will be set to 0 to represent the button no longer being pressed. The CreateButton function can be used to create a button using the passed through functions, this is something which can be implemented using pygame functions so can simply be written as a single function.

2.1.4: Place Object

a. They will then be able to select a grid tile and the value for that grid in the array will change, changing its colour to red in order to show this. It will then become not depressed to also highlight this. There are commands within the pygame library for this already so I will use this to create the button. They will then be able to select a grid tile and the value for that grid in the array will change, changing its colour to white in order to show this.

```
TileChanger:
    for row in 10:
        for column in 10:
            colour = WHITE
            if Land[row][column] == 1:
                colour = RED
            pygame.rectangle(screen, colour,
                             [(MARGIN + WIDTH) * column + MARGIN),
                             (MARGIN + HEIGHT) * row + MARGIN,
                             WIDTH,
                             HEIGHT])
```

In my "While not done" procedure I can add a reference to this function so that every second this function is checked 60 times. Allowing the buttons and tiles to update as they are changed by the user. TileChanger will redraw each tile to the correct colour and if one of the tiles' elements in the array Land is changed it will change to red.

Testing for each sprint will be conducted after I have completed that part of code and will be updating a testing table with any bugs and whether that part of the code was successful. This will allow me to adapt to any bugs that I encounter along the way and document these in a table.

2.1.5: Variables and Functions

To give a wider picture of what variables and programming techniques are being used to create my solution I have compiled a list of variables and programming techniques used in sprint 1.

Variable Name	Variable Type	Explanation
screen	Object	Used to create the window that pygame runs in
done	Boolean	Used to show if the user wants to quit
clock	Object	Used to regulate how often the window refreshes
pos	Array	Used to represent the coordinates of where the user clicked in the window
x	Integer	Represents x coordinate of cursor in window
y	Integer	Represents y coordinate of cursor in window

WHITE	Tuple	Used to represent white in RGB colour space
RED	Tuple	Used to represent red in RGB colour space
GRAY	Tuple	Used to represent gray in RGB colour space
WINDOWSIZE	Array	Used to represent the size of the pygame window
WINDOWTITLE	String	Used to represent the title of the pygame window
WIDTH	Integer	Represents the width of each tile in the grid
HEIGHT	Integer	Represents the height of each tile in the grid
MARGIN	Integer	Represents the margin between each tile in the grid
depressed	Boolean	Represents whether a button is pressed
pressed	Integer	Represents which button is pressed
Land	Two-dimensional Array	Represents the grid of tiles

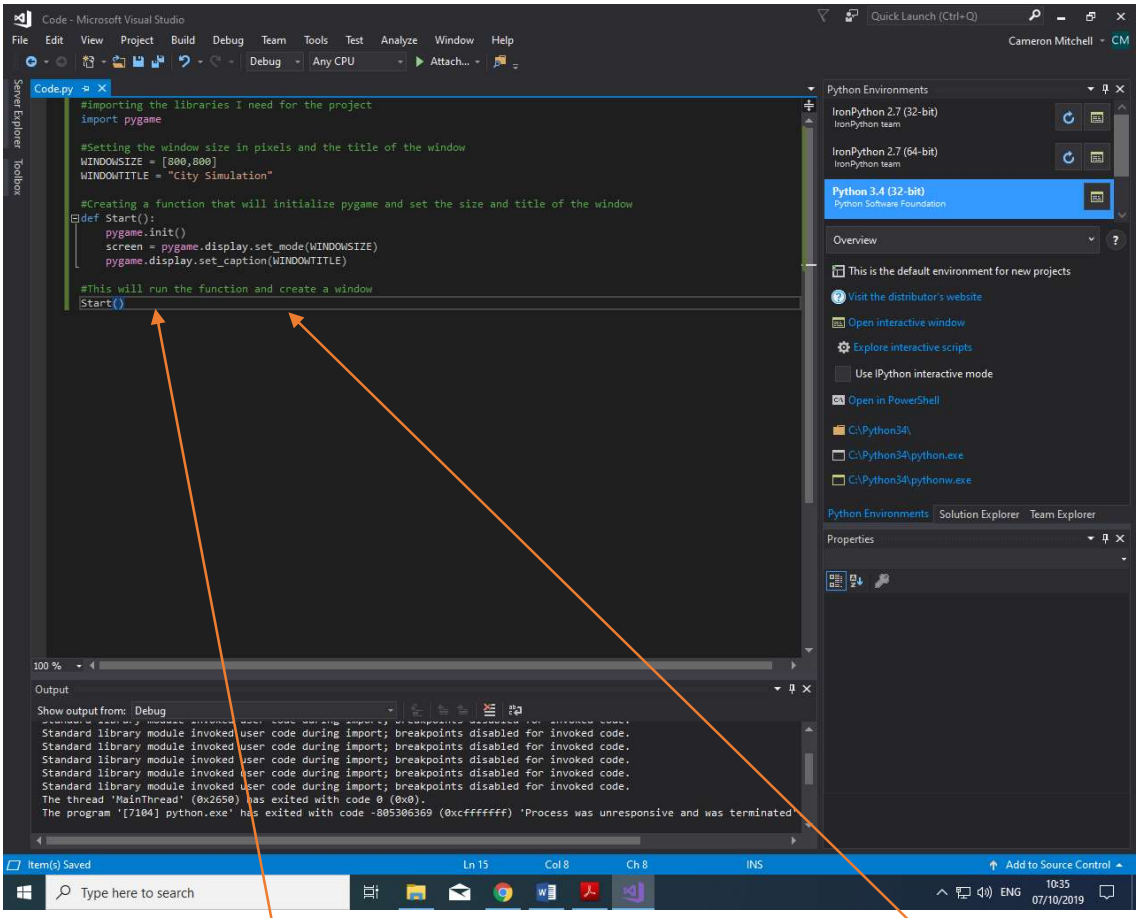
These are some of the programming techniques I should use in order to achieve some of the functions that I need the solution to perform.

Programming Technique	Explanation
Count-controlled Loops	Used to create a certain number of rows and columns in the Land array
	Used to iterate through each element in the Land array when being checked if the tile should have its colour changed
Selection	Used throughout the program to determine things such as whether the user clicked on a button or a tile and which one of these was clicked on
Event-driven programming	I need to use this in order to detect when the user attempts to perform an action inside the generated window

2.2: Development

2.2.1: Game Environment – Window

I will start by creating a basic window of size 800x800, this can be done through the pygame library, where many functions to do this already exist.

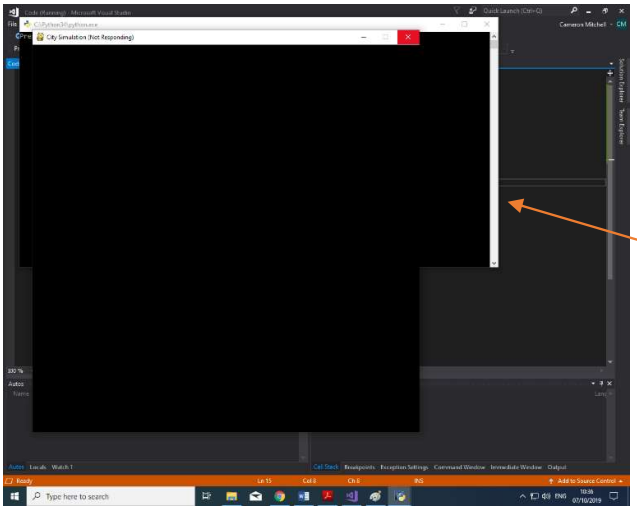


I used the visual studio IDE because it already has many useful python libraries such as pygame when you install python with the visual studio installer. I then imported the pygame library that I will be using to achieve much of my game's features such as the graphical user interface.

I then created two variables called "WINDOWSIZE" and "WINDOWTITLE", these will be used to set the size and title for window that the user will see. Then I defined a function to initialise the pygame library and set the various window attributes and ran this function.

In the end I did not need to use a different variable for the background colour of the window as the default colour is black, so therefore I do not need to change it. The IDE also features an autocomplete feature which sped up development because it automatically completed function and variable names as well as check my syntax and logic as I am developing my code.

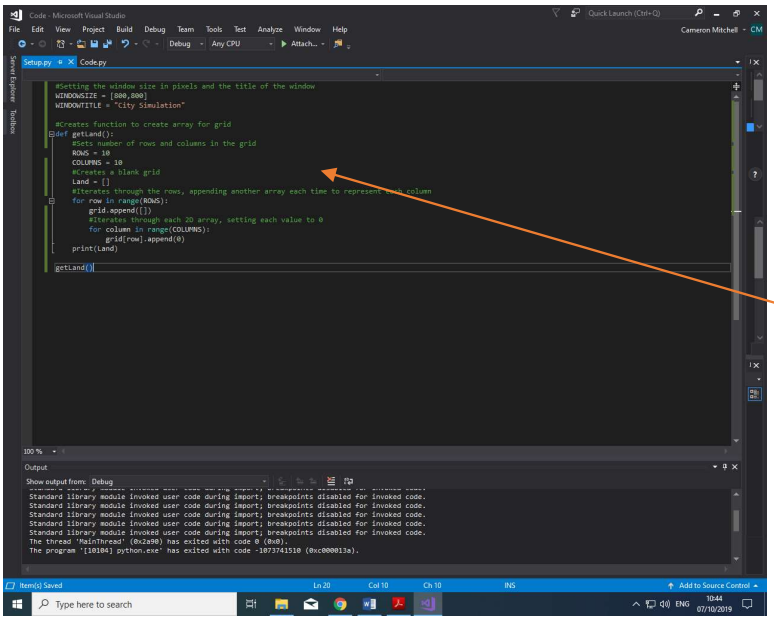
Test reference 1.1



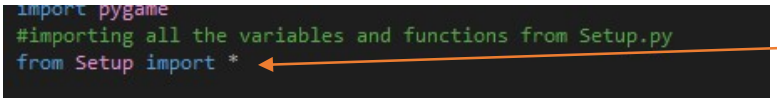
I then tested my solution and the solution ran as expected and produced a window of size 800 by 800 pixels. It seems like it would be big enough for all the features that I need to include in my solution as well as it takes up a large area on a 1920 x 1080 monitor so would generally be enough for the everyday user.

2.2.1: Game Environment – Grid

Next, I will create a function to create the array that I need to store the statuses of each tile in the grid so that I can change them and allow functionality and interactivity to the game.

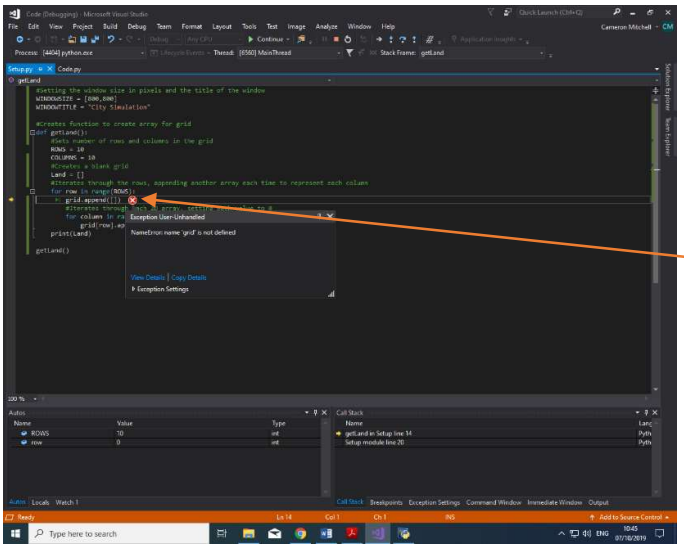


I decided to create a second python file called “Setup.py” and move the WINDOWSIZE and WINDOWTITLE variables into it so I can more easily organise the various functions and variables in the solution. I created a function called “getLand” which iterates to form a two-dimensional array with each element in the array being equal to 0. The array is 10 x 10 in size, equal to the grid so that I can use the array to represent this.

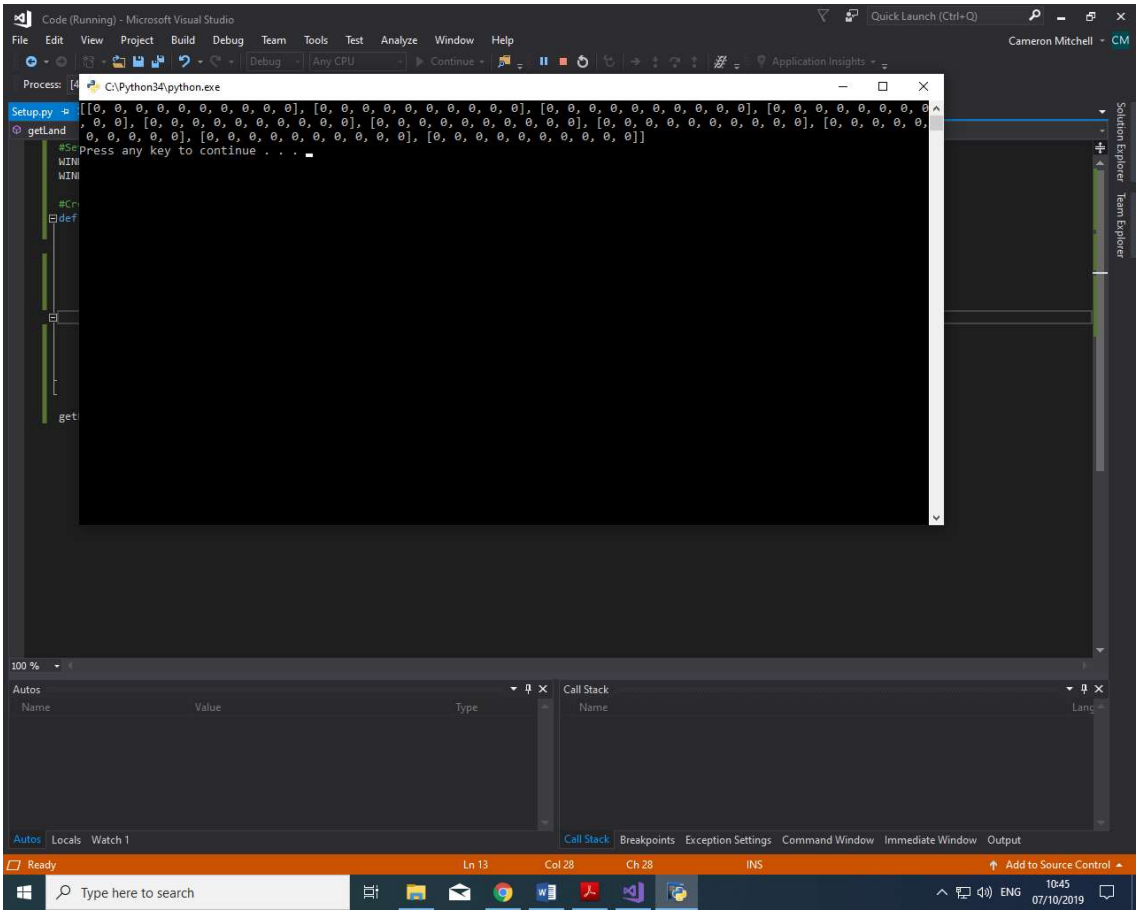


I then linked this python file to the main python one by using the import function to bring all the functions and variables from “Setup.py” to the main file. This allows you to efficiently manage your functions and variables without separating them completely.

Test reference 1.1.2

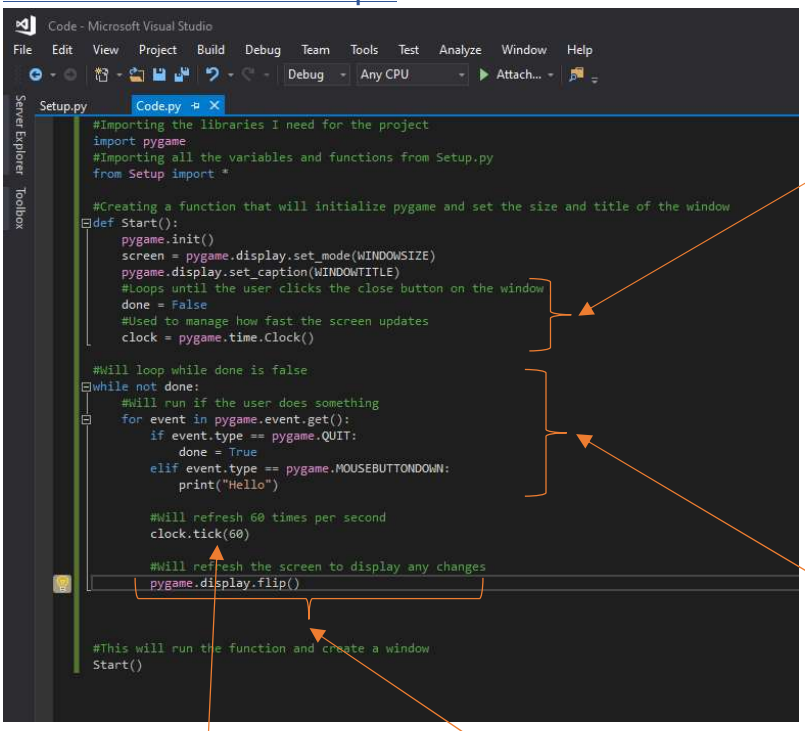


I had a syntax error whereby the variable name of the array I was incrementing did not exist. I fixed this by changing the name of the array to the correct one: grid → Land. After this the solution ran as expected and produced an array of 10 x 10 as required. I printed the variable out in the console in order to show this as I do not have the visual array-backed grid programmed into the solution at this stage.



Using the print function, I was able to print the entire array into the console and see what values each element in the array has. These '0's can be changed to different values to represent different things being placed in that grid.

2.2.1: Detect Grid – User Input

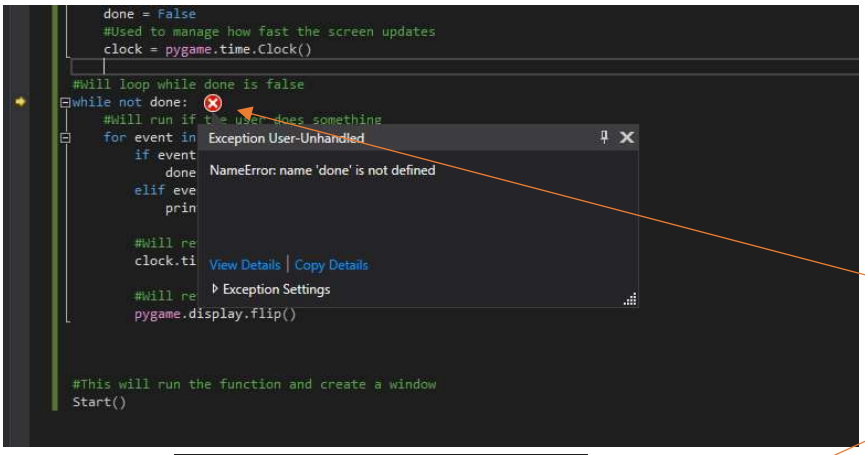


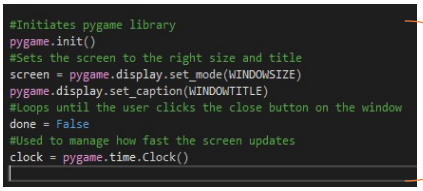
In my "Start" function I added a variable called done which was set to false, this is used to determine when the user quits the game and when to stop running the code. The clock variable is used to set how often the screen refreshes per second for the user when updates to things like tiles are made.

I then created a function that keeps running until the user quits the pygame window. If the user does an action it will determine what type of action the user did. If the user quit the window, the code will stop running. If the user clicks inside the window "Hello" will be outputted to the console. This allows me to detect when the user has clicked anywhere in the window.

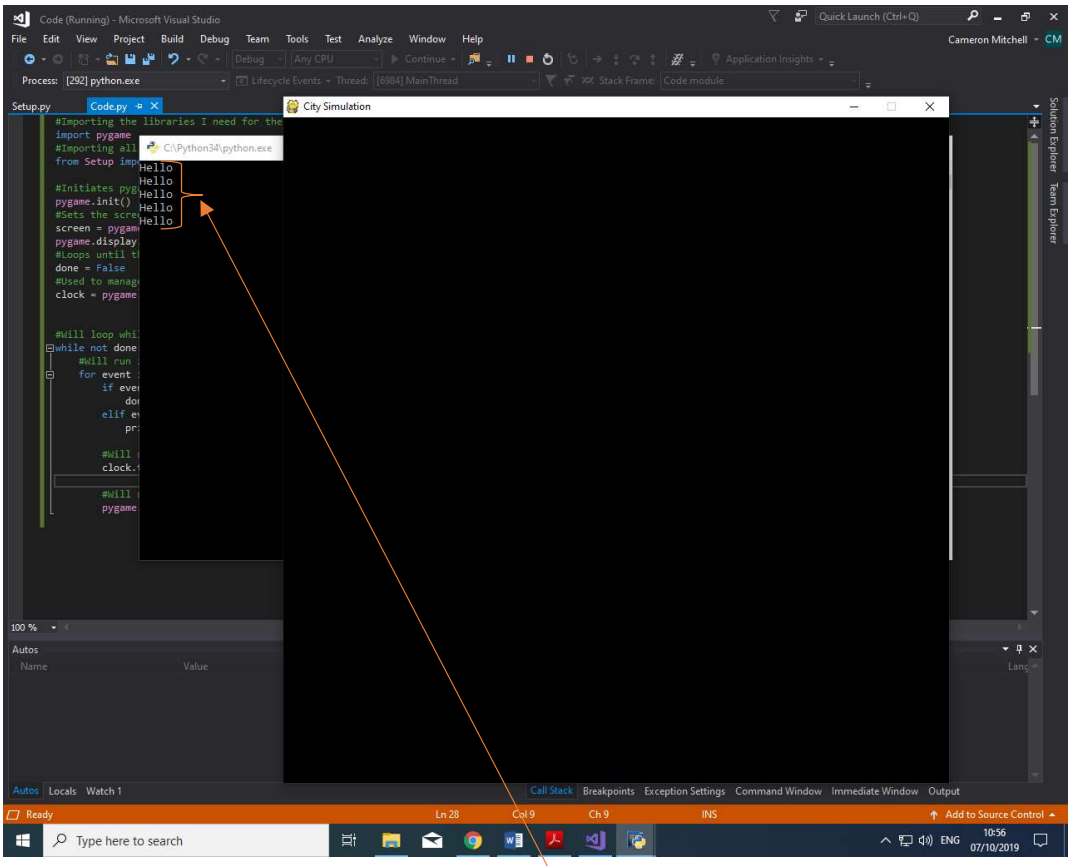
I then set the clock to refresh 60 times a second to update the screen. It will then "flip" the display which will show any updates to the display.

Test reference 1.2.1





I then got an error as I was trying to use a variable that wasn't in the scope of the other function. I then fixed this by moving all the variables from the "Start" function to the main body of the file, outside of any function. This not only fixed the problem but would allow me to access the variables elsewhere if needed.



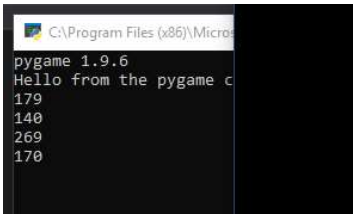
The solution ran as expected and when I tried to click on somewhere in the grid, it produced “Hello” in the console. I attempted this multiple times and the result was the same in each.

2.2.2: Detect Grid – Specific Grid

```
#Will loop while done is false
while not done:
    #Will run if the user does something
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        elif event.type == pygame.MOUSEBUTTONDOWN:
            pos = pygame.mouse.get_pos()
            x = pos[0]
            y = pos[1]
            print(x)
            print(y)
        #Will refresh 60 times per second
        clock.tick(60)

        #Will refresh the screen to display any changes
        pygame.display.flip()
```

I then changed what happens if a user button press is detected. The variable “pos” would get the coordinates of the cursor when the button is pressed inside the window. I would then break this array into variable x and y representing the x and y coordinates in the window. I then would output x and y to see if I can detect where the cursor exactly is.

Test reference 1.2.2



I then tested my solution and it worked as expected, outputting the x coordinate and then y coordinate on the next line. This will allow me to see where the user has clicked and use this to determine which tile in the grid the user has – or hasn’t – clicked on.

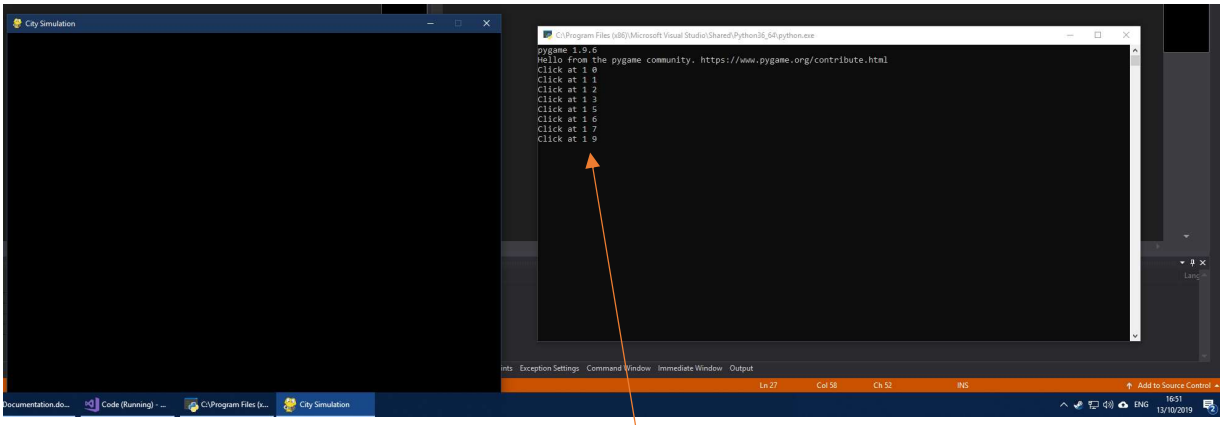
```
        elif event.type == pygame.MOUSEBUTTONDOWN:
            #Gets x and y position of the mouse
            pos = pygame.mouse.get_pos()
            #Creates two variables to make the coordinates easier to work with
            x = pos[0]
            y = pos[1]
            #If the user clicks inside the area for the game, it runs tileDeterminer
            if (x <= 600) and (y <= 600):
                tileDeterminer(x, y)
```

I then changed the code so that if the x and y coordinates where less than or equal to 600 (inside the grid space) they would be passed to a “tileDeterminer” function.

```
Setup.py
tileDeterminer
#Setting the window size in pixels and the title of the window
WINDOWSIZE = [800,800]
WINDOWTITLE = "City Simulation"

#Defining grid sizes
WIDTH = 55
HEIGHT = 55
MARGIN = 5

#Creates function to create array for grid
def getLand():
    #Sets number of rows and columns in the grid
    ROWS = 10
    COLUMNS = 10
    #Creates a blank grid
    Land = []
    #Iterates through the rows, appending another array each time to represent each column
    for row in range(ROWS):
        Land.append([])
        #Iterates through each 2D array, setting each value to 0
        for column in range(COLUMNS):
            Land[row].append(0)

def tileDeterminer(x, y):
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        print("Click at " + str(row) + " " + str(column))
```

In “Setup.py” I added a tile Determiner function that takes x and y as inputs and determines which tile in the grid the user has clicked on. This is done by getting the x and y coordinates and dividing these by the width and height of the block added to the margin. This gives an accurate approximation for which tile the user has clicked on and I can test this using the print function that I have added at the end of the function. I also added the “WIDTH”, “HEIGHT” and “MARGIN” variables so that I can change the size of each tile in the grid, to add more tiles for example, during a later stage of development.

Test reference 1.2.3



Testing my changes, I clicked around the left-half of the grid and in the console the tile that was clicked on was outputted, telling me that my code works and that I can use this to determine which tile the user clicked on with rough accuracy. I can then use this information to implement things like colour changing of clicked tiles in the future. At this stage of development, I did not have the grid visually on the window, but this will be added in the next stage of development to further test this solution.

```
Setup.py
tileDeterminer
    #Defining Colours
    WHITE = (255, 255, 255)
    RED = (255, 0, 0)
```

I then setup two variables called “WHITE” and “RED”. These will be used in the construction of the visual version of the array in order to show the difference to a blank tile and one which has been altered. These are set to be tuples of size 3 that represent the values of red, green and blue in the colour using the RGB colour system.

```
            x = pos[0]
            y = pos[1]
            #If the user clicks inside the area for the game, it runs tileDeterminer
            if (x <= 600) and (y <= 600):
                tileDeterminer(x, y, Land)

        #Checks that the colour for each tile is correct
        for row in range(10):
            for column in range(10):
                #Default tile will be set to white colour
                colour = WHITE
                #If the tile has been clicked on then the colour will change to red
                if Land[row][column] == 1:
                    colour = RED
                #Draws the rectangle for each grid tile
                pygame.draw.rect(screen,
                                 colour,
                                 [(MARGIN + WIDTH) * column + MARGIN,
                                  (MARGIN + HEIGHT) * row + MARGIN,
                                  WIDTH,
                                  HEIGHT])
```

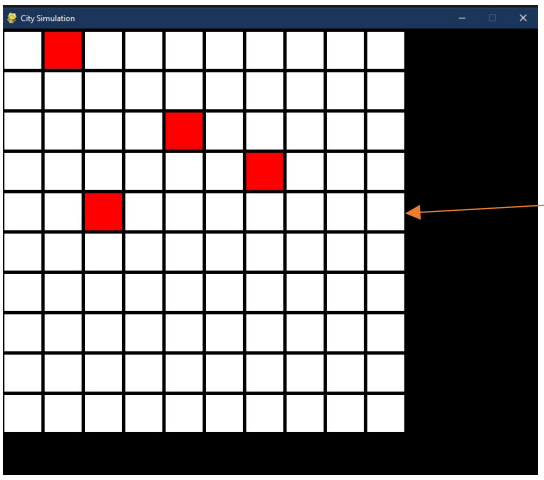
I then added a count-controlled loop which will iterate through the size of the array and will set a variable called "colour" to WHITE and if that element has a 1, indicating it has been changed, its colour will be set to RED. The "pygame.draw.rect" function will then be used to create a rectangle for each of the element in the array. It being the colour corresponding to whether it has been altered.

```
#Generates land variable from setup.py
Land = getLand()
```

```
def tileDeterminer(x, y, Land):
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        Land[row][column] = 1
```

I then created a Land variable to represent the grid, this is inside the main python file so it can be used within the while not done function as well as passed between functions. And changed the tileDeterminer function to change the value of the element in the grid if it is clicked on.

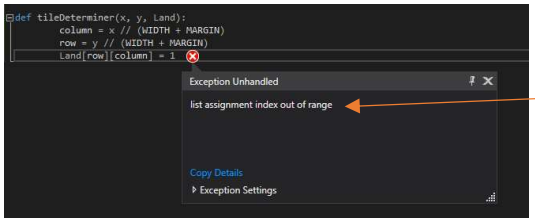
Test reference 1.2.4



I then ran the solution and tried to click a few tiles to see if:

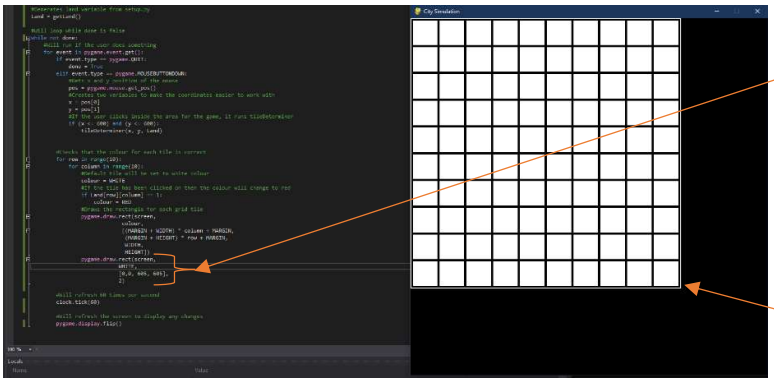
1. The tileDeterminer function would run and change the value of the element that represents the tile that has been clicked
2. Whether the tile changes colour once the value of its element has been changed in the array

I think there is enough evidence to suggest that both conditions were met, and each tile changed colour once it was clicked on. I did, however, encounter a bug whereby if you clicked slightly to the right of the farthest most tile it would crash as it would be trying to store a value in the array position of 10 which, as it is from 0 → 9 is not possible so I got an IndexError.

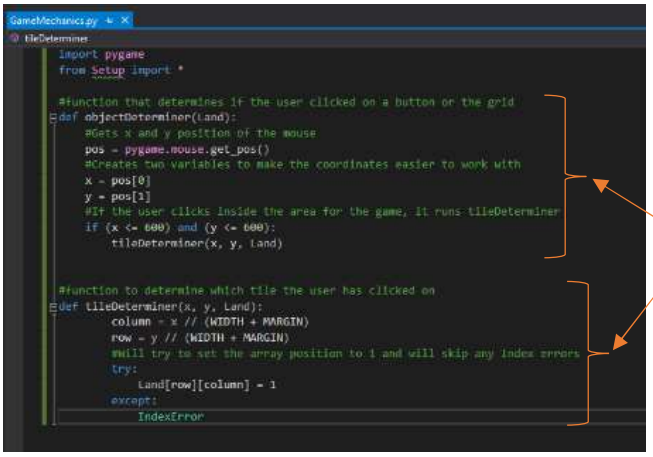



```
def tileDeterminer(x, y, Land):
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        try:
            Land[row][column] = 1
        except:
            IndexError
```

To fix this error I used a try and except system whereby the function would simply do nothing if an index error occurs. This is not the most efficient way to account for an error like this, but as it only effects around 5 pixels each way, it isn't necessary to write anymore code to try to fix it, as I would not place objects such as buttons this within 5 pixels of the grid anyway.



To make the 600 x 600 area where the grid is contained more obvious, I added an outline (including the 5px margin after the last tile) so the player knows where to click more accurately. I made this white to contrast it to the black background and linked it to the tiles. I then tested this, and it produced the expected result.



```
import pygame
from Setup import *

#function that determines if the user clicked on a button or the grid
def objectDeterminer(Land):
    #Gets x and y position of the mouse
    pos = pygame.mouse.get_pos()
    #Creates two variables to make the coordinates easier to work with
    x = pos[0]
    y = pos[1]
    #If the user clicks inside the area for the game, it runs tileDeterminer
    if (x <= 600) and (y <= 600):
        tileDeterminer(x, y, Land)


#function to determine which tile the user has clicked on
def tileDeterminer(x, y, Land):
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        #Will try to set the array position to 1 and will skip any index errors
        try:
            Land[row][column] = 1
        except:
            IndexError
```

To more easily organise my code, I created a third file called “GameMechanics.py” where I moved my tileDeterminer function and created a new file called objectDeterminer. I can then use this to determine what action the user is attempting to perform by using where they clicked and therefore the x and y coordinates to refer to either a button or a tile in the grid.

2.2.3: Create Button

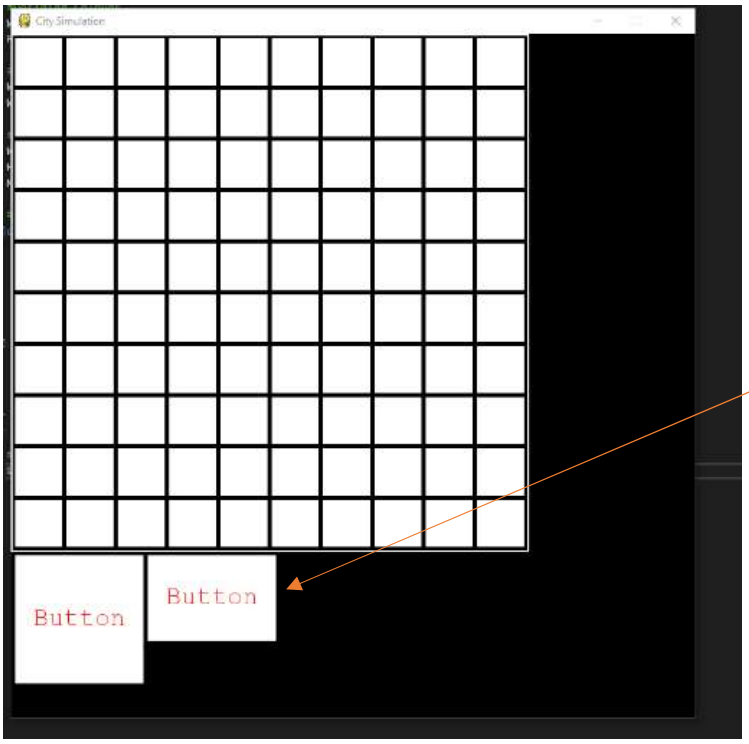
```
import pygame
from Setup import *

#function to draw a button with text
def createButton(screen, buttoncolour, x, y, width, height, text, textcolour):
    #Sets the font for the text that will be added over the button
    font = pygame.font.SysFont("Courier", 30)
    #Draws a rectangle on the window with the size specified
    pygame.draw.rect(screen, buttoncolour, [x, y, width, height])
    #Creates the text based on the text variable and colour variable
    text = font.render(text, True, textcolour)
    #Places the text object on top of the rectangle object and centres it
    screen.blit(text, (x+((width-text.get_rect().width)/2),y+((height-text.get_rect().height)/2)))
```

I then created a new python file called “DrawBandO” which stands for “Draw Buttons and Objects”. I then created a function inside this called “createButton”, this takes in button colour, x coordinate, y coordinate, width, height, text and text colour as inputs. This streamlines the process to create a button as it allows me to create a button wherever with whatever colour, size and text so that I don’t have to constantly write out each button individually.

```
createButton(screen, WHITE, 5, 610, 150, 150, "Button", RED)
createButton(screen, WHITE, 160, 610, 150, 100, "Button", RED)
```

In the main code.py I then called the function in DrawBandO.py by using “from DrawBandO import *”. I then added my requirements as inputs to be put into the function and used this to test to see if my solution achieved what it needed to.

Test reference 1.3.1



When I tested my solution, I found that the buttons of differing dimensions were created:

- One with dimensions (150x150) and starting at (5,610)
- Another with dimensions (150x100) and starting at (160, 610)

This means that my createButton function can be used to create any size and colour of button at any position, this will allow me to easily add more functionality to the game as I can easily develop buttons as I will not have to worry about them being visually represented on the screen. These buttons currently have no functionality, and this will be added in my next stage of development.

2.2.4: Place Object

```
#Setting starting variables for button depressed and button pressed
depressed = False
pressed = 0
```

I created two variables called "depressed" and "pressed" in setup.py. Depressed represents whether a button is depressed and pressed represents that no button is pressed at the start of the program.

```
#Draws border round grid
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
createButton(screen, WHITE, 5, 610, 150, 100, "Button", RED)

#Will loop while done is false
while not done:
    #Will run if the user does something
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        elif event.type == pygame.MOUSEBUTTONDOWN:
            #Gets x and y position of the mouse
            pos = pygame.mouse.get_pos()
            #Creates two variables to make the coordinates easier to work with
            x = pos[0]
            y = pos[1]
            #If the user clicks inside the area for the game, it runs tileDeterminer
            if (x <= 600) and (y <= 600):
                #passes x, y coordinates and the status of which button is pressed
                tileDeterminer(x, y, pressed)
            else:
                #changes the value of pressed and depressed depending on what button is pressed
                depressed, pressed = buttonDeterminer(x, y, depressed, pressed, screen)
        #Updates the tiles in the grid
        tileChecker(screen)
```

I then moved the pos and x and y variables to the main code.py file so that the other functions can much more easily access those variables. I also moved the objectDeterminer to the main code so that the x and y values are passed straight to the tile determiner function in GameMechanics.py to simplify the process so it only must pass to one function, not two. If the grid is not clicked the variables depressed and pressed will be passed to ButtonDeterminer, a new function that handles what happens if the user clicks on a button in DrawBandO.py. The function tileChecker will also be run which will update the colours of the tiles if they change.

At the start of the program the initial button will also be drawn onto the window, as well as the outline of the grid area, this is to setup the button to be changed later. Later buttons and permanent features will also be drawn at the start of the program and will be added to this section to do that.

```python
#function to determine what button has been pressed
def buttonDeterminer(x, y, depressed, pressed, screen):
    #coordinate range for button 1
    if (5 <= x <= 155) and (610 <= y <= 710):
        #if the button is not depressed (already clicked on) it will change to reflect it
        if depressed == False:
            #Redraws the button to represent its depressed state
            createButton(screen, GRAY, 5, 610, 150, 100, "Button", RED)
            #Sets the value of what should be set in the array when a tile is clicked on
            pressed = 1
            #Will change it to false to represent its depressed state
            depressed = True
        #If the button is depressed (already clicked on) it will change to reflect it
        else:
            #Redraws the button to represent its non depressed state
            createButton(screen, WHITE, 5, 610, 150, 100, "Button", RED)
            #Sets the value of what should be set in the array when a tile is clicked on
            pressed = 0
            #Will change it to false to represent its non-depressed state
            depressed = False
    #Will return these variables to update them for the other functions
    return depressed, pressed
```

The buttonDeterminer function is contained in the DrawBand0.py file and is used to determine which button the user has clicked on and what should then happen. It takes in x coordinate, y coordinate, depressed, pressed and screen as inputs. Based on the status of depressed allows the button to be pressed on and off with a visual representation to represent whether the button is down or up (represented by grey and white). If the button is not depressed the button will become depressed and pressed will be set to 1 to represent that button is pressed. If it is true, then the button will be redrawn as white and pressed set to 0 to represent that the button is not pressed. The values of depressed and pressed will then be returned.

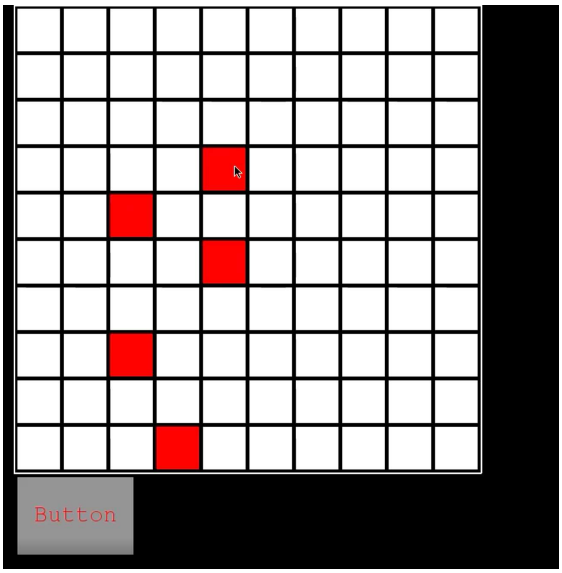
This will allow me to add functionality to buttons and allow the user to do interact differently with the grid based off what buttons they have pressed and where they click on the grid.

```python
#function that checks that the tiles are the right colour
def tileChecker(screen):
    for row in range(10):
            for column in range(10):
                #Default tile will be set to white colour
                colour = WHITE
                #If the tile has been clicked on then the colour will change to red
                if Land[row][column] == 1:
                    colour = RED
                #Draws the rectangle for each grid tile
                pygame.draw.rect(screen,
                                 colour,
                                 [(MARGIN + WIDTH) * column + MARGIN,
                                  (MARGIN + HEIGHT) * row + MARGIN,
                                  WIDTH,
                                  HEIGHT])
```

The tileChecker function in the DrawBand0.py file is just the same procedure as what was in the main code before but put into a function to more easily organise the code. This also allows me to add different colours for what the tile should be based off the button pressed.

Test reference 1.4.1, 1.4.2 and 1.4.3

Link To Video



From the video and this screenshot, you can see that the button's colour is changed from a white to a grey once the user clicks on it. If the user clicks on it again however, it goes back to white to represent it being now not pressed. If the button is pressed and the user clicks on a tile in the grid then the tile turns red, representing that an "object" has been placed at that grid and its element value has been changed. Once the user unclicks the button however and clicks back onto the tile it turns back to white, to represent that the user hasn't pressed anything so the tile resets to its default white state.

2.2.5: Functions and Variables Created

To summarise what different functions were defined during the development, based upon my designed algorithms at the start of this sprint.

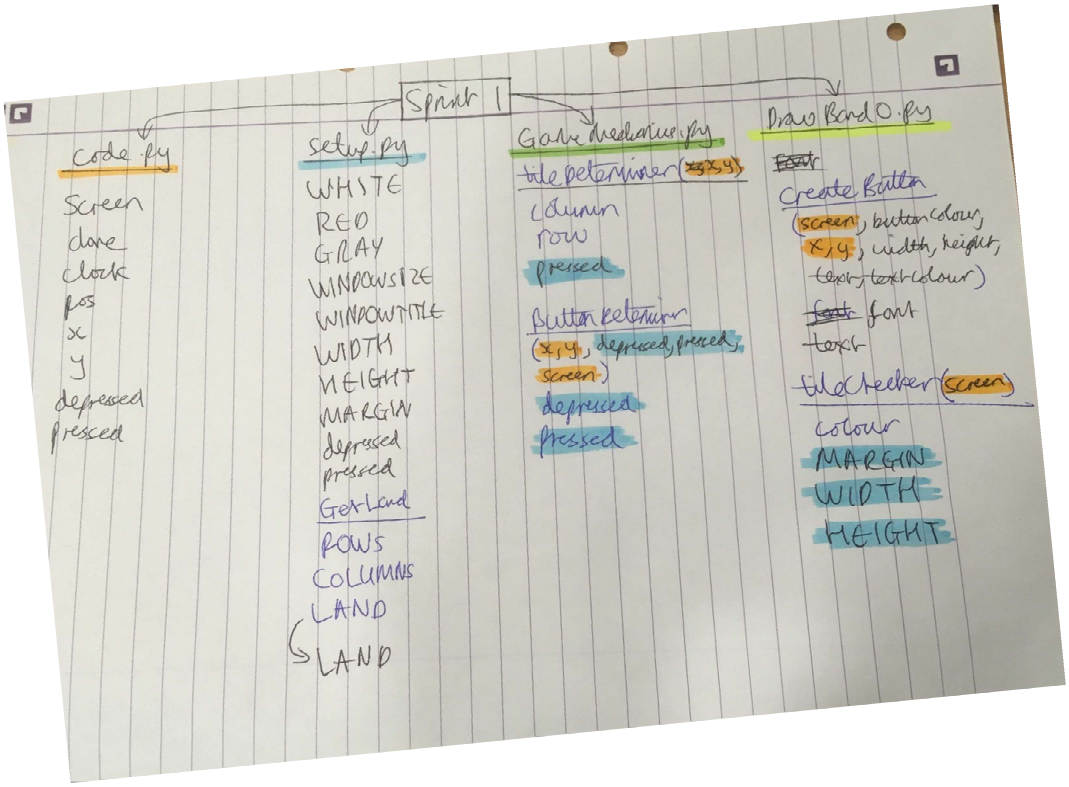
Function Name	Input Variables	Returned Variable	Explanation	File Declared
GetLand	-	Land	Used to create the Two-Dimensional Array Land	Setup.py
tileDeterminer	x, y, pressed	-	Used to determine which tile the user has clicked on and to then change the value of that tile in the Land array	GameMechanics.py
buttonDeterminer	x, y, depressed, pressed, screen	depressed, pressed	Used to determine which button the user has clicked on and to change the state of each button to be either depressed or not and identify which button is pressed if it it.	GameMechanics.py
createButton	screen, buttoncolour, x, y, width, height, text, textcolour	-	Used to create and render each button in the pygame window	DrawBand0.py
tileChecker	screen	-	Used to change the colour of each tile if the value of each corresponding element has changed	DrawBand0.py

These are variables which have not been mentioned in design but used as local variables inside other variables.

Variable	Variable Type	Description	File
column	Integer	Represents the column of the grid the user had clicked on	GameMechanics.py (tileDeterminer)

row	Integer	Represents the row of the grid the user has clicked on	GameMechanics.py (tileDeterminer)
font	Object	Used to load in the font for the text on the buttons	DrawBandO.py (createButton)
text	Object	Used to load the text ready to be rendered on the screen	DrawBandO.py (createButton)
colour	Tuple	Used to determine what colour each tile should be changed to	DrawBandO.py (tileChecker)
ROWS	Integer	Represents the number of rows in the grid	Setup.py (getLand)
COLUMNS	Integer	Represents the number of columns in the grid	Setup.py (getLand)

Diagram to show how variables are shared between different python files, where the variables are highlighted represents where they have been brought in from another file. The highlighting of the variable is corresponding the underlined highlighting under the file name.



2.3: Testing and Evaluation

Sprint 1 has involved test references 1.1.1 – 1.4.3, refer to the testing table for a summary of the tests or use the below table for quick reference.

Feature	Test Reference	Test Question	Test Type	Case	Expected Output	Actual Output	Bug	Bug Fixed	Success
Game Environment	1.1.1	Does a window appear?	Verify	Run Solution	Window of 800 x 800 appears	Window of 800 x 800 appeared	-	-	☑
	1.1.2	Does the grid get created?	Verify	Run Solution	An array will be printed in console	An array was printed in console	Syntax Error, forgot to change the name of the array when appending it	☑	☑
Detect Grid	1.2.1	Does user input get registered?	Experimental	Click in window	"Hello" would be printed in console	"Hello" was printed in console	Scope Error, tried to access variable in other function	☑	☑
	1.2.2	Does coordinates of user input get outputted?	Experimental	Click in window	x and y coordinates would printed in console	x and y coordinates printed in console	-	-	☑
	1.2.3	Does the grid that the user clicked on get outputted in console?	Experimental	Click on left-part of screen	Row and column values for tile outputted to console	Row and column values for tile outputted to console	-	-	☑
	1.2.4	Does the grid change colour once clicked?	Experimental	Click on tiles in grid	Clicked on tiles change colour	Clicked on tiles changed colour	Index Error, tried to save value to a non-existant element	☑	☑
Create Button	1.3.1	Do buttons appear how they are meant to?	Verify	Run Solution	Two buttons of different dimensions should be generated	Two buttons were created of differing dimensions	-	-	☑
Place Object	1.4.1	Does the button change colour to represent it being pressed?	Verify	Press on button	Button turns grey	Button turns grey	-	-	☑
	1.4.2	Does the tile change to red once clicked on after the button has been pressed?	Experimental	Press on tile	Tile turns red	Tile turns red	-	-	☑
	1.4.3	Does the button revert to its white/non-depressed state once the user clicks back on the button?	Verify	Press on button	Button turns white	Button turns white	-	-	☑

Sprint 1 Unit Tests from Testing Table.xlsx

Overall, all the components of the sprint were coded and works as intended. There were a few bugs in 1.1.2 where there was a syntax error wherein the variable name was incorrect and fixed this by changing its name to the correct variable name. Another bug I encountered was in 1.2.1 whereby a variable was not transferred across to another function and it was not global so therefore the variable couldn't be accessed by this other function. The final bug I encountered was in 1.2.4 whereby a non-existent element in the two-dimensional array was trying to be altered and an error occurred as the element did not exist. I then edited this to skip any index errors that the program encounters which seemed to fix the issue with my solution.

This development sprint completed success criteria required for this sprint:

1. Game environment – creates a window in which a map is visible and contains a grid of squares so that the user can eventually place things inside this grid, the grid will be made of white squares.
2. Detect grid – creates a way for the program to know what grid has been pressed on or whether they haven't pressed on any to allow the user to place objects in a grid of their choosing.
3. Create button – creates a button for the user to choose an object to place to allow the user to place an object of their choosing.
4. Place Object – once the user has clicked on the button and on a grid tile the tile will turn black to test the placement is working.

This sprint was overall successful, and I fulfilled the requirements for all my design points for this sprint. Next sprint I will add to this current sprint, adding many more features and functionality to the foundations created by sprint 1. I will add more features like more buttons, more tile colours and a monetary system.

3.0: 2nd Sprint

3.1: Design

In my second sprint I will add much more functionality to my game by allowing the user to exit via a button, create multiple buttons to allow the user to change the colour of tiles to multiple different colours. I will also add a statistics UI and assign wealth and population to each tile to simulate the city's function to the user. I decompose my success criteria into multiple, smaller segments that I will need to implement during this sprint:

1. Exit Button:
    a. Create a button labelled with text "Exit" and assign a quit function to it
2. Create multiple buttons:
    a. Create variables for a range of colours that the user can set the tile to
    b. Create the buttons using my pre-made function and set the labels of these to what colour it sets the tile to.
    c. Add a button that sets the colour of the button to the default to simulate demolition
3. Create statistics UI:
    a. Create variables for "population" and "money"
    b. Create a section of the viewable window where the user can see the value of the population and money variables
4. Assign wealth:
    a. Make each button created cost a certain amount of "money" and then if they place that subtract it from the money variable.
    b. Make a label under the main label of each button to highlight how much money it costs to place that colour
    c. Make sure the user cannot select a button which costs more than the amount stored in the money variable
5. Assign population:
    a. When the user changes the colour of a tile it adds to the population variable
6. Create Utilities:
    a. Create a separate section of the viewable window to be dedicated to "utilities"
    b. Create buttons for different utilities with similar functions as the other buttons
7. Timed Game:
    a. Add a function to refresh the money variable every 60th of a second, the same as the clock variable for the program
8. Maintenance Costs and Taxes:
    a. Each utility building will have a set cost that will reduce the value of the money variable by a certain amount, depending on the utility, per time unit

b. Assign the buttons that aren't the exit or utility buttons to be "residential" buildings that will produce a set increase to money every time unit
c. Assign a certain amount of each utility to be present on each residential building to simulate demand
d. Add utilities to the UI and reduce the amount of money residential buildings generate based on a percentage of what residential buildings have been built and require

3.1.1: Exit Button

I will use my pre-existing function that creates a button that the button can click on to exit the program, ending the game. I will then use the range of coordinates that this button object covers to determine whether the user has clicked on this button and then will exit if this is the intention.

Code.py
```
CreateButton(startx = 610, starty = 5, height = 100, width = 50, WHITE, "Quit")
```

GameMechanics.py
```
elif (610 <= x <= 710) and (5 <= y <= 55):
    quit()
```

In my code.py I will use my CreateButton function to draw the button on the display and add a conditional if/else to determine whether the user clicked on the button each time the user clicks on the screen. It will then run the quit() function which will exit the program.

3.1.2: Create Multiple Buttons

a. First I will create a range of colours that I can use for both my various button text and also the placed colour on the tiles. These are based upon the RGB colour range and I will create a few basic colours to begin with and then expand these as necessary throughout the project.

Setup.py
```
WHITE = (255,255,255)
RED = (255,0,0)
GRAY = (128,128,128)
LIME = (255,0,0)
BLUE = (0,0,255)
YELLOW = (255,255,0)
CYAN = (0,255,255)
```

b. I will then use my pre-defined createButton function to create another button called "Blue"). This will be the same size as the previous button but positioned alongside it. I will then add a conditional if/else statement that will run if the user clicks

Code.py
```
CreateButton(startx = 160, starty = 610, height = 150, width = 100, WHITE, "Blue")

GameMechanics.py
buttonDeterminer:
    if (160 <= x <= 310) and (610 <= y <= 710):
        if depressed == False:
            CreateButton(startx = 160, starty = 610, height = 150, width = 100, GRAY, "Blue")
            depressed == True
        else:
            CreateButton(startx = 160, starty = 610, height = 150, width = 100, WHITE, "Blue")
            depressed == False

tileDeterminer:
    if depressed == True:
        Land[row][column] = 2

DrawBandO.py
tileChecker:
    if Land[row][column] == 2:
        colour = BLUE
```

on the coordinates that represent the button. If the button is depressed and the user clicks on a tile, then the value of that tile in the Land array will change to represent that blue colour that has been selected by changings its stored array value to 2 and not 0 (blank) or 1 (red). When the tiles are then being checked the colour of the tile will be changed to blue if the value of it in the array is blue.

```
Code.py
CreateButton(startx = 315, starty = 610, height = 150, width = 100, WHITE, "Clear")

GameMechanics.py
buttonDeterminer:
    if (315 <= x <= 465) and (610 <= y <= 710):
        if depressed == False:
            CreateButton(startx = 315, starty = 610, height = 150, width = 100, GRAY, "Clear")
            depressed == True
        else:
            CreateButton(startx = 315, starty = 610, height = 150, width = 100, WHITE, "Clear")
            depressed == False

tileDeterminer:
    if depressed == True:
        Land[row][column] = 0
```

c. Then I will create another button using the same function, creating it directly next to the other two buttons. This will set the clicked tile to 0, which will set the colour of the tile to white as if it was not changed from the default tile, effectively acting as a sort of "demolition" button if the tiles represent buildings.

3.1.3: Create Statistics UI

a. In setup.py I will create a variable for both money and population, as these are imported to code.py, I can use these in other functions.

```
Setup.py
int money = 10000
int population = 0
```

b. I will then create a box with the "pygame.draw" function and then within that create text using a new function that will put the text on top of that box. This will represent the area where the user can see the statistics of their city. I will then also add text inside the box that will display the money and population variables to the user, using that same label function.

```
Code.py
pygame.draw(WHITE, 610, 55, 185, 550)
createLabel(625, 65, 155, 30, BLACK, "Statistics")
createLabel(625, 100, 155, 10, BLACK, money)
createLabel(625, 155, 155, 10, BLACK, population)

DrawBandO.py
createLabel(startx, starty, height, width, colour, text):
    pygame.draw(startx, starty, height, width, colour, text)
```

3.1.4: Assign Wealth

```
Setup.py
int redCost = 100
int blueCost = 500

GameMechanics.py
if Land{row][column] == 0:
    if pressed == 1:
        money = money – redCost
        Land[row][column] = pressed
    if pressed == 2:
        money = money – blueCost
        Land[row][column] = pressed
```

a. In setup.py I will create two new variables representing the "costs" of changing the tiles using the buttons. Then in mechanics.py I will add a function so that the money variable decreases when the colour of the tile changes using the buttons.

```
Code.py
createLabel(5, 660, 150, 50, BLACK, redCost)
createLabel(160, 660, 150, 50, BLACK, blueCost)
```

b. By using the label function I can add text to the bottom of the buttons to represent their costs to the user before they have clicked on them.

```
GameMechanics.py
if pressed == 1:
    if money – redCost >= 0:
        if Land[row][column] == 0:
            money = money – redCost
if pressed == 2:
    if money – blueCost >= 0:
        if Land[row][column] == 0:
            money = money - blueCost
```

c. I will then edit the GameMechanics.py to make it so the user cannot change a tile colour unless they can afford it by editing the conditional statements to only allow a tile change if the money variable is higher than the cost to change the tile.

3.1.5: Assign Population

a. I will edit the tilechecker function to change the population variable when the tiles are updated every 60th second and this will allow the user to see how much population is present in the city.

```
DrawBandO.py
tilechecker:
    if Land[row][column] == 1:
        colour = RED
        population = population + 100
    if Land[row][column] == 2:
        colour = BLUE
        population = population + 200
```

3.1.6: Create Utilities

a. I will then draw a box using pygame.draw and inside that create text which signifies that the box is dedicated to "Utilities" where I will include more buttons that will allow the user to change the tiles to different colours that will represent different utilities.

```
Code.py
CreateButton(startx = 630, starty = 650, height = 145, width = 65, BLACK, "Power")
CreateButton(startx = 630, starty = 720, height = 145, width = 65, BLACK, "Water")

Setup.py
int waterCost = 200
int powerCost = 200

GameMechanics.py
if pressed == 4:
    if money – powerCost >= 0:
        if Land[row][column] == 0:
            money = money – powerCost
            Land[row][column] = pressed
if pressed == 5:
    if money – waterCost >= 0:
        if Land[row][column] == 0:
            money = money – waterCost
            Land[row][column] = pressed

DrawBandO.py
if Land{row][column] == 4:
    colour = YELLOW
if Land[row][column] == 5:
    colour = CYAN
```

b. Using my createButton function I will create two buttons inside of the utilities box called power and water. I will also setup two variables called waterCost and powerCost and use these in a modified GameMechanics.py in order to check to make sure the user can afford to buy and place the power and water tiles. Then I have edited the DrawBandO.py to change the colour of the tiles when they are updated to the correct colours if they have been changed.

3.1.7: Timed Game

a. The program already refreshed every 60th seconds through "clock.tick(60)"

3.1.8: Maintenance Costs and Taxes

```
Setup.py
int power = 0
int water = 0
int powerTiles = 0
int waterTiles = 0

DrawBandO.py
tilechecker(population, power, water, powerTiles, waterTiles):
    population = 0
    powerTiles = 0
    waterTiles = 0
    return population, power, water, powerTiles, waterTiles

GameMechanics.py
variableCalc(population, money, power, water, powerTiles, waterTiles):
    money = money – ((powerTiles * 0.5) + (waterTiles * 0.5))
    return money

Code.py
tileChecker(population, power, water, powerTiles, waterTiles)
variableCalc(population, money, power, water, powerTiles, waterTiles)
```

a. I will create four new variables to represent the net power and water but also powerTiles and waterTiles. Then I will edit the DrawBandO.py file to reset the population, powerTiles and waterTiles variables to be reset so when the tiles get refreshed to recount the number of power and water tiles and total population. I have then edited the GameMechanics.py to add a new function variableCalc to calculate the maintenance cost based upon the number of power and water tiles.

```
Code.py
createLabel(625, 210, 155, 10, BLACK, power)
createLabel(625, 265, 155, 10, BLACK, water)
createLabel(10, 615, 585, 30, BLACK, “Residental Buildings”)

GameMechanics.py
money = money + ((population * 0.005) – ((powerTiles * 0.5) + (waterTiles * 0.5)))
```

b. I will edit my Code.py to show the values of power and water variable and add a title above my buttons. I will then edit the GameMechanics.py to calculate money based upon the population (based on the place tiles) and power and water tiles.

```
GameMechanics.py
power = ((powerTiles * 100) – (population * 0.05))
water = ((waterTiles * 1150) – (population * 0.142))
```

c. Edit GameMechanics.py to calculate power and water variables which will be updated and shown to the user based upon the population and power and water tile variable.

d. I will replace power and water in Setup.py with power supply, demand and balance as well as water supply, demand, balance and tax. Then in GameMechanics.py I have calculated the powerBalance, waterBalance and tax based upon the population size and number of tiles and water tiles. This will then calculate how much the money value will increase, or decrease, by when the variables are refreshed.

Setup.py

```
int powerDemand = 0
int powerSupply = 0
int powerBalance = 0
int waterDemand = 0
int waterSupply = 0
int waterBalance = 0
int tax = 0
```

GameMechanics.py

```
powerDemand = population * 0.1
powerSupply = powerTiles * 100
powerBalance = powerSupply – powerDemand
waterDemand = population * 0.142
waterSupply = waterTiles * 350
waterBalance = waterSupply – waterDemand
tax = population * 0.0005
if powerBalance < 0:
    tax = tax * (powerSupply/powerDemand)
if waterBalance < 0:
    tax = tax * (waterSupply/waterDemand)
money = money + tax
```

3.1.9: Variables and Functions

To give a wider picture of what variables and programming techniques are being used to create my solution I have compiled a list of variables and programming techniques used in sprint 2.

Variable Name	Variable Type	Explanation
WHITE	Tuple	Represents the RGB colour values for white
RED	Tuple	Represents the RGB colour values for red
GRAY	Tuple	Represents the RGB colour values for gray
LIME	Tuple	Represents the RGB colour values for lime
BLUE	Tuple	Represents the RGB colour values for blue
YELLOW	Tuple	Represents the RGB colour values for yellow
CYAN	Tuple	Represents the RGB colour values for cyan
CreateButton	Function	A function to create a box with text on top to represent a button the user can click on
Startx	Integer	Represents the start x coordinate

Starty	Integer	Represents the start y coordinate
Height	Integer	Represents the height in pixels
Width	Integer	Represents the width in pixels
depressed	Boolean	Represents the state of buttons that the user has clicked on
money	Integer	Represents the amount of money the user has
population	Integer	Represents the population of the city
redCost	Integer	Represents the amount of money to place a red tile
blueCost	Integer	Represents the amount of money to place a blue tile
waterCost	Integer	Represents the amount of money to place a water tile
powerCost	Integer	Represents the amount of money to place a power tile
Variablecalc	Function	Calculates and returns the new values for money
powerTiles	Integer	Represents the number of power tiles
waterTiles	Integer	Represents the number of water tiles
powerDemand	Integer	Represents the power demand based on population
powerSupply	Integer	Represents the power supply based on the number of power tiles
powerBalance	Integer	Represents the overall power supply/demand
waterDemand	Integer	Represents the water demand based on population
waterSupply	Integer	Represents the water supply based on the number of water tiles
waterBalance	Integer	Represents the overall water supply/demand

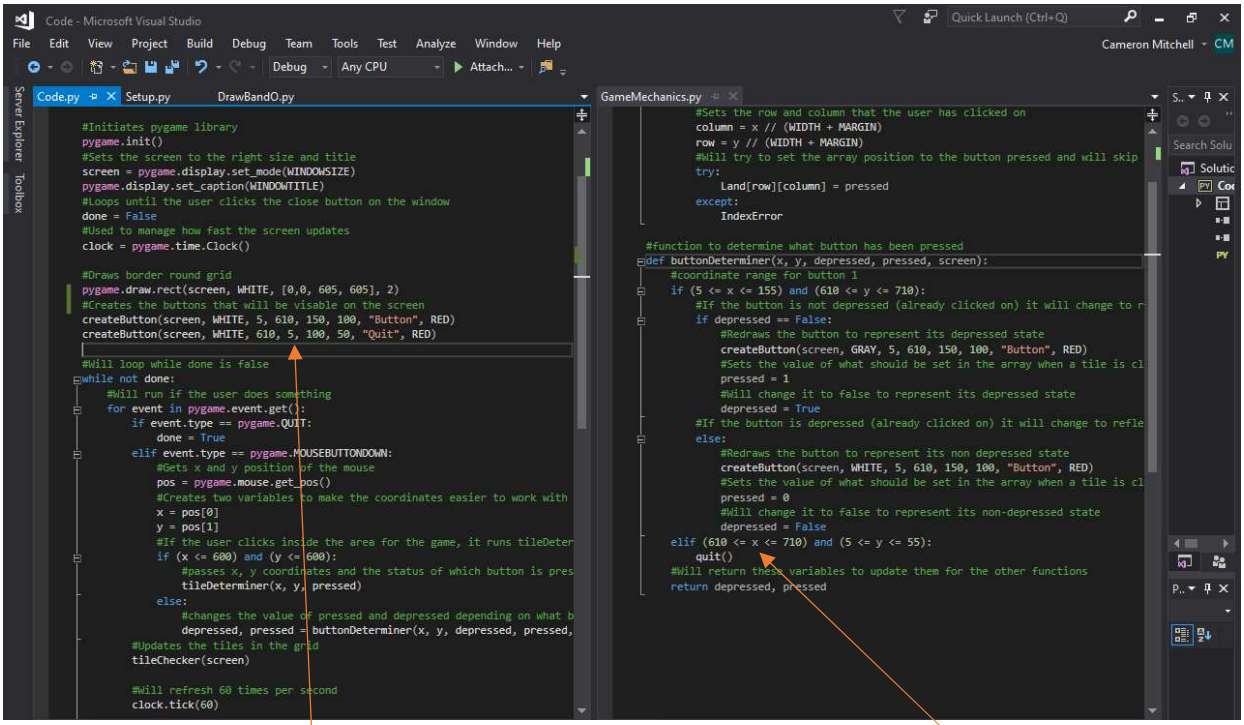
tax	Integer	Represents the money income based on water and power balance

These are some of the programming techniques I should use in order to achieve some of the functions that I need the solution to perform.

Programming Technique	Explanation
Selection	Used throughout the program to determine things such as whether the user clicked on a button or a tile and which one of these was clicked on
Event-driven programming	I need to use this in order to detect when the user attempts to perform an action inside the generated window

3.2: Development

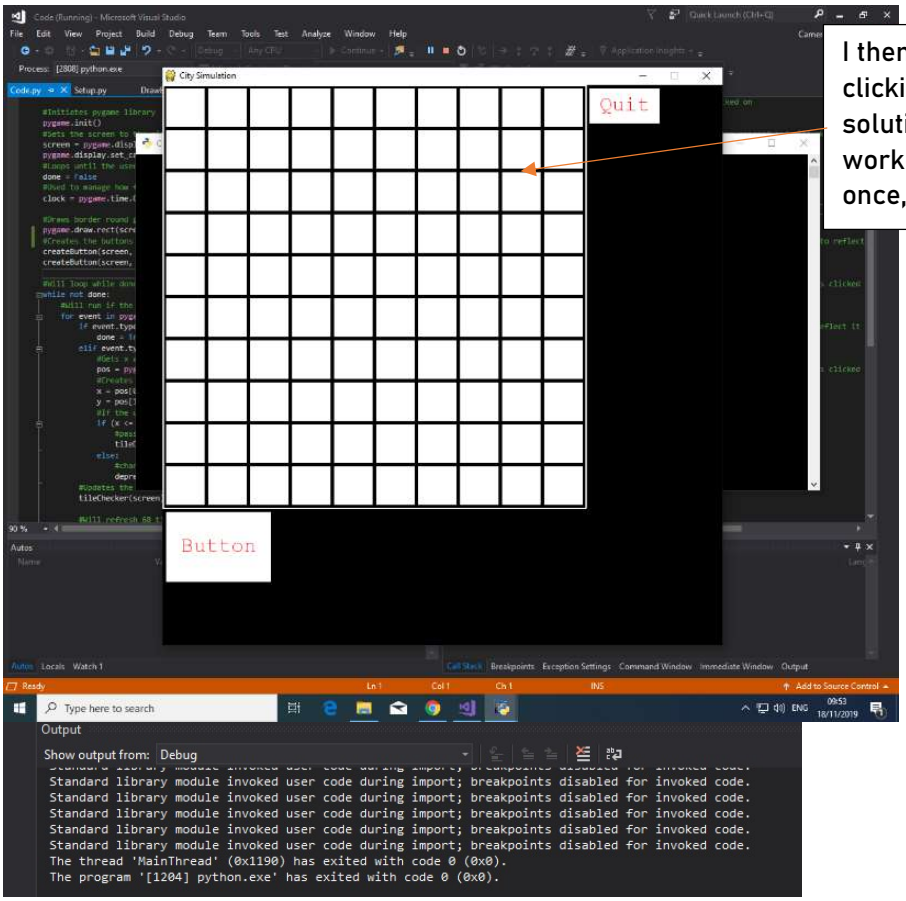
3.2.1: Exit Button



I used my createButton function that I created last sprint to create a new button that has the text “quit”.

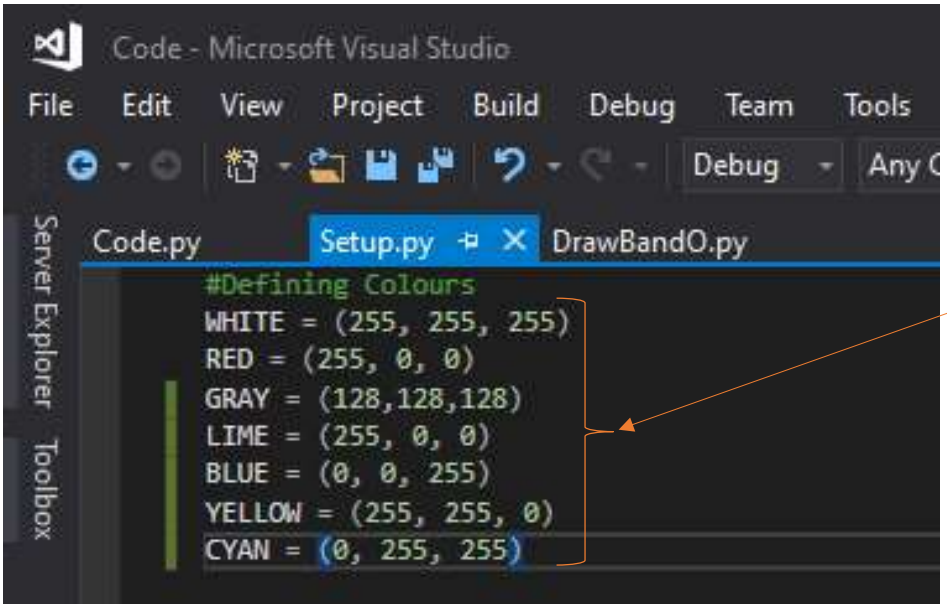
Editing my buttonDeterminer function I set I added a conditional statement so that if the user clicks in the generated box representing the quit button, it will quit the program.

Test Reference 2.1.1



I then tested my solution by clicking on the button once the solution loaded up and it worked by exiting the program once, I clicked on it.

3.2.2: Create Multiple Buttons



I then created variables that represents the RGB colours for different colours I will use during the solution. These have been setup in setup.py to allow them to be used in all other python files due to my import statements created in the previous sprint.

```
#Draws border round grid
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
#Creates the buttons that will be visable on the screen
#Space | Button Colour | x Start | y Start | Width | Height | Button Text | Text Colour
createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED)
createButton(screen, WHITE, 610, 5, 100, 50, "Quit", RED)
createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE)

#Will loop while done is false
while not done:
```

I then used my createButton function to create another button called blue with blue text that will change a tile to blue.

I then changed the depressed variable from a Boolean to an array to represent the states of all the buttons and I can change the size of it to fit the number of buttons I have in the UI.

```
#Setting starting variables for button depressed and button pressed
depressed = [0,0,0]
```

```
#function to determine what button has been pressed
def buttonDeterminer(x, y, depressed, screen):
    alreadypressed = False
    for buttons in depressed:
        if depressed[buttons] == 1:
            alreadypressed = True
    #coordinate range for button 1
    if (5 <= x <= 155) and (610 <= y <= 710):
        #If the button is not depressed (already clicked on) it will change to reflect it
        if alreadypressed == False:
            #Redraws the button to represent its depressed state
            createButton(screen, GRAY, 5, 610, 150, 100, "Red", RED)
            #Will change it to false to represent its depressed state
            depressed[0] = 1
        #If the button is depressed (already clicked on) it will change to reflect it
        else:
            #Redraws the button to represent its non depressed state
            createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED)
            #Will change it to false to represent its non-depressed state
            depressed[0] = 0
    if (160 <= x <= 310) and (610 <= y <= 710):
        #If the button is not depressed (already clicked on) it will change to reflect it
        if alreadypressed == False:
            #Redraws the button to represent its depressed state
            createButton(screen, GRAY, 160, 610, 150, 100, "Blue", BLUE)
            #Will change it to false to represent its depressed state
            depressed[1] = 1
        #If the button is depressed (already clicked on) it will change to reflect it
        else:
            #Redraws the button to represent its non depressed state
            createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE)
            #Will change it to false to represent its non-depressed state
```

In GameMechanics.py I edited the buttonDeterminer function to include an alreadypressed variable which I will use to determine whether the user has already pressed a button. It will then cycle through the depressed array looking for a 1, which represents the button being pressed. Then if this is false and the user has clicked on the drawn box area it will depress one of the buttons. I also added a conditional statement that allows the user to also depress the new blue button, just like the red one. I also edited tileDeterminer so it will set the value of the tile in the Land array to the correct value depending on which button is depressed at that time.

```
from DrawBandO import *

#function to determine which tile the user has clicked on
def tileDeterminer(x, y, depressed):
        #Sets the row and column that the user has clicked on
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        #Will try to set the array position to the button pressed and will skip any Index
        try:
            #Will make it so if you haven't depressed a button you won't clear the tile
            for buttons in depressed:
                if depressed[buttons] == 1:
                    pressed = buttons + 1
            Land[row][column] = pressed
        except:
            IndexError

#function to determine what button has been pressed
def buttonDeterminer(x, y, depressed, screen):
```

Test Reference 2.2.1

Link to Video

As you can see from the video the colours override one another even if the tile already has a colour. To fix this I added a conditional statement to tileDeterminer to make sure the user can only change the colour of a tile if it is 0.

```
try:
    #Will make i
    for buttons
        if depre
            pressed = buttons + 1
    if Land[row][column] == 0:
        Land[row][column] = pressed
except:
    IndexError
```

GameMechanics.py

buttonDeterminer

```
import pygame
from Setup import *
from DrawBandO import *

#function to determine which tile the user has clicked on
def tileDeterminer(x, y, depressed):
        #Sets the row and column that the user has clicked on
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        #Will try to set the array position to the button pressed and will skip any Index errors
        try:
            #Will make it so if you haven't depressed a button you won't clear the tile
            for buttons in range(len(depressed)):
                if depressed[buttons] == 1:
                    pressed = buttons + 1
            if Land[row][column] == 0:
                Land[row][column] = pressed
```

In GameMechanics.py I also edited the tileDeterminer function to include a count-controlled loop to loop through the depressed array and will set the value of the pressed variable to that of the button number set to 1 in the depressed array. It then sets the value of that tile in the Land array to the value of pressed if the value is originally 0 so users can't replace one colour with another.

```
#Draws border round grid
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
#Creates the buttons that will be visable on the screen
#Space | Button Colour | x Start | y Start | Width | Height | Button Text | Text Colour | First | Button Number
createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED, True, 1)
createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE, True, 2)
createButton(screen, WHITE, 315, 610, 150, 100, "Clear", CYAN, True, 3)
createButton(screen, WHITE, 610, 5, 100, 50, "Quit", RED, True, 4)
```

```
for buttons in range(len(depressed)):
    if depressed[buttons] == 1:
        pressed = buttons + 1
if Land[row][column] == 0:
    Land[row][column] = pressed
if pressed == 3:
    Land[row][column] = 0
```

I then created another button with the text "Clear" with cyan text colour. Then in GameMechanics.py I edited the tileDeterminer function to include a condition so that if the user has pressed on the clear button the value of that tile in the Land array is set to 0, effectively clearing it.

```
def getLocations():
    #Sets number of rows and columns in the grid
    ROWS = 800
    COLUMNS = 800
    #Creates a blank grid
    Locations = []
    #Iterates through the rows, appending another array each time to represent each column
    for row in range(ROWS):
        Locations.append([])
        #Iterates through each 2D array, setting each value to 0
        for column in range(COLUMNS):
            Locations[row].append(0)
    return Locations

#Generates land variable
Land = getLand()
Locations = getLocations()
```

To simplify the detection of which button the user has clicked on I created a new function inside setup.py that creates an 800 element array that each have 800 elements inside them, effectively creating an 800 x 800 grid in array form by using the first dimension of the array as the y coordinate and the second dimension the x coordinate. I then create the 800 x800 array using this.

DrawBandO.py

tileChecker

```
import pygame
from Setup import *

#function to draw a button with text
def createButton(screen, buttoncolour, x, y, width, height, text, textcolour, first, button):
    #Sets the font for the text that will be added over the button
    font = pygame.font.SysFont("Courier", 30)
    #Draws a rectangle on the window with the size specified
    pygame.draw.rect(screen, buttoncolour, [x, y, width, height])
    #Creates the text based on the text variable and colour variable
    text = font.render(text, True, textcolour)
    #Places the text object on top of the rectangle object and centres it
    screen.blit(text, (x+((width-text.get_rect().width)/2),y+((height-text.get_rect().height)/2)))
    #If this is the first time that the button is being created
    if first == True:
        #Will assign the values to the location array
        objectLocations(x, y, width, height, button)

#A function to set values in the location array
def objectLocations(x, y, width, height, button):
    #As arrays start from 0, I have subtracted 1 from each variable
    starty = y - 1
    endy = y + height - 1
    startx = x - 1
    endx = x + width - 1
    #This will cycle through the location array to set the correct values
    while endy >= starty:
        tempx = endx
        while tempx >= startx:
            Locations[endy][tempx] = button
            tempx -= 1
        endy -= 1
```

Then in DrawBandO.py I edited my createButton function so that the Locations array is updated to replace the values of the represented coordinates in the array with the corresponding button if it is the first time the button is being created. This is done by passing a Boolean and button number when using the create button function. This is then fed into a new function called objectLocations which updates the array, this is only done if the first variable is set to true however which will only be done if it's the first time the button has been created. If not, the array is not updated.

```
#Creates the buttons that will be visable on the screen
#Space | Button Colour | x Start | y Start | Width | Height | Button Text | Text Colour | F
createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED, True, 1)
createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE, True, 2)
createButton(screen, WHITE, 315, 610, 150, 100, "Clear", CYAN, True, 3)
createButton(screen, WHITE, 610, 5, 100, 50, "Quit", RED, True, 4)
```

I then edited the previous createButton statements to conform with the new inputs required by the function and in Code.py when the buttons are first being created set the first variable to true.

```
#function to determine what button has been pressed
def buttonDeterminer(x, y, depressed, screen):
    alreadypressed = False
    for pressed in range(len(depressed)):
        if depressed[pressed] == 1:
            alreadypressed = True
    #coordinate range for button 1
    print(Locations[y][x])
    if Locations[y][x] == 1:
        #If the button is not depressed (already clicked on) it will change to reflect it
        if alreadypressed == False:
            #Redraws the button to represent its depressed state
            createButton(screen, GRAY, 5, 610, 150, 100, "Red", RED, False, 1)
            #Will change it to false to represent its depressed state
            depressed[0] = 1
        #If the button is depressed (already clicked on) it will change to reflect it
        else:
            #Redraws the button to represent its non depressed state
            createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED, False, 1)
            #Will change it to false to represent its non-depressed state
            depressed[0] = 0
    if Locations[y][x] == 2:
        #If the button is not depressed (already clicked on) it will change to reflect it
        if alreadypressed == False:
            #Redraws the button to represent its depressed state
            createButton(screen, GRAY, 160, 610, 150, 100, "Blue", BLUE, False, 2)
            #Will change it to false to represent its depressed state
            depressed[1] = 1
        #If the button is depressed (already clicked on) it will change to reflect it
        else:
            #Redraws the button to represent its non depressed state
            createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE, False, 2)
            #Will change it to false to represent its non-depressed state
            depressed[1] = 0
    if Locations[y][x] == 3:
        #If the button is not depressed (already clicked on) it will change to reflect it
        if alreadypressed == False:
            #Redraws the button to represent its depressed state
            createButton(screen, GRAY, 315, 610, 150, 100, "Clear", CYAN, False, 3)
            #Will change it to false to represent its depressed state
            depressed[2] = 1
        #If the button is depressed (already clicked on) it will change to reflect it
        else:
            #Redraws the button to represent its non depressed state
            createButton(screen, WHITE, 315, 610, 150, 100, "Clear", CYAN, False, 3)
            #Will change it to false to represent its non-depressed state
            depressed[2] = 0
    if Locations[y][x] == 4:
```

I then edited the buttonDeterminer function in order to use the new locations array. It now only has the determine the value of the Locations array at that specific coordinate, which has been altered when the buttons were originally created.

Test Reference 2.2.2

Link to Video

As you can see from the video, the clear button and function works correctly, as well as the new determining function using the locations array.

3.2.3: Create Statistics UI

```
#Setup of game variables
money = 10000
population = 0
```

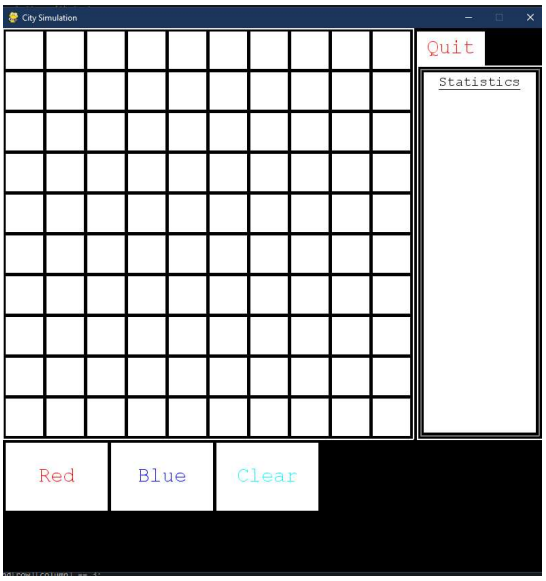
In the setup.py I created two new variables, money and population that will represent the relative amount of money the user has, and population present in the city.

```
def createLabel(screen, labelcolour, x, y, width, height, text, textcolour, fontsize, underline):
    #Sets the font for the text that will be added over the button
    font = pygame.font.SysFont("Courier", fontsize)
    #Draws a rectangle on the window with the size specified
    pygame.draw.rect(screen, labelcolour, [x, y, width, height])
    #Creates the text based on the text variable and colour variable
    font.set_underline(underline)
    text = font.render(text, True, textcolour)
    #Places the text object on top of the rectangle object and centres it
    screen.blit(text, (x+((width-text.get_rect().width)/2),y+((height-text.get_rect().height)/2)))
```

Through copying and modifying my createButton function I can create a function which will draw a label and can underline it through a Boolean variable underline. I have then created three filled rectangular boxes, each smaller and within each other to create a box effect with a border. Then inside the box text using the createLabel function with the text “statistics” to highlight that the box is for statistics to be shown to the user.

```
#Draws border round grids
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
pygame.draw.rect(screen, WHITE, [610,55, 185,550], 2)
pygame.draw.rect(screen, GRAY, [615,60, 175,540], 2)
pygame.draw.rect(screen, WHITE, [620,65, 165,530])
#Creates the buttons that will be visable on the screen
#Space | Button Colour | x Start | y Start | Width | Height | Button Text | Text Colour | First | Button Number
createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED, True, 1)
createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE, True, 2)
createButton(screen, WHITE, 315, 610, 150, 100, "Clear", CYAN, True, 3)
createButton(screen, WHITE, 610, 5, 100, 50, "Quit", RED, True, 4)
#By editing my createButton function I can create one to create a label instead
createLabel(screen, WHITE, 625, 65, 155, 30, "Statistics", BLACK, 20, True)
```

Test Reference 2.2.3



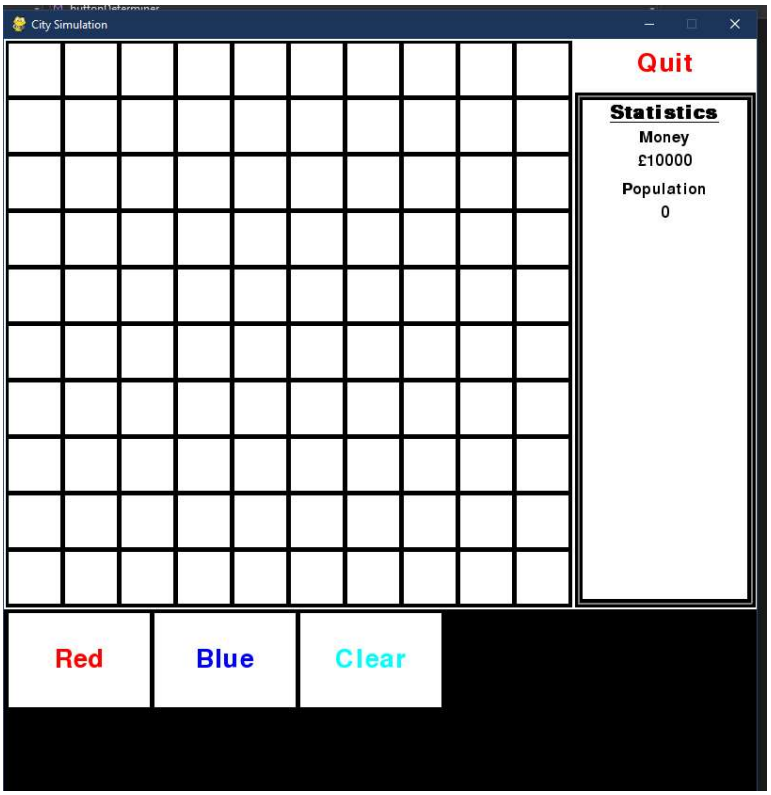
As you can see from the screenshot of the solution, the boxes are drawn correctly with the grey border. The text is also drawn correctly with the underline and the text size.

```
def createLabel(screen, labelcolour, x, y, width, height, text, textcolour, fontsize, underline, bold):
    #Sets the font for the text
    font = pygame.font.SysFont("Roboto", fontsize)
    #Draws a rectangle on the window with the size specified
    pygame.draw.rect(screen, labelcolour, [x, y, width, height])
    #Creates an underline if underline = True
    font.set_underline(underline)
    font.set_bold(bold)
    #Creates the text based on the text variable and colour variable
    text = font.render(text, True, textcolour)
    #Places the text object on top of the rectangle object and centres it
    screen.blit(text, (x+((width-text.get_rect().width)/2),y+((height-text.get_rect().height)/2)))
```

I edited the createLabel function to also include a Boolean variable called bold which, if true will make the text bold. I then used this to create four labels that will represent the population and money to the user.

```
#By editing my createButton function I can create one to create a label instead
#Space | Background Colour | x Start | y Start | Width | Height | Text | Text Colour | Font Size | Underline | Bold
createLabel(screen, WHITE, 625, 65, 155, 30, "Statistics", BLACK, 30, True, True)
createLabel(screen, WHITE, 625, 100, 155, 10, "Money", BLACK, 25, False, False)
createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False)
createLabel(screen, WHITE, 625, 155, 155, 10, "Population", BLACK, 25, False, False)
createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False)
```

Test Reference 2.2.4



You can see from the screenshot that the text created by the createLabel functions work correctly.

3.2.4: Assign Wealth

```
#Setup Button Values
redCost = 100
blueCost = 500
```

In setup.py I then created two variables, redCost and blueCost which represent the amount of money it costs to change the colour of a tile based upon the button pressed and therefore colour chosen to change the tile to.

```
    else:
        if button == 1:
            createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)

def createLabel(screen, labelcolour, x, y, width, height, text, textcolour, fontsize, underline, bold, transparent):
    if transparent == True:
        transparent = -1
    #Sets the font for the text
    font = pygame.font.SysFont("Roboto", fontsize)
    #Draws a rectangle on the window with the size specified
    rectangle = pygame.draw.rect(screen, labelcolour, [x, y, width, height], transparent)
    #Creates an underline if underline = True
    font.set_underline(underline)
    font.set_bold(bold)
    #Creates the text based on the text variable and colour variable
    text = font.render(text, True, textcolour)
    #Places the text object on top of the rectangle object and centres it
    screen.blit(text, (x+((width-text.get_rect().width)/2),y+((height-text.get_rect().height)/2)))
```

I then edited the createLabel function to also include a transparency variable so that I can remove the white background from the text under the main text for the buttons in order to highlight the relative cost for each button below the main text for each button. I also edited the createButton function so that if the button is redrawn then the text that represents the cost of the button is also redrawn. Then I have done this for the red button and displayed the redCost variable below it.

```
createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)
```

```
Code - GameMechanics.py
meMechanics.py
tileDeterminer

nction to determine which tile the user has clicked on
tileDeterminer(x, y, depressed, money, screen):
    #Sets the row and column that the user has clicked on
    column = x // (WIDTH + MARGIN)
    row = y // (WIDTH + MARGIN)
    #Sets values for each pressed button
    costs = [0, redCost, blueCost]
    #Will try to set the array position to the button pressed and will skip any Index errors
    try:
        #Will make it so if you haven't depressed a button you won't clear the tile
        for buttons in range(len(depressed)):
            if depressed[buttons] == 1:
                pressed = buttons + 1
        if pressed == 3:
            Land[row][column] = 0
        if (money - costs[pressed]) >= 0:
            if Land[row][column] == 0:
                money = (money - costs[pressed])
                Land[row][column] = pressed
        elif (money - costs[pressed]) <= 0:
            if pressed == 1:
                createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), RED, 15, False, False, True)
            if pressed == 2:
                createLabel(screen, WHITE, 160, 660, 150, 50, ("£" + str(blueCost)), RED, 15, False, False, True)
    except:
        IndexError
    pygame.draw.rect(screen, WHITE, [625, 120, 155, 15])
    #Space | Background Colour | x Start | y Start | Width | Height | Text | Text Colour | Font Size | Underline | B
    createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False, False)
    return money
```

I then edited the tileDeterminer function so that it includes a new variable called costs that is a one-dimensional array that stores the costs for each button. I then added conditional statements that means that the colour of each tile only changes if the user has enough money to do it. If not, it will redraw the text under the text of the button as red to signify that the user cannot actually afford to change that tile's colour.

```
#Draws border round grids
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
pygame.draw.rect(screen, WHITE, [605,55, 195,550], 2)
pygame.draw.rect(screen, GRAY, [610,60, 185,540], 2)
pygame.draw.rect(screen, WHITE, [615,65, 175,530])
#Creates the buttons that will be visable on the screen
#Space | Button Colour | x Start | y Start | Width | Height | Button Text | Text Colour | First | Button Number
createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED, True, 1)
createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE, True, 2)
createButton(screen, WHITE, 315, 610, 150, 100, "Clear", CYAN, True, 3)
createButton(screen, WHITE, 605, 0, 195, 55, "Quit", RED, True, 4)
#By editing my createButton function I can create one to create a label instead
#Space | Background Colour | x Start | y Start | Width | Height | Text | Text Colour | Font Size | Underline | Bold | Transparent
createLabel(screen, WHITE, 625, 65, 155, 30, "Statistics", BLACK, 30, True, True, False)
#Money Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 100, 155, 10, "Money", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False, False)
#Population Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 155, 155, 10, "Population", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
#Labels Underneath the buttons
createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 160, 660, 150, 50, ("£" + str(blueCost)), BLACK, 15, False, False, True)
```

I then reordered my code in code.py so that it is easier to identify which parts of the code create different parts of the user interface and this will make editing my code easier in the future. I also provided guides in the comments as to what variables need to be passed to each function.

Test Reference 2.2.5

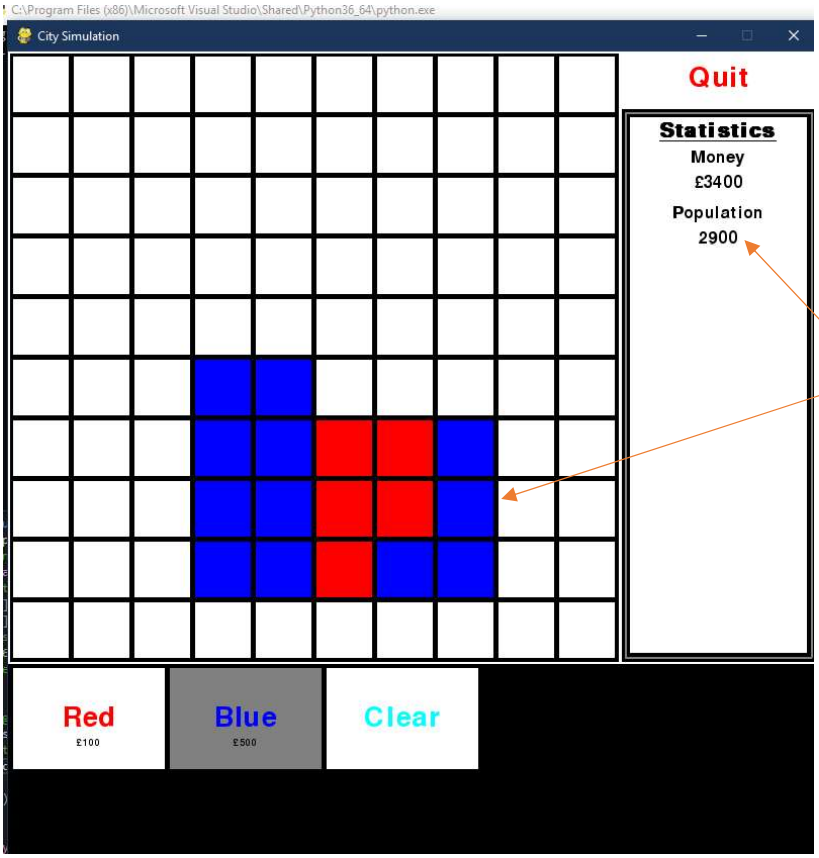
Link to Video

As you can see from the video the variables update and labels redrawn both on the buttons and on the statistics overview as intended based on my code.

3.2.5: Assign Population

```
#function that checks that the tiles are the right colour
def tileChecker(screen, population):
    population = 0
    for row in range(10):
        for column in range(10):
            if Land[row][column] == 0:
                colour = WHITE
            #If the tile has been clicked on then the colour will change to red
            if Land[row][column] == 1:
                colour = RED
                population += 100
            if Land[row][column] == 2:
                colour = BLUE
                population += 200
            if Land[row][column] == 3:
                colour = WHITE
            #Draws the rectangle for each grid tile
            pygame.draw.rect(screen,
                             colour,
                             [(MARGIN + WIDTH) * column + MARGIN,
                              (MARGIN + HEIGHT) * row + MARGIN,
                              WIDTH,
                              HEIGHT])
    pygame.draw.rect(screen, WHITE, [625, 175, 155, 15])
    createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
```

Editing the tileChecker function I added a mechanic which will increase the population variable for each colour in the grid, effectively assigning each tile colour a population value. Then the population label is redrawn to update the value of population for the user and a white background is redrawn to cover up the previous value as well.



As you can see from the screenshot the program runs as expected and the population variable is correctly increasing and is drawn on the user interface correctly. This increases based upon the amount of red (5) and blue (12) tiles that have been drawn which is correct.

3.2.6: Create Utilities

```
#Draws border round grids
#Space | Background Colour | x Start | y Start | Width | Height | Nothing = fill in, 2 = border with no fill
#Border for Main Grid
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
#Border 1 for Statistics Box
pygame.draw.rect(screen, WHITE, [605,55, 195,550], 2)
#Border 2 for Statistics Box
pygame.draw.rect(screen, GRAY, [610,60, 185,540], 2)
#Border 3 for Statistics Box
pygame.draw.rect(screen, WHITE, [615,65, 175,530])
#Border 1 for Utilities Box
pygame.draw.rect(screen, WHITE, [605,605, 195,195], 2)
#Border 2 for Utilities Box
pygame.draw.rect(screen, GRAY, [610,610, 185,185], 2)
#Border 3 for Utilities Box
pygame.draw.rect(screen, WHITE, [615,615, 175,175])
```

I then edited my code.py file and added more comments to signify which function creates which part of the display for the user. I then added 3 more draw functions to create a box with a border for the utilities section, where I will add more buttons.

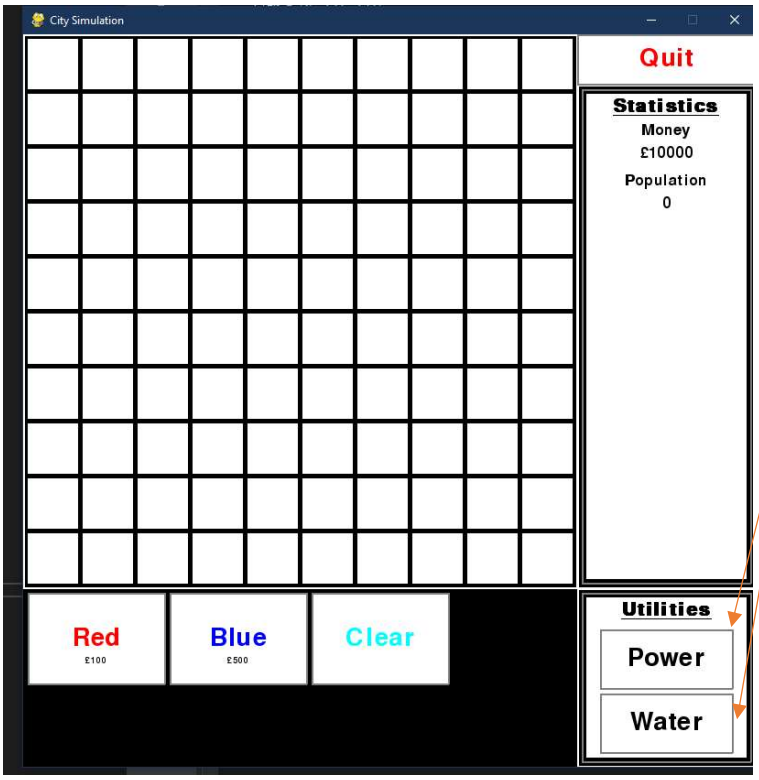
```
#Creates the buttons that will be visable on the screen
#Space | Button Colour | x Start | y Start | Width | Height | Button Text | Text Colour | First | Button Number
createButton(screen, WHITE, 5, 610, 150, 100, "Red", RED, True, 1, True)
createButton(screen, WHITE, 160, 610, 150, 100, "Blue", BLUE, True, 2, True)
createButton(screen, WHITE, 315, 610, 150, 100, "Clear", CYAN, True, 3, True)
createButton(screen, WHITE, 605, 0, 195, 55, "Quit", RED, True, 4, True)
#Utility Buttons
createButton(screen, WHITE, 630, 650, 145, 65, "Power", BLACK, True, 5, True)
createButton(screen, WHITE, 630, 720, 145, 65, "Water", BLACK, True, 6, True)
#By editing my createButton function I can create one to create a label instead
#Space | Background Colour | x Start | y Start | Width | Height | Text | Text Colour | Font Size | Underline | Bold | Transparent
createLabel(screen, WHITE, 625, 65, 155, 30, "Statistics", BLACK, 30, True, True, False)
#Money Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 100, 155, 10, "Money", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False, False)
#Population Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 155, 155, 10, "Population", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
#Labels Underneath the buttons
createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 160, 660, 150, 50, ("£" + str(blueCost)), BLACK, 15, False, False, True)
#Labels for the Utilties Box
createLabel(screen, WHITE, 625, 615, 155, 30, "Utilities", BLACK, 30, True, True, False)
```

I then added 2 new buttons using the createButton function, one of these represents a power building the user can place and then other will be a water building that can be placed. I have set the first variable to be true so that the locations array will be updated when these buttons are drawn. I have also added a label in the box for utilities that says “Utilities” to show the user where the power and water buttons will be located.

```
rawBandO.py
createButton
    import pygame
    from Setup import *

    #function to draw a button with text
def createButton(screen, buttoncolour, x, y, width, height, text, textcolour, first, button, border):
    #Sets the font for the text that will be added over the button
    font = pygame.font.SysFont("Roboto", 40)
    #Draws a rectangle on the window with the size specified
    pygame.draw.rect(screen, buttoncolour, [x, y, width, height])
    #Creates the text based on the text variable and colour variable
    text = font.render(text, True, textcolour)
    #Places the text object on top of the rectangle object and centres it
    screen.blit(text, (x+((width-text.get_rect().width)/2),y+((height-text.get_rect().height)/2)))
    #If this is the first time that the button is being created
    if first == True:
        #Will assign the values to the location array
        objectLocations(x, y, width, height, button)
    else:
        if button == 1:
            createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)
        if button == 2:
            createLabel(screen, WHITE, 160, 660, 150, 50, ("£" + str(blueCost)), BLACK, 15, False, False, True)
    if border == True:
        pygame.draw.rect(screen, GRAY, [x, y , width, height], 2)
```

I also added a border Boolean variable which, when on a conditional statement will run which creates a grey background to the button to help the user determine where the button actually ends by giving a border.

Test Reference 2.2.7



I also added a border Boolean variable which, when on a conditional statement will run which creates a grey background to the button to help the user determine where the button actually ends by giving a border to the buttons in the bottom right.

```
MARGIN = 5

#Setting starting variables for button depressed and button pressed
depressed = [0,0,0,0,0]
colour = WHITE

#Setup of game variables
money = 10000
population = 0

#Setup Button Values
redCost = 100
blueCost = 500
powerCost = 200
waterCost = 200
clearCost = 0

#Creates function to create array for grid
def getLand():
```

I then created 3 new variables to represent the costs of placing a power, water and to clear a tile. I will then use these to display the cost to the user for each button and to subtract this from their money when they do place the object.

```
createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
#Labels Underneath the buttons
createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 160, 660, 150, 50, ("£" + str(blueCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 630, 680, 150, 50, ("£" + str(powerCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 630, 750, 150, 50, ("£" + str(waterCost)), BLACK, 15, False, False, True)
#Labels for the Utilties Box
createLabel(screen, WHITE, 625, 615, 155, 30, "Utilities", BLACK, 30, True, True, False)
```

Using these newly created variables I used my createLabel function in order to display to the user these costs under the buttons which they represent, allowing the user to easily see what buttons cost what amount of money.

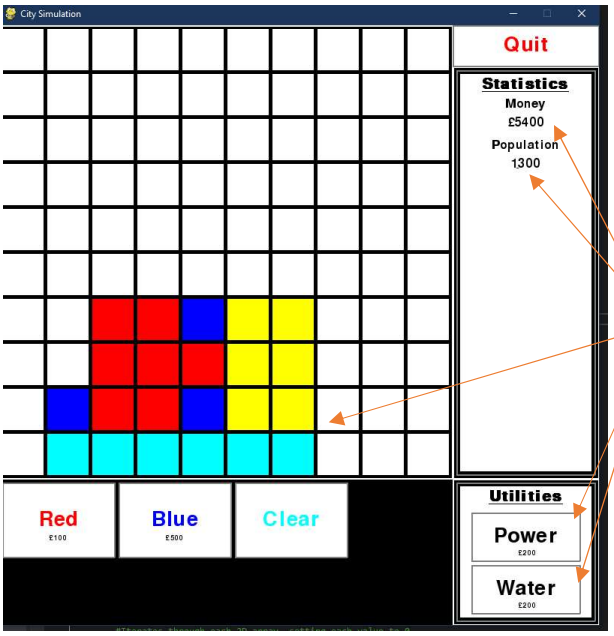
```
ction to determine which tile the user has clicked on
tileDeterminer(x, y, depressed, money, screen, population):
    #Sets the row and column that the user has clicked on
    column = x // (WIDTH + MARGIN)
    row = y // (WIDTH + MARGIN)
    #Sets values for each pressed button
    costs = [redCost, blueCost, clearCost, powerCost, waterCost]
    #Will try to set the array position to the button pressed and will skip any Index errors
    try:
        #Will make it so if you haven't depressed a button you won't clear the tile
        for buttons in range(len(depressed)):
            if depressed[buttons] == 1:
                pressed = buttons + 1
        if pressed == 3:
            Land[row][column] = 0
        elif (money - costs[pressed-1]) >= 0:
            if Land[row][column] == 0:
                money = (money - costs[pressed-1])
                Land[row][column] = pressed
        elif (money - costs[pressed-1]) <= 0:
            if pressed == 1:
                createLabel(screen, WHITE, 5, 660, 150, 50, ("£" + str(redCost)), RED, 15, False, False, True)
            if pressed == 2:
                createLabel(screen, WHITE, 160, 660, 150, 50, ("£" + str(blueCost)), RED, 15, False, False, True)
            if pressed == 4:
                createLabel(screen, WHITE, 630, 680, 150, 50, ("£" + str(powerCost)), RED, 15, False, False, True)
            if pressed == 5:
                createLabel(screen, WHITE, 630, 750, 150, 50, ("£" + str(waterCost)), RED, 15, False, False, True)
    except:
        IndexError
    pygame.draw.rect(screen, WHITE, [625, 120, 155, 15])
    #Space | Background Colour | x Start | y Start | Width | Height | Text | Text Colour | Font Size | Underline |
    createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False, False)
    return money
```

I then added the new variables to the costs array, I have then used this to subtract from the money variable when the object is placed and to redraw the cost onto the button so that the user can again see how much that particular button costs to place.

```
#function that checks that the tiles are the right colour
def tileChecker(screen, population):
    population = 0
    for row in range(10):
        for column in range(10):
            if Land[row][column] == 0:
                colour = WHITE
            if Land[row][column] == 1:
                colour = RED
                population += 100
            if Land[row][column] == 2:
                colour = BLUE
                population += 200
            if Land[row][column] == 3:
                colour = WHITE
            if Land[row][column] == 4:
                colour = YELLOW
            if Land[row][column] == 5:
                colour = CYAN
            #Draws the rectangle for each grid tile
            pygame.draw.rect(screen,
                             colour,
                             [(MARGIN + WIDTH) * column + MARGIN,
                              (MARGIN + HEIGHT) * row + MARGIN,
                              WIDTH,
                              HEIGHT])
        pygame.draw.rect(screen, WHITE, [625, 175, 155, 15])
        createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
```

I then changed the tile checker function so that the tile will go yellow if the power button is used and cyan if the water button is used.

Test Reference 2.2.8



From testing the solution, I find that the labels are drawn correctly and that the money and population variables change as things are placed and the colours of the tiles change accordingly.

3.2.7: Timed Game

This functionality is already built into the solution using the code:

```
pygame.display.flip()
#Will refresh 60 times per second
clock.tick(60)
```

3.2.8: Maintenance Costs and Taxes

```
#Setup of game variabl
money = 10000
population = 0
power = 0
water = 0
powerTiles = 0
waterTiles = 0
```

I then created four more variables. Representing the number of power and water tiles present in the grid as well as the total power and water output of those tiles.

```
#function that checks that the tiles are the right colour
def tileChecker(screen, population, power, water, powerTiles, waterTiles):
    population = 0
    powerTiles = 0
    waterTiles = 0
    for row in range(10):
        for column in range(10):
            #Empty Land
            if Land[row][column] == 0:
                colour = WHITE
            #Red
            if Land[row][column] == 1:
                colour = RED
                population += 100
            #Blue
            if Land[row][column] == 2:
                colour = BLUE
                population += 200
            #Power
            if Land[row][column] == 4:
                colour = YELLOW
                powerTiles += 1
            #Water
            if Land[row][column] == 5:
                colour = CYAN
                waterTiles += 1
            #Draws the rectangle for each grid tile
            pygame.draw.rect(screen,
                             colour,
                             [(MARGIN + WIDTH) * column + MARGIN,
                              (MARGIN + HEIGHT) * row + MARGIN,
                              WIDTH,
                              HEIGHT])
    return population, power, water, powerTiles, waterTiles
```

I then added these new variables as parameters into the tileChecker function. The population and tile variables are all set to 0 and then as the grid array is cycled through it will add to these variables, it then returns these to the main file where it has been called in order for these new variables to be used globally.

```
#Updates the tiles in the grid
population, power, water, powerTiles, waterTiles = tileChecker(screen, population, power, water, powerTiles, waterTiles)
```

```
def variableCalc(screen, population, money, power, water, powerTiles, waterTiles):
    money -= ((powerTiles * 0.5) + (waterTiles * 0.5))
    pygame.draw.rect(screen, WHITE, [625, 120, 155, 15])
    #Space | Background Colour | x Start | y Start | Width | Height | Text | Text Colour | Font Size | Underline | Bo
    createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False, False)
    return money
```

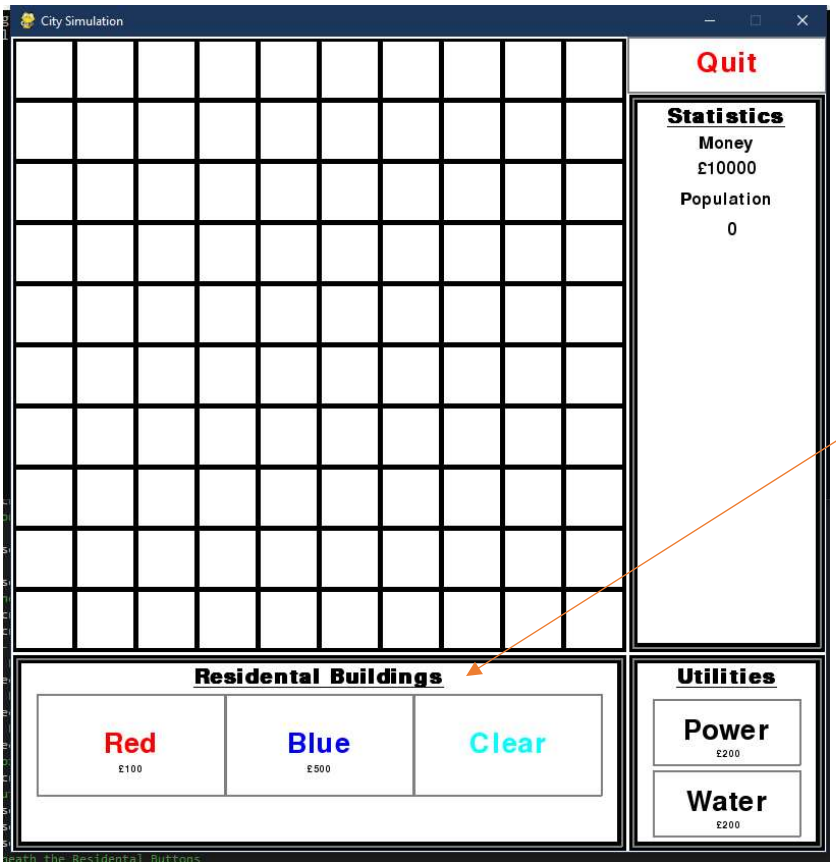
I then created a new function in GameMechanics.py and it takes in many parameters and then uses these to calculate a value for the money variable and then returns this to the main file so this new value of money can be used globally.

```
    population, power, water, powerTiles, waterTiles = tileChecker(screen, population, power
    #Will correct the variables
    money = variableCalc(screen, population, money, power, water, powerTiles, waterTiles)
    #Will refresh the screen to display any changes
    pygame.display.flip()
```

```
pygame.draw.rect(screen, WHITE, [0,0, 605, 605], 2)
#--------------------Statistics Box--------------------
#Border 1 for Statistics Box
pygame.draw.rect(screen, WHITE, [605,55, 195,550], 2)
#Border 2 for Statistics Box
pygame.draw.rect(screen, GRAY, [610,60, 185,540], 2)
#Border 3 for Statistics Box
pygame.draw.rect(screen, WHITE, [615,65, 175,530])
#Statistics Box Label
createLabel(screen, WHITE, 625, 65, 155, 30, "Statistics", BLACK, 30, True, True, False)
#Money Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 100, 155, 10, "Money", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(money)), BLACK, 25, False, False, False)
#Population Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 155, 155, 10, "Population", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
#--------------------Utility Box--------------------
#Border 1 for Utilities Box
pygame.draw.rect(screen, WHITE, [605,605, 195,195], 2)
#Border 2 for Utilities Box
pygame.draw.rect(screen, GRAY, [610,610, 185,185], 2)
#Border 3 for Utilities Box
pygame.draw.rect(screen, WHITE, [615,615, 175,175])
#Utility Box Label
createLabel(screen, WHITE, 625, 615, 155, 30, "Utilities", BLACK, 30, True, True, False)
#Utility Buttons
#Power Button
createButton(screen, WHITE, 630, 650, 145, 65, "Power", BLACK, True, 4, True)
#Water Button
createButton(screen, WHITE, 630, 720, 145, 65, "Water", BLACK, True, 5, True)
#Labels Underneath the Utility Buttons
createLabel(screen, WHITE, 630, 680, 150, 50, ("£" + str(powerCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 630, 750, 150, 50, ("£" + str(waterCost)), BLACK, 15, False, False, True)
#--------------------Residental Box--------------------
#Border 1 for Residental Box
pygame.draw.rect(screen, WHITE, [0,605, 605,195], 2)
#Border 2 for Residental Box
pygame.draw.rect(screen, GRAY, [5,610, 595,185], 2)
#Border 3 for Residental Box
pygame.draw.rect(screen, WHITE, [10,615, 585,175])
#Residental Box Label
createLabel(screen, WHITE, 10, 615, 585, 30, "Residental Buildings", BLACK, 30, True, True, False)
#Residental Buttons
createButton(screen, WHITE, 25, 645, 185, 100, "Red", RED, True, 1, True)
createButton(screen, WHITE, 210, 645, 185, 100, "Blue", BLUE, True, 2, True)
createButton(screen, WHITE, 395, 645, 185, 100, "Clear", CYAN, True, 3, True)
#Labels Underneath the Residental Buttons
createLabel(screen, WHITE, 25, 695, 185, 50, ("£" + str(redCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 210, 695, 185, 50, ("£" + str(blueCost)), BLACK, 15, False, False, True)
#--------------------Quit Button--------------------
createButton(screen, WHITE, 605, 0, 195, 55, "Quit", RED, True, 6, True)

#Will loop while done is false
while not done:
    #Will run if the user does something
    for event in pygame.event.get():
```

I then restructured my code with comments to improve the readability of it, organising it into different sections based upon where the code effects the screen. I then created a border around the three lower left buttons and recentred these within this newly created border with text heading the box that indicates to the user that these buttons are for residential buildings.

Test Reference 2.2.9



As you can see from the screenshot, the border around the boxes is drawn correctly and the heading clearly states to the user that the buttons represent "residential buildings", with the border making it clear to the user where to click.

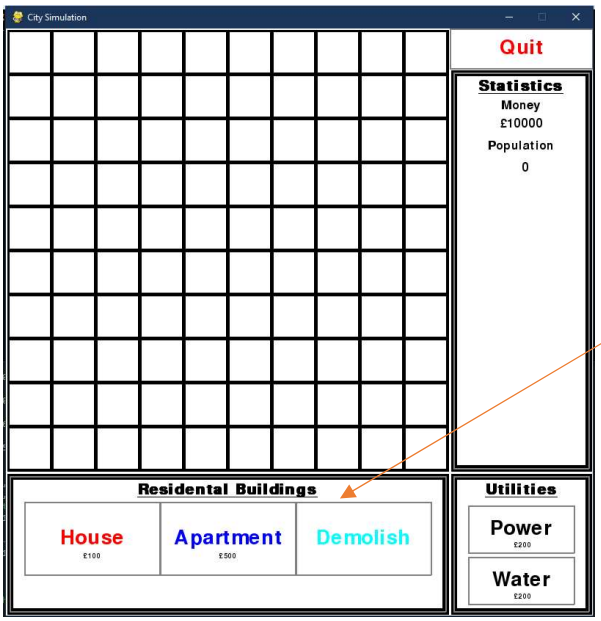
```
createLabel(screen, WHITE, 10, 615, 585, 30, "Residental Buildings", BLACK, 30, True, True, False)
#Residental Buttons
createButton(screen, WHITE, 25, 645, 185, 100, "House", RED, True, 1, True)
createButton(screen, WHITE, 210, 645, 185, 100, "Apartment", BLUE, True, 2, True)
createButton(screen, WHITE, 395, 645, 185, 100, "Demolish", CYAN, True, 3, True)
```

I then renamed all the variables and labels that were "Red" → "House", "Blue" → "Apartment" and "Clear" → "Demolish". This relates the buttons to the various real-world objects, meaning the game is more relatable and realistic as a city building game.

```
#function to determine what button has been pressed
def buttonDeterminer(x, y, depressed, screen):
    alreadypressed = False
    for pressed in range(len(depressed)):
        if depressed[pressed] == 1:
            alreadypressed = True
    #If the button is not depressed (already clicked on) it will change to reflect it
    if alreadypressed == False:
        #Redraws the button to represent its depressed state
        if Locations[y][x] == 1:
            createButton(screen, GRAY, 25, 645, 185, 100, "House", RED, False, 1, True)
        if Locations[y][x] == 2:
            createButton(screen, GRAY, 210, 645, 185, 100, "Apartment", BLUE, False, 2, True)
        if Locations[y][x] == 3:
            createButton(screen, GRAY, 395, 645, 185, 100, "Demolish", CYAN, False, 3, True)
        if Locations[y][x] == 4:
            createButton(screen, GRAY, 630, 650, 145, 65, "Power", BLACK, False, 4, True)
        if Locations[y][x] == 5:
            createButton(screen, GRAY, 630, 720, 145, 65, "Water", BLACK, False, 5, True)
        if Locations[y][x] == 6:
            quit()
        #Will change it to false to represent its depressed state
        depressed[(Locations[y][x] - 1)] = 1
    #If the button is depressed (already clicked on) it will change to reflect it
    else:
        #Redraws the button to represent its non depressed state
        if Locations[y][x] == 1:
            createButton(screen, WHITE, 25, 645, 185, 100, "House", RED, False, 1, True)
        if Locations[y][x] == 2:
            createButton(screen, WHITE, 210, 645, 185, 100, "Apartment", BLUE, False, 2, True)
        if Locations[y][x] == 3:
            createButton(screen, WHITE, 395, 645, 185, 100, "Demolish", CYAN, False, 3, True)
        if Locations[y][x] == 4:
            createButton(screen, WHITE, 630, 650, 145, 65, "Power", BLACK, False, 4, True)
        if Locations[y][x] == 5:
            createButton(screen, WHITE, 630, 720, 145, 65, "Water", BLACK, False, 5, True)
        if Locations[y][x] == 6:
            quit()
```

```
#Setup Button Values
houseCost = 100
apartmentCost = 500
powerCost = 200
waterCost = 200
clearCost = 0
```

Test Reference 2.2.10



As you can see from the screenshot, the labels on the buttons have now been changed to the correct ones after I changed the values for each button.

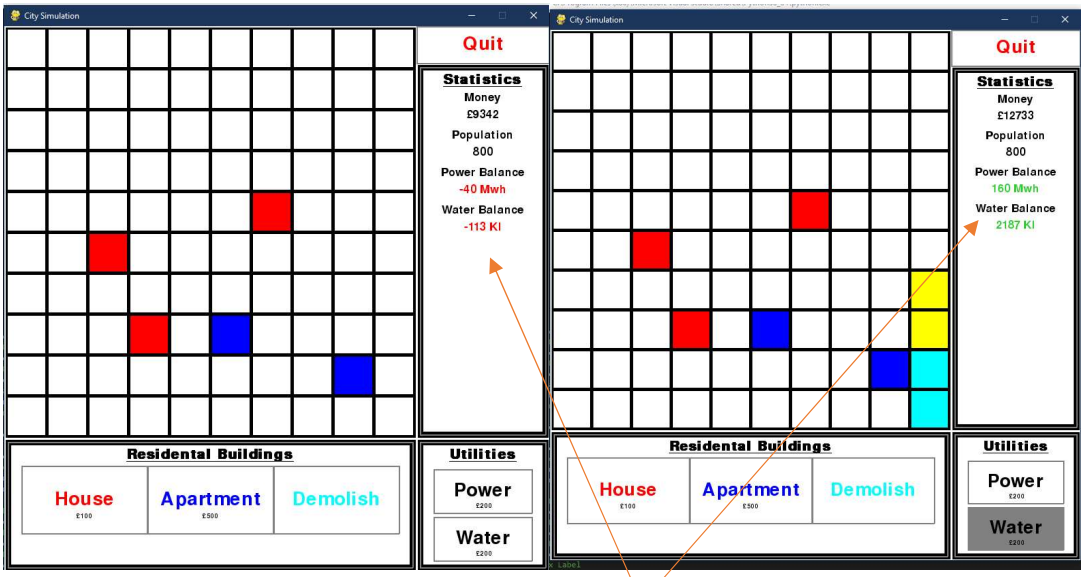
```
#Power Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 210, 155, 10, "Power Balance", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 235, 155, 10, str(power) + " Kwh", BLACK, 25, False, False, False)
#Water Label and Counter in Statistics Box
createLabel(screen, WHITE, 625, 265, 155, 10, "Water Balance", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 290, 155, 10, str(water) + " Kl", BLACK, 25, False, False, False)
```

I then added code in the statistics box section of my main code file and added labels and values for the total balance of both power and water, this should help the user to determine what they need to build more of and also how many new buildings they can build without having to build more utility buildings.

```
def variableCalc(screen, population, money, power, water, powerTiles, waterTiles):
    #Updates money variable
    money += ((population * 0.005) - ((powerTiles * 0.5) + (waterTiles * 0.5)))
    pygame.draw.rect(screen, WHITE, [625, 120, 165, 20])
    createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(math.ceil(money))), BLACK, 25, False, False, False)
    #Updates the population variable
    pygame.draw.rect(screen, WHITE, [625, 170, 165, 20])
    createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
    #Updates the power variable
    power = ((powerTiles * 100) - (population * 0.05))
    pygame.draw.rect(screen, WHITE, [625, 225, 165, 20])
    if power >= 0:
        createLabel(screen, WHITE, 625, 235, 155, 10, str(math.ceil(power)) + " Mwh", GREEN, 25, False, False, False)
    else:
        createLabel(screen, WHITE, 625, 235, 155, 10, str(math.ceil(power)) + " Mwh", RED, 25, False, False, False)
    #Updates the water variable
    water = ((waterTiles * 1150) - (population * 0.142))
    pygame.draw.rect(screen, WHITE, [625, 280, 165, 20])
    if water >= 0:
        createLabel(screen, WHITE, 625, 290, 155, 10, str(math.ceil(water)) + " Kl", GREEN, 25, False, False, False)
    else:
        createLabel(screen, WHITE, 625, 290, 155, 10, str(math.ceil(water)) + " Kl", RED, 25, False, False, False)
    return money
```

I then imported the math library into python so that I can use it to round up or down the variables as required. I updated the money variable so that it works at a rate which I found to be optimum based on the number of water and power tiles. Used the math library to round up the money variable so that it isn't displayed to an unnecessarily high number of decimal places. I then calculated the total power output based on the population and number of power tiles and made it so the value for this is displayed in green if this is positive and red if this is negative. I then used water usage data to calculate the total water balance accurately and then displayed this as green or red depending on the negative or positive value of that variable, using the correct units for each as required.

Test Reference 2.2.11



From the screenshots you can see that the power balance label and variable as well as the water balance label and variable are printed correctly and with the correct colour based on the positive or negative nature of the variable.

```
ecker
        tempx = endx
        while tempx >= startx:
            Locations[endy][tempx] = button
            tempx -= 1
        endy -= 1

#function that checks that the tiles are the right colour
def tileChecker(screen, population, power, water, powerTiles, waterTiles):
    population = 0
    powerTiles = 0
    waterTiles = 0
    for row in range(10):
        for column in range(10):
            #Empty Land
            if Land[row][column] == 0:
                colour = WHITE
            #Red
            if Land[row][column] == 1:
                colour = RED
                population += 50
            #Blue
            if Land[row][column] == 2:
                colour = BLUE
                population += 125
            #Power
            if Land[row][column] == 4:
                colour = YELLOW
                powerTiles += 1
            #Water
            if Land[row][column] == 5:
                colour = CYAN
                waterTiles += 1
            #Draws the rectangle for each grid tile
            pygame.draw.rect(screen,
                             colour,
                             [(MARGIN + WIDTH) * column + MARGIN,
                              (MARGIN + HEIGHT) * row + MARGIN,
                              WIDTH,
                              HEIGHT])
    return population, power, water, powerTiles, waterTiles
```

I then edited the tileChecker function so that each button, including the apartment and house tiles provide less population as this was a bit too much for each, and was a little unrealistic for each tile.

```
def variableCalc(screen, population, money, powerBalance, waterBalance, powerTiles, waterTiles):
    pygame.draw.rect(screen, WHITE, [625, 120, 165, 20])
    createLabel(screen, WHITE, 625, 125, 155, 10, ("£" + str(math.ceil(money))), BLACK, 25, False, False, False)
    #Updates the population variable
    pygame.draw.rect(screen, WHITE, [625, 170, 165, 20])
    createLabel(screen, WHITE, 625, 180, 155, 10, str(population), BLACK, 25, False, False, False)
    #Updates the power variable
    powerDemand = population * 0.1
    powerSupply = powerTiles * 100
    powerBalance = powerSupply - powerDemand
    pygame.draw.rect(screen, WHITE, [625, 225, 165, 20])
    if powerBalance > 0:
        createLabel(screen, WHITE, 625, 235, 155, 10, str(math.ceil(powerBalance)) + " Mwh", GREEN, 25, False, Fa
    else:
        createLabel(screen, WHITE, 625, 235, 155, 10, str(math.ceil(powerBalance)) + " Mwh", RED, 25, False, Fals
    #Updates the water variable
    waterDemand = population * 0.142
    waterSupply = waterTiles * 350
    waterBalance = waterSupply - waterDemand
    pygame.draw.rect(screen, WHITE, [625, 280, 165, 20])
    if waterBalance > 0:
        createLabel(screen, WHITE, 625, 290, 155, 10, str(math.ceil(waterBalance)) + " Kl", GREEN, 25, False, Fal
    else:
        createLabel(screen, WHITE, 625, 290, 155, 10, str(math.ceil(waterBalance)) + " Kl", RED, 25, False, False
    #Calculates Tax Income
    tax = (population * 0.0005)
    #Calculates Penalty for not fulfilling demand
    if powerBalance < 0:
        tax = tax * (powerSupply/powerDemand)
    if waterBalance < 0:
        tax = tax * (waterSupply/waterDemand)
    #Updates money variable
    money += tax
    #Returns the money variable
    return money
```

To make it so that the amount of money the user earns or loses per second is based on the power and water balance, I edited the variableCalc function. It calculated powerDemand based on population and powerSupply based on the number of power tiles placed. It then uses these to find the difference between them, if the demand is greater than the supply this value will be negative. I also did the same with waterDemand, waterSupply and waterBalance but with different, more realistic values. A tax variable is then calculated that's based upon the total population; this represents the user's income. Then this tax variable is then changed depending on if either the water or power balance variables are negative, the fraction of demand fulfilled then this amount of tax is added to the money variable. I also increase the cost of the house and apartment and made the power and water tiles so expensive as to compensate for the maintenance costs of the power and water facilities in one purchase.

```
#Setup Button Values
houseCost = 1000
apartmentCost = 5000
powerCost = 2500
waterCost = 3000
clearCost = 0
```

Test Reference 2.2.12

Link to Video

From the video you can see that the variables, labels and buttons are redrawn and update with time and that the variables are rounded up to help the user to more easily see how much money they have.

3.2.9: Functions Created

To summarise the functions created throughout the development of this sprint I will put these into the table below:

Function Name	Input Variables	Returned Variable	Explanation	File Declared
createLabel	Screen, labelcolour, x, y, width, height, text, textcolour, fontsize, underline, bold, transparent	-	Used to draw text on the screen of the game.	DrawBand0.py
objectLocations	x, y, width, height, button	-	Used to populate the locations array to keep track of where each button is drawn	DrawBand0.py
tileChecker	Screen, population, power, water, powerTiles, waterTiles	Population, power, water, powerTiles, waterTiles	Used to draw each tile and determine what the population and power and water tiles should be	DrawBand0.py
variableCalc	Screen, population, money, powerBalance, waterBalance, powerTiles, waterTiles	money	Used to calculate the money variable	GameMechanics.py
getLocations	-	Locations	Used to create the Locations array	GameMechanics.py

3.3: Testing and Evaluation

Sprint 2 has involved test references 2.1.1 – 2.2.12, refer to the testing table for a summary of the tests or use the below table for quick reference.

Exit Button	2.1.1	Does the program quit if the user clicks on the quit button?	Verify	Press on quit button	Program will quit	Program quits	-	-	☑
Create Multiple Buttons	2.2.1	Do the two buttons work as intended?	Experimental	Press on buttons and tiles	Tiles will go correct colour based upon the the but	Tiles turn correct colour	Colour overrides preivous colour even if it is not white	☑	☑
	2.2.2	Does the clear button work?	Experimental	Press on button and tile	Tiles will be cleared of their colour	Tiles are cleared of their colour	-	-	☑
Create Statistics UI	2.2.3	Does the box and text get drawn correctly?	Verify	Run Solution	Boxes and text drawn in window	Boxes and text drawn in window	-	-	☑
	2.2.4	Does the text get produced correctly?	Experimental	Run Solution	Text Produced correctly	Text Produced correctly	-	-	☑
Assign Wealth	2.2.5	Do the labels get drawn and redrawn correctly?	Experimental	Press buttons and tiles	Labels drawn and redrawn correctly	Labels drawn and redrawn correctly	-	-	☑
Assign Population	2.2.6	Does the population variable increase correctly?	Experimental	Press buttons and tiles	Population variable increases and text redrawn	Population variable increases and text redrawn	-	-	☑
Create Utilities	2.2.7	Do the boxes and button boxes get drawn correctly?	Verify	Run Solution	Boxes and button boxes drawn correcly	Boxes and button boxes drawn correcly	-	-	☑
	2.2.8	Do the labels get drawn and redrawn correctly?	Experimental	Press buttons and tiles	Labels drawn and redrawn correctly	Labels drawn and redrawn correctly	-	-	☑
laintenance Costs and Taxe	2.2.9	Do the new boxes and borders get drawn correctly?	Verify	Run Solution	Boxes and borders drawn correctly	Boxes and borders drawn correctly	-	-	☑
	2.2.10	Do the buttons display the new, correct text?	Verify	Run Solution	New text on the buttons displayed correctly	New text on the buttons displayed correctly	-	-	☑
	2.2.11	Do the new variables and labels get drawn correctly?	Experimental	Press on buttons and tiles	New variables and labels drawn correctly	New variables and labels drawn correctly	-	-	☑
	2.2.12	Do the variables, labels and buttons update correctly?	Experimental	Press on buttons, tiles and label	Buttons, variables and tiles update and drawn cor	Buttons, variables and tiles update and drawn correctly	-	-	☑

Sprint 2 Unit Tests from Testing Table.xlsx

Overall, all the components of the sprint were coded and works as intended. There was a bug in 2.2.1 where there was a programming error wherein the colour of each tile would be overridden even if it was not set as white. I also had to draw extra white boxes atop each variable drawn within the statistics box or else the text would just lay over the top of each other and make it unreadable.

This development sprint completed success criteria required for this sprint:

1. Exit Button – create a button to allow the user to quit the game.
2. Create multiple buttons – create buttons that allow the user to select what colour they want to place; this will make sure the user can switch between the various types of buildings that they want to place.
3. Create statistics UI – Add a box to allow the user to see their "money" and "population", this will allow the user to see what types of buildings they can build.
4. Assign wealth – Assign a wealth to each button and set the money to some figure greater than this and don't allow the user to place that object if they don't have enough money and display this in the UI.
5. Assign population – Assign a population to placed objects and then display this in the UI.
6. Create Utilities – Create/Assign buttons for each of the utilities in the game, the amount of these required will be determined off the population size.
7. Timed Game – Create a clock which refreshes money every half-second so that the game seems fluid.
8. Maintenance Costs and Taxes – Create a system whereby the utilities reduce the money variable by a set amount per utility per 15 secs, and each residential building gives you a set amount of money per 15 secs with the intention to act as a sort of tax and that you have to keep the money stable.

This sprint was overall successful, and I fulfilled the requirements for all my design points for this sprint. Next sprint I will add to this current sprint, adding many more features and functionality to the foundations created by sprint 2. I will add more

specialised art to each tile instead of just solid colours and create a road system whereby each residential building will need to be built next to a road to generate money.

4.0: 3rd Sprint

4.1: Design

In my last sprint I added much of the game's functionality into the program and developed many of the interactive features that the user can interact with in the game. This sprint I will focus on the final parts of the game development such as a road system, a proper aesthetic, wealth system and a way to upgrade the various buildings in order to expand the complexity of the game. I will base my development off my success criteria:

1. Check Road – Checks the placed object is next to a road and if not, then the user is unable to place the desired object.
    a. Create new buttons and labels for the roads and assign them a colour
    b. Determine where buildings are placed around the roads and if you can place them there
2. Assign proper aesthetic – Create a system whereby empty tiles are green to represent grass and residential and utility buildings have different colours based on their purpose to allow the user to more easily see what services they have and what they need.
    a. Create images for each of the buildings
    b. Add these to the grid when the buildings are built
3. Add wealth-system – Create 3 types of residential housing, $, $$, $$$ to illustrate the various levels of wealth of each house. The amount of services required will increase as you go from $ to $$$ as to increase the difficulty. Have these assigned different colours or graphics.
    a. Create images for each level of residential building
    b. Allow the user to upgrade each type of residential building
4. Save Game – Add an ability for the user to save their game to resume it another time through saving the grid, money and population in a text file.
    a. Create way of saving the game in a file or array
5. Default Map – Add a default/new-game grid, money and population to load when a new game is started.
6. Main Menu – Create a start menu that comes up when you load the program that will allow the user to select load save game and start new game.
    a. Allow the user to start a game based on buttons that they press

4.1.1: Check Road

```
Setup.py
roadCost = 500

Code.py
createButton(screen, WHITE, 625, 525, 145, 50, “Road”, BLACK, True, 6, True)
createLabel(screen, WHITE, 625, 550, 150, 35, (“£” + str(roadCost)), BLACK, False, False, True)

GameMechanics.py
if Locations[y][x] == 6:
        createButton(screen, GRAY, 625, 525, 145, 50, “Road”, BLACK, False, 6, True)

If pressed == 6:
        createLabel(screen, WHITE, 625, 550, 150, 35, (“£” + str(roadCost)), RED, 15, False, False, True)
```

I can add a roadCost variable to the Setup.py file so I can use it to reduce the money variable when I build a road. I then can add a road button and label to represent the cost to place a road. I can then change the locations array so that it will store the x and y coordinates that represent the x and y values for the road buttons. I can then change the tileDeterminer function so that the cost of placing a road can turn red if the user cannot afford it.

```
GameMechanics.py
roadConditions = [Land[row][column-1], Land[row][column+1], Land[row-1][column-1], Land[row-1][column+1], Land[row+1][column], Land[row+1][column-1], Land[row+1][column+1]]

If (Land[row][column] == 0) and ((6 in roadConditions) or (pressed == 6)):
        money -= costs[pressed-1]
        Land[row][column] = pressed
```

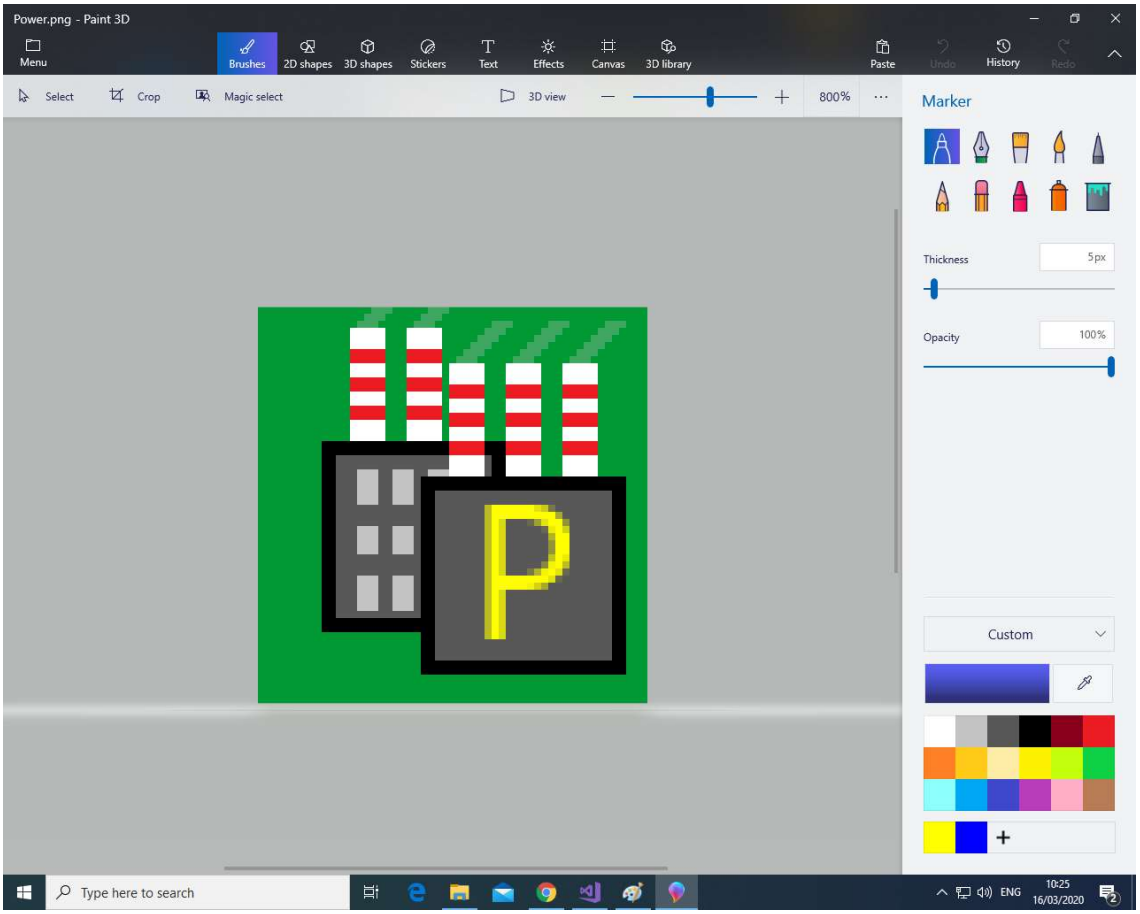
I can then use the roadConditions array to make sure that there is a road anywhere around the user so that, in the conditional statement, they can place a building. I have then included this in the conditional statement along except for the road being pressed so that the user can place a road at the beginning.

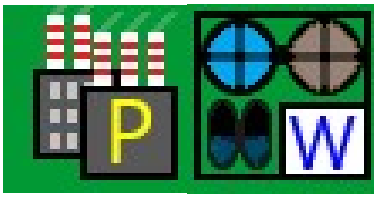
```
DrawBandO.py
if Land[row][column] == 6:
        colour = GRAY
```

I can then change the colour of the tile to grey for the user to visually see that a road has been built in that tile.

4.1.2: Assign Proper Aesthetic

This step will involve me creating new icons for each of the buildings to allow the game to be more visual than just blocks of colour. I have already designed some of these using the Paint3D software on the Windows 10 OS:





Above are the icons that I have designed for water and power, the other icons I will create during the development stage of the sprint.

DrawBandO.py

```
if Land[row][column] == 4:
        image = 'Power.png'
        screen.blit(image, startx, starty)
```

I will then use the images to load into the grids using the pygame "blit" function which places the image onto the grid.

I will then use a function to decide which kind of road needs to be placed as the image in the grid. I want the road to connect and face other roads and the power and water plants but not the residential buildings.

DrawBandO.py

```
def roadDecider(row, column):
        tilesaround = [Land[row-1][column], Land[row][column-1], Land[row][column+1],
Land[row+1][column]]
        top = False
        bottom = False
        left = False
        right = False
        if tilesaround[0] != 0:
                top = True
        if tilesaround[1] != 0:
                left = True
        if tilesaround[2] != 0:
                right = True
        if tilesaround[3] != 0:
                bottom = True
```

I will then define a function that uses an array called tilesaround in order to determine where around the road to be placed other buildings are. I can then use this to determine what type of road to place. In the development stage I can add in the conditional statements in order to place each type of road based off the unique circumstances that each road needs to be placed in.

4.1.3: Add Wealth System

Setup.py

```
houseUpgradeCost = 2000
apartmentUpgradeCost = 8000
```

I will then add two more variables into the setup.py file so that the cost of upgrading both the house and the apartment can be defined for the user and so they can be used in calculating the monetary cost of upgrading each building.

Code.py

```
createButton(screen, WHITE, 170, 645, 40, 100, "^", RED, True, 7, True)
createButton(screen, WHITE, 365, 645, 40, 100, "^", BLUE, True, 7, True)
```

Then, I can add the buttons onto the main screen, with each upgrade button being at the side of each regular building button. This will mean the upgrade buttons are placed in a way which makes it clear to the user which upgrade button is for which building.

I will then design 3 images for each of the basic buildings in order to show the 3 stages of wealth that the user can create by upgrading the base buildings. I have already designed the icons for each of these:

The colour that I have layered over the top of each of the base designed corresponds to metals perceived as valuable in the real world to easily show the user which buildings are of a higher quality and wealth. The first upgrade goes from plain to bronze, a valuable metal found in many parts of the modern-day world from wires to industrial pipes. The final upgrade goes from bronze to gold – one of the most valuable metals on the planet due to its rarity. There is also a helpful scale from one to 3 $ signs to also help the user to recognise which buildings are of which wealth level.

GameMechanics.py

```
TileDeterminer( ):
        if pressed == 7:
                if Land[row][column] == 7:
                        money -= costs[pressed-1]
                        Land[row][column] = 8
                if Land[row][column] == 1:
                        money -= costs[pressed-1]
                        Land[row][column] = 7

        if pressed == 8:
                if Land[row][column] == 9:
                        money -= costs[pressed-1]
                        Land[row][column] = 10
                if Land[row][column] == 2:
                        money -= costs[pressed-1]
                        Land[row][column] = 9
```

I will then have to edit the tile determiner function so that the upgrade button upgrades both the medium wealth and low wealth variants of each building. I will then have to change the DrawBand0.py file so that each value in the Land array will have a picture corresponding to it for the new medium and high wealth variants.

DrawBandO.py

```
roadDecider(row, column):
        if tilesaround[0] != 0,1,2,7,8,9,10:
                top = True
        if tilesaround[1] != 0,1,2,7,8,9,10:
                left = True
        if tilesaround[2] != 0,1,2,7,8,9,10:
                right = True
        if tilesaround[3] != 0,1,2,7,8,9,10:
                bottom = True
```

I will then have to edit the roadDecider function so that it excludes the numerous buildings that aren't power or water buildings so that they are orientated the correct way in the game.

4.1.4: Time-Based and End Menu

I then decided that instead of having the save game and default map success criteria. I should instead have the game time based rather than having an unlimited time as this makes the game more fun and quick-paced as compared to a bit pointless without a time limit as you could just wait and get infinite money, even with the worst buildings. And add a screen at the end to summarise the results of the game and how well the user has done during the game. As for the default map, there already is one so that can easily be discarded as a success criterion.

1. ~~Save Game – Add an ability for the user to save their game to resume it another time through saving the grid, money and population in a text file.~~
    a. ~~Create way of saving the game in a file or array~~
2. ~~Default Map – Add a default/new-game grid, money and population to load when a new game is started.~~

Code.py

```
import time

startTime = time.time() #Gets the start time of the program
if (int(time.time() – startTime)) < 300:
        if (x <= 600) and (y <= 600):
                money = tileDeterminer(x, y, depressed…)
        else:
                ….
if (time.time() – startTime) < 300:

        population, powerBalance, …
        money = …

if round((time.time() – startTime), 1) == 300.0:
        pygame.draw.rect(screen,WHITE, [0,0,800,800]
        pygame.display.flip()
        pygame.mixer.music.load('x.mp3')
```

I can then start recording the time from when the program starts in order to tell when the user starts the program to count until the 5 mins or 300 seconds. This will add the time-based element to the game, it will allow the game's functions to be carried out until the current time – the start time is greater than 300 second. At which point all these functions will stop and the screen will be wiped with white and I will use the pygame mixer function to load a piece of music at the end to allow the user to feel a sense of accomplishment from completing the game, I think I'll use this piece:
endGame.mp3

I can then add labels for each of the statistics in order to allow the user to see how well they did in the game.

Code.py
```
createLabel(screen, WHITE, 110, 110, 580, 50, "CONGRATULATIONS", RED, 50, True, False, True)
time.sleep(1) #For extra emphasis on the results they will be staggered by 1 second each
createLabel(screen, WHITE, 110, 160, 580, 50, "Population: " + population, BLACK, 40, False, False, True)
time.sleep(1)
createLabel(screen, WHITE, 110, 210, 580, 50, "Money: " + money, BLACK, 40, False, False, True
```

This will allow the user to clearly see what their population and money was at the end of the game.

Code.py
```
createLabel(screen, WHITE, 625, 315, 155, 20, "Time Left", BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 340, 155, 20, (300 – (time.time() – startTime)), BLACK, 25, False, False, False)
```

This will allow the user to see how many seconds they have until the game ends and the end screen will appear, keeping the pressure on the user to do the best they can. Also allows for in-depth strategies to be built based on when to do certain actions and to plan for future development, assessing whether the user will make a gain from their investments.

Code.py
```
start = False
if start == False:
        createLabel(screen, WHITE, 625, 365, 155, 20, "Press Enter to start the game", BLACK, 15, False, False, False)
if event.type == pygame.KEYDOWN:
        start = True
        startTime = time.time()
```

This will make it so that the user can start when they press enter (or any key) and this will then start the game rather than just when they open the program. These are some of the programming techniques I will need to use to complete this sprint.

Programming Technique	Explanation
Selection	Used throughout the program to determine things such as whether the user clicked on a button or a tile and which one of these was clicked on
Event-driven programming	I need to use this in order to detect when the user attempts to perform an action inside the generated window

4.2: Development

4.2.1: Check Road

```
roadCost = 500
```

I defined the roadCost variable so that I can use it in other parts of the code and to show it to the user in the form of a label.

```
#Road Label and Button
createButton(screen, WHITE, 625, 525, 145, 50, "Road", BLACK, True, 6, True)
createLabel(screen, WHITE, 625, 550, 150, 35, ("£" + str(roadCost)), BLACK, 15, False, False, True)
```

I then created the road button and labels so that the user can select and place the roads using the button and see the price below it before they place it.

Code.py Setup.py GameMechanics.py DrawBandO.py

```
#If the button is not depressed (already clicked on) it will change to reflect it
if alreadypressed == False:
    #Redraws the button to represent its depressed state
    if Locations[y][x] == 1:
        createButton(screen, GRAY, 25, 645, 185, 100, "House", RED, False, 1, True)
    if Locations[y][x] == 2:
        createButton(screen, GRAY, 210, 645, 185, 100, "Apartment", BLUE, False, 2, True)
    if Locations[y][x] == 3:
        createButton(screen, GRAY, 395, 645, 185, 100, "Demolish", CYAN, False, 3, True)
    if Locations[y][x] == 4:
        createButton(screen, GRAY, 630, 650, 145, 65, "Power", BLACK, False, 4, True)
    if Locations[y][x] == 5:
        createButton(screen, GRAY, 630, 720, 145, 65, "Water", BLACK, False, 5, True)
    if Locations[y][x] == 6:
        createButton(screen, GRAY, 625, 525, 145, 50, "Road", BLACK, False, 6, True)
    if Locations[y][x] == 7:
        quit()
    #Will change it to false to represent its depressed state
    depressed[(Locations[y][x] - 1)] = 1
#If the button is depressed (already clicked on) it will change to reflect it
else:
    #Redraws the button to represent its non depressed state
    if Locations[y][x] == 1:
        createButton(screen, WHITE, 25, 645, 185, 100, "House", RED, False, 1, True)
    if Locations[y][x] == 2:
        createButton(screen, WHITE, 210, 645, 185, 100, "Apartment", BLUE, False, 2, True)
    if Locations[y][x] == 3:
        createButton(screen, WHITE, 395, 645, 185, 100, "Demolish", CYAN, False, 3, True)
    if Locations[y][x] == 4:
        createButton(screen, WHITE, 630, 650, 145, 65, "Power", BLACK, False, 4, True)
    if Locations[y][x] == 5:
        createButton(screen, WHITE, 630, 720, 145, 65, "Water", BLACK, False, 5, True)
    if Locations[y][x] == 6:
        createButton(screen, WHITE, 625, 525, 145, 50, "Road", BLACK, False, 6, True)
    if Locations[y][x] == 7:
        quit()
    #Will change it to false to represent its non-depressed state
    depressed[(Locations[y][x] - 1)] = 0
#Will return these variables to update them for the other functions
return depressed
```

I then altered the buttonDeterminer function to include the road button so that it will be redrawn based on the state at which the button is in such as whether it has been pressed on or not. I also altered the createButton function so that the road label will be redrawn if the user presses on or off the button. As well as this I added another element to the depressed array so that the state of the road button can also be tracked and used within the buttonDeterminer function.

```
        if button == 5:
            createLabel(screen, WHITE, 630, 750, 150, 50, ("£" + str(waterCost)), BLACK, 15, False, False, True)
        if button == 6:
            createLabel(screen, WHITE, 625, 550, 150, 35, ("£" + str(roadCost)), BLACK, 15, False, False, True)
    if border == True:
        pygame.draw.rect(screen, GRAY, [x, y , width, height], 2)
```

```
    elif (money - costs[pressed-1]) <= 0:
        if pressed == 1:
            createLabel(screen, WHITE, 25, 695, 185, 50, ("£" + str(houseCost)), RED, 15, False, False, True)
        if pressed == 2:
            createLabel(screen, WHITE, 210, 695, 185, 50, ("£" + str(apartmentCost)), RED, 15, False, False, True)
        if pressed == 4:
            createLabel(screen, WHITE, 630, 680, 150, 50, ("£" + str(powerCost)), RED, 15, False, False, True)
        if pressed == 5:
            createLabel(screen, WHITE, 630, 750, 150, 50, ("£" + str(waterCost)), RED, 15, False, False, True)
        if pressed == 6:
            createLabel(screen, WHITE, 625, 550, 150, 35, ("£" + str(roadCost)), RED, 15, False, False, True)
    except:
        IndexError
    return money
```

I then changed the tileDeterminer function so that it include the same functionality as the other buttons in the sense that if the user doesn't have a high enough money variable the label will be redrawn under the button as red to show the user they cannot afford to place a road.

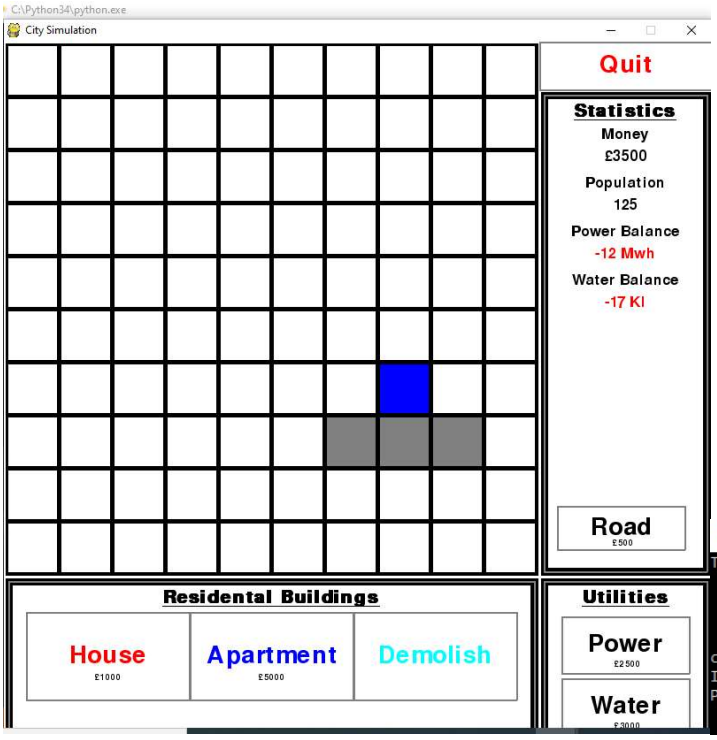
```
tion to determine which tile the user has clicked on
ileDeterminer(x, y, depressed, money, screen, population):
    #Sets the row and column that the user has clicked on
    column = x // (WIDTH + MARGIN)
    row = y // (WIDTH + MARGIN)
    #Road Conditions
    roadConditions = [Land[row][column-1], Land[row][column+1], Land[row-1][column], Land[row-1][column-1], Land[row-1][column+1], Land[row+1][column], Land[row+1][column-1], Land[row+1][column+1]]
    #Sets values for each pressed button
    costs = [houseCost, apartmentCost, clearCost, powerCost, waterCost, roadCost]
    #Will try to set the array position to the button pressed and will skip any Index errors
    try:
        #Will make it so if you haven't depressed a button you won't clear the tile
        for buttons in range(len(depressed)):
            if depressed[buttons] == 1:
                pressed = buttons + 1
        if pressed == 3:
            Land[row][column] = 0
        elif (money - costs[pressed-1]) >= 0:
            if (Land[row][column] == 0) and ((6 in roadConditions) or (pressed == 6)):
                money = (money - costs[pressed-1])
                Land[row][column] = pressed
        elif (money - costs[pressed-1]) <= 0:
```

Then I added a new array called “roadConditions” and this is used to determine if there is a road in the surrounding tiles of where the user wants to place the building or another road. I have then altered the conditional statement if (Land[row][column] == 0) to also include if there is a 6(representing a road) in the roadConditions array, meaning that there is a road in one of the surrounding tiles. But also if the pressed is 6 also allow it to continue as you need a road at the start.

```
#Road
if Land[row][column] == 6:
    colour = GRAY
#Draws the rectangle for eac
pygame.draw.rect(screen,
```

I then altered the DrawBandO.py file so that when the grid is being redrawn after changes are made if the value for that tile in the grid is 6, representing a road, then it is changed to the grey colour.

Test Reference 3.1.1

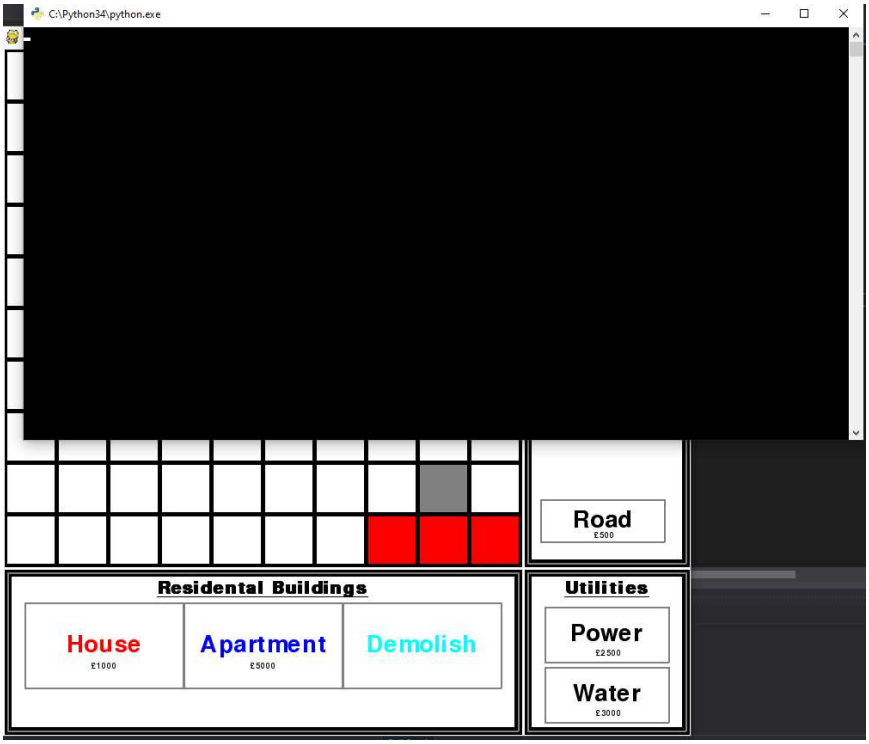


As you can see from my screenshot the grey tile is rendered correctly and as expected. I did, however, encounter an error when I tried to click on either side of the grid, this was due to the face I was checking all tiles around the one I clicked on, and as there are not tiles to the right of the right-most column this creates an error as it is outside the array.

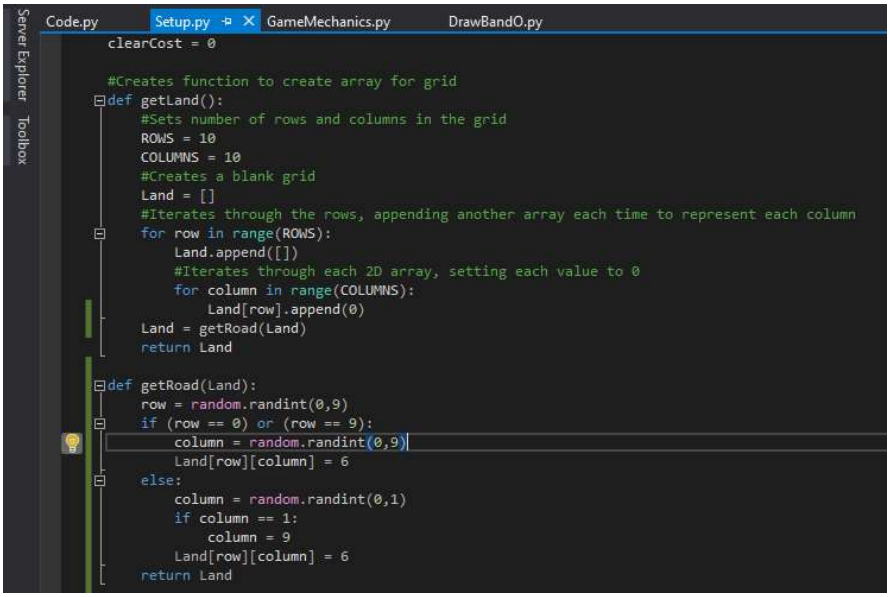
```
C:\Python34\python.exe
Traceback (most recent call last):
  File "C:\Users\21121\Downloads\OneDrive_2
    money = tileDeterminer(x, y, depressed,
  File "C:\Users\21121\Downloads\OneDrive_2
    roadConditions = [Land[row][column-1],
column+1], Land[row+1][column], Land[row+1]
IndexError: list index out of range
Press any key to continue . . .
```

```
row = y // (WIDTH + MARGIN)
#Road Conditions
roadConditions = [[row,column-1],[row,column+1],[row-1,column],[row-1,column-1],[row-1,column+1],[row+1,column],[row+1,column+1],[row+1,column-1]]
x = 8
while x > -1:
    try:
        roadConditions[x-1] = Land[roadConditions[x-1][0]][roadConditions[x-1][1]]
    except:
        IndexError
    x -= 1
```

To fix this I created a loop to try to create a value for each element in the roadConditions array. This would try to fill in each element in turn and if it encountered an index error it would skip that one and go onto the next.



This appeared to work and allowed me to place a building and/or road at the edge of the grid without having any errors, fixing the issue that I had been having.



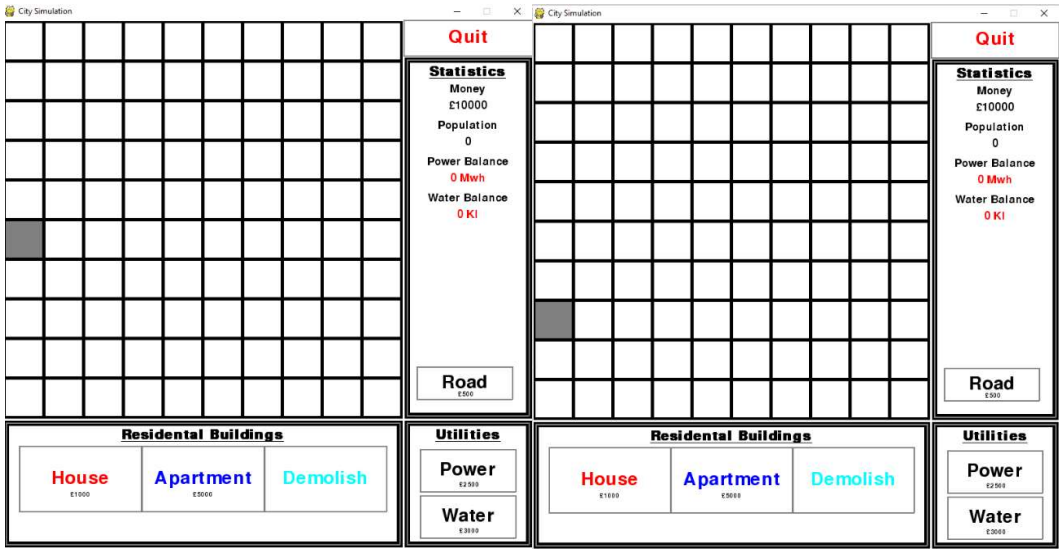

```
    clearCost = 0

    #Creates function to create array for grid
def getLand():
    #Sets number of rows and columns in the grid
    ROWS = 10
    COLUMNS = 10
    #Creates a blank grid
    Land = []
    #Iterates through the rows, appending another array each time to represent each column
    for row in range(ROWS):
        Land.append([])
        #Iterates through each 2D array, setting each value to 0
        for column in range(COLUMNS):
            Land[row].append(0)
    Land = getRoad(Land)
    return Land

def getRoad(Land):
    row = random.randint(0,9)
    if (row == 0) or (row == 9):
        column = random.randint(0,9)
        Land[row][column] = 6
    else:
        column = random.randint(0,1)
        if column == 1:
            column = 9
        Land[row][column] = 6
    return Land
```

I still had an issue with the fact I could place a road anywhere on the grid due to the previous “or pressed == 6”, to get around this I decided to create a function in setup.py so that a single road tile is placed randomly along the edge of the grid in the starting Land array so that the city can be built off that road and I can remove the “or pressed == 6” condition as the user doesn’t now need to place a starting road.

```
        if (6 in roadConditions):
            money = (money - costs[pressed-1])
            Land[row][column] = pressed
```

I then altered the conditional statement to remove the previous pressed == 6 condition.

Test Reference 3.1.2



You can see from the above pictures that the road has been randomly placed on start of the program, this gets rid of the need for the user to place one at the start – while also providing a unique game experience every time.

Test Reference 3.1.3

Sprint 3 Road System.mp4

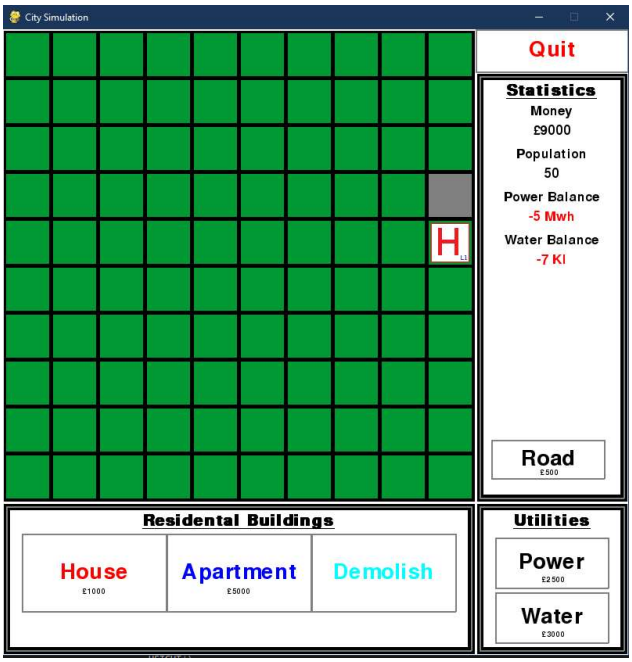
As you can see from the video, buildings can only be built around a road and the starting road is randomly placed and is the starting point for the game.

4.2.2: Assign Proper Aesthetic

```
                population += 125
            #Power
            if Land[row][column] == 4:
                colour = YELLOW
                powerTiles += 1
            #Water
            if Land[row][column] == 5:
                colour = CYAN
                waterTiles += 1
            #Road
            if Land[row][column] == 6:
                colour = GRAY
            #Draws the rectangle for each grid tile
            pygame.draw.rect(screen,
                             colour,
                             [(MARGIN + WIDTH) * column + MARGIN,
                              (MARGIN + HEIGHT) * row + MARGIN,
                              WIDTH,
                              HEIGHT])
            if Land[row][column] == 1:
                house = pygame.image.load('HouseL1.png')
                screen.blit(house, [(MARGIN + WIDTH) * column + MARGIN,
                                    (MARGIN + HEIGHT) * row + MARGIN,
                                    WIDTH,
                                    HEIGHT])
    return population, power, water, powerTiles, waterTiles
```

I had designed a HouseL1.png so that I could test out the functionality of the pygame.image.load function and apply it to the game, this would replace the colour with an image, using the screen.blit function as a way of putting the image on the screen and placing it in the correct position.

Test Reference 3.2.1

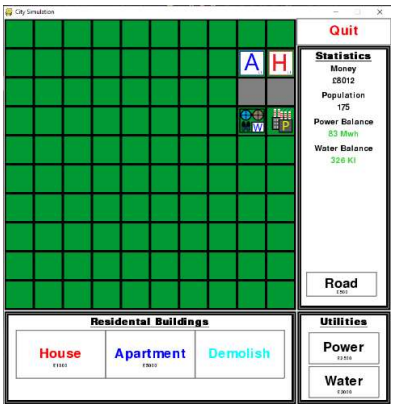


As you can see from the image the house image is drawn in place of a colour, meaning that I can apply this to the other buildings and make the game much more aesthetically pleasing to the user.

```
            colour = CYAN
            waterTiles += 1
        #Road
        if Land[row][column] == 6:
            colour = GRAY
        #Draws the rectangle for each grid tile
        pygame.draw.rect(screen,
                         colour,
                         [(MARGIN + WIDTH) * column + MARGIN,
                          (MARGIN + HEIGHT) * row + MARGIN,
                          WIDTH,
                          HEIGHT])
        if Land[row][column] == 1:
            image = pygame.image.load('HouseL1.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                          (MARGIN + HEIGHT) * row + MARGIN,
                          WIDTH,
                          HEIGHT])
        if Land[row][column] == 2:
            image = pygame.image.load('ApartL1.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                          (MARGIN + HEIGHT) * row + MARGIN,
                          WIDTH,
                          HEIGHT])
        if Land[row][column] == 4:
            image = pygame.image.load('Power.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                          (MARGIN + HEIGHT) * row + MARGIN,
                          WIDTH,
                          HEIGHT])
        if Land[row][column] == 5:
            image = pygame.image.load('Water.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                          (MARGIN + HEIGHT) * row + MARGIN,
                          WIDTH,
                          HEIGHT])
    return population, power, water, powerTiles, waterTiles
```

I then created icons for each of the current buildings and then applied the same method that I used for the house in order to draw the image of the building on the tile to further highlight to the user what buildings are placed where.

Test Reference 3.2.2

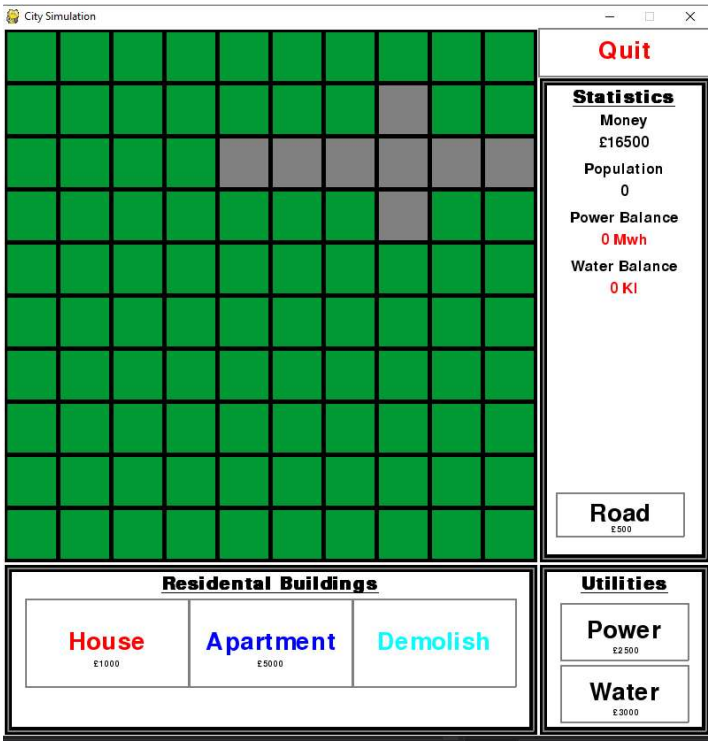


As you can see from the image the icons for the buildings are drawn correctly and as intended, and from the user's perspective they can now more easily identify what is built where and adds a lot more visual detail to the game.

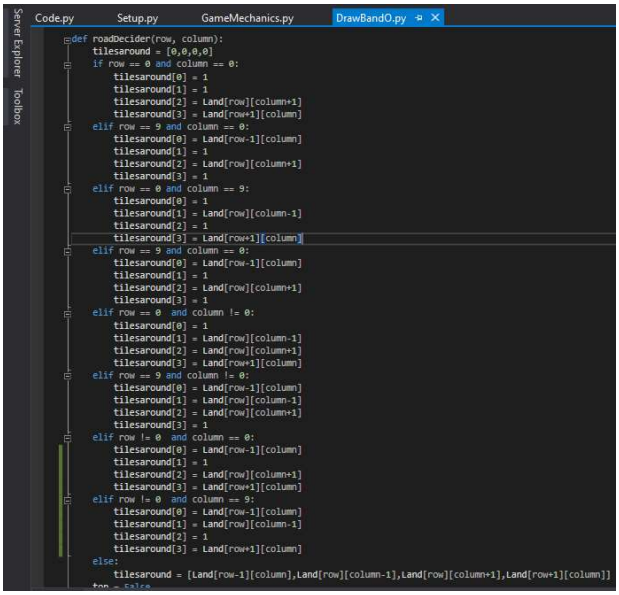
```
        column = x // (WIDTH + MARGIN)
        row = y // (WIDTH + MARGIN)
        #Road Conditions
        roadConditions = [[row,column-1],[row,column+1],[row-1,column],[row+1,column]]
        x = 4
        while x > -1:
            try:
                roadConditions[x-1] = Land[roadConditions[x-1][0]][roadConditions[x-1][1]]
            except:
                IndexError
            x -= 1
```

I then changed the roadConditions array so that buildings could only be built directly next to the road so not diagonally. This improves the importance of building roads and makes the game more realistic. I also changed the value of x from 8 to 4 to represent this as only 4 places can contain a building around the road now.

Test Reference 3.2.3

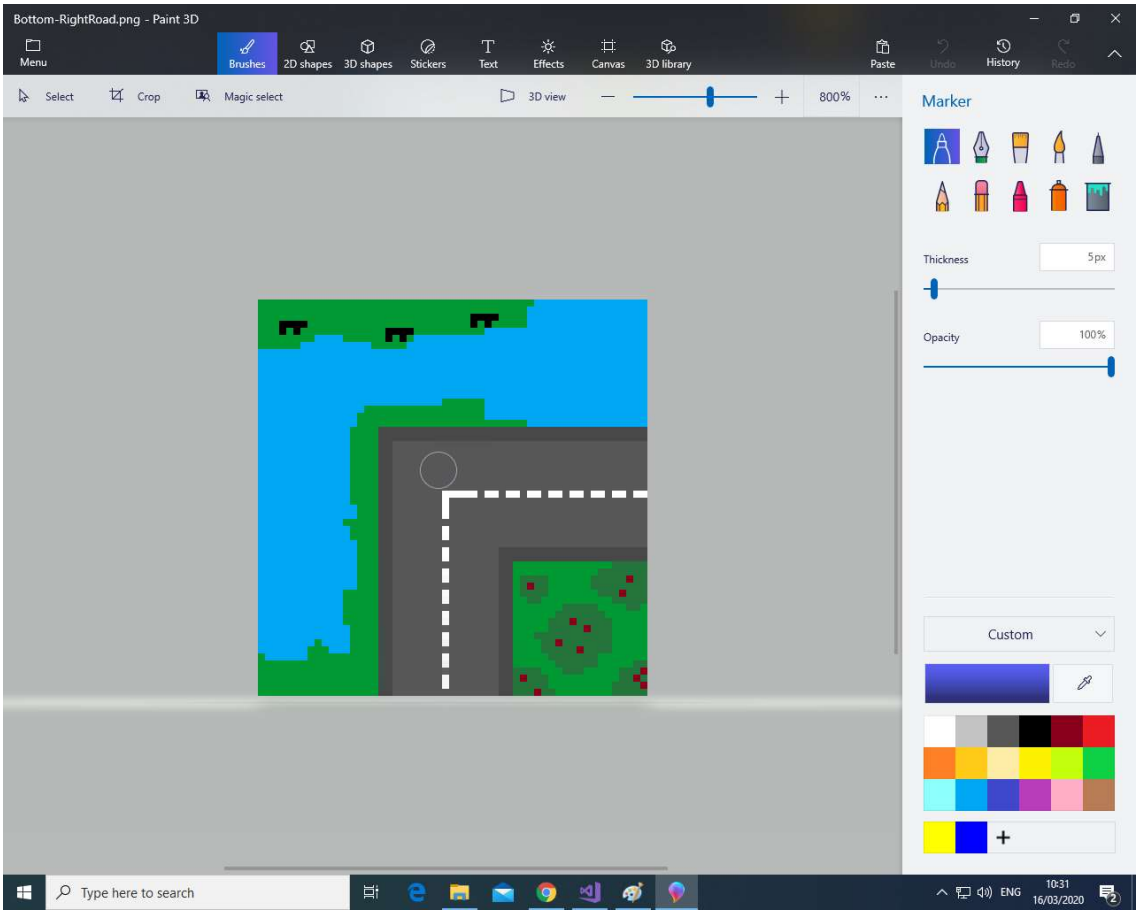


As you can see from the image, I could only place the roads either above, below, left, or right of each other. This means that you can only build buildings in these directions of each other.

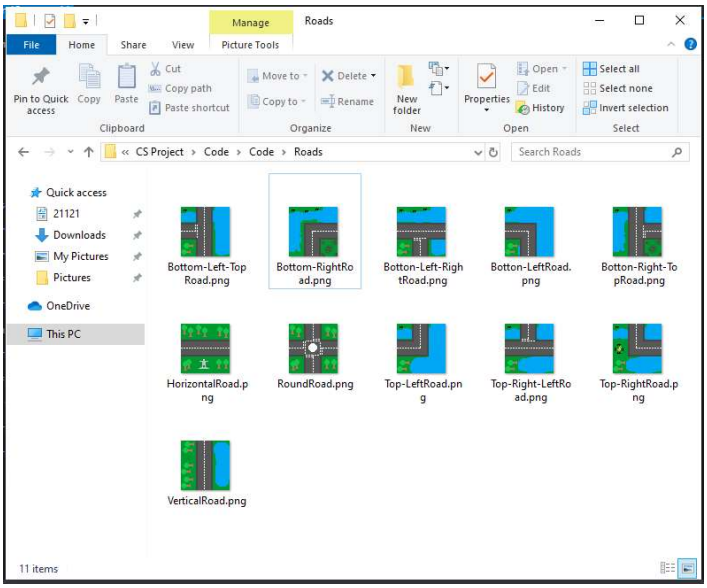



```
def roadDecider(row, column):
    tilesaround = [0,0,0,0]
    if row == 0 and column == 0:
        tilesaround[0] = 1
        tilesaround[1] = 1
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = Land[row+1][column]
    elif row == 9 and column == 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = 1
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = 1
    elif row == 0 and column == 9:
        tilesaround[0] = 1
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = 1
        tilesaround[3] = Land[row+1][column]
    elif row == 9 and column == 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = 1
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = 1
    elif row == 0  and column != 0:
        tilesaround[0] = 1
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = Land[row+1][column]
    elif row == 9 and column != 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = 1
    elif row != 0  and column == 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = 1
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = Land[row+1][column]
    elif row != 0  and column == 9:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = 1
        tilesaround[3] = Land[row+1][column]
    else:
        tilesaround = [Land[row-1][column],Land[row][column-1],Land[row][column+1],Land[row+1][column]]
    top = False
```

I then designed a new function called roadDecider that contains a new array called tilesaround. It edits the values of these elements based on where the road is placed, I have made it so that the roads along the edges of the grid appear to have a connection to outside of the grid via a huge amount of conditional statements. The values in tilesaround represent top, left, right and bottom respectively.



I then designed all the various road images in paint3D, here are all of them:



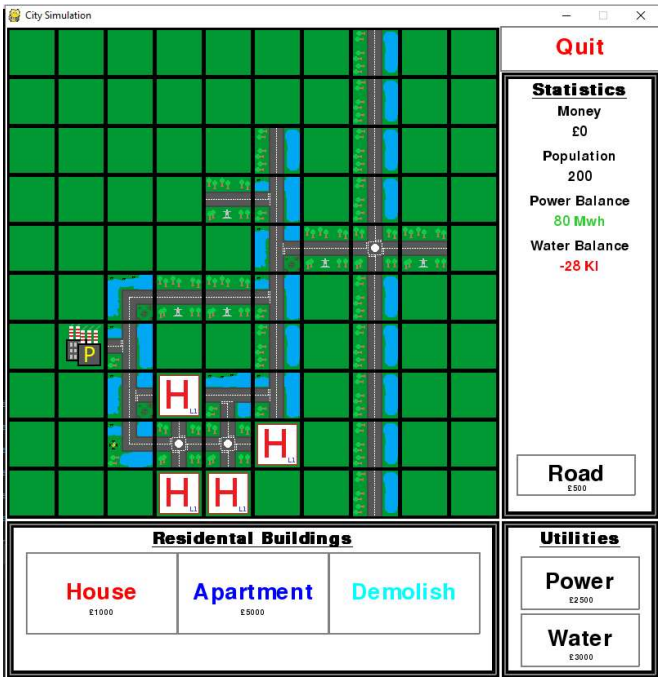
These represent all the possible arrangements of connecting roads that can be used in the game. These will change when the grid is refreshed if there is a road or certain buildings connecting to it.

```
Setup.py    GameMechanics.py    DrawBandO.py
else:
    tilesaround = [Land[row-1][column],Land[row][column-1],Land[row][column+1],Land[row+1][column]]
top = False
left = False
right = False
bottom = False
if tilesaround[0] != 0:
    top = True
if tilesaround[1] != 0:
    left = True
if tilesaround[2] != 0:
    right = True
if tilesaround[3] != 0:
    bottom = True
if (top == True) and (left == True) and (right == False) and (bottom == False):
    road = "Top-LeftRoad.png"
elif (top == True) and (left == True) and (right == True) and (bottom == False):
    road = "Top-Right-LeftRoad.png"
elif (top == True) and (left == True) and (right == True) and (bottom == True):
    road = "RoundRoad.png"
elif (top == True) and (left == False) and (right == False) and (bottom == True):
    road = "VerticalRoad.png"
elif (top == True) and (left == False) and (right == True) and (bottom == False):
    road = "Top-RightRoad.png"
elif (top == True) and (left == True) and (right == False) and (bottom == True):
    road = "Bottom-Left-TopRoad.png"
elif (top == True) and (left == False) and (right == True) and (bottom == True):
    road = "Botton-Right-TopRoad.png"
elif (top == False) and (left == True) and (right == False) and (bottom == True):
    road = "Botton-LeftRoad.png"
elif (top == False) and (left == False) and (right == True) and (bottom == True):
    road = "Bottom-RightRoad.png"
elif (top == False) and (left == True) and (right == True) and (bottom == True):
    road = "Botton-Left-RightRoad.png"
elif (top == False) and (left == True) and (right == True) and (bottom == False):
    road = "HorizontalRoad.png"
elif (top == False) and (left == False) and (right == False) and (bottom == False):
    road = "RoundRoad.png"
elif (top == True) or (bottom == True):
    road = "VerticalRoad.png"
elif (right == True) or (left == True):
    road = "HorizontalRoad.png"
return road
```

I then finished editing the roadDecider function and set it up so that the boolean variables top, left, right and bottom are changed based off the elements in the tilesaround array. These boolean variables will then determine which road is used based on a huge amount of conditional statements, each representing a differing arrangement of buildings and/or roads around a certain road. This will then return the name of the road to be used to the DrawBandO.py file.

```
                                    HEIGHT])
            if Land[row][column] == 6:
                image = pygame.image.load(roadDecider(row,column))
                screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                    (MARGIN + HEIGHT) * row + MARGIN,
                                    WIDTH,
                                    HEIGHT])
    return population, power, water, powerTiles, waterTiles
```

Test Reference 3.2.4



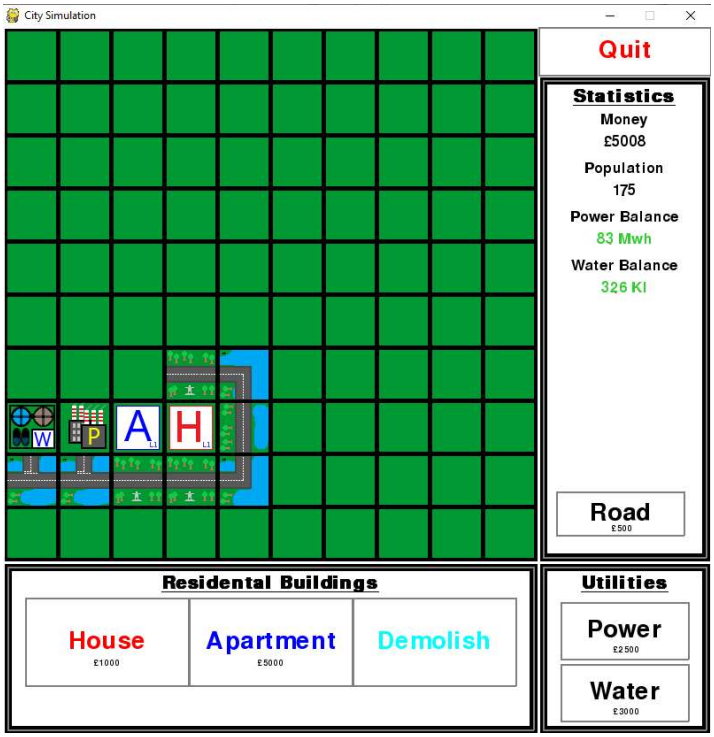
As you can see from the image the roads are changed depending on the tiles around the road. The roads appear to connect to each other this way and the buildings seem to all be connected, adding much more realism to the game.

```
Setup.py    GameMechanics.py    DrawBandO.py
def roadDecider(row, column):
    tilesaround = [0,0,0,0]
    if row == 0 and column == 0:
        tilesaround[0] = 3
        tilesaround[1] = 3
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = Land[row+1][column]
    elif row == 9 and column == 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = 3
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = 3
    elif row == 0 and column == 9:
        tilesaround[0] = 3
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = 3
        tilesaround[3] = Land[row+1][column]
    elif row == 9 and column == 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = 3
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = 3
    elif row == 0  and column != 0:
        tilesaround[0] = 3
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = Land[row+1][column]
    elif row == 9 and column != 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = 3
    elif row != 0  and column == 0:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = 3
        tilesaround[2] = Land[row][column+1]
        tilesaround[3] = Land[row+1][column]
    elif row != 0  and column == 9:
        tilesaround[0] = Land[row-1][column]
        tilesaround[1] = Land[row][column-1]
        tilesaround[2] = 3
        tilesaround[3] = Land[row+1][column]
```

I then changed all the values of the outer tiles from 1 to 3 so that I can make it so that the houses and apartments aren't counted and so aren't connected to the roads, as this would be unrealistic. I then changed the conditional statements to exclude the values for the house and apartment, so they are not considered when deciding which road to use.

```
bottom = False
if (tilesaround[0] != 0) and (tilesaround[0] != 1) and (tilesaround[0] != 2):
    top = True
if (tilesaround[1] != 0) and (tilesaround[1] != 1) and (tilesaround[1] != 2):
    left = True
if (tilesaround[2] != 0) and (tilesaround[2] != 1) and (tilesaround[2] != 2):
    right = True
if (tilesaround[3] != 0) and (tilesaround[3] != 1) and (tilesaround[3] != 2):
    bottom = True
```

Test Reference 3.2.5



As you can see from the image, the roads do not appear to connect to the houses and apartments but do connect to each other and the water and power plants.

4.2.3: Add Wealth System

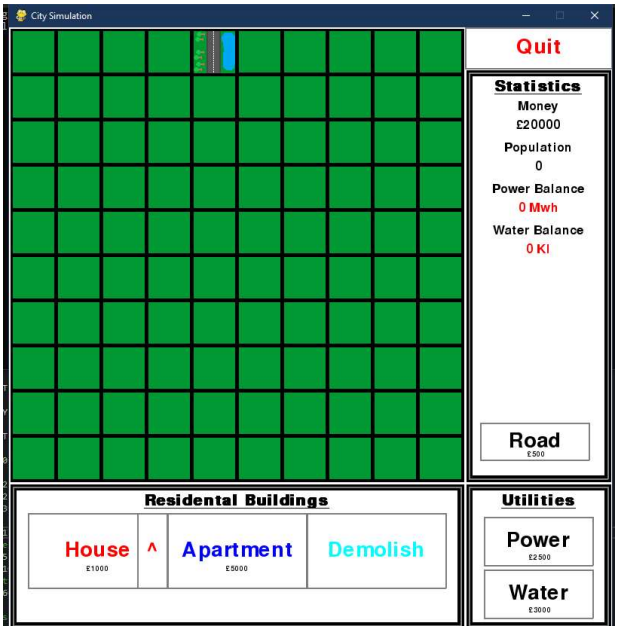
```
#Setting starting variables for button depressed and button pressed
depressed = [0,0,0,0,0,0,0,0,0,0,0]
colour = WHITE

#Setup of game variables
money = 20000
population = 0
powerBalance = 0
waterBalance = 0
powerTiles = 0
waterTiles = 0

#Setup Button Values
houseCost = 1000
houseUpgradeCost = 2000
apartmentCost = 5000
apartmentUpgradeCost = 8000
powerCost = 2500
waterCost = 3000
roadCost = 500
clearCost = 0
```

I added two new variables, the houseUpgradeCost and the apartmentUpgradeCost. These represent the cost to upgrade the house and apartment respectively. I also updated the depressed array to include the new buttons.

```
#Upgrade Buttons
createButton(screen, WHITE, 170, 645, 40, 100, "^", RED, True,  7, True)
```

I then added a new button with the button number 7 to represent the new upgrade button for upgrading houses and used a "^" to represent an up arrow so the user can deduce it is the upgrade button. I added this next to the house button to allow the user to more easily understand that it is the button to upgrade the houses.

Test Reference 3.3.1

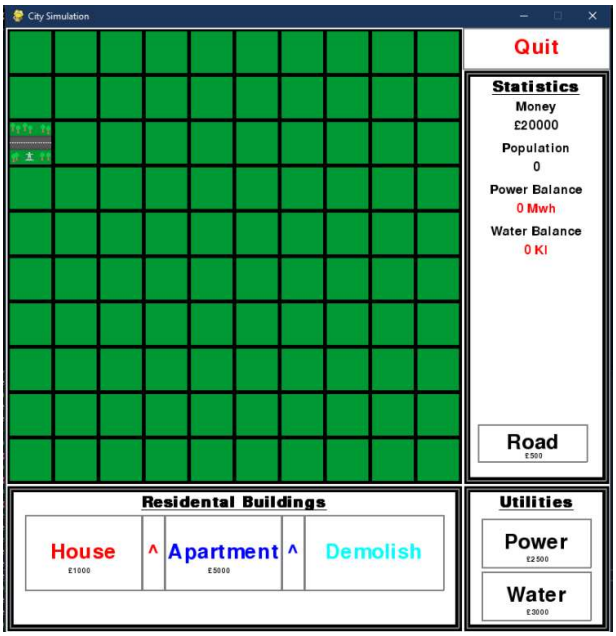


As you can see from the image, the house upgrade button is correctly placed and clearly represents the button to upgrade the houses due to its: position next to the house button, red colour, and up arrow symbol.

```
#--------------------Residental Box--------------------
#Border 1 for Residental Box
pygame.draw.rect(screen, WHITE, [0,605, 605,195], 2)
#Border 2 for Residental Box
pygame.draw.rect(screen, GRAY, [5,610, 595,185], 2)
#Border 3 for Residental Box
pygame.draw.rect(screen, WHITE, [10,615, 585,175])
#Residental Box Label
createLabel(screen, WHITE, 10, 615, 585, 30, "Residental Buildings", BLACK, 30, True, True, False)
#Residental Buttons
createButton(screen, WHITE, 25, 645, 155, 100, "House", RED, True, 1, True)
createButton(screen, WHITE, 210, 645, 155, 100, "Apartment", BLUE, True, 2, True)
createButton(screen, WHITE, 395, 645, 185, 100, "Demolish", CYAN, True, 3, True)
#Upgrade Buttons
createButton(screen, WHITE, 180, 645, 30, 100, "^", RED, True,  7, True)
createButton(screen, WHITE, 365, 645, 30, 100, "^", BLUE, True,  7, True)
#Labels Underneath the Residental Buttons
createLabel(screen, WHITE, 25, 695, 145, 50, ("£" + str(houseCost)), BLACK, 15, False, False, True)
createLabel(screen, WHITE, 210, 695, 145, 50, ("£" + str(apartmentCost)), BLACK, 15, False, False, True)
```

I then added in another upgrade button for the apartments. I then altered the width and height values for the house and apartment buttons and labels so that they are central to the main button and not the main button and upgrade button as this makes them look unsymmetrical and poor quality.

Test Reference 3.3.2



As you can see from the image the upgrade buttons are drawn in correctly and the house and apartment buttons and labels are now centred – making them look a lot better.

```
            image = pygame.image.load('HousePoor.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 2:
            image = pygame.image.load('ApartPoor.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 4:
            image = pygame.image.load('Power.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 5:
            image = pygame.image.load('Water.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 6:
            image = pygame.image.load(roadDecider(row,column))
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 7:
            image = pygame.image.load('HouseMiddle.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 8:
            image = pygame.image.load('HouseRich.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 9:
            image = pygame.image.load('ApartMiddle.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
        if Land[row][column] == 10:
            image = pygame.image.load('ApartRich.png')
            screen.blit(image, [(MARGIN + WIDTH) * column + MARGIN,
                                (MARGIN + HEIGHT) * row + MARGIN,
                                WIDTH,
                                HEIGHT])
    return population, power, water, powerTiles, waterTiles
```

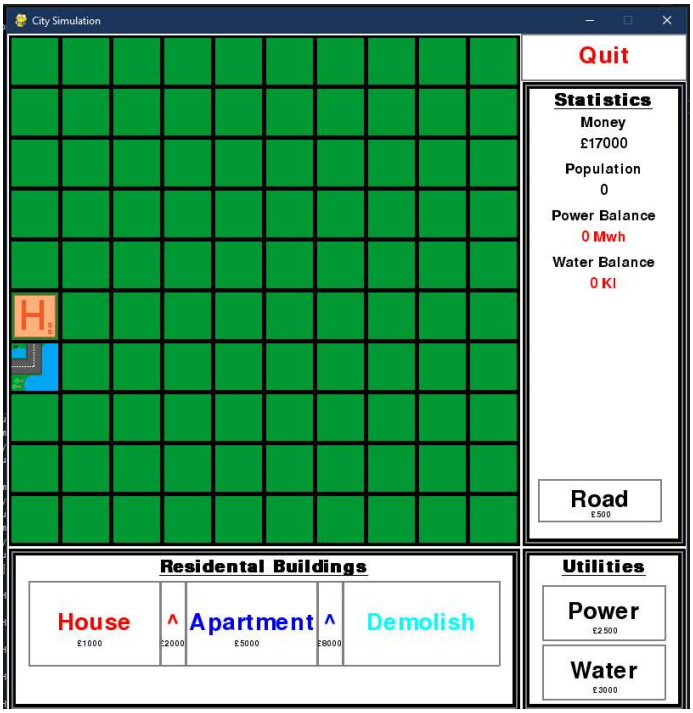
I then used my previously designed images for each of the wealth levels and assigned these to values of the Land array that represent the new wealth levels. I also changed the image files for the basic houses and apartments to fit this new wealth design. This also led me to add the new buttons and labels for those buttons into the functions where they are to be redrawn such as in the tileDeterminer function if the user doesn't have a high enough money variable.

```
if pressed == 7:
    createLabel(screen, WHITE, 180, 695, 30, 50, ("£" + str(houseUpgradeCost)), RED, 15, False, False, True)
if pressed == 8:
    createLabel(screen, WHITE, 365, 695, 30, 50, ("£" + str(apartmentUpgradeCost)), RED, 15, False, False, True)
```

```
#Sets values for each pressed button
costs = [houseCost, apartmentCost, clearCost, powerCost, waterCost, roadCost,
#Will try to set the array position to the button pressed and will skip any I
try:
    #Will make it so if you haven't depressed a button you won't clear the ti
    for buttons in range(len(depressed)):
        if depressed[buttons] == 1:
            pressed = buttons + 1
    if pressed == 3:
        Land[row][column] = 0
    elif (money - costs[pressed-1]) >= 0:
        print(pressed)
        if (6 in roadConditions):
            if (pressed != 7 and pressed != 8) and Land[row][column] == 0:
                money = (money - costs[pressed-1])
                Land[row][column] = pressed
            if pressed == 7:
                if Land[row][column] == 7:
                    money = (money - costs[pressed-1])
                    Land[row][column] = 8
                if Land[row][column] == 1:
                    money = (money - costs[pressed-1])
                    Land[row][column] = 7
            if pressed == 8:
                if Land[row][column] == 9:
                    money = (money - costs[pressed-1])
                    Land[row][column] = 10
                if Land[row][column] == 2:
                    money = (money - costs[pressed-1])
                    Land[row][column] = 9
```

I then changed the tileDeterminer function so that if the button is 7 (house upgrade) or 8 (apartment upgrade) then there will be other conditional statements since it can change tiles to both middle and high wealth. If it is 7 that is pressed then the function will first check if the tile is 7 and if the user has enough money it will change it to 8 representing an upgrade from medium to high wealth and if it is 1 and they have enough money then it will change to 7, representing a change from low to medium wealth. I put it this way round as if you put it the other way round it will upgrade a low to high wealth as it will first change it from 1 to 7 as the first statement is true but then go onto the second statement and that is also true so will go from 7 to 8 and this is not what we want. I then did the same to the apartment variants if 8 is pressed.

Test Reference 3.3.3



As you can see from the image the house has been built (£1000) and then the upgrade has been applied via the upgrade button (£2000) and the money has been reduced by the correct amount (20000 – 1000 – 2000 = 17000) so the upgrade has been applied correctly.

```
if (tilesaround[0] != 0) and (tilesaround[0] != 1) and (tilesaround[0] != 2) and (tilesaround[0] != 7) and (tilesaround[0] != 8) and (tilesaround[0] != 9) and (tilesaround[0] != 10):
    top = True
if (tilesaround[1] != 0) and (tilesaround[1] != 1) and (tilesaround[1] != 2) and (tilesaround[1] != 7) and (tilesaround[1] != 8) and (tilesaround[1] != 9) and (tilesaround[1] != 10):
    left = True
if (tilesaround[2] != 0) and (tilesaround[2] != 1) and (tilesaround[2] != 2) and (tilesaround[2] != 7) and (tilesaround[2] != 8) and (tilesaround[2] != 9) and (tilesaround[2] != 10):
    right = True
if (tilesaround[3] != 0) and (tilesaround[3] != 1) and (tilesaround[3] != 2) and (tilesaround[3] != 7) and (tilesaround[3] != 8) and (tilesaround[3] != 9) and (tilesaround[3] != 10):
    bottom = True
```

I then altered the roadDeterminer function so that it would also disregard the other levels of houses and apartments as well as the base ones.

Test Reference 3.3.4

As you can see from the image the various levels of house didn't change the type of road when it is changed, this shows that my changes are working as intended.

```
        x -= 1
    #Sets values for each pressed button
    costs = [houseCost, apartmentCost, clearCost, powerCost, waterCost, roadCos
    pygame.mixer.music.load('construction.mp3')
    #Will try to set the array position to the button pressed and will skip any
    try:
        #Will make it so if you haven't depressed a button you won't clear the
        for buttons in range(len(depressed)):
            if depressed[buttons] == 1:
                pressed = buttons + 1
        if pressed == 3:
            Land[row][column] = 0
        elif (money - costs[pressed-1]) >= 0:
            if (6 in roadConditions):
                if (pressed != 7 and pressed != 8) and Land[row][column] == 0:
                    money = (money - costs[pressed-1])
                    Land[row][column] = pressed
                    pygame.mixer.music.play()
                if pressed == 7:
                    if Land[row][column] == 7:
                        money = (money - costs[pressed-1])
                        Land[row][column] = 8
                        pygame.mixer.music.play()
                    if Land[row][column] == 1:
                        money = (money - costs[pressed-1])
                        Land[row][column] = 7
                        pygame.mixer.music.play()
                if pressed == 8:
                    if Land[row][column] == 9:
                        money = (money - costs[pressed-1])
                        Land[row][column] = 10
                        pygame.mixer.music.play()
                    if Land[row][column] == 2:
                        money = (money - costs[pressed-1])
                        Land[row][column] = 9
                        pygame.mixer.music.play()
        elif (money - costs[pressed-1]) <= 0:
```

I then decided to use the pygame.mixer function to play a sound every time the user builds a building or road to give a much more immersive experience for the user. I decided to use this sound: construction.mp3
As it sounds like a construction site.

Test Reference 3.3.5

Sprint 3 Upgrade System.mp4

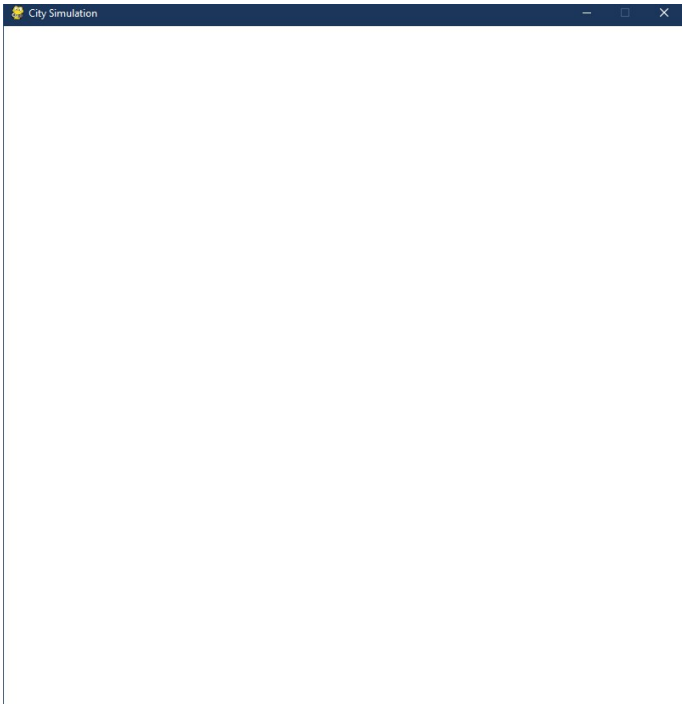
As you can see in the video the upgrade system and sounds work as intended.

4.2.3: Time-Based and End Menu

```
    #Border 3 for Residental Box
    pygame.draw.rect(screen, WHITE, [10,615, 585,175])
    #Residental Box Label
    createLabel(screen, WHITE, 10, 615, 585, 30, "Residental Buildings", BLACK, 30, True, True, False)
    #Residental Buttons
    createButton(screen, WHITE, 25, 645, 155, 100, "House", RED, True, 1, True)
    createButton(screen, WHITE, 210, 645, 155, 100, "Apartment", BLUE, True, 2, True)
    createButton(screen, WHITE, 395, 645, 185, 100, "Demolish", CYAN, True, 3, True)
    #Upgrade Buttons
    createButton(screen, WHITE, 180, 645, 30, 100, "^", RED, True,  7, True)
    createButton(screen, WHITE, 365, 645, 30, 100, "^", BLUE, True,  8, True)
    #Labels Underneath the Residental Buttons
    createLabel(screen, WHITE, 25, 695, 145, 50, ("£" + str(houseCost)), BLACK, 15, False, False, True)
    createLabel(screen, WHITE, 210, 695, 145, 50, ("£" + str(apartmentCost)), BLACK, 15, False, False, True)
    createLabel(screen, WHITE, 180, 695, 30, 50, ("£" + str(houseUpgradeCost)), BLACK, 15, False, False, True)
    createLabel(screen, WHITE, 365, 695, 30, 50, ("£" + str(apartmentUpgradeCost)), BLACK, 15, False, False, True)
    #--------------------Quit Button--------------------
    createButton(screen, WHITE, 605, 0, 195, 55, "Quit", RED, True, 9, True)

    startTime = time.time()
    end = False
    #Will loop while done is false
    while not done:
        #Will run if the user does something
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                done = True
            elif event.type == pygame.MOUSEBUTTONDOWN:
                #Gets x and y position of the mouse
                pos = pygame.mouse.get_pos()
                #Creates two variables to make the coordinates easier to work with
                x = pos[0]
                y = pos[1]
                if (int(time.time() - startTime)) < 5:
                    #If the user clicks inside the area for the game, it runs tileDeterminer
                    if (x <= 600) and (y <= 600):
                        #passes x, y coordinates and the status of which button is pressed
                        money = tileDeterminer(x, y, depressed, money, screen, population)
                    else:
                        #changes the value of pressed and depressed depending on what button is pressed
                        depressed = buttonDeterminer(x, y, depressed, screen)
        if (time.time() - startTime) < 5:
            #Updates the tiles in the grid
            population, powerBalance, waterBalance, powerTiles, waterTiles = tileChecker(screen, population, powerBa
            #Will correct the variables
            money = variableCalc(screen, population, money, powerBalance, waterBalance, powerTiles, waterTiles)
            #Will refresh the screen to display any changes
            pygame.display.flip()
        print(round((time.time() - startTime), 1))
        if round((time.time() - startTime), 1) == 5.0:
            #Blank Screen and Music
            pygame.draw.rect(screen, WHITE, [0,0,800,800])
            pygame.display.flip()
            pygame.mixer.music.load('endGame.mp3')
            pygame.mixer.music.play()
        #Will refresh 60 times per second
        clock.tick(60)
```

I decided to make the game time-based so what I did was import time into the code.py file and then create a variable startTime and set it to the time at which the game started. I then added conditional statements to each of the game mechanics, such as before the money and button changes were being calculated and the grid was going to be redrawn. These conditional statements made it so that if the current time – the start time was greater than 5 these processes would stop and if it was, then the screen would be filled with white and a new sound (endGame.mp3) would be played highlighting the end of the game.

Test Reference 3.4.1



As you can see from the image, at the time 5 seconds the screen gets covered in white and the music plays.

```
        #Blank Screen and Music
        pygame.draw.rect(screen, WHITE, [0,0,800,800])
        pygame.display.flip()
        pygame.mixer.music.load('endGame.mp3')
        pygame.mixer.music.play()
        #Total box and score
        pygame.draw.rect(screen, GRAY, [100,100,600,600],2)
        pygame.draw.rect(screen, BLACK, [110,110,580,580],2)
        pygame.display.flip()
        time.sleep(1)
        createLabel(screen, WHITE, 110, 110, 580, 50, "CONGRATULATIONS", RED, 50, True, False, True)
        pygame.display.flip()
        time.sleep(1)
        createLabel(screen, WHITE, 110, 160, 580, 50, "Population:  " + str(population), BLACK, 40, False, False, True)
        pygame.display.flip()
        time.sleep(1)
        createLabel(screen, WHITE, 110, 210, 580, 50, "Money:  " + str(money), BLACK, 40, False, False, True)
        pygame.display.flip()
        time.sleep(1)
        createLabel(screen, WHITE, 110, 260, 580, 50, "Power Balance:  " + str(powerBalance), BLACK, 40, False, False, True)
        pygame.display.flip()
        time.sleep(1)
        createLabel(screen, WHITE, 110, 310, 580, 50, "Water Balance:  " + str(waterBalance), BLACK, 40, False, False, True)
        pygame.display.flip()
        time.sleep(1)
        createLabel(screen, WHITE, 110, 360, 580, 50, "Rank:", BLACK, 40, True, False, True)
        pygame.display.flip()
        time.sleep(1)
        if money < 10000:
            image = pygame.image.load('Bronze.png')
            screen.blit(image, [300, 410, 580, 50])
        if money < 50000:
            image = pygame.image.load('Silver.png')
            screen.blit(image, [300, 410, 580, 50])
        if money > 100000:
            image = pygame.image.load('Gold.png')
            screen.blit(image, [300, 410, 580, 50])
        pygame.display.flip()
    #Will refresh 60 times per second
    clock.tick(60)
```

powerBalance: int, float

I then added a box at the centre of the white screen that has a grey border with a smaller black border within that, this is to add some emphasis to the final results screen and I used the draw.rect function to accomplish this with the width of the rectangle set to 2. I then added each of the variables and a “CONGRATULATIONS” to be outputted on the final results screen via a label with a time.sleep function in between to stagger the results by 1 second each time, using the display.flip function to refresh the screen each second. I then calculated a rank based on how well the user did, based on the money variable. I then designed 3 images to represent 3 tiers of success: Bronze, Silver and Gold.

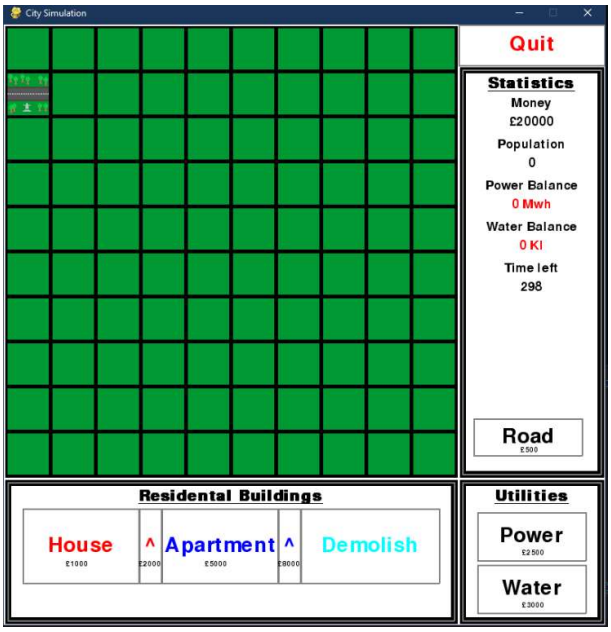
Test Reference 3.4.2

From the image you can see that the borders, congratulations and variable labels are outputted correctly along with the rank image.

```
createLabel(screen, WHITE, 625, 315, 155, 20, "Time left" , BLACK, 25, False, False, False)
createLabel(screen, WHITE, 625, 340, 155, 20, str(300 - round(time.time() - startTime)) , BLACK, 25, False, False, False)
```

I then changed all the second values to 300 seconds, to represent 5 minutes rather than the 5 seconds I was using before to test. I then created a label that outputs the text “Time left” and below that a label that outputs and shows the user the number of seconds remaining before the game ends and the end screen pops up.

Test Reference 3.4.3



You can see from the image that the time left, and seconds remaining labels are outputted correctly on the screen. This is taken 2 seconds after the program has started, 300 – 298 = 2.

```
start = False
startTime = time.time()
beginTime = 0
#Will loop while done is false
while not done:
    #Will run if the user does something
    for event in pygame.event.get():
        if start == False:
            createLabel(screen, WHITE, 625, 365, 155, 20, "Press Enter to start the game" , BLACK, 15, False, False, False)
            if event.type == pygame.KEYDOWN:
                start = True
                beginTime = time.time() - startTime
                createLabel(screen, WHITE, 625, 365, 155, 20, "GAME STARTED!" , RED, 15, False, False, False)
        if start == True:
            if event.type == pygame.QUIT:
                done = True
            elif event.type == pygame.MOUSEBUTTONDOWN:
                #Gets x and y position of the mouse
                pos = pygame.mouse.get_pos()
                #Creates two variables to make the coordinates easier to work with
                x = pos[0]
                y = pos[1]
                if (round((time.time() - (startTime + beginTime)), 1) < 300):
                    #If the user clicks inside the area for the game, it runs tileDeterminer
                    if (x <= 600) and (y <= 600):
                        #passes x, y coordinates and the status of which button is pressed
                        money = tileDeterminer(x, y, depressed, money, screen, population)
                    else:
                        #changes the value of pressed and depressed depending on what button is pressed
                        depressed = buttonDeterminer(x, y, depressed, screen)
    if (round((time.time() - (startTime + beginTime)), 1) < 300):
        #Updates the tiles in the grid
        population, powerBalance, waterBalance, powerTiles, waterTiles = tileChecker(screen, population, powerBalance, waterBalance, powerTiles, waterTiles)
        #Will correct the variables
        money = variableCalc(screen, population, money, powerBalance, waterBalance, powerTiles, waterTiles)
        #Will refresh the screen to display any changes
        pygame.display.flip()
    if round((time.time() - (startTime + beginTime)), 1) == 300.0:
        #Blank Screen and Music
```

I then decided that the user should start the game themselves rather than beginning when the user loads the program. To accomplish this, I created 2 new variables called beginTime and start. Start represents whether the user has started the game and begin time represents the time when they start the game so that it can be added to the start time so that the seconds is restored to 300 when they start the game themselves. I added a label which is below the seconds remaining label and is used to tell the user to press enter to start the game (although this could be any key). If they user then presses down on a key start will be set to true, beginTime will be calculated and the label will change from “Press Enter to start the game” to “GAME STARTED!” in red. Changing the bool start to True allows the other event types to become accessible and allow the user to play the game and the game’s mechanics to become active. I replaced all the time.time() – startTime to time.time() – (startTime + beginTime) to represent this change.

```
time.sleep(1)
createLabel(screen, WHITE, 110, 110, 580, 50, "CONGRATULATIONS", RED, 50, True, False, True)
pygame.display.flip()
time.sleep(1)
createLabel(screen, WHITE, 110, 160, 580, 50, "Population:  " + str(population), BLACK, 40, False, False, True)
pygame.display.flip()
time.sleep(1)
createLabel(screen, WHITE, 110, 210, 580, 50, "Money:  " + str(round(money)), BLACK, 40, False, False, True)
pygame.display.flip()
time.sleep(1)
createLabel(screen, WHITE, 110, 260, 580, 50, "Rank:", BLACK, 40, True, False, True)
pygame.display.flip()
time.sleep(1)
```

I then removed the power balance and water balance variables from the end screen as I think they are irrelevant at showing how well the user did at the game. I also rounded up the money to the nearest integer rather than having a bunch of unnecessary decimal places.

Test Reference 3.4.4

You can see from the image that the end screen is indeed outputted correctly, and the rank image is outputted as intended.

Test Reference 3.4.5

Sprint 3 Final.mp4

From the video you can see that all the elements have come together, and the solution is now a complete game. You can see that the start when the user presses enter is working and starts counting down from 300 then. The label updated correctly to “GAME STARTED!” and the game progressed as expected. The end screen displayed the correct population and money as well as the correct rank image and played the sounds correctly.

4.3: Testing and Evaluation

This sprint involved the test references 3.1.1 – 3.4.5 and these were based upon the original success criteria stated at the start of the project. Here is a copy of the test references that were involved in this project:

Check Road	3.1.1	Does the road appear grey?	Experimental	Press on buttons and tiles	Tiles appear grey when clicked on	Tiles appeared grey when clicked on	Index Error, when clicking on tile at edge of grid	☑	☑
	3.1.2	Does the road get randomly placed?	Verify	Run Solution	Road gets randomly placed on program start	Road was randomly placed on program start	-	-	☑
	3.1.3	Does the road system work as intended?	Experimental	Press on buttons and tiles	Objects can only be built next to a road	Objects could only be built next to a road	-	-	☑
Create Proper Aesthetic	3.2.1	Does the image get drawn in correctly?	Experimental	Press on buttons and tiles	House image appears in place of a colour	House image appeared in place of a colour	-	-	☑
	3.2.2	Do the building icons get drawn correctly?	Experimental	Press on buttons and tiles	Building icons appear correctly	Building icons appeared correctly	-	-	☑
	3.2.3	Can you only place buildings in 4 directions?	Experimental	Press on buttons and tiles	You can only place buildings directly next to roads	You could only place buildings directly next to roads	-	-	☑
	3.2.4	Does the correct road get drawn?	Experimental	Press on buttons and tiles	The correct road is drawn depending on the tiles around the road	The correct road was drawn depending on the tiles around the road	-	-	☑
	3.2.5	Does the road connect to the right buildings?	Experimental	Press on buttons and tiles	The roads only connect to each other and the power and water tiles	The roads only connected to each other and the power and water tiles	-	-	☑
Add Wealth System	3.3.1	Does the house upgrade button get drawn?	Verify	Run Solution	The upgrade button is drawn in correctly	The upgrade button was drawn in correctly	-	-	☑
	3.3.2	Do the updated buttons get drawn correctly?	Verify	Run Solution	The new and updated buttons are drawn correctly	The new and updated buttons were drawn correctly	-	-	☑
	3.3.3	Does the upgrade button work?	Verify	Press on buttons and tiles	The low wealth house will go to a medium wealth house	The low wealth house did go to a medium wealth house	-	-	☑
	3.3.4	Does the road disregard different levels of buildings	Verify	Press on buttons and tiles	The road won't change based off the level of house	The road didn't change based off the level of house	-	-	☑
	3.3.5	Does the music and upgrade system work as intended?	Experimental	Press on buttons and tiles	The buildings can be upgraded and the sound will play when something is built or upgraded	The buildings could be upgraded and the sound did play when something is built or upgraded	-	-	☑
ime-based and End Menu	3.4.1	Does the end-music play and does the screen go white?	Verify	Run Solution	After 5 seconds the screen will go white and the end music will play	After 5 seconds the screen did go white and the end music did play	-	-	☑
	3.4.2	Does the end-screen get outputted correctly?	Verify	Run Solution	The end screen should output the labels, values and rank image correctly	The end screen did output the labels, values and rank image correctly	-	-	☑
	3.4.3	Does the time left and remaining seconds labels get outputted correctly?	Verify	Run Solution	The time left and value for the remaining seconds should be outputted correctly	The time left and value for the remaining seconds were outputted correctly	-	-	☑
	3.4.4	Does the end screen get outputted correctly?	Verify	Run Solution	The end screen labels and image should get outputted correctly	The end screen labels and image should did get outputted correctly	-	-	☑
	3.4.5	Does the game work successfully and does the start time when enter is pressed work successfully?	Verify	Run solution and play game	The game starts when the user pressed down enter and the game is then playable with an end screen	The game started when the user pressed down enter and the game was then playable with an end screen	-	-	☑

Overall the sprint was a success, however I did change some of the original success criteria in order to more successfully fit the aim and nature of the game:

1. Check Road – Checks the placed object is next to a road and if not, then the user is unable to place the desired object.
2. Assign proper aesthetic – Create a system whereby empty tiles are green to represent grass and residential and utility buildings have different colours based on their purpose to allow the user to more easily see what services they have and what they need.
3. Add wealth-system – Create 3 types of residential housing, $, $$, $$$ to illustrate the various levels of wealth of each house. The amount of services required will increase as you go from $ to $$$ as to increase the difficulty. Have these assigned different colours or graphics.
4. Save Game – Add an ability for the user to save their game to resume it another time through saving the grid, money and population in a text file.
5. Default Map – Add a default/new-game grid, money and population to load when a new game is started.
6. Main Menu – Create a start menu that comes up when you load the program that will allow the user to select load save game and start new game.

The first 3 success criteria and the related development and design sections went very successfully, with the only bug being in the check road part of the development where I encountered an index error while trying to place buildings and roads at the edge of the grid and fixed this with conditional statements for those edge cases. The last 3 success criteria I did not complete as I replaced these with better ideas and solutions. Instead of a save game function I used a time-based method whereby the game only goes on for a certain amount of time to keep the game fast paced and fun. This makes the save game feature irrelevant due to the now time-based nature of the game. For the default map criterion, I already have this in the game, so I did not need to design or develop this. And instead of a main menu, I replaced this idea with one

where the user presses the enter button to start the game rather than clicking on more buttons so that the user can start the game when they want to rather than when the game loads in the first place. This also allows the user to get ready to begin placing buildings the moment that the user starts the game rather than having to move the mouse pointer across the screen to do so.

I will now move onto an evaluation of the entire project and assess how successful my entire project has been.

5.0: Evaluation

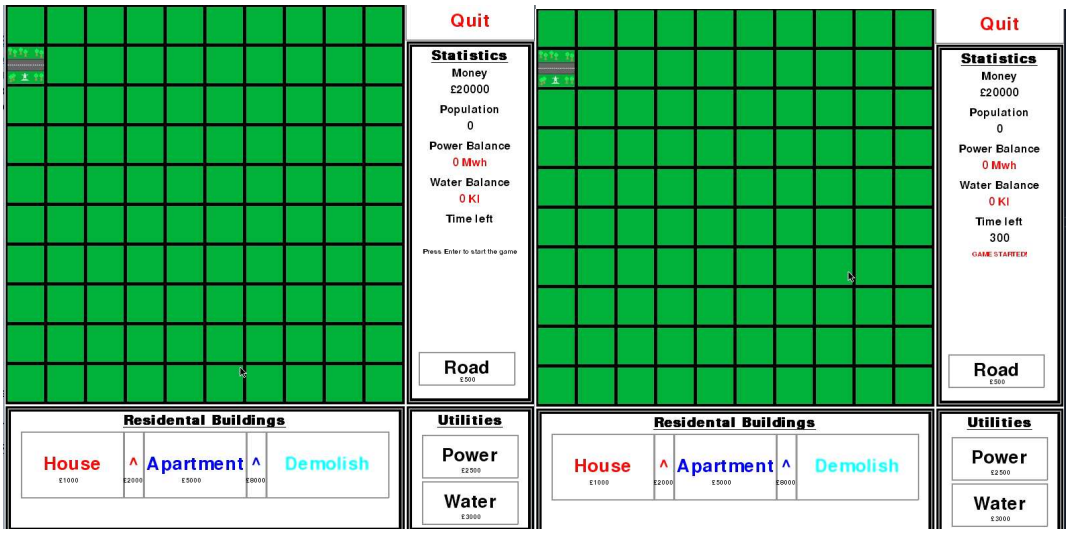
5.1: Post Development Testing for functionality and robustness

I will need to do some post-development testing on the functionality and robustness of the solution. If I find any bugs or errors in the post-development testing, I will not fix these but instead give ways they could be fixed. The areas I will test will include:

1. Time – Does the game start when the user presses a key down, and can you play the game before this start time?
2. Placing Buildings – Can you place a building anywhere on the grid and if not does it have to be next to a road?
3. Money – Does it increase with the population and does it decrease when objects are placed?
4. Buttons – Can you click on multiple at once and what happens if you click on the same one twice?
5. Labels – Do they get redrawn when the button is clicked on and do the ones in the statistics column get updated correctly?
6. Quit – Does the quit button work as intended?
7. Roads – Do the roads orient themselves correctly when the grid is updated and do, they connect to houses and apartments?
8. End Screen – Does the end screen appear when the time left is 0, does it play the music, and do the variables appear as intended?

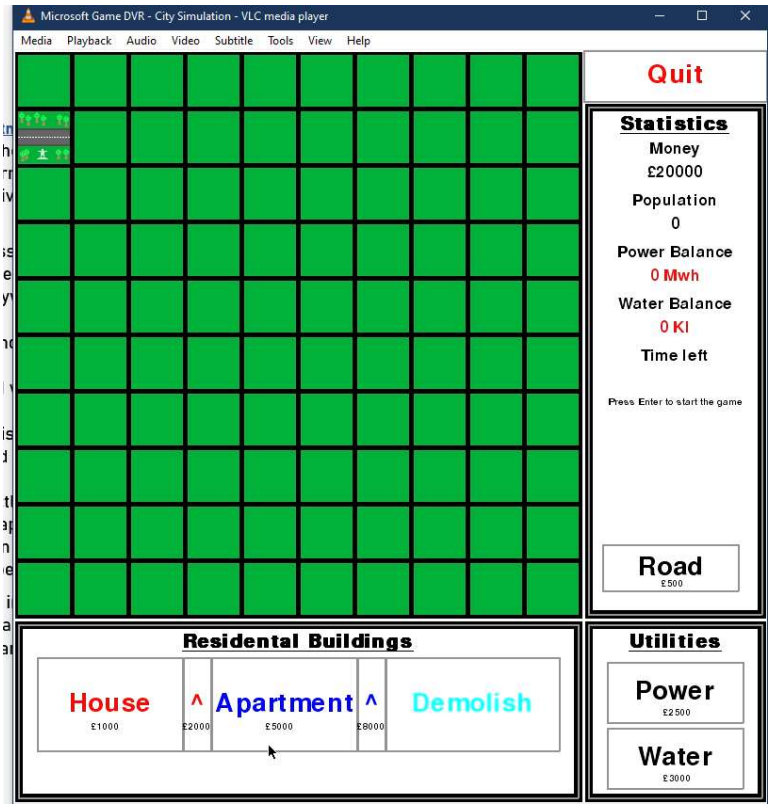
The tests referenced in this section will be contained in the "Functionality and Robustness" sheet of the Testing Table.xlsx file much like the sprint tests in the "Main Table" sheet of the same file.

Test Reference 1.1



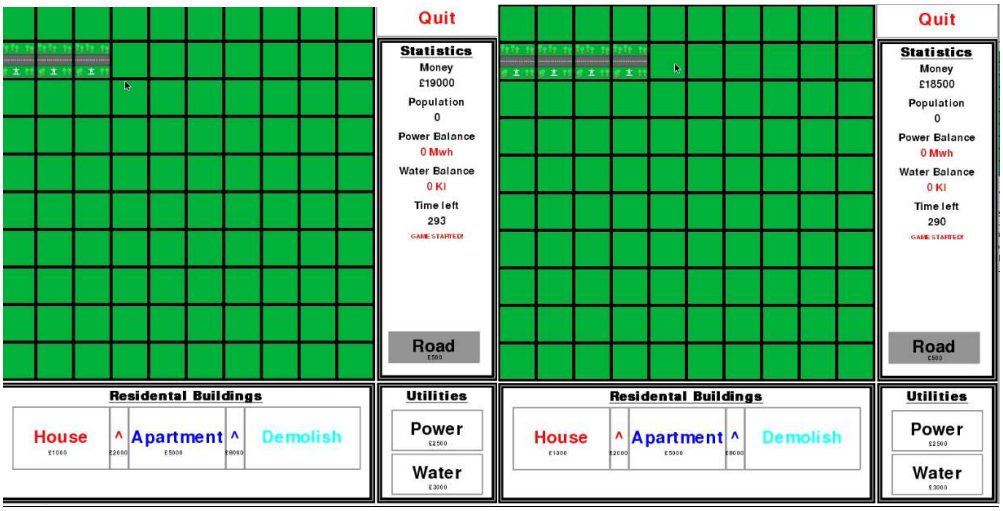
As you can see from the two pictures, the label changes when the enter key is pressed from "Press Enter to start the game" to "GAME STARTED!". The time remaining also appears, this is evidence to suggest that the game starts when the user presses a key. Satisfying 1.1.

Test Reference 1.2



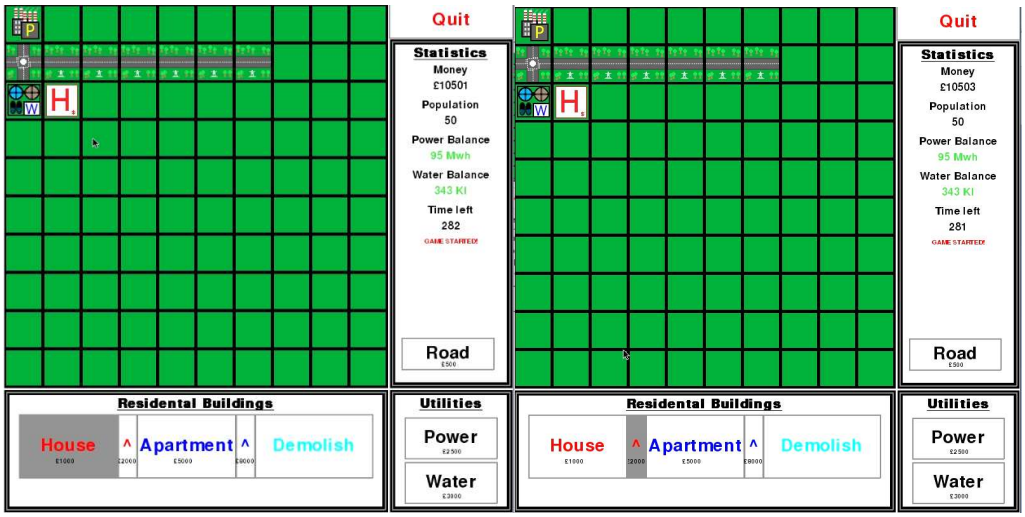
From the picture I am trying to click on the apartment button, but this is however, not allowing me to, and you can see this as the button is not depressed in the picture after I have clicked on it. This shows that the game cannot be played before the user has pressed enter and the game has started. Satisfying 1.2.

Test Reference 2.1 and 2.2

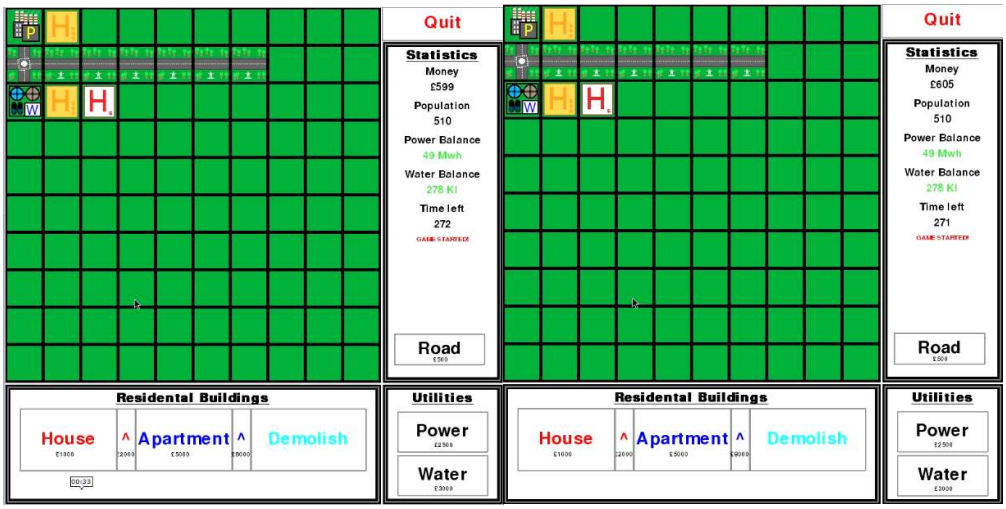


From the pictures you can see that I have tried to click on the tile diagonally opposite a road tile and the road hasn't been placed, showing that you cannot place a building anywhere. The second image also shows that you must place a building next to a road for it to be built and to show on the screen. Satisfying 2.1 and 2.2.

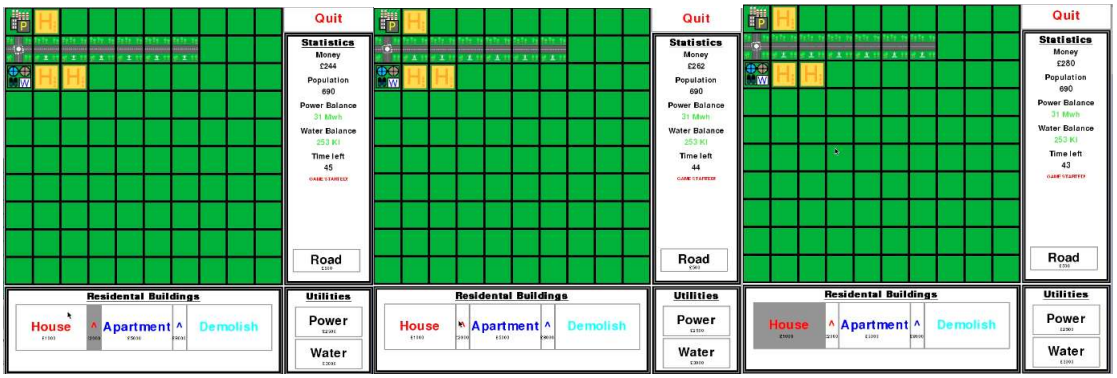
Test Reference 3.1 and 3.2



From the above images you can see that the single level 1 house has increased the money variable by 2 in 1 second. This shows that the population increases the money, but below you will see that 2 level 2 houses and 1 level 1 house has increased the money by 6 in 1 second, proving that the money scales with the population. As I have built more houses and upgrades and the money has been reduced, this also shows that the money is reduced when an object is placed by the correct amount. Satisfying 3.1 and 3.2.

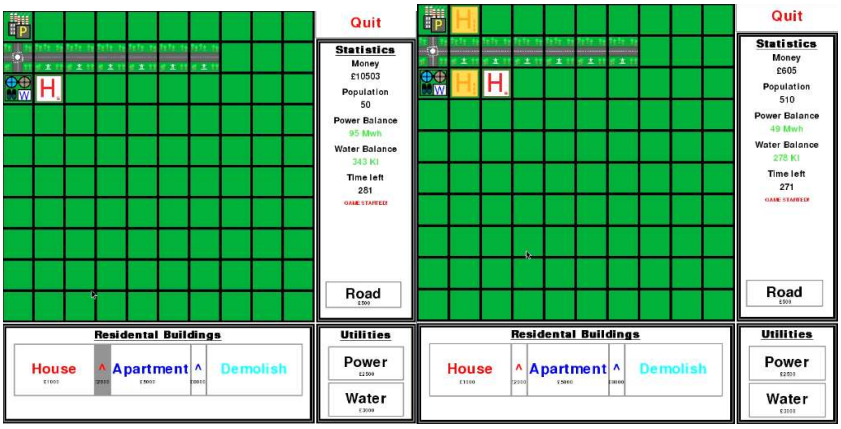


Test Reference 4.1, 4.2 and 5.1



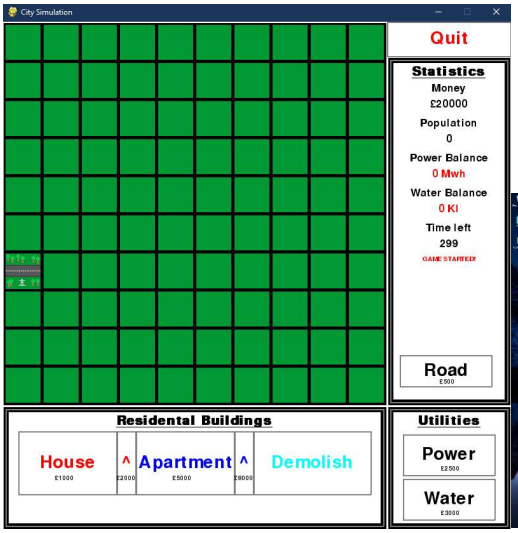
From the images above, you can see that I couldn't press on the house button when I had the house upgrade button selected and had to deselect the button in order to be able to select the house button and place a house. This shows that you cannot click on two buttons at the same time and shows that the visual state of the button changes when you click onto it again, to represent being deselected. You can also see that the labels, which represent the cost of placing each building, is redrawn when the state of the button changes and is selected or deselected. Satisfying 4.1, 4.2 and 5.1.

Test Reference 5.2



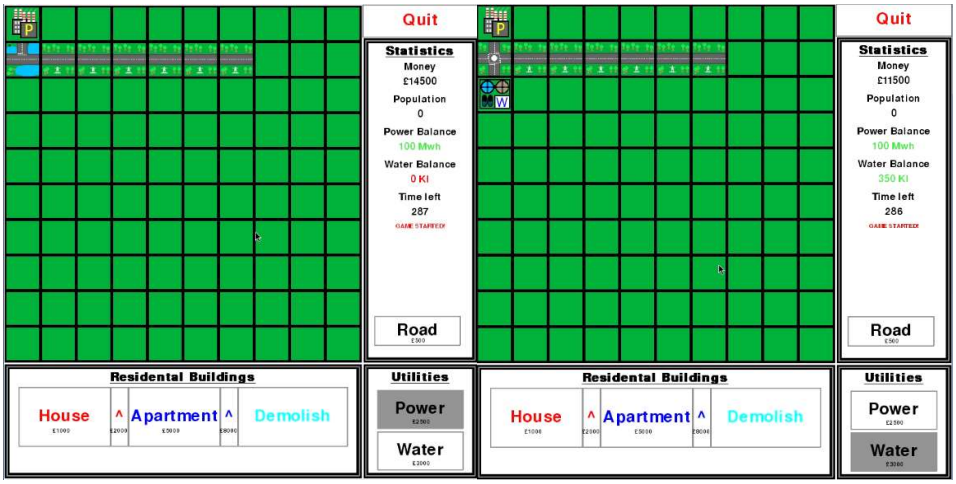
The images show that the labels represent each variable such as money, population, power balance, water balance and time left. All these variables update correctly from the first image to the later second image and show that the labels change correctly when they are redrawn. Satisfying 5.2.

Test Reference 6.1

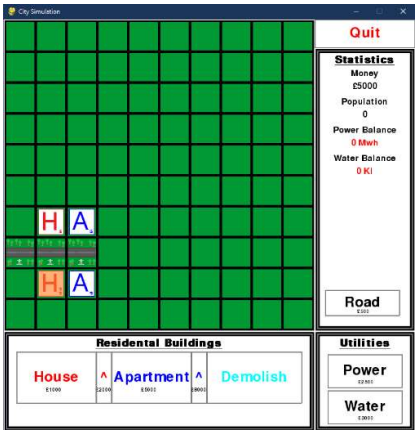


The left image shows the game just before I click on the "Quit" button, when I do so it exits me to my desktop as the game has quit as demonstrated below. Satisfying 6.1.

Test Reference 7.1 and 7.2

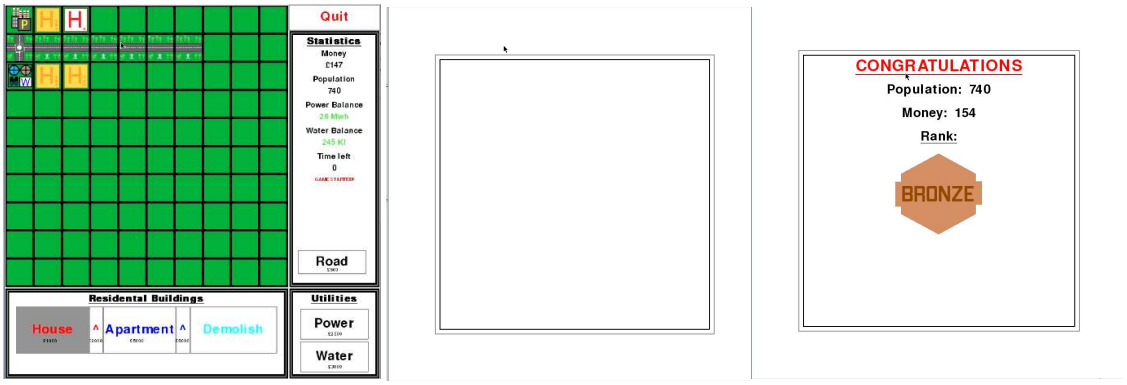


The above images show that when the grid is updated, with the new water building for example the road automatically changes to the correct orientation when the grid is updated. Satisfying 7.1.



The image to the left also shows that when the grid is updated the road doesn't orientate to the houses when they are placed next to it. Satisfying 7.2.

Test Reference 8.1, 8.3



The images above show that after the time reaches 0, the white screen appears with a grey and black border. Showing that the end screen appears when the time is 0. Then the variables appear as intended, after they come up one by one. Satisfying 8.1 and 8.3

Test Reference 8.2

Sprint 3 Final.mp4

In the video, you can hear the sounds at the end when the end screen appears, and when the buildings are being constructed. Proving that it does play the end music. Satisfying 8.2

5.2: Usability Testing

Now I have developed my solution I must allow the stakeholders to give feedback on my game and highlight anything that could be improved and highlight any bugs within the solution. I asked them these questions for each feature:

- How easy did you find this feature to use?
- Are there any bugs with this feature?
- Are there any improvements to this specific feature that could be made?
- Highlight some of the good aspects about this feature

Features to be tested:

- Starting the Game
  - Pressing a key to start it
- Placing buildings
  - Grid Placement
  - Road Placement
- Money
  - Rate of increase based on population

- Buttons
    - Use of to control game and select which buildings to place
    - Deselect function
- Labels
    - Drawn correctly?
- Quit Button
- Roads
    - Automatic orientation
    - Placement requirements
- End Screen
    - Appearance
    - Music
    - Variable values match ones in game

The results from carrying out this usability test are summarised below:

- Some people found that they would accidentally press a key that wasn't enter and it would start the game unintentionally
- Some people found that the grid was far too large based on the small time given and the amount of time should be increased higher than 5 mins
- Some people found that the population didn't give enough money, leading to people being idle not doing anything for long periods of time
- Some people found it annoying that they had to deselect a button to click on another
- Some wanted more statistics on the end screen rather than just the population, money and rank
- Not much variety in the number of buildings available

5.3: Evaluation of Solution

I will now compare the final solution to the original success criteria, this will allow an overall conclusion to be used to decide whether the solution is suitable and if it will help the problem that it aimed to solve. It will also include maintenance issues, limitations and how the solution could be developed further.

Success Criteria 1 – Game Environment

My solution creates a window, which is 800x800 pixels wide and contains a grid of same-sized tiles which can be used to place buildings and roads inside, this is shown by test reference 1.1.1 which shows how a window of 800 x 800 does appear and test reference 1.1.2 which shows that a grid does get created and an array of the grid is printed in the console to show that the grid is array-based and can be changed to include various buildings and roads. This is further shown by the post-development tests which show that a window and grid do appear for the user to place objects inside of.

However, some users said that the number of tiles in the grid was too high for the time given as they couldn't all realistically be filled in during the 5-minute duration of the game. I could improve this in the future if I increased the duration of the game to allow for the user to create a massive sprawling city instead.

Success Criteria 2 – Detect Grid

In the solution the user input is detected when they click inside of the window, this is shown by test reference 1.2.1 where "Hello" is outputted to the console when the user clicks inside the window. This input is then converted to the x and y coordinates corresponding to the pixel clicked on in the grid and this is shown in test reference 1.2.2 where the coordinates are outputted to the console. If they click within the area the grid has been created in then it will convert the coordinates into the row and column values for that tile in the grid array and this is shown by test reference 1.2.3 where these row and column values are outputted to the console. The tile that the user has clicked on with these row and column values are then changed depending on a number of factors like money and current tile value, but the general idea of it being changed once clicked on is shown by 1.2.4 which proves that when the user clicks on a tile it changes colour to demonstrate that the tile has

been clicked on. This allows the user to place buildings later when I developed the icons for these buildings instead of using colours.

Success Criteria 3 – Create Button

In the solution I created a function to create a rectangle with text at its centre, this was then used to create buttons that allow the user to select which buildings they would like to place later. I prove that I can create this rectangle with text in the middle through test reference 1.3.1 where I create two buttons with differing dimensions, this allows me to create the differing sized buttons of the power and house buttons.

Success Criteria 4 – Place Object

In the solution I made it so that the buttons change colour once pressed on to represent that the user has selected to place that specific building or road, this is shown by test reference 1.4.1 which shows that the button turns grey once clicked on. I then made it so that once the button is depressed the tile will turn red to simulate placing a building, shown by test reference 1.4.2 and this is used in later development to change the picture of what is in each tile. I then edited my code so that when the user clicked on the button again it would turn white, to simulate deselection, this is proven by test reference 1.4.3.

Some users didn't like having to unselect the button every time they wanted to select a new one when I carried out the usability test on my stakeholders, I could alter this in the future by automatically deselecting a button once you click on another button.

Success Criteria 5 – Exit Button

I created a button that quits when pressed, this is so that the user can leave the game at any point if they wish to. I show that this works in test reference 2.1.1 as the program quits when the user clicks on the quit button.

Success Criteria 6 – Create Multiple Buttons

There are multiple buttons in the game, and these originally included: red, blue, clear and quit. But later these were renamed house, apartment, demolish and quit. I used the same function when creating these buttons and then can all be selected and deselected in the same way but not at the same time. I have shown multiple buttons work at

the same time through test reference 2.2.1 and that I can revert the value of a tile to its original, pre building placement, through test reference 2.2.2 where I used the clear button to remove the colour change. Originally the buttons represented colour changes, but these changed to buildings at later stages of development and I added a few extra buttons such as power, water and the upgrade buttons. I have allowed the user to select which building they want to place and place it using these buttons.

Some users found that the number of buildings and therefore buttons was limited when I carried out the usability test, I could improve this by adding more buttons with different buildings such as shops and factories.

Success Criteria 7 – Create Statistics UI

In the solution there is a box at the right side of the window which is used to contain all the information about the game in one box. This box holds information such as money, population, water and power balance and time left. I have shown that the text and box can be drawn correctly and as intended through test references 2.2.3 and 2.2.4, I also show this in my post-development tests as the variables are updated correctly in the box as the game progresses.

Success Criteria 8 – Assign Wealth

During the game, the user places buildings and roads that have an assigned cost through selecting buttons and clicking on the grid, these costs are represented by the label underneath each button highlighting the costs of placing each building. The player starts with 20000 as the money variable and if they change a tile to a different one using the buttons the associated cost is taken off the money variable, this money variable changes due to success criteria 12 but is shown that the money variable and labels under the buttons change in test reference 2.2.5.

Success Criteria 9 – Assign Population

The various house and apartment buildings have an associated population, this is used to determine the needs from the population and the amount of money that the user will get from them. This amount of people is static and can be changed by deleting the building and placing another or by upgrading the existing building as upgrading the houses

and apartments leads to higher population values. I have shown that the population increases by the number of houses and apartments through test reference 2.2.6.

In the usability test some users found that the population didn't give enough money to the user and so they were left idle for long periods of time, I could improve this by changing the value of how much each building adds to the population variable.

Success Criteria 10 – Create Utilities

In the game the population will only generate money if the powerBalance and waterBalance variable are positive, the values of these are calculated based on the population and the number of power and water tiles present in the grid. This is used to represent the real world needs of a population and makes the game more realistic. I have proven that the buttons and labels associated with the utilities are created and act as intended through test reference 2.2.7 and 2.2.8.

Success Criteria 11 – Timed Game

This criterion was already fulfilled when I started using the pygame library and you need to use a clock to update the grid and label values when the grid is refreshed every 1/60 second.

Success Criteria 12 – Maintenance Costs and Taxes

Originally, I was going to create a system whereby the utilities (power and water stations) reduce the money variable by a set amount while each residential building increases the money variable by a set amount, with the intention to act as a sort of tax. I decided to replace this with a system of supply and demand as stated in criterion 10. This was a much better way of doing it as it is more realistic as without water or power, people are not going to live in the buildings and therefore won't pay tax. This is where I have added in more statistics to the statistics box to represent the variable and this is proven by test references 2.2.9, 2.2.10, 2.2.11, 2.2.12.

Success Criteria 13 – Check Road

In the game when the game starts a road is randomly placed around the edge of the grid, proven by test reference 3.1.2 and other roads or buildings can only be built around this road and other, joining roads as

long as the roads are one of above, below, right or left of the building, proven by test reference 3.2.3. This is proven by test reference 3.1.3 and I prove that the roads do get placed in 3.1.1. They also automatically orientate visually to match the placement of other roads and the power and water buildings but not the residential buildings as proven by test references 3.2.4 and 3.2.5.

Success Criteria 14 – Assign Proper Aesthetic

Instead of changing the tile's colours I instead decided to make empty tiles green to represent grass and residential and utility buildings to have proper building pictures and icons, so they look like the houses and apartments. I designed pictures for each building level to make them distinct from each other and shown that they are drawn correctly in test references 3.2.1 and 3.2.2. This makes the game a lot more visually appealing to the user and adds a lot more visual depth to the game. Alongside this I also added a sound for when buildings are placed or upgraded so that there is an audio depth to the game to make it more appealing, test reference 3.3.5.

Success Criteria 15 – Add Wealth System

In the game the user can choose to upgrade both the houses and apartments via buttons at the right side of the base building buttons, this allows the user to have a lot more variety of strategies and experiment with when they should or shouldn't upgrade the buildings. I have shown that these new buttons are drawn correctly through test reference 3.3.1 and I updated the size of the original base buttons to centre their text in a now smaller button due to the new upgrade buttons, shown by test reference 3.3.2. These upgrade buttons allow the user to click on an existing house or apartment and upgrade it to the next wealth level, providing more population and therefore money a second, this is done by changing the value in the land array and is shown the work in test reference 3.3.3. These upgraded buildings also act the same with roads as base houses and apartments, shown by 3.3.4.

Success Criteria 16,17,18

At sprint 3 I decided against these success criteria so I couldn't complete these, replacing them with one success criterion: "Time-

based and End menu". As I was making it Time-based there was no need for a save game option as this would defeat the point of the fast-paced nature of the time-based game. There was already a default map and so success criteria 17 was already filled from the start. Instead of a main menu, I instead used a way of the player starting the game with a key press.

New Success Criteria 16 – Time-based and End menu

The game is timed from when the user presses a key once the program has been opened, this means that the user doesn't have to start the game as soon as the program opens and allows them to plan based off of the initial location of the road before they start. When the user enters a key, they'll have 300 seconds to get as much money as possible and after that time the game processes stop and a white screen appears while music plays signifying the end of the game, shown by 3.4.1 and 3.4.5. The user is shown how much time they have left to earn the money, 3.4.3. While the end screen also shows the end, variable values such as money, population and a rank based on the money variable, 3.4.2 and 3.4.4.

When I conducted the usability test, some users wanted more statistics on the end screen, I could improve this by adding more information on the end screen such as the number of buildings and what type of buildings they have built the most of. I also found that some users accidentally started the game by pressing a button before they meant to, inadvertently starting the game, I could improve this by changing the event type to only the enter key and not just any key press.

Overall, the solution fulfilled a large amount of success criteria, and changed some to suit a more engaging and fun experience for the user. There are, however, still some issues with the usability and some features that the stakeholders would like, further development would be needed to fully solve the problem my solution aims to solve. It does help children and teenagers build a sense of the value of money and how to most effectively use it through various strategies and methods, but needs to include some more realistic features to more a better solution for the problem such as shops and factories.

5.3: Maintenance and Limitations

A limitation of my solution is that it contains a set size of grid, and that means that if the user gets to a point when they have filled the entire grid they cannot progress any further, future development on the solution could lead to the game having more grid sizes so that the user has to adapt to a more expansive environment in the game.

Another limitation of my solution is that the game has a set difficulty, with the amount of money being constant at the start of the game, in future development the game could be made harder with features like maintenance for roads or houses to make it so the larger the city gets the higher the maintenance and so the mistakes of the player and planning the city become a lot more important. You could also increase the difficulty by adding a varying size of starting money so that the player must make more wise and difficult choices on what to build at the start of the game to maximise their efficiency.

An additional limitation could be the lack of building variety in the game, with only the houses and apartments to upgrade and build, the player may need more variety as highlighted in the usability report. A solution to this would be to add more building types and upgrades through the introduction of more buttons and labels that represent buildings such as shops, factories, parks or any other types of buildings which would be a useful addition. Along with these types of buildings could come new, engaging variables. These could include employment which could be based off the number of factories and the population, happiness could be based on the number of parks and people and consumer goods based on the number of shops and people. This could mean that some of the other limitations could be avoided such as the set difficulty and with the more complex buildings and variables, the difficulty would scale with the size of the city.

Another limitation is that the user has no way of storing their scores or ranks after the game has ended, this could be improved by adding a way of storing the high scores and then coming back to them at a later time to compare the progress the user has made. This could also tie in well with a main menu system, which I failed to develop for this solution but could be developed as an additional feature in the future. The user would then be able to access the many extra features that I would say could be developed in the future such as the size of the grids,

difficulty increases through different levels of money at the start, could all be accessed through the main menu when the user chooses to start a new game.

5.4: Bibliography

Used for	Link
Construction Noise	http://www.orangefreesounds.com/
Pygame Help	https://pythonprogramming.net/
	https://nerdparadise.com/
Endgame Noise	http://soundbible.com/