H446 A24cmp - 24752 Baldwin, James

Analysis:

Project Identification and Summary of Solution:

I am planning on making a military-themed tower defence style game where the player is challenged to defend their base against waves of enemies. It will be a 2D game featuring a variety of towers with different abilities that the user can strategically place on the board to defeat the enemies. Users can purchase new towers or upgrade current ones using a currency system which they earn for defeating individual enemies and completing a level. There are multiple types of enemy, with different health and speeds, as the game progresses. To create the game, I will use the p5 library with javascript and a modular approach. To make the game challenging and engaging, I will implement levels with increasing difficulty and multiple maps, each with a unique gameplay feature (e.g. all enemies move faster) and a boss at the end of the map, making my tower defence game unique. To make the game user friendly, I will clearly show the instructions and use sound effects/visual cues to provide feedback to the player.

A1 - Stakeholders:

Stakeholder	**Description**	**Role**	**Stakeholder Requirements**
William Thorne	Year 12 Student at Farnborough Sixth Form College. William is a casual gamer but has little programming knowledge. William also has poor eyesight, even with glasses (which he has worn since childhood).	Casual User	William would like an engaging game with many game modes (he suggests campaign and arcade). He is not a competitive gamer so is only looking to play a game casually a few times to cure boredom. William needs the game to have contrasting and bold colours with large text and a clear font. He needs these features so he can clearly see the game.
Dominic Keyworth	Year 12 Student at Farnborough Sixth Form. Dominic is a casual gamer and has little programming knowledge. Dominic has a special	Casual User	Dominic would like towers from multiple military eras so he doesn't get bored playing with the same towers over and over. This is because he is both a

	interest in this project due to his love for military history.		casual gamer and a military history enthusiast. Dominic would like multiple maps, each from different eras. By incorporating this with the previous requirement, he would like the maps and towers to be from the same era, e.g. a desert map based on the Gulf War should have towers used in said conflict. Dominic has never played a tower defence game so would need the game to be explained clearly and possibly even a tutorial mode. This is so he can understand the game mechanics and rules before he starts playing.
Sam Russell	Year 12 Student of Computer Science at Farnborough Sixth Form College. Sam is a competitive gamer and likes comparing his score to others.	Regular User	Sam would like a leaderboard. This is because he plans on playing the game seriously and wants to compare his scores to other players. Sam would like a way to save his progress and return later. This is because he wants to eventually complete the game and doesn't want to sit there for hours at the time trying to do this, he would rather play it across multiple small sessions.

Harvey Miller	Year 12 Student of Computer Science at Farnborough Sixth Form College. Harvey is an experienced programmer and he runs a blog where he reviews video games.	Game Reviewer	Harvey rarely has time to sit down and play games at a computer desk between college and his blog. He would like the game to be available on a mobile phone and to interact with the game via a touchscreen. Harvey would like the game to be unique enough to write a review about to stand out from other similar games. Unique features will help him write an engaging review about my game.

A2 - Research:

Some existing solutions include Bloons TD 6, Plants vs Zombies and Toy Defense

Bloons TD 6:

Figure 1: Bloons TD 6

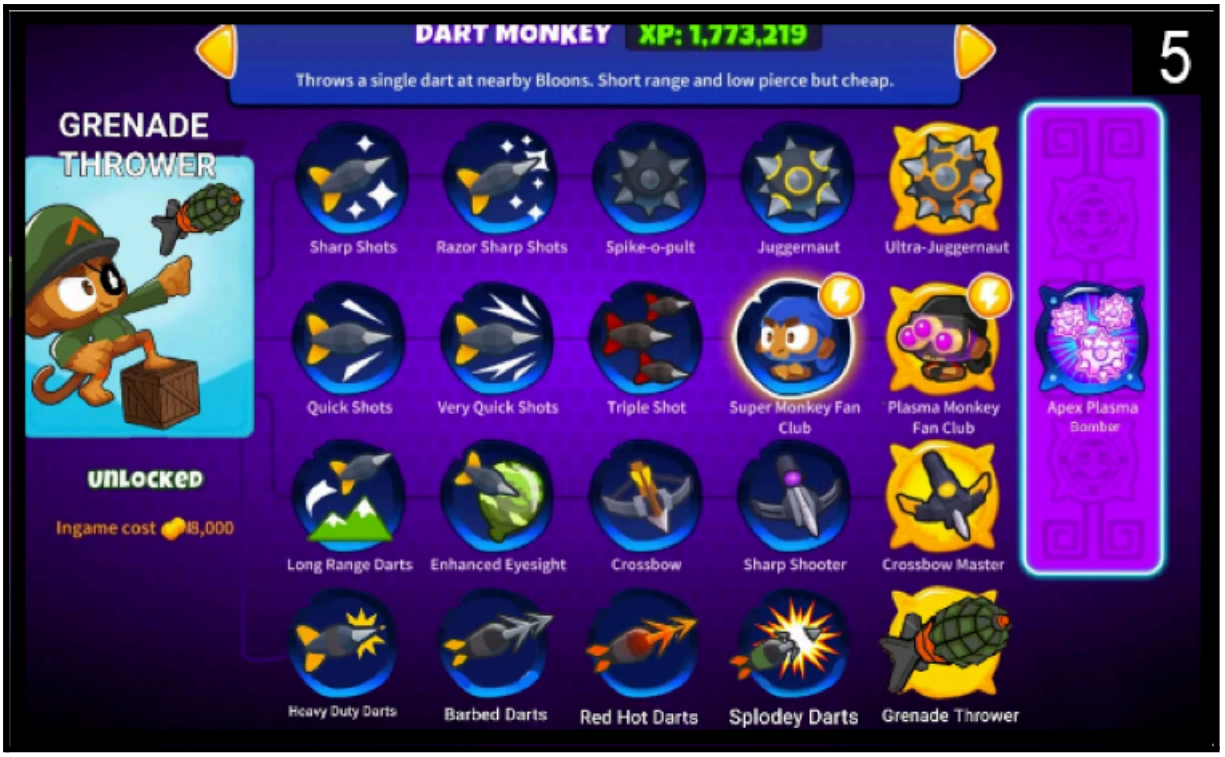


Figure 2: The tower upgrade screen for Bloons TD 6

Figure 3: Multiple maps from Bloons TD 6

1. Towers and prices clearly shown on the right. This allows the player to quickly select a new tower and place it. Towers which the player cannot afford are also greyed out to make this clear.
2. Base health, coins and level number all clearly displayed. This is all important information for the player and is displayed at all times while the game is running.

3. Multiple tower types and enemies all revolving around the central theme of monkeys and balloons respectively. This makes all the towers and enemies similar to each other to immerse the player in this world. It also keeps with themes of previous games from the franchise.
4. Cartoonish and child-like feel. This means the game is not overcomplicated with a relatively simple look and makes sure the game is kid-friendly.
5. Multiple upgrades and upgrade paths allowing users to fully customise each tower. This adds depth and repeatability to the game since there are multiple upgrade paths. It also allows players to customise the towers to their likes and needs for that level.
6. Multiple enemy types. This increases difficulty to the game since the speed and health is different for each enemy type meaning the player has to adjust their strategy accordingly.
7. Effective 2.5d perspective, not overly complicated but still has a modern and cartoonish (characters popping out) feeling.
8. (Not shown in any figure). Multiplayer adds an element of teamwork to the game since players are challenged to work together to defend the base. It could also increase the popularity of the game since players are more likely to recommend and play it with friends.
9. Multiple maps increase the repeatability of the game since each map has a different route for the enemies meaning there are different spots where it is best to place towers. It also makes the game more engaging since the player is not greeted with the same background the whole game.
10. Multiple weapon types (fire, plasma, etc.) each having different effects on enemies. This increases difficulty and repeatability since a fire tower may deal damage slowly over time; an ice tower may slow an enemy down; meaning the player has to strategically place their towers to get the most out of these effects and they may want to replay the level with different tower types.
11. (Not shown in any figure). Certain maps have obstacles which units can’t see over. This increases difficulty since blind spots for towers are created and the player has to work with this.

Plants vs Zombies:

Figure 4: Plants vs Zombies

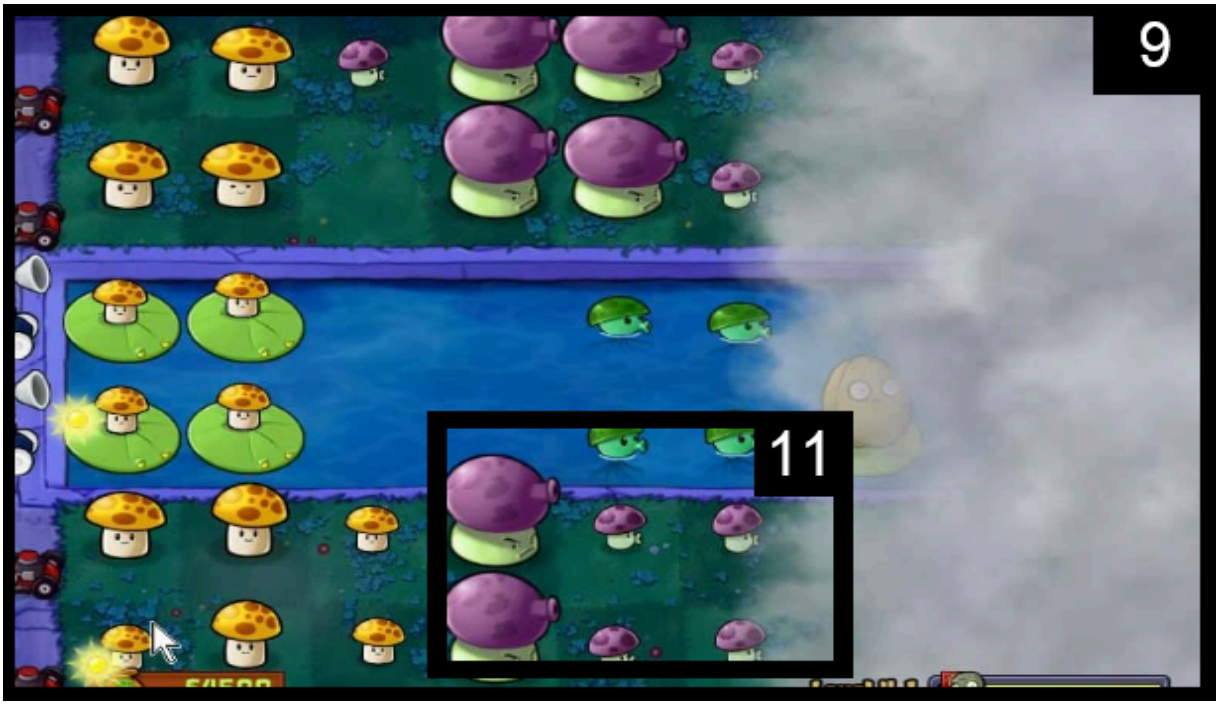


Figure 5: A different (foggy) map from Plants vs Zombies

1. Towers and prices clearly shown at the top. This allows the player to quickly select a new tower and place it. Towers which the player cannot afford or are on cooldown are also greyed out to make this clear.

2. Level and how long is left in said level shown in bottom right. This is important information for the player and is displayed at all times while the game is running.
3. Menu button in top right. This is displayed at all times while the game is running and allows the player to quickly edit settings such as volume or return to the home screen.
4. All towers are based around a central theme of plants/gardens and all enemies are zombies. This makes all the towers and enemies similar to each other to immerse the player in this world.
5. Currency (sun) is given randomly throughout the round and can be produced by the player using a sunflower. Displayed in the top left. The player doesn't receive currency for killing enemies or completing the round, unlike other tower defence games. This means the player has to think carefully about how they spend this and makes the game more challenging.
6. Cartoonish and child-like feel. This means the game is not overcomplicated with a relatively simple look and makes sure the game is (relatively) kid-friendly.
7. Effective 2.5d perspective, not overly complicated but still has a modern and cartoonish (characters popping out) feeling.
8. Multiple enemy types. This increases difficulty to the game since the speed and health is different for each enemy type meaning the player has to adjust their strategy accordingly.
9. Multiple maps increase the repeatability of the game since each map has a different route for the enemies meaning there are different spots where it is best to place towers. There is also water on some maps, which towers cannot be placed on without placing a lilypad first, and fog on others, which covers the very right of the screen covering any zombies beneath it. It also makes the game more engaging since the player is not greeted with the same background the whole game.
10. Multiple weapon types (fire, ice, etc.) each having different effects on enemies. This increases difficulty and repeatability since a fire tower may deal damage slowly over time; an ice tower may slow an enemy down; meaning the player has to strategically place their towers to get the most out of these effects and they may want to replay the level with different tower types.
11. On maps which are set at night, there are new towers (mushrooms) which only work on these maps. This helps make the game more engaging and possibly challenging by adding more towers which the player should use if they want to beat certain levels, pushing them away from towers they may prefer to use.

Toy Defense:

Figure 6: Toy Defense

Figure 7: A different map from Toy Defense

Figure 8: The skill tree from Toy Defense

1. Coin count and health clearly shown in top left. This is important information for the player and is displayed at all times while the game is running.
2. Towers and prices on bottom right. This allows the player to quickly select a new tower and place it. Towers which the player cannot afford are also greyed out to make this clear.
3. All towers and enemies are based around the central theme of toy soldiers. This makes all the towers and enemies similar to each other to immerse the player in this world.
4. Special abilities on the right. These make the game more complex by giving the player the option to affect the enemy behaviour, e.g. slowing them all down, adding layers and strategy to the game as to when they should use the abilities.
5. Multiple maps the repeatability of the game since each map has a different route for the enemies meaning there are different spots where it is best to place towers. It also makes the game more engaging since the player is not greeted with the same background the whole game.
6. Can sell and upgrade towers. This adds layers to the game as the player may have to decide between buying a new tower or upgrading a current one.
7. (Not shown in any figure). Can save some placed towers after the round and get a discount on placing these at the start of the next round. This idea is unique to this game and encourages players to upgrade their current towers instead of spamming low level towers all over the map.
8. Skill tree. This increases the repeatability of the game since the player can customise the way every tower plays (e.g. increase damage for all towers, increase fire rate for all towers) and may want to replay with a different skill tree setup.

9. Pause and settings in top right. This is important information for the player and is displayed at all times while the game is running.
10. Simpler 2.5d perspective, no overcomplicated tower models and 2.5d effect only created by shadows.
11. Obstacles take space on the map. This makes the game harder as some spots on the map are weaker meaning the player has to consider placing better towers around these obstacles to counteract the fewer towers possible. In this case, it also makes the game more realistic since barbed wire is common on a battlefield (or at least in this era).

Features from my research which I will incorporate into my solution:

- Towers and prices clearly shown on the right. This allows the player to quickly select a new tower and place it. Towers which the player cannot afford are also greyed out to make this clear.
- Multiple tower types and enemies all revolving around a central theme. This makes all the towers and enemies similar to each other to immerse the player in this world.
- Base health, coins and level number all clearly displayed. This is all important information for the player and is displayed at all times while the game is running.
- Multiple upgrades and upgrade paths allowing users to fully customise each tower. This adds depth and repeatability to the game since there are multiple upgrade paths. It also allows players to customise the towers to their likes and needs for that level.
- Multiple enemy types. This increases difficulty to the game since the speed and health is different for each enemy type meaning the player has to adjust their strategy accordingly.
- Multiple maps increase the repeatability of the game since each map has a different route for the enemies meaning there are different spots where it is best to place towers. It also makes the game more engaging since the player is not greeted with the same background the whole game.
- Menu button in top right. This is displayed at all times while the game is running and allows the player to quickly edit settings such as volume or return to the home screen.
- Can sell and upgrade towers. This adds layers to the game as the player may have to decide between buying a new tower or upgrading a current one.
- Skill tree. This increases the repeatability of the game since the player can customise the way every tower plays (e.g. increase damage for all towers, increase fire rate for all towers) and may want to replay with a different skill tree setup.
- Obstacles take space on the map. This makes the game harder as some spots on the map are weaker meaning the player has to consider placing better towers around these obstacles to counteract the fewer towers possible.

A3 - Essential Features:

1. A score counter. This is displayed at all times whilst the game is running. The user can increase their score by defeating enemies. They can also add a bonus to this by completing it with no damage to their tower. This gives the player a general idea of how well they're doing.
2. Base health shown at all times. It is a heart shape which is coloured from a gradient of green to red as the base health is damaged by enemies, as well as an exact

percent of health that is left. This gives the user an idea of how much more damage they can afford to take before they will lose the game.

3. An in-game currency system. This will be split into two groups: cash and battle medals. The player will earn cash for defeating enemies or from selling existing towers. This can be used to purchase more towers or upgrade existing ones. The user's current cash is displayed at all times whilst the game is running.The player will earn battle medals for completing levels. These can be used to purchase upgrades in the skill tree.
4. Multiple towers, in this case from multiple military eras. Towers could include soldiers, tanks, armoured vehicles, planes, helicopters, artillery, SAM (surface-to-air missile) batteries, cyberwarfare soldiers. Some of these towers cannot destroy certain types of enemy (e.g. a soldier tower cannot destroy an enemy plane). These towers are all different prices and have different attributes (e.g. rate of fire, damage). This helps keep the game engaging because there are multiple types of tower and challenging because some towers cannot defeat some enemies.
5. Upgrades for towers increase attributes for individual towers using in game currency. This is done by a button in the bottom left corner. If clicked, the player can then click the tower they wish to upgrade. The player is given two options each time, meaning they have to make a choice between what attribute they upgrade. Each tower can be upgraded a limited number of times. This helps repeatability since users can retry levels with different upgrade paths and it also increases difficulty since the user may have to choose between purchasing a new tower or upgrading a current one.
6. Multiple enemy types, again from different military eras. Enemy types are the same as towers. Each enemy type has a different speed and health. Some enemy types cannot be destroyed by some towers (e.g. a soldier cannot destroy a plane). This makes the game more engaging and also challenging since enemy types later into the game are harder to destroy. The enemies health is displayed as a bar underneath itself with a colour gradient from green to red. This gives the user an idea of how much more they need to destroy each enemy to destroy it.
7. Multiple maps, again from different military eras. These maps will have towers and enemies relevant to that era, adding an educational aspect to the game. Each map will have a different path the enemies follow and obstacles in different places, increasing the difficulty since the player has to learn the ideal spots to place towers on each map.
8. There is a boss at the end of each level. This can either be a bigger wave of enemies all at once, or a single enemy with better attributes. This adds additional challenges to each level and makes the game more unique.
9. Each map has a different special ability the player can use to their advantage (e.g. call in an airstrike or deploy a minefield). This adds complexity to the game since the player only has one of these per level and has to use it wisely. It also makes the game more unique.
10. Skill tree, which is found in the main menu and affects attributes of all towers. It can be upgraded with battle medals, which the user earns from completing levels. Attributes affected are similar to individual tower upgrades (e.g. rate of fire, damage) but also unique gameplay multiplayers such as increased cash for defeating enemies. This makes the game more engaging since the user can customise how their towers work to suit their gameplay style.

11. Settings, which is found in the main menu and allows the user to customise audio using a slider. There is a slider for both music and sound effects.
12. A tutorial, which can be started from the main menu. This helps the user learn the basic mechanics of the game. This is important to keep users who may not have played a tower defence game before engaged and enjoying the game.
13. Multiple game modes, which are found in the main menu. These include campaign and arcade. This makes the game more enjoyable for all users, since casual players can play arcade whereas competitive players can try and beat the campaign.

A4 - Limitations of Proposed Solution:

1. The game will not be on a mobile device. This is because this would be outside the scope of the project. It would also be because the game screen is too large for a mobile screen and the user would struggle to interact with the user interface effectively (to place towers).
2. The game will not have a leaderboard. This is because the game is not meant to be played competitively and this feature will only appeal to a small percentage of players. Furthermore, this may put some players off playing since it may be intimidating to see a leaderboard when you open the game.
3. The game will not have a 2.5d perspective. This is because p5.js is mainly a 2d-based library and although it is possible to use 3d graphics, this is significantly less efficient and rendering and frame rate are severely affected, reducing overall gameplay.
4. The game will not have multiplayer because this would require a reliable and robust network of servers to handle multiplayer. This would be difficult to implement in a solo project (and possibly expensive).
5. The game will not display how long is left in the level. This adds an element of unpredictability to the game and encourages the player to rely on strategic skills instead of unfair knowledge on how long is left in the round.
6. Currency will not be given randomly throughout the round or produced by towers themselves. This is because players should be encouraged to try and destroy enemies instead of relying on random spawning or spamming towers that spawn currency, possibly exploiting the mechanics of the game.
7. The towers will not have multiple weapon types. This is because the game is partially educational and having soldiers with plasma or flame weapons is not realistic. Having multiple weapon types also creates a steeper learning curve since new players will have to learn which weapons work the best against which enemies, which may not be appealing for the casual player.

A5 - Computational Approach and Methods:

A computer-based solution is appropriate for my tower defence game because of the complex algorithms and calculations required. For example, when a tower fires a projectile at an enemy, a pathfinding algorithm is used to find the optimal route (and realistic route) the projectile should take towards the enemy. Computers can handle multiple processes at once e.g. enemy spawning, enemy movement and collisions with projectiles (fired from towers). As well as this, computers can be used to easily manage the scalability of a tower defence

game e.g. upgrades to towers since multipliers can be applied to whatever attribute is being upgraded. Computers can handle graphics, audio and user input, making the game more engaging for the player.

The computational methods needed for my essential features are as follows:

1. Score counter:
    - Output - the user is displayed a score as an output. This allows the user to have a general idea of how they are doing on the level. This is suitable because the user would most likely want to know how they are doing on the level.
2. Base health:
    - Output - the base health is displayed as a bar underneath itself. This conveys the base health information to the user via a graphical display. This is suitable because the user will need to make strategic decisions based on the base health.
3. Multiple towers:
    - Input - the user clicks the tower they would like to place and clicks the tile they want to place it on. This is suitable because it provides a precise input and because it is easier to receive the input, and easier for the user to use.
    - Reusable components - the different types of tower will inherit from a parent class with all the common attributes (e.g. rate of fire, damage) and methods (e.g. method to place to the tower). This is to save rewriting code and is suitable because all the towers are based on a central theme (so share most attributes).
    - Abstraction - the towers are displayed by a combination of 2d shapes instead of a complex 3d model, which may be less accurate but will greatly increase rendering and efficiency of the program.
    - Heuristics - projectiles which are fired by the towers at the enemy use a pathfinding algorithm to find the enemy. This is suitable because the projectile needs to take the most realistic (and therefore most direct route) towards the enemy.
4. Upgrades for towers:
    - Input - the user clicks whether they want to upgrade or sell the tower. If they want to upgrade the tower, the two options for upgrading pop up (which are again clicked). This is suitable because the input will need to be precise (since the user doesn't want to click the wrong option) and because it is easier to receive the input, and easier for the user to use.
5. Multiple enemies:
    - Reusable components - the different types of enemy will inherit from a parent class with all the common attributes (e.g. rate of fire, damage) and methods. This is to save rewriting code and is suitable because all the enemies are based on a central theme (so share most attributes).
    - Abstraction - the enemies are displayed by a combination of 2d shapes and a sprite instead of a complex 3d model, which may be less accurate but will greatly increase rendering and efficiency of the program.
    - Output - the enemy's health is displayed as a bar underneath itself. This conveys the enemy's health information to the user via a graphical display.

This is suitable because the user will need to make strategic decisions based on the enemy's health.

6. Multiple maps:
    - Reusable components - all maps share the same core code, including the recognition of obstacles where towers cannot be placed. The only difference is the layout of the path that the enemies take as well as the placement of obstacles. The map can therefore use reusable components to save writing repeated code.
7. Skill tree:
    - Input - the user clicks the upgrade they want to use. This is suitable because the input will need to be precise (since the user doesn't want to click the wrong option) and because it is easier to receive the input, and easier for the user to use.
8. Tutorial:
    - Problem recognition - some users will have never played a tower defence game. There will be a tutorial option in the main menu to solve this. This is suitable because it doesn't force players who are familiar with tower defence games to play through a tutorial but new players still have the option.

A6 - Success Criteria:

1. The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.
2. There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players.
3. There is an arcade game mode. The user selects one of the maps and plays endlessly with increasing difficulty over time until they lose their base. This gamemode is aimed at casual players.
4. The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.
5. The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.
6. A projectile fired from the tower moves towards the enemy. When it hits the enemy, this collision is detected and the enemy takes damage. The projectile disappears after this. If the damage was enough to destroy the enemy, the enemy also disappears. This ensures that the game works as required for a tower defence game.
7. When the base health is zero, the game ends and resets. This means the user can play again if they are playing arcade or restart the level if they are playing the campaign.

8. All the appropriate information is displayed to the user at all times while the game is running (base health, score, cash & battle medal counts). This is all important information to the player and displaying this at all times ensures the player has all the information needed for the game.
9. The user can click the upgrade button in the bottom left corner. If they subsequently click a tile with a tower on, they are prompted to click one of two options which to upgrade. If they click the upgrade button again, upgrade mode turns off. This allows for an easy way for the user to upgrade towers.
10. There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.
11. The user can access the skill tree from the main menu. They can spend their battle medals and purchase universal upgrades. The user can therefore customise their towers to suit their playstyle, making the game more enjoyable.

A7 - Requirements:

Software requirements for developer:

- Visual Studio Code is the code editor I will be using. An alternative could be replit, however, it is quite slow when editing and debugging code so a local editor is more efficient as the project gets larger. Visual Studio code also has a variety of extensions making programming and debugging easier such as code snippets (saving some time when writing code) and live server (allowing changes to be seen instantly).
- A browser to host the live server when developing. A browser is also needed to access replit to upload versions that require testing by stakeholders and to upload the final version.
- Google drive to store files for the game. Hosting the files in the cloud makes it easier to develop the game no matter the location, even though I'm using an offline editor. There will also be a local version to ensure progress cannot be lost.

Hardware requirements for developer:

- A computer/laptop to edit the code, upload the files to replit, create images to use in the game, etc.
- A router to access the internet. This is essential since a computer cannot connect to the internet (and therefore the game) without an internet connection, which is managed by the router.
- A keyboard and mouse to edit the code.

Software requirements for end user:

- A browser to access the replit which hosts the game. Alternatively, the game could be packaged as a desktop app instead but this requires additional libraries and the game is intended for casual users rather than dedicated players. Therefore, since a website is quicker to access, it is more suitable.

Hardware requirements for end user:

- A computer/laptop to access the game. The user can then use a browser to access the url and play the game. An alternative could be making a mobile game, however, this would add complexity to the project and is overall outside the scope of the

project. A mobile phone may also not be suitable to take user input since the user needs to precisely click on the tile they wish to place the tower on.

- A router to access the internet. This is essential since a computer cannot connect to the internet (and therefore the game) without an internet connection, which is managed by the router.
- A mouse to click the towers they wish to place. This is necessary because the user input is taken from the mouse. An alternative could be using a keyboard, however, it is much simpler and easier to use a mouse to input since a keyboard would require the tile code to be inputted, adding additional validation, or using the arrow keys and enter to navigate to the tile, adding unnecessary complexity.

Iteration 1 - Iterative Design, Develop, Test & Remedy, Review:

Iterative Design:

Success Criteria:

In this iteration, I will be working towards the following success criteria:

- (1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.
- (4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.
- (8) All the appropriate information is displayed to the user at all times while the game is running (base health, score, cash & battle medal counts). This is all important information to the player and displaying this at all times ensures the player has all the information needed for the game.

D1 - Decomposition:

Figure 9: An idea of what the menu should look like

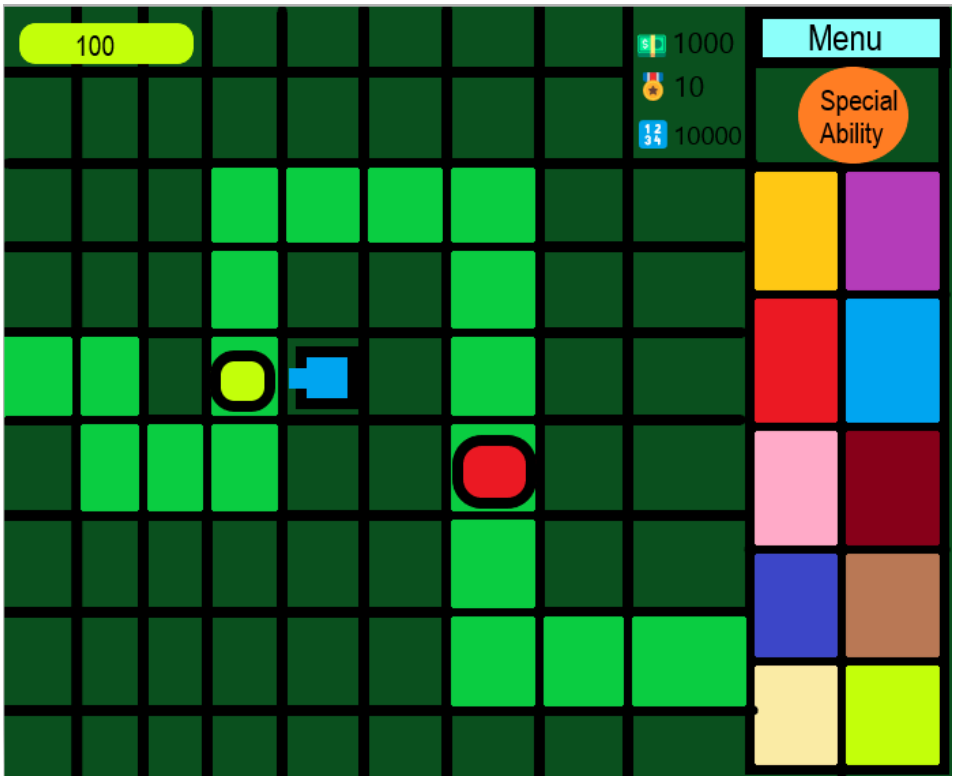


Figure 10: An idea of what the game should look like. The tower selection on the right is blank in this iteration.

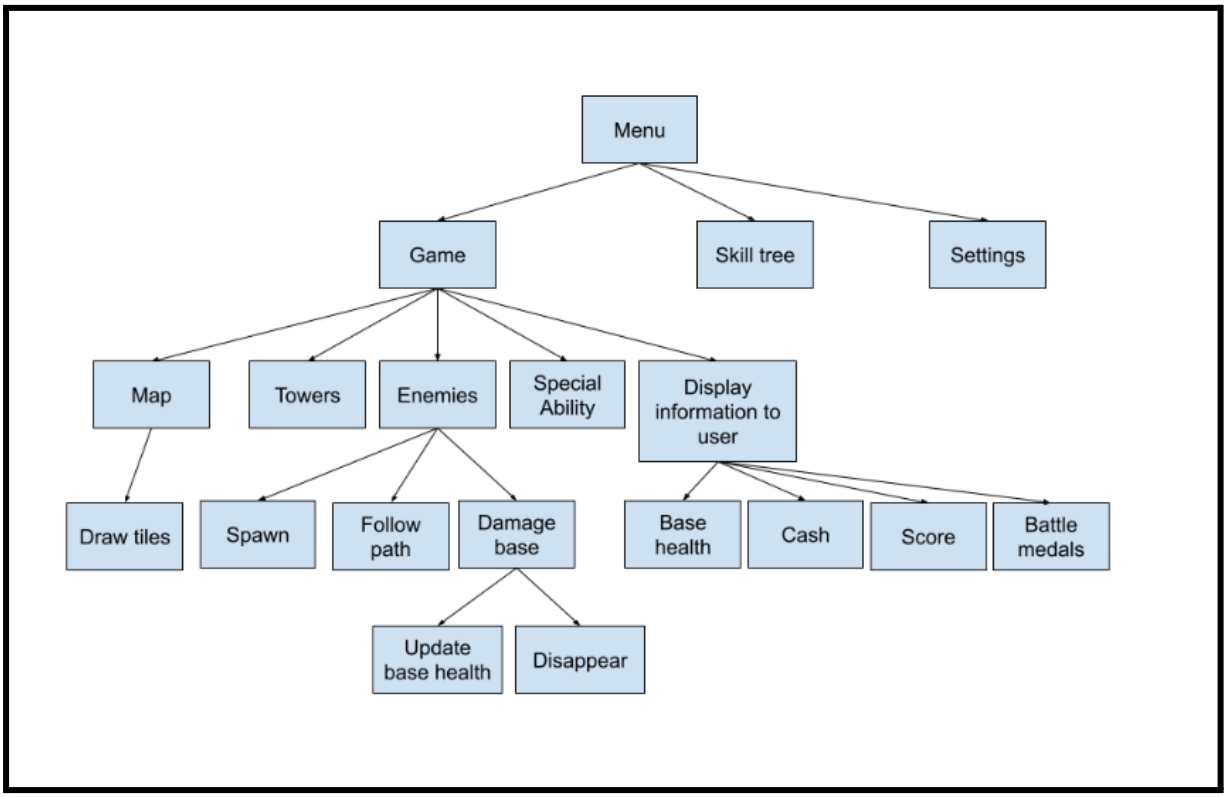


Figure 11: Structure chart for this iteration

Menu breaks into the three main sections: *Game* (includes campaign, arcade and the tutorial), *Skill tree* and *Settings*. This is the most sensible way of breaking up the solution since these are the options the user sees when they first launch the game. Game breaks into *Map, Towers, Enemies, Special ability* and *Display information to user*. This highlights the main areas seen in Figure 10 and logically breaks down the subroutines needed to create the game. Skill tree and settings are not covered in this iteration. *Map* includes drawing the tiles and colouring them in. There is only one map in this iteration, although there will be more in later iterations. *Display information to user* breaks into *Base health, Cash, Score* and *Battle medals*. As discussed previously, this is the information that the user will want to see at all times and is therefore the best way of breaking this down. *Enemies* breaks into *Spawn, Follow path* and *Damage base.* All the enemies will need to spawn and follow the path towards the base. *Damage base* breaks into *Update base health* and *Disappear*. If an enemy hits the base, the base health needs to be updated and the enemy needs to disappear.

D2 - Structure of Solution:

```
computer-science-coursework/
├─ img/
│  ├─ mainMenuImage.png
├─ lib/
│  ├─ p5.js
│  ├─ p5.sound.js
├─ src/
```

```
│   ├── classes/
│   │   ├── enemy/
│   │   │   ├── enemyBasic.js
│   │   │   ├── enemyPlane.js
│   │   │   ├── enemyVehicle.js
│   │   ├── enemyQueue.js
│   ├── setup/
│   │   ├── enemySetup.js
│   │   ├── gameSetup.js
│   │   ├── masterSetup.js
│   │   ├── menuSetup.js
│   ├── index.html
│   ├── jsconfig.json
│   ├── sketch.js
│   ├── style.css
```

enemyBasic
+ posX + posY + nextX + nextY + health + speed + colour + map[][] + mapPassed[][] + pointer + move + changeOfDirection
+ constructor(health, speed, map, colour, pointer, changeOfDirection):void + update(queue):void

enemyVehicle (subclass of enemyBasic)
+ armour
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour):void

enemyPlane (subclass of enemyVehicle)
+ flight + stealth
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour, stealth):void

enemyQueue
+ queue[] + head + tail + timeElapsed + spawnRate + type
+ constructor(spawnRate, type):void + enqueue(newItem):void + dequeue():object + checkType():void + updatePosistion():void + spawn():void

D3 - Algorithms:

randomInteger(min, max) - takes two parameters and returns a random number between them (inclusive for the lower value, exclusive for the upper value).

```
function randomInteger(min, max)
  return randomBetweenZeroAndOne() * (max-min) + min
end function
```

distance(x1, y1, x2, y2) - takes two coordinates and returns the distance between them.

```
function distance(x1, y1, x2, y2)
  return sqrt((x2-x1)^2 + (y2-y1)^2)
end function
```

linearInter(min, max, ratio) - takes two points and interpolates between them by the given ratio. This is used to help the enemies move and partially solves the "*Follow Path"* branch of the decomposition chart.

```
function linearInter(min, max, ratio)
  ratio = returnMaxOf(0, returnMinOf(ratio, 1))
  return (max - min) * ratio + min
end function
```

enemyQueue - class for the queue structure where the enemies are stored. This is a simplified version of the final queue class. The final version will mostly solve the "*Enemy*" branch of the decomposition chart.

```
class enemyQueue
  procedure new()
    queue = []
    head = 0
```

```
    tail = 0
  end procedure

  procedure enqueue(newItem)
    if tail == queue.length then
      print("Queue is full")
    else
      queue[tail] = newItem
      tail++
    end if
  end procedure

  function dequeue()
    if head == tail then
      print("Queue is empty")
      return false
    else
      item = queue[head]
      head++
      return item
    end if
  end function
end class
```

drawMapVietnam() - Displays the Vietnam-themed map on the screen. This solves the "*Draw tiles*" branch of the decomposition chart.

```
procedure drawMapVietnam()
  for i=0 to MAP_WIDTH + CELL_SIZE, STEP CELL_SIZE
    line(i, 0, i, MAP_HEIGHT)
    if i < MAP_HEIGHT + CELL_SIZE then
      line(0, i, MAP_WIDTH, i)
    end if
  next i

  for i=0 to MAP_VIETNAM.length
    for let j=0 to MAP_VIETNAM[0].length
      if MAP_VIETNAM[i][j] = 0 then
        fillColor("dark green")
        rectangle(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE)
      else if MAP_VIETNAM[i][j] = 1 then
        fillColor("brown")
        rectangle(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE)
      else if MAP_VIETNAM[i][j] = 2 then
        fillColor("black")
        rectangle(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE)
```

```
      else if MAP_VIETNAM[i][j] = 3 then
        fillColor(“gray”)
        rectangle(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE)
      end if
    next j
  next i
end procedure
```

displayInformation() - displays all the important information to the user including score, cash and battle medals. This partially solves the “*Display information to user*” branch of the decomposition chart.

```
procedure displayInformation()
  textSize(25)
  text(“Score: “ + playerScore, 1640, 100)
  text(“Cash: “ + playerCash, 1640, 140)
  text(“Medals: “ + playerBattleMedals, 1640, 180)
end procedure
```

displayBaseHealth(health, max) - displays the current base health as a colour between green and red and fills a bar with the percentage of the health remaining. This completes the “*Display information to user*” branch of the decomposition chart.

```
procedure displayBaseHealth(health, max)
  text(“Base Health:”, 1640, 220)
  percentage = health/max
  start = color(“green”)
  end = color(“red”)
  gradientColour = lerpColor(start, end, percentage)
  barWidth = mapNumbers(percentage, 0, 1, 0, 205)
  rectangle(1640, 230, 205, 30)
  if health>0 then
    fillColor(gradientColour)
    rectangle(1640, 230, barwidth, 30)
  end if
end procedure
```

D4 - Key variables and data structures including validation:

Variable Name	**Description and Justification**	**Validation**
gameState	An integer between 1-6 (inclusive) which helps the *sketch.js* file branch to the correct part of the program.	Cannot be null because the draw function in *sketch.js* checks this variable to see what part of the program the

	The value is based on the part of the game the user is in (menu, campaign, arcade, tutorial, skill tree, settings).	user wants to go to so initialised as 1 since this is the value that takes the user to the main menu screen (which they are greeted by when they first open the game). Cannot be below zero or above 6 because the branch in the *draw* function only checks between 1-6. Only changed by the buttons on the main menu or by a button returning the user to the main menu so is not dependent on user input and therefore is never outside of the range 1-6.
playerScore	This variable contains the player's current score. This resets to zero at the end of each level. This variable is displayed in the top right corner in the campaign, arcade and tutorial game mode. The number gives the user an idea of how they are doing and in the campaign gamemode the score is rated out of 3 stars.	Must be an integer. Cannot be below zero - initialised as zero and never subtracted from, only added to. Cannot be above 9,999,999 since there is not enough room on the screen to display a number larger than this. Cannot be changed by the user.
playerCash	This variable contains the player's current cash. This resets to zero at the end of each level since it is used to purchase towers for that level. This variable is displayed in the top right corner in the campaign, arcade and tutorial game mode. The starting cash depends on the level of difficulty.	Must be an integer. Cannot be below zero - initialised depending on the difficulty of the level and additional validation is used whenever this value is subtracted from (i.e. ensuring the user has enough cash to purchase a tower). Cannot be above 99,999,999 since there is not enough room on the screen to display a number larger than this. Cannot be changed by the user (directly - the user can earn cash by destroying enemies, however, this is automatically added).
playerBattleMedals	This variable contains the player's current battle medal total. Unlike the previous two variables, this does not	Must be an integer. Cannot be below zero - initialised depending on the difficulty of the level and additional

	reset to zero at the end of each level. This variable is displayed in the top right corner in the campaign, arcade, tutorial and skill tree. The user starts with no battle medals, although they receive some for completing the tutorial.	validation is used whenever this value is subtracted from (i.e. ensuring the user has enough battle medals to purchase an upgrade). Cannot be above 999,999 since there is not enough room on the screen to display a number larger than this. Cannot be changed by the user (directly - the user can earn battle medals by completing levels in campaign or getting past checkpoints in arcade, however, this is automatically added).
baseHealth	Contains the current health of the enemy base. This is displayed to the user on the right hand side of the screen via a colour gradient and the amount the bar is filled.	Declared as equal to *baseHealthMax*. Cannot be negative (whenever an enemies health is subtracted, validation is used to ensure *baseHealth>0*). Can never be over *baseHealthMax* since dividing them needs to give a percentage between 0 and 100 (although this can never happen since the user cannot edit this value and data is only ever subtracted, not added, so considering they start the same this is impossible). Must be a positive rational number.
baseHealthMax	Contains the maximum (original) health of the base. This is used in the *displayBaseHealth* method to find the percent of the base health remaining (to reduce the use of magic numbers). This value changes in campaign depending on the level difficulty although it is constant in the arcade gamemode (100.	Declared depending on the level, although generally 1000 and in this iteration it is only 1000. Unchanged from the start of the level in the campaign or once the “Arcade” button is pressed in the arcade. Cannot be zero since this value is divided with *baseHealth* to find the percentage of the base health remaining and therefore must also be greater than or equal to *baseHealth*. Must be an integer.

returnToMenuButton	This button is in the top right corner at all times (except when on the main menu) to ensure the user can quickly return to the main menu. When clicked, gameState is changed to 1 and the user is returned to the main menu.	If the gameState is 2-4 (inclusive) then the user is playing campaign, arcade or the tutorial so additional validation is needed to ensure they don't accidentally click this button and exit to the menu. If they click the button in one of these states, then an additional confirmation message appears prompting the user to click "Ok" to return to the menu, otherwise they stay where they are. In states 5 and 6 (skill tree or settings), it is less important to have this additional validation so clicking the button immediately returns the user to the menu.
campaignButton, arcadeButton, tutorialButton, skillTreeButton, settingsButton	These buttons are displayed on the main menu and change gameState to branch to whatever part of the program the user wants. They are then hidden from the screen.	Validation is not necessary since the user cannot edit the variable themselves, they can only change gameState to a discrete value. Furthermore, the user can quickly return to the menu with the button in the top right corner if needed.
mainMenuImage	Since the image is loaded within a function (*preload*), the variable (and therefore memory) for it needs to be declared outside so it is in the correct scope and can be displayed in another function (*draw*) so it is declared in *menuSetup.js*. Contains the image of a tank firing a shell which is displayed at the bottom of the main menu screen.	Initialised in *menuSetup.js* (for the reason described left) as a blank variable. A p5 image element is loaded into this variable in the preload loop and it is unchanged from here.
CELL_SIZE	This variable contains the cell size (height and width, since the cells are square). This is used to reduce magic numbers in the code and improve its readability.	Declared as a constant (96) and unchanged from here.

MAP_HEIGHT	This variable contains the map height. This is used to reduce magic numbers in my code and improve its readability. This is the same height as the canvas since the map takes up the whole screen vertically.	Declared as a constant (864) and unchanged from here.
MAP_WIDTH	This variable contains the map width. This is used to reduce magic numbers in my code and improve its readability. This is not the same width as the canvas since the map does not take up the whole screen horizontally (there is room on the right needed to display the towers and other information, as shown in **Figure 10**.	Declared as a constant (1632) and unchanged from here.
MAP_TILE_X[]	This array contains the x values for each cell (the x value of the top right corner of the tile). This is used to reduce magic numbers in my code and improve its readability.	Declared as a constant but declared blank. This is because a constant in javascript isn't a constant value but constant reference. In the next block of code after it is declared, it is filled using a simple for loop and unchanged from here.
MAP_TILE_Y[]	This array contains the y values for each cell (the y value of the top right corner of the tile). This is used to reduce magic numbers in my code and improve its readability.	Declared as a constant but declared blank. This is because a constant in javascript isn't a constant value but constant reference. In the next block of code after it is declared, it is filled using a simple for loop and unchanged from here. Additional validation is used declaring this, compared to *MAP_TILE_X*, since this array needs to be a shorter length (the height of the screen is less than the width) and since they are declared using the same loop, a check that the loop is on less than its tenth iteration (*i<10)* is used.

MAP_VIETNAM[][]	This 2D array contains integers between 0 and 3 (inclusive) and these determine the colour and picture displayed. This is achieved using a nested for loop and rectangles built into the p5.js library. The variable is called this because in later iterations there will be more maps (and this is the one based off the Vietnam War).	Declared as a constant 2D array. Unchanged from here.
enemySoldier, enemySpecialForces	These are instances of the *enemyQueue* class where each index in the queue contains an instance of the *enemyBasic* class. This is used to keep all the basic enemies in the same place and there will later be more queues for each enemy type. Each enemy type has the same speed as each other, so a queue is a suitable data structure (since the first enemy to spawn in will be the first to damage the base and despawn)	Declared as an instance of the *enemyQueue* class and controlled by its native methods from there.
enemyTruck, enemyAPC, enemyIFV, enemyTank	These are instances of the *enemyQueue* class where each index in the queue contains an instance of the *enemyVehicle* class. This is used to keep all the basic enemies in the same place and there will later be more queues for each enemy type. Each enemy type has the same speed as each other, so a queue is a suitable data structure (since the first enemy to spawn in will be the first to damage the base and despawn)	Declared as an instance of the *enemyQueue* class and controlled by its native methods from there.
enemyHelicopter, enemyJet, enemyBomber, enemyStealthBomber	These are instances of the *enemyQueue* class where each index in the queue contains an instance of the	Declared as an instance of the *enemyQueue* class and controlled by its native methods from there.

	enemyPlane class. This is used to keep all the basic enemies in the same place and there will later be more queues for each enemy type. Each enemy type has the same speed as each other, so a queue is a suitable data structure (since the first enemy to spawn in will be the first to damage the base and despawn)	

enemyBasic
+ posX + posY + nextX + nextY + health + speed + colour + map[][] + mapPassed[][] + pointer + move + changeOfDirection
+ constructor(health, speed, map, colour, pointer, changeOfDirection):void + update(queue):void

Attribute Name	**Description and Justification**	**Validation**
posX	Contains the x position of the enemy.	Not passed into the constructor method, instead preset as *MAP_TILE_X[0]* (this is the starting tile for the path). Must be an integer or float since it is changed by the *linearInter* method which interpolates between two given numbers by a given ratio.
posY	Contains the y position of	Not passed into the

	the enemy.	constructor method, instead preset as *MAP_TILE_Y[4]* (this is the starting tile for the path). Must be an integer or float since it is changed by the *linearInter* method which interpolates between two given numbers by a given ratio.
nextX	Contains the x position of the next tile that the enemy needs to move to.	Not passed into the constructor method, instead preset as *MAP_TILE_X[1]* (this is the first tile the enemy needs to move to). It is changed in the *update()* method of this class and only ever changed by adding or subtracting *CELL_SIZE*. This is because the enemy needs to fit on the tiles (rather than between them) so by only amending *nextX* by this value, this ensures this stays true. Must be an integer (or float, although it never is) since *posX* is made equal to this at the end of each tile. Since the enemies would move far too slowly (the speed of them slows over time as they get closer to the end of the tile by the nature of linear interpolation) if we waited for them to exactly reach the end of each tile, they move to the next tile when the distance to it is less than 1 so there will be slight variations in x and y position and the enemies may slightly move off the tiles if we didn't make *poxX* equal to this at the end of each tile.
nextY	Contains the y position of the next tile that the enemy needs to move to.	Not passed into the constructor method, instead preset as *MAP_TILE_Y[4]* (this is the starting tile for the path). It is changed in

		the *update()* method of this class and only ever changed by adding or subtracting *CELL_SIZE*. This is because the enemy needs to fit on the tiles (rather than between them) so by only amending next*Y* by this value, this ensures this stays true. Must be an integer (or float, although it never is) since *posY* is made equal to this at the end of each tile. Since the enemies would move far too slowly (the speed of them slows over time as they get closer to the end of the tile by the nature of linear interpolation) if we waited for them to exactly reach the end of each tile, they move to the next tile when the distance to it is less than 1 so there will be slight variations in x and y position and the enemies may slightly move off the tiles if we didn't make *poxY* equal to this at the end of each tile.
health	Contains the health of the enemy.	Must be a positive rational number (and the same data type as *baseHealth)*. This is because it is subtracted from *baseHealth* when the enemy reaches and damages the base, so this must be true to avoid an error.
speed	Contains the speed of the enemy.	Cannot be null. Must be a positive rational number between 0-1 since it is used to interpolate the enemy position (and therefore move the enemy).
colour	Contains the colour that the enemy is coloured in with. The colour of the enemy determines the enemy type.	Cannot be null. Must be a string - although the *fill()* method from the p5.js library can take multiple values (i.e. RGB or HSB), three

		variables would be needed for this alternative approach. Therefore, *colour* must be a string containing a common colour (i.e. one recognised by CSS since this is what the p5.js library uses, list here: CSS Colors)
map[][]	Contains a copy of the current map the user is playing so the values from it (and therefore the path the enemies need to move down) can be read.	Validation is already covered when the maps are declared since one of the three maps is passed into the constructor method and unchanged from here.
mapPassed[][]	Contains a 2D array of the same size as the maps. This keeps a track of where the enemy has already travelled down the path so that it doesn't go back on itself.	Declared as the same size and dimensions as map. This is needed since values from these data structures are directly compared. If the enemy needs to move to another tile, the current tile is marked as one so that it doesn't go back on itself (since when looking to move to another tile, the tile is looked up in this array to make sure the value is zero).
pointer	Contains the index position of the given enemy in the queue.	This is unused in this iteration since the *dequeue* method is sufficient (since there are no towers to change the order in which the enemies arrive at the base, when they despawn the first one to spawn will be the first one to despawn for each enemy type (each type has its own queue)). In later iterations, it will be used to work out what index needs to be removed if a tower destroys the enemy.
move	Contains an indicator of whether the enemy is changing *posX* or *posY*. This is because only one direction is changing at a time so a switch-case acts on this variable to see which	Must be a boolean which is suitable since it only ever takes two values. Declared as false since this means the enemy is moving right/left (and every map starts with the enemy

	direction it is.	moving right). It is only ever changed in the *update* method between true and false, depending on which direction the next tile lays.
changeOfDirection	Contains the distance to the next tile where the enemy moves. By the nature of linear interpolation, as the enemy moves closer to the end of each tile it slows down drastically. For faster enemies, this is not noticeable but for slower enemies this makes them too slow. So for these enemies, they move to the next tile when they are a slightly further distance away from it.	Must be either 1, 2 or 5. Declared as one of these values, depending on the enemy type and unchanged from here.

enemyVehicle (subclass of enemyBasic)
+ armour
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour):void

Attribute Name	**Description and Justification**	**Validation**
armour	Contains the armour of the enemy. This must be destroyed by the tower before the enemy's health is affected. This does not damage the base, only the enemy health.	Unused in this iteration

enemyPlane (subclass of enemyVehicle)
+ flight + stealth
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour, stealth):void

Attribute Name	**Description and Justification**	**Validation**

flight	True for all objects of type *enemyPlane*. Tells the program that this enemy can fly (and therefore cannot be shot down by some towers).	Unused in this iteration, although declared as true and unchanged from here
stealth	Contains whether or not the plane has stealth features or not.	Unused in this iteration, although passed as a parameter into the creation of each enemy object and unchanged from here.

enemyQueue
+ queue[] + head + tail + timeElapsed + spawnRate + type
+ constructor(spawnRate, type):void + enqueue(newItem):void + dequeue():object + checkType():void + updatePosistion():void + spawn():void

Attribute Name	**Description and Justification**	**Validation**
queue[]	An array containing the enemy objects. For each enemy type (i.e. move at the same speed), the first one to spawn in will be the first to reach the base and damage it (assuming it is not destroyed) therefore a queue (FIFO) data structure is suitable. Note there has to be a new queue for each enemy type since some move faster than others so a different type can spawn after one before but overtake it.	Declared as blank. Amended by *enqueue()* and *dequeue()* therefore *spawn()* (since this uses enqueue). Validation is used in these (e.g. checking the queue is not empty when using *dequeue()* as explained in **D12 - Evidence of validation**.
head	Contains the head value of the queue (i.e. the first item to be removed; the first item	Must be an integer since it is used to check the position in the queue structure. Must be

	in the queue).	above zero for the same reason. Declared as 0 and controlled by *dequeue()* from there on (i.e. incremented if an item is removed).
tail	Contains the tail value of the queue (i.e. the position of the next item to be added)	Must be an integer since it is used to check the position in the queue structure. Must be above zero for the same reason. Declared as 0 and controlled by *enqueue()* from there on (i.e. incremented if an item is added).
timeElapsed	Contains the time elapsed since the last time an enemy spawned. This is compared to the *spawnRate* to see when the enemy needs to move.	Cannot be null since it is checked every time *draw()* is run (therefore 60 times a second) and compared to the value of *spawnRate*. *deltaTIme* (a property from the p5.js library which contains the time since the beginning of the last frame) is added to this value each frame and when it is above *spawnRate*, an enemy spawns and it is reset to zero. It needs to be the same data type as *spawnRate*.
spawnRate	Contains the spawn rate of the enemies. The lower the value, the more frequently the enemies spawn.	Cannot be null. Must be the same data type as *timeElapsed* since the two values are compared.
type	Contains the type of enemy.	Cannot be null. Must be an integer between 0-9 (inclusive).

D5 - Usability Features:

- As of this iteration, the load time is instant. Even using the p5.js *preload* function, this screen is barely noticeable and navigating between menus is instant. This improves overall user experience and gives the user a good first impression of the game. This is implemented by minimal use of the *preload* function and by keeping the code modular.
- The game can easily be navigated from the menu. The menu clearly displays 5 buttons that take the user to any part of the game and this guides the user around the

main parts of the game. This is further implemented by a button in the top right of each part of the game (other than the menu) which returns the user to this menu.

- All of the buttons are in an aqua colour. This is a unique colour not used anywhere else in my game and stands out to the user, even in their peripheral vision. This ensures the buttons stand out and they don't accidentally press a button. Furthermore, the menu button is in the top right corner no matter what part of the game the user is on and this standardisation across the game ensures the user is aware of and doesn't accidentally press the menu button.
- If the user wants to return to the menu from either the campaign, arcade or tutorial gamemodes, a prompt appears for the user to confirm this decision. This is to ensure the user doesn't accidentally return to the menu in the middle of a game.
- All the fonts used are clear and commonly used around the web. The font size is large enough to be clearly read but it doesn't take up unnecessary space. This makes the game have a more professional feel.
- The user score, cash and battle medal count are displayed while they are playing the game. This is to ensure the user has all the appropriate information available to them at all times (e.g. if the user wants to buy a new tower, they need to quickly be able to check whether they have enough cash).
- The current base health is displayed as a gradient between green and red depending on the health remaining (compared to the max/original health). The bar displaying the colour also shrinks as the health shrinks. This is to give users a second way of seeing the base health remaining, or to give colorblind users a way to recognise this. Displaying the base health remaining benefits the user since they may play more or less aggressively depending on the health remaining and they will clearly want to know if they are about to lose.
- Each enemy type is coloured differently. This differentiates between them and, in later iterations, easily allows the user to see what enemies they need to be more aware of. In later iterations I might add a sprite-like graphic to each enemy but this seems unnecessary and time consuming at the moment.

D6 - Sample Data and Iteration Tests:

Iteration Number.Test Number	**Success Criteria**	**Test Details**	**Expected Outcome**
1.1	(1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.	1. Run index.html with the live server Visual Studio Code extension	The main menu appears with the title, buttons and image at the bottom

1.2	(1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear.
1.3	(1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Menu” button 4. Press “Ok” on the prompt that pops up	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear. After the “Menu” button is clicked, a confirmation prompt appears. Once “Ok” is clicked, the main menu screen appears again, including the buttons redisplaying.
1.4	(1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Menu” button 4. Press “Cancel” on the prompt that pops up	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear. After the “Menu” button is clicked, a confirmation prompt appears. Once “Cancel” is clicked, nothing changes and the arcade screen is still displayed.
1.5	(1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Skill Tree” button 3. Press the “Menu” button	The skill tree loads and the menu button appears in the top right corner. The buttons from the menu disappear. After the “Menu” button is clicked, there is no prompt and the main menu immediately loads.
1.6	(8) All the appropriate information is displayed to the user at all times while	1. Run index.html with the live server Visual Studio Code extension	The arcade game mode loads and the user is greeted with a blank copy of the map (there are no

	the game is running (base health, score, cash & battle medal counts). This is all important information to the player and displaying this at all times ensures the player has all the information needed for the game.	2. Press the “Arcade” button	enemies yet). The user’s score, cash and battle medal counts are displayed in the top right corner, underneath the menu button.
1.7	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade button”	The arcade game mode loads and a single red rectangle (enemy) should move along the path.
1.8	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade button”	The arcade game mode loads and multiple red rectangles (enemy) should spawn and move along the path.
1.9	(8) All the appropriate information is displayed to the user at all times while the game is running (base health, score, cash & battle medal counts). This is all important information to the player and displaying this at all times ensures the player has all the information needed for the game.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade button”	The arcade game mode loads and on the right hand side there is a rectangle displaying the current base health as a gradient between red and green. This should steadily go down over time, to show off the gradient.

1.10	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button	The arcade game mode loads and on the right hand side there is a rectangle displaying the current base health as a gradient between red and green. When an enemy hits the base, it deals its current health as damage to the base and disappears. The base health is updated.
1.11	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button	The arcade game mode loads and enemies of different colours and speeds start to move down the path.
1.12	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button	The arcade game mode loads and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
1.13	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button	The arcade game mode loads and there are multiple enemies with different colours, speeds and spawn rates.

Iterative Development:

D8 - Iterative Development:

Iteration 1 - Main Menu page

The first area of functionality that needs to be developed is the main menu. This is the first thing the user sees as they open the game and takes them to other areas of the game. It contains buttons which change the global variable *gameState* to achieve this. A main menu meets success criteria 1.

The main menu will have 5 buttons (which when clicked make *gameState* equal to the value in the brackets), as seen in **Figure 9**, which are Campaign (*gameState* = *2*), Arcade (*gameState* = *3*), Tutorial (*gameState* = *4*), Skill Tree (*gameState* = *5*) and Settings (*gameState* = *6*). *gameState* = *1* is the menu itself. The *gameState* variable helps the p5.js *draw* function branch to the correct part of the program. An alternative to using a variable and a switch-case is using a series of loops and functions, using the buttons to move between them. However, this could get complicated and unreadable very quickly so it is much more suitable to use a variable with a discrete number of values and branching since the current part of the program can easily be read if needed.

```
// Declares all the buttons seen on the main menu screen.
function menuButtonsSetup() {
    returnToMenuButton = createButton("Menu");
    returnToMenuButton.position(1640, 15);
    returnToMenuButton.id("returnToMenuButtonID");
    returnToMenuButton.mousePressed(returnToMenuButtonFunction);

    campaignButton = createButton("Campaign");
    campaignButton.position(740, 225);
    campaignButton.class("mainMenuButtonClass");
    campaignButton.mousePressed(campaignButtonFunction);

    arcadeButton = createButton("Arcade");
    arcadeButton.position(740, 325);
    arcadeButton.class("mainMenuButtonClass");
    arcadeButton.mousePressed(arcadeButtonFunction);

    tutorialButton = createButton("Tutorial");
    tutorialButton.position(740, 425);
    tutorialButton.class("mainMenuButtonClass");
    tutorialButton.mousePressed(tutorialButtonFunction);

    skillTreeButton = createButton("Skill Tree");
```

```
    skillTreeButton.position(740, 525);
    skillTreeButton.class("mainMenuButtonClass");
    skillTreeButton.mousePressed(skillTreeButtonFunction);

    settingsButton = createButton("Settings");
    settingsButton.position(740, 625);
    settingsButton.class("mainMenuButtonClass");
    settingsButton.mousePressed(settingsButtonFunction);
}
```

```
// Shows or hides the return to menu button depending on the input.
function returnToMenuButtonDisplay(display) {
    if(display == true) {
        returnToMenuButton.show();
    } else if(display == false) {
        returnToMenuButton.hide();
    }
}

// Shows or hides the menu buttons depending on the input.
function mainMenuButtonsDisplay(display) {
    if(display == true) {
        campaignButton.show();
        arcadeButton.show();
        tutorialButton.show();
        skillTreeButton.show();
        settingsButton.show();
    } else if(display == false) {
        campaignButton.hide();
        arcadeButton.hide();
        tutorialButton.hide();
        skillTreeButton.hide();
        settingsButton.hide();
    }
}

// When the menu button is clicked on any part of the game, the menu loads
(because gameState is changed to 1).
function returnToMenuButtonFunction() {
    if(gameState == 2 || gameState == 3 || gameState == 4) {
        if(confirm("Are you sure you want to return to menu?")) {
            gameState = 1;
```

```
        }
    } else {
        gameState = 1;
    }
}

// When the campaign button is clicked on the main menu, the campaign loads
(because gameState is changed to 2).
function campaignButtonFunction() {
    gameState = 2;
}

// When the arcade button is clicked on the main menu, the arcade loads
(because gameState is changed to 3).
function arcadeButtonFunction() {
    gameState = 3;
}
```

Figure 12: The final version the menu.

Iteration 1 - Setting up the game

The second area of functionality that needs to be developed is the background and setup for the game. This includes drawing the tiles, setting up and displaying the important information to the user (cash count, battle medal count, etc.) and creating the base as well as displaying its health as a coloured gradient. This clearly needs to all be setup before I can have enemies moving towards the base or towers defending the base. Displaying all this information to the user achieves success criteria 7.

The tiles are drawn using the arrays *MAP_TILE_X* and *MAP_TILE_Y.* Alternatively, only one array could be used since the first nine values of both are the same. However, two are used to make it clearer where the values are coming from and to distinguish between the two dimensions in the code. The method *drawMapVietnam* draws the Vietnam-themed map (in

later iterations there are more maps the user can choose from). This is achieved by a for loop and the *line* method from the p5.js library. This is suitable because the cell sizes are even so a line can be drawn from top to bottom (or left to right) in one iteration of the loop and then the loop variable can be incremented by the cell size to draw another line at this new position. After the cells are drawn, a nested for loop is used to go through the *MAP_VIETNAM* 2D array and check the value at a given position. 0 means the cell can have a tower placed there (coloured green on this map), 1 means the cell is part of the path the enemies move down (coloured brown on this map), 2 means the cell is the base (coloured black) and 3 means the cell is an obstacle (coloured grey on this map). Next, all the appropriate information is displayed to the user using the method *displayInformation()* which simply uses the p5.js *text* method to display the variables containing this information.

```
// Draws the map including the lines and filling in the tiles
function drawMapVietnam() {

    // Drawing the tiles
    stroke("black");
    for(let i=0; i<MAP_WIDTH+CELL_SIZE; i+=CELL_SIZE) {
        line(i, 0, i, MAP_HEIGHT);
        if(i<MAP_HEIGHT+CELL_SIZE) {
            line(0, i, MAP_WIDTH, i);
        }
    }

    // Filling in the tiles needed
    for(let i=0; i<MAP_VIETNAM.length; i++) {
        for(let j=0; j<MAP_VIETNAM[0].length; j++) {
          if(MAP_VIETNAM[i][j] == 0) {
            fill(3, 38, 11);
            rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

          } else if(MAP_VIETNAM[i][j] == 1){
            fill(66, 66, 32);
            rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

          } else if(MAP_VIETNAM[i][j] == 2){
            fill("black");
            rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

          } else if(MAP_VIETNAM[i][j] == 3){
            fill(69, 69, 69);
            rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

          }
```

```
        }
      }
}

// Displays the user score, cash and battle medals
function displayInformation() {
    textSize(25);
    text("🔢Score: " + playerScore, 1640, 100);
    text("💵Cash: " + playerCash, 1640, 140);
    text("🎖️Medals: " + playerBattleMedals, 1640, 180);
}
```

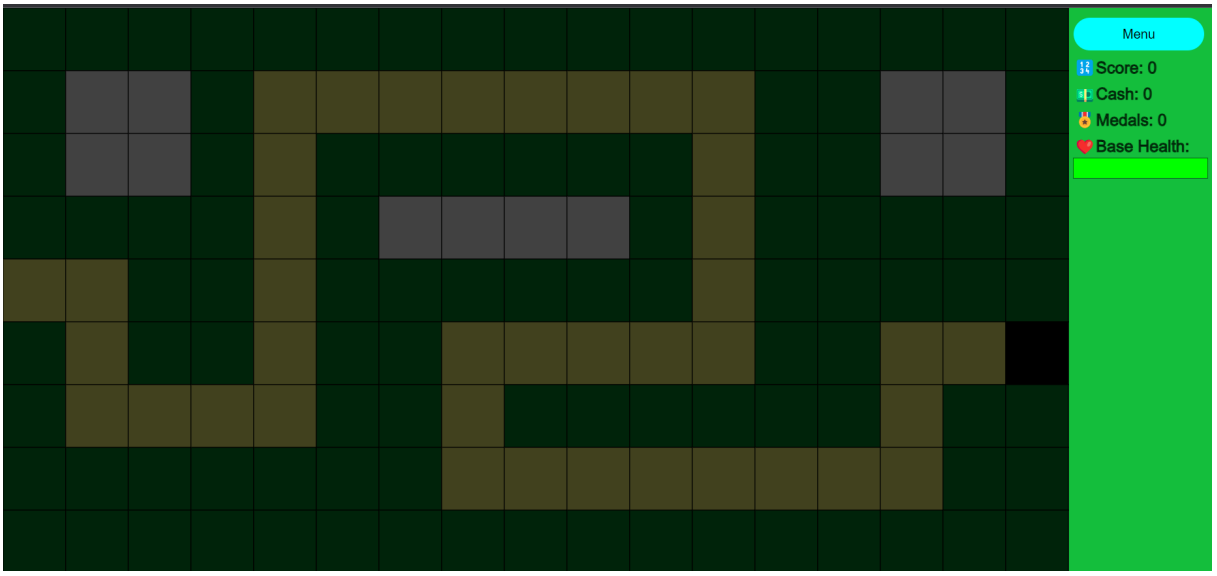


Figure 13: The final version of the game screen (ignore the health bar currently).

Iteration 1 - Creating the enemies

The third area of functionality that needs to be developed is the enemies. This is made of multiple parts, as shown in **Figure 11**. All the enemies are based off a class called *enemyBasic* and inherit from said class. At this point, the enemies need to spawn on the leftmost tile of the path (coloured brown in this iteration), move down the path towards the base. Having enemies move across the map achieves success criteria 4. The starting position is the same for all enemies (and across all maps in future iterations) so these attributes (*posX* and *posY*) are set as *MAP_TILE_X[0]* and *MAP_TILE_Y[4]* respectively. The attributes *health, speed, map* and *colour* are all passed into the constructor method since these are different for each type of enemy, or individual enemy. The *map* attribute contains a copy of the current user map because in the *update* method, each direction is checked to see which way the path the enemies move down is heading. *colour* contains the colour of the enemy, which is different for each type (and may be replaced with pixel art in later iterations). The update method controls the movement and display of the enemies by calculating the current position of the enemy relative to the map and checking the corresponding tiles to see which direction the enemy needs to move. A small rectangle is then displayed at *posX, posY* and filled with *colour*.

Iteration 1 - Creating multiple enemies

The fourth area of functionality that needs to be developed is spawning multiple enemies consecutively. Having multiple enemies achieves success criteria 4. This is relatively simple since arrays can store objects in JavaScript. Since the first enemy to spawn will be the first one to reach the base (and therefore despawn), a queue can be used to store the enemies. Although in later iterations enemies may not reach the base in the order they spawn in (i.e. a tower destroys the first enemy in the queue), a queue should still be used instead of a plain array since the queue class stores the head and tail pointers. These are used to see what part of the queue array needs to be checked to update the enemy positions since the whole array does not need to be checked if some indexes contain enemies that have been destroyed (saving memory and time). To solve the issue of enemies being destroyed in the wrong order for a queue, the native array method *pop* can be used since the specific enemy *pointer* in the array is stored in the enemy object (I assume; I might be proven wrong in later iterations) instead of *dequeue* since this will change the value of head which may not needed.

Iteration 1 - Setting up the base and displaying its health

The fifth area of functionality that needs to be developed is creating the variables used with the base and displaying its health to the user. Completing this meets success criteria 4. The variables *baseHealth* and *baseHealthMax* are used to manage the base health. *baseHealthMax* contains the maximum (original) base health and this value can change in campaign depending on the level and its difficulty, although it is constant in arcade. *baseHealth* contains the current base health and this value decreases over time as enemies damage the base (the enemy health remaining is subtracted from this value). The base health remaining is calculated and displayed by the *displayBaseHealth* method, which takes two parameters: *health* and *max*. The percentage of health remaining is calculated and this value is used with the *lerpColor* method from the p5.js library, which takes two colours and an value between 0-1 and finds the colour between them depending on the value given (e.g. in my program green and red are the two colours and a value of 0.5 would find the colour exactly between them, a value of 0.25 would find a colour closer to red, etc.). Therefore, as the base health decreases, the colour displayed becomes more red to indicate to the player that they are losing health. Furthermore, the *map* method from the p5.js library is used, which takes five parameters: the value you want to map; the start value of the current range; the end value of the current range; the start value of the new range; the end value of the new range. The value is mapped from the current range to the new range (e.g. a value of 50 in a range of 0-100 mapped to a range of 0-200 would be 100). Therefore, the colour only fills part of the display area, also depending on the health remaining.

```
// Displays the base health as a gradient.
function displayBaseHealth(health, max) {
  text("❤️Base Health:", 1640, 220);

  let percentage = health/max; // Percentage of original health remaining.
  let start = color(255, 0, 0);
  let end = color(0, 255, 0);
```

```
  let gradientColour = lerpColor(start, end, percentage); // Colours the
health bar a percentage between red and green depending on the health
remaining.

  let barWidth = map(percentage, 0, 1, 0, 205) // Makes the size of the health
bar proportional to the health remaining.

  stroke("black");
  rect(1640, 230, 205, 30);

  if(health>0){
    fill(gradientColour);
    noStroke();
    rect(1640, 230, barWidth, 30);
  }
}
```

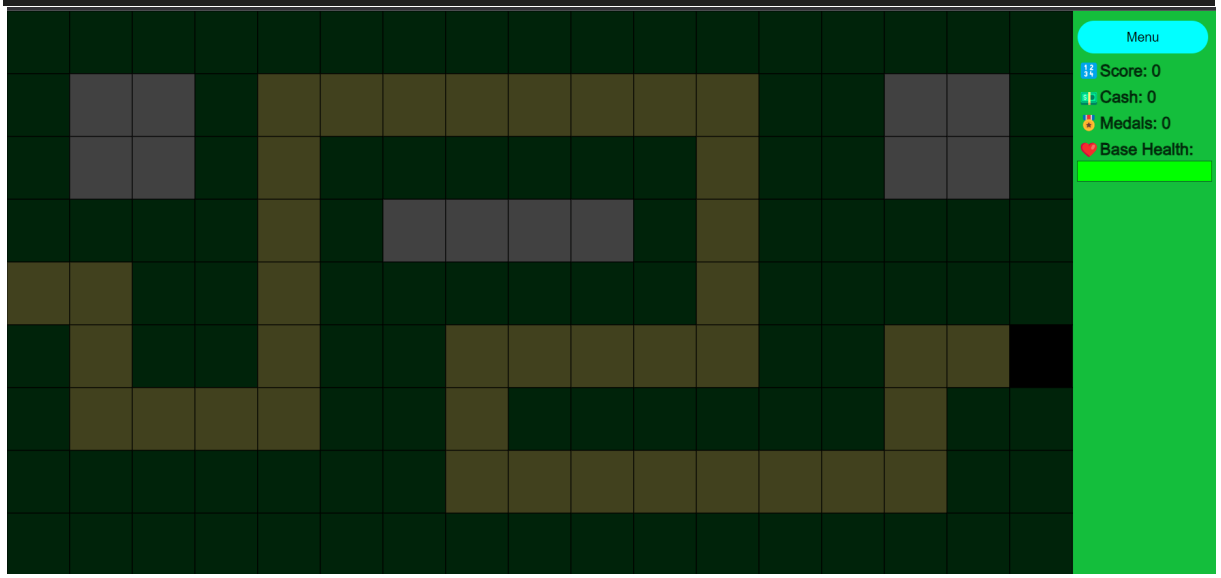


Figure 13: The final version of the game screen

Iteration 1 - Removing base health when an enemy hits and making the enemy disappear

The sixth area of functionality that needs to be developed is decreasing the base health and making the enemies disappear when they reach the base. Completing this meets success criterias 4 and 7. I added another branch in the *update* method of the *enemyBasic* class to check whether the tile to the right of the enemy has value 2 (the path the enemies follow has value 1) since this is the value of the base. This does mean that on all maps (in future iterations), the tile on the path before the base has to be to the left of it so alternatively I could check all 3 possible directions (the base is on the edge of the screen so there is no need to check whether it is to the left of the previous tile). However, only one tile is predetermined by this and all the other tiles of the path are free to be placed in any order so I don't see this as necessary. If the next tile is the base tile (and the base health is over zero) then the current enemy health is subtracted from the base health. The enemy queue

dequeue function is called which makes the enemy disappear since the *head* value is incremented and only the values in the queue from the *head* to the *tail* are checked to update and display the enemy position.

```
else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) { // If the next tile is the user base then the enemy damages it and disappears.
        if (baseHealth > 0) {
          baseHealth -= this.health;
        }
        queue.dequeue();
      }
```

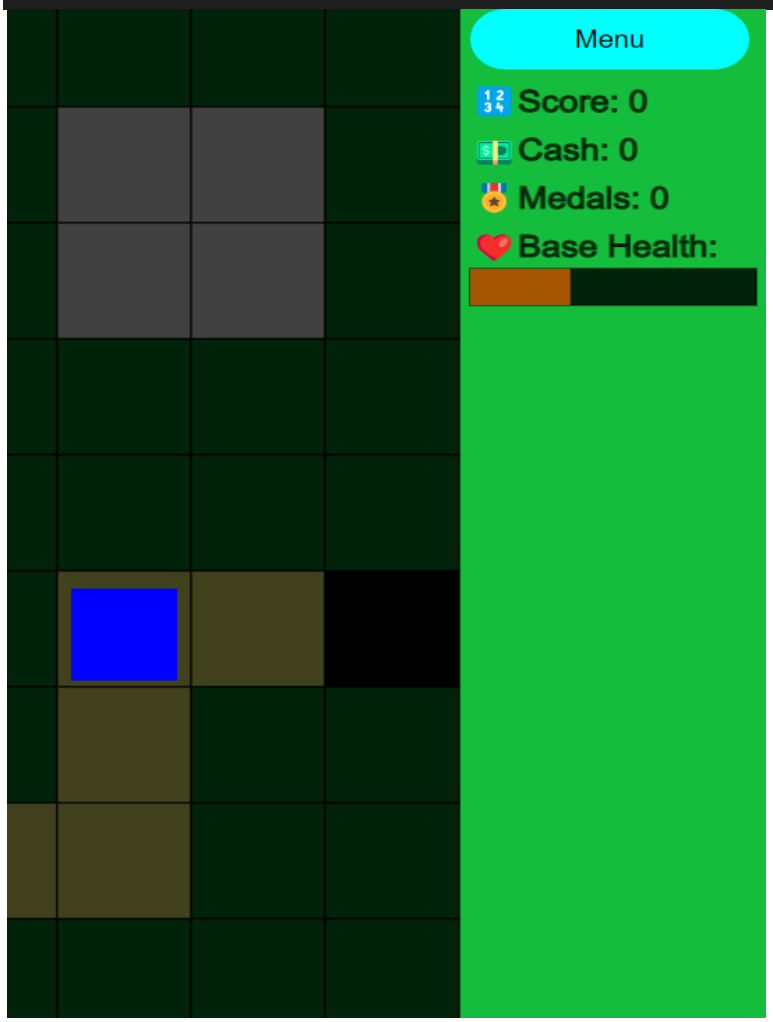


Figure 14: The base health has gone down and the enemies have disappeared that damaged it.

Iteration 1 - Multiple enemy types

The final area of functionality that needs to be developed for this iteration is multiple enemy types, each with different health, speed and spawn rate. Having multiple enemy types solves success criteria 4. Adding multiple enemy types is pretty simple because of the systems I have already used to make the enemies, I just need to make multiple queues, each

containing enemies of a different type. This is suitable because although enemies of different speeds may overtake each other (so the first one to spawn may not be the first to reach the base), enemies of the same speed do follow a queue-like structure. Each queue manages spawning and updating the position of the enemies. The spawn rate is controlled by *timeElapsed* and *spawnRate*. *spawnRate* is passed into the queue when it is declared and controls how often the enemies spawn, because an enemy spawns when *timeElapsed* (since the last enemy spawned) is greater than *spawnRate*. Therefore, each queue can spawn enemies at a different rate by having a different value for *spawnRate*. The enemy type is checked by the *checkType* method and this type of enemy is pushed to the queue. Also while developing this, I made the enemies movement smoother, since they moved in jumps before. This led to the creation of another attribute for the *enemyBasic* class (and its subclasses), *changeOfDirection.* The enemies now move by linear interpolation (by the *linearInter* method) and by the nature of this, the enemies slow down drastically towards the end of the tile, as the current position moves closer and closer to the next position (at the end of the tile). For faster enemies, this is not noticeable but for slower enemies, they move extremely slowly during this period. Therefore, the enemy moves to the next tile when the distance to it is less than *changeOfDirection* (this value is greater for slower enemies to mask the slowdown due to linear interpolation).

```
// Checks the enemy type to see what needs to be displayed
    checkType() {
        switch(this.type) {
            case 0:
                this.enqueue(new enemyBasic(100, 0.01, MAP_VIETNAM, "red",
this.tail, 5));
                break;

            case 1:
                this.enqueue(new enemyBasic(150, 0.02, MAP_VIETNAM, "yellow",
this.tail, 5));
                break;

            case 2:
                this.enqueue(new enemyVehicle(200, 0.03, MAP_VIETNAM, "gray",
this.tail, 2, 10));
                break;

            case 3:
                this.enqueue(new enemyVehicle(250, 0.04, MAP_VIETNAM, "blue",
this.tail, 2, 50));
                break;

            case 4:
                this.enqueue(new enemyVehicle(300, 0.05, MAP_VIETNAM, "green",
this.tail, 2, 100));
```

```
                break;

            case 5:
                this.enqueue(new enemyVehicle(350, 0.06, MAP_VIETNAM,
"orange", this.tail, 1, 200));
                break;

            case 6:
                this.enqueue(new enemyPlane(400, 0.07, MAP_VIETNAM, "purple",
this.tail, 1, 250, false));
                break;

            case 7:
                this.enqueue(new enemyPlane(450, 0.08, MAP_VIETNAM, "lime",
this.tail, 1, 250, false));
                break;

            case 8:
                this.enqueue(new enemyPlane(450, 0.09, MAP_VIETNAM, "white",
this.tail, 1, 250, false));
                break;

            case 9:
                this.enqueue(new enemyPlane(500, 0.1, MAP_VIETNAM, "black",
this.tail, 1, 250, true));
                break;

            default:

        }
    }
```

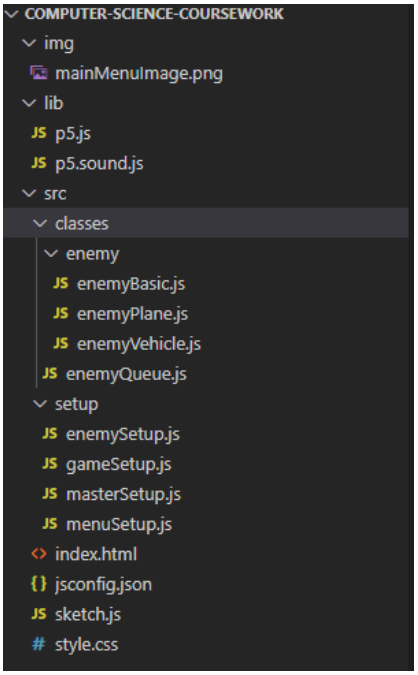
```
// Updates the enemies position each frame.
    update(queue) {
      let i = Math.floor(this.posX / CELL_SIZE);
      let j = Math.floor(this.posY / CELL_SIZE);

      // Checks if adjacent tiles are path tiles and need to be moved onto.
      if (this.map[i + 1][j] == 1 && this.mapPassed[i + 1][j] == 0 &&
distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextX = (i + 1) * CELL_SIZE;
```

```
        this.move = false;
      } else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j]
== 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextX = (i - 1) * CELL_SIZE;
        this.move = false;
      } else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0 &&
distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextY = (j + 1) * CELL_SIZE;
        this.move = true;
      } else if (j >= 1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1]
== 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextY = (j - 1) * CELL_SIZE;
        this.move = true;
      } else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) {
// If the next tile is the user base then the enemy damages it and disappears.
        if (baseHealth > 0) {
          baseHealth -= this.health;
        }
        queue.dequeue();
      }

      // Checks whether the x or y direction needs to be changed and changes
it, as well as the next tile.
      switch (this.move) {
        case false:
          if (Math.abs(this.nextX - this.posX) < this.changeOfDirection) {
            this.posX = this.nextX;
          } else {
            this.posX = linearInter(this.posX, this.nextX, this.speed);
          }
          break;
        case true:
          if (Math.abs(this.nextY - this.posY) < this.changeOfDirection) {
            this.posY = this.nextY;
          } else {
            this.posY = linearInter(this.posY, this.nextY, this.speed);
          }
          break;
      }
```

```
      // If the end of the current tile is reached, it is marked as passed.
      if (distance(this.posX, this.posY, this.nextX, this.nextY) <
this.changeOfDirection) {
        this.mapPassed[i][j] = 1;
      }

      fill(this.colour);
      rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
    }
```

D9 - Structure and modularity:

The screenshot below is of the directory containing my code so far:



The two screenshots below show the main *sketch.js* file and clearly shows the modular nature of my code, there is little to no code that isn't a subroutine:

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
14/06/23. This is the main file which combines code from other files to run
the program.

"use strict";

// Preload function from the p5.js library. Contains images which need to be
loaded before the main menu appears.
function preload() {
  mainMenuImage = loadImage("./../img/mainMenuImage.png");
}
```

```
// Setup function from the p5.js library. This function is run once at the
start of the program and never again, so it is used for creating the canvas,
etc.
function setup() {
  createCanvas(1856, 864);
  menuButtonsSetup();
}

// Draw function from the p5.js library. This function runs 60 times per
second and is used for animation (i.e. updating object positions, drawing map,
etc.).
function draw() {
  background(21, 191, 61);

  switch(gameState) {
    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      returnToMenuButtonDisplay(false);
      break;

    // Loads the campaign game mode.
    case 2:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      break;

    // Loads the arcade game mode.
    case 3:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      drawMapVietnam();
      displayInformation();
      displayBaseHealth(baseHealth, baseHealthMax);
      enemySpawn();
      enemyUpdate();
      enemySoldier.updatePosistion();
      enemySoldier.spawn();
      break;
```

```
    // Loads the tutorial.
    case 4:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    // Loads the skill tree.
    case 5:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    // Loads the settings.
    case 6:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    default:
      gameState = 1;
  }
}
```

Below are all the classes I have used so far.

enemyBasic
+ posX + posY + nextX + nextY + health + speed + colour + map[][] + mapPassed[][] + pointer + move + changeOfDirection
+ constructor(health, speed, map, colour, pointer, changeOfDirection):void + update(queue):void

enemyVehicle (subclass of enemyBasic)
+ armour
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour):void

enemyPlane (subclass of enemyVehicle)
+ flight + stealth
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour, stealth):void

enemyQueue
+ queue[] + head + tail + timeElapsed + spawnRate + type
+ constructor(spawnRate, type):void + enqueue(newItem):void + dequeue():object + checkType():void + updatePosistion():void + spawn():void

D10 - Code annotation and comments:

sketch.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
14/06/23. This is the main file which combines code from other files to run
the program.

"use strict";

// Preload function from the p5.js library. Contains images which need to be
loaded before the main menu appears.
function preload() {
  mainMenuImage = loadImage("./../img/mainMenuImage.png");
}
```

```
// Setup function from the p5.js library. This function is run once at the
start of the program and never again, so it is used for creating the canvas,
etc.
function setup() {
  createCanvas(1856, 864);
  menuButtonsSetup();
}

// Draw function from the p5.js library. This function runs 60 times per
second and is used for animation (i.e. updating object positions, drawing map,
etc.).
function draw() {
  background(21, 191, 61);

  switch(gameState) {
    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      returnToMenuButtonDisplay(false);
      break;

    // Loads the campaign game mode.
    case 2:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      break;

    // Loads the arcade game mode.
    case 3:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      drawMapVietnam();
      displayInformation();
      displayBaseHealth(baseHealth, baseHealthMax);
      enemySpawn();
      enemyUpdate();
      enemySoldier.updatePosistion();
      enemySoldier.spawn();
      break;

    // Loads the tutorial.
```

```
    case 4:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    // Loads the skill tree.
    case 5:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    // Loads the settings.
    case 6:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    default:
      gameState = 1;
  }
}
```

enemyBasic.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
19/06/23. This file contains the base class for the enemy, used for soldiers
and special forces.

"use strict";

class enemyBasic {

    // Constructor method for the class.
    constructor(health, speed, map, colour, pointer, changeOfDirection) {
        this.posX = MAP_TILE_X[0];
        this.posY = MAP_TILE_Y[4];
        this.nextX = MAP_TILE_X[1];
        this.nextY = MAP_TILE_Y[4];
        this.health = health;
```

```
        this.speed = speed; // The rate at which the enemy moves to the text
square.
        this.colour = colour;
        this.map = map; // Current map.
        this.mapPassed = [ // Tracks where the enemy has already been on the
map so it doesn't go back on itself.
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
          ];
        this.pointer = pointer; // Position in the queue.
        this.move = false;
        this.changeOfDirection = changeOfDirection
    }

    // Updates the enemies position each frame.
    update(queue) {
      let i = Math.floor(this.posX / CELL_SIZE);
      let j = Math.floor(this.posY / CELL_SIZE);

      // Checks if adjacent tiles are path tiles and need to be moved onto.
      if (this.map[i + 1][j] == 1 && this.mapPassed[i + 1][j] == 0 &&
distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextX = (i + 1) * CELL_SIZE;
        this.move = false;
      } else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j]
== 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
```

```
        this.mapPassed[i][j] = 1;
        this.nextX = (i - 1) * CELL_SIZE;
        this.move = false;
      } else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0 &&
distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextY = (j + 1) * CELL_SIZE;
        this.move = true;
      } else if (j >= 1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1]
== 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
        this.mapPassed[i][j] = 1;
        this.nextY = (j - 1) * CELL_SIZE;
        this.move = true;
      } else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) {
// If the next tile is the user base then the enemy damages it and disappears.
        if (baseHealth > 0) {
          baseHealth -= this.health;
        }
        queue.dequeue();
      }

      // Checks whether the x or y direction needs to be changed and changes
it, as well as the next tile.
      switch (this.move) {
        case false:
          if (Math.abs(this.nextX - this.posX) < this.changeOfDirection) {
            this.posX = this.nextX;
          } else {
            this.posX = linearInter(this.posX, this.nextX, this.speed);
          }
          break;
        case true:
          if (Math.abs(this.nextY - this.posY) < this.changeOfDirection) {
            this.posY = this.nextY;
          } else {
            this.posY = linearInter(this.posY, this.nextY, this.speed);
          }
          break;
      }

      // If the end of the current tile is reached, it is marked as passed.
```

```
      if (distance(this.posX, this.posY, this.nextX, this.nextY) <
this.changeOfDirection) {
        this.mapPassed[i][j] = 1;
      }

      fill(this.colour);
      rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
    }
}
```

enemyQueue.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/06/23. This file contains the queue data structure used for the enemies.

"use strict";

class enemyQueue {

    // Constructor method for the class.
    constructor(spawnRate, type) {
        this.queue = [];
        this.head = 0;
        this.tail = 0;
        this.timeElapsed = 0; // Time elapsed since the last time an enemy was
spawned.
        this.spawnRate = spawnRate; // The rate at which enemies spawn. The
lower the value, the more frequently they spawn.
        this.type = type; // The type of enemy.
    }

    // Pushes a given item to the queue.
    enqueue(newItem) {
        if(this.tail!=this.queue.length){
            console.log("Queue is full");
        } else {
            this.queue[this.tail] = newItem;
            this.tail++;
        }
    }

    // Removes the item at the start of the queue.
    dequeue() {
```

```
        if(this.head==this.tail){
            console.log("Queue is empty");
            return false;
        } else {
            let item = this.queue[this.head];
            this.head++;
            return item;
        }
    }

    // Checks the enemy type to see what needs to be displayed
    checkType() {
        switch(this.type) {
            case 0:
                this.enqueue(new enemyBasic(100, 0.01, MAP_VIETNAM, "red",
this.tail, 5));
                break;

            case 1:
                this.enqueue(new enemyBasic(150, 0.02, MAP_VIETNAM, "yellow",
this.tail, 5));
                break;

            case 2:
                this.enqueue(new enemyVehicle(200, 0.03, MAP_VIETNAM, "gray",
this.tail, 2, 10));
                break;

            case 3:
                this.enqueue(new enemyVehicle(250, 0.04, MAP_VIETNAM, "blue",
this.tail, 2, 50));
                break;

            case 4:
                this.enqueue(new enemyVehicle(300, 0.05, MAP_VIETNAM, "green",
this.tail, 2, 100));
                break;

            case 5:
                this.enqueue(new enemyVehicle(350, 0.06, MAP_VIETNAM,
"orange", this.tail, 1, 200));
                break;
```

```
            case 6:
                this.enqueue(new enemyPlane(400, 0.07, MAP_VIETNAM, "purple",
this.tail, 1, 250, false));
                break;

            case 7:
                this.enqueue(new enemyPlane(450, 0.08, MAP_VIETNAM, "lime",
this.tail, 1, 250, false));
                break;

            case 8:
                this.enqueue(new enemyPlane(450, 0.09, MAP_VIETNAM, "white",
this.tail, 1, 250, false));
                break;

            case 9:
                this.enqueue(new enemyPlane(500, 0.1, MAP_VIETNAM, "black",
this.tail, 1, 250, true));
                break;

            default:

        }
    }

    // For each item in the queue, its position is updated (using the update
method from the enemy class).
    updatePosistion() {
        for(let i=this.head; i<this.tail; i++) {
                this.queue[i].update(this);
            }
     }

    // Spawns new enemies.
    spawn() {
        this.timeElapsed += deltaTime;
        if(this.timeElapsed>this.spawnRate) {
          this.checkType();
          this.timeElapsed = 0;
        }
    }
```

```
}
```

masterSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on 14/06/23. This file sets up all the most important global variables and subroutines.

"use strict";

let gameState = 1;

let playerScore = 0;
let playerCash = 0;
let playerBattleMedals = 0;

let baseHealthMax = 1000;
let baseHealth = baseHealthMax;

// Returns a random integer between a min (inclusive) and max (exclusive) using the Javascript built-in Math library.
function randomInteger(min, max) {
    return Math.random() * (max - min) + min;
}

// Returns the distance between 2 given points
function distance(x1, y1, x2, y2) {
    return Math.sqrt((x2 - x1)** 2 + (y2 - y1)** 2);
}

// Linear interpolates between a given number using the third parameter
function linearInter(min, max, ratio) {
    ratio = Math.max(0, Math.min(ratio, 1)); //ensures t is between 0 and 1 to allow ease of multiplication
    return (max - min) * ratio + min;
}
```

Above are a few examples of code annotation and comments, every file has similar annotation to the screenshots above.

D11 - Naming:

Variables	Data Structures and Objects	Subroutines
• gameState • playerScore • playerCash • playerBattleMedals • baseHealthMax • baseHealth • CELL_SIZE • MAP_HEIGHT • MAP_WIDTH	• returnToMenuButton • campaignButton • arcadeButton • tutorialButton • skillTreeButton • settingsButton • mainMenuImage • enemyBasicQueue • MAP_TILE_X • MAP_TILE_Y • MAP_VIETNAM • enemySolider • enemySpecialForces • enemyTruck • enemyAPC • enemyIFV • enemyTank • enemyHelicopter • enemyJet • enemyBomber • enemyStealthBomber	• randomInteger(min, max) • distance(x1, y1, x2, y2) • linearInter(min, max, ratio) • titleAndImageSetup() • menuButtonsSetup() • returnToMenuButtonDisplay(display) • mainMenuButtonsDisplay(display) • returnToMenuButtonFunction() • campaignButtonFunction() • arcadeButtonFunction() • tutorialButtonFunction() • skillTreeButtonFunction() • settingsButtonFunction() • drawMapVietnam() • displayInformation() • displayBaseHealth(health, max) • enemySpawn() • enemyUpdate()

enemyBasic
+ posX + posY + nextX + nextY + health + speed + colour + map[][] + mapPassed[][] + pointer + move + changeOfDirection
+ constructor(health, speed, map, colour, pointer, changeOfDirection):void + update(queue):void

enemyVehicle (subclass of enemyBasic)
+ armour
+ constructor(health, speed, map, colour, pointer, changeOfDirection, armour):void

enemyPlane (subclass of enemyVehicle)
+ flight

+	stealth
+	constructor(health, speed, map, colour, pointer, changeOfDirection, armour, stealth):void

enemyQueue	
+	queue[]
+	head
+	tail
+	timeElapsed
+	spawnRate
+	type
+	constructor(spawnRate, type):void
+	enqueue(newItem):void
+	dequeue():object
+	checkType():void
+	updatePosistion():void
+	spawn():void

D12 - Evidence of validation:

- If the user clicks the menu button while in either the campaign, arcade or tutorial modes then an additional confirmation message appears, prompting the user to click “Ok” to confirm the decision to return to the menu. This is because progress could be lost in these gamemodes, more so than in the skill tree or settings where the user can quickly return to where they were.
- In the *update* method of the enemyBasic class, the tiles surrounding the enemy cannot be checked all at once since each direction requires a different change in x or y value. They are checked in the order right, left, up, down. Therefore, a track of where the enemy has already been is needed and this is stored in the *mapPassed* variable. Without this additional validation, say an enemy has to move three tiles downwards, as soon as it gets to the middle tile it will move back upwards, since this direction is checked before downwards.
- Also in the *update* method of the *enemyBasic* class, when checking for the tiles to the left or below the enemy, an index one less than the current one is checked. Therefore, in these two branches additional validation is used to ensure that the current index is over or equal to one (since if the current index is zero, the negative first index is checked which clearly doesn’t exist).

```
else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
```

- In the *enqueue* method of the *enemyQueue* class, it is checked that *tail* is not equal to the queue length. If it is equal, the queue is full and the queue is unchanged. This isn’t entirely necessary in javascript since arrays (i.e. the queue) are dynamic but it is generally used in queue structures so I left it.

```
// Pushes a given item to the queue.
```

```
    enqueue(newItem) {
        if(this.tail!=this.queue.length){
            console.log("Queue is full");
        } else {
            this.queue[this.tail] = newItem;
            this.tail++;
        }
    }
```

- In the *dequeue* method of the *enemyQueue* class, it checked that *tail* is not equal to *head* since this would mean that the queue is empty. If it is, the queue is unchanged. If this additional validation wasn't used, in the method *updatePosistion* (for example), the method would try to update the position of the enemy object at index -1 (since head would have been decremented by the *dequeue* method).

```
// Removes the item at the start of the queue.
    dequeue() {
        if(this.head==this.tail){
            console.log("Queue is empty");
            return false;
        } else {
            let item = this.queue[this.head];
            this.head++;
            return item;
        }
    }
```

- When *MAP_TILE_X* and *MAP_TILE_Y* are set up, they are filled using the same for loop. Since the height of the screen is less than the width, *MAP_TILE_Y* doesn't need to have as many values as *MAP_TILE_X* so validation is used to ensure that *i<10* (there are nine tiles vertically). Again, this isn't necessary since javascript arrays are dynamic so all that would occur is *MAP_TILE_Y* would be filled with a few more values which are never used. However, it does save some memory space (although clearly more could be saved by only having one array since the first nine values are the same, however I chose not to do this to make it clear where values come from and what attributes of an object are being changed by a function i.e. are the x or y values being changed).

```
// Fills the arrays declared above with the x and y posistions where
each tile starts to make referencing them easier and clearer.
for(let i=0; i<18; i++) {
    MAP_TILE_X[i] = i*CELL_SIZE;
    if(i<10){
        MAP_TILE_Y[i] = i*CELL_SIZE;
    }
}
```

- In the method *displayBaseHealth*, the base health is displayed as a colour gradient from green to red. If the health is less than 0, the outer rectangle does not need to be filled with anything since (in later iterations), the game will end. Therefore, validation is used to check that *health>0* before the colour gradient is displayed, otherwise a blank rectangle will be displayed.

```
if(health>0){
    fill(gradientColour);
    noStroke();
    rect(1640, 230, barWidth, 30);
  }
```

- In the method *drawMapVietnam()*, the lines (creating the tiles) are drawn using a single for loop. Similar to filling *MAP_TILE_X* and *MAP_TILE_Y*, the height of the screen is less than the width so additional validation is needed to check the current value of *i* within the loop is less than *MAP_HEIGHT+CELL_SIZE* (since the loop is incremented by *CELL_SIZE* and we want to draw it all the way to the edge of the screen, not one cell less than the edge of the screen). The loop then continues to draw the vertical lines (depending on the width values).

```
// Drawing the tiles
    stroke("black");
    for(let i=0; i<MAP_WIDTH+CELL_SIZE; i+=CELL_SIZE) {
        line(i, 0, i, MAP_HEIGHT);
        if(i<MAP_HEIGHT+CELL_SIZE) {
            line(0, i, MAP_WIDTH, i);
        }
    }
```

- In the method *linearInter*, the ratio is checked to make sure that it is between 0 and 1. This is to make the multiplication, and overall linear interpolation easier. It does mean there are limits to this method though: if a number greater than 1 is passed into it, 1 will be used instead (however a number outside of the range 0<ratio<1 is never used in this iteration).

```
ratio = Math.max(0, Math.min(ratio, 1)); //ensures ratio is between 0
and 1 to allow ease of multiplication
```

D13 - Prototypes:

The video below shows all the functionality for this iteration, at the end of it:
https://drive.google.com/file/d/1b0HCjmJyMbTIU8fuV9FcDg0L5eK6QoVR/view?usp=drive_link

D14 - Review:

So far, I have completed the following success criteria:

- (1) The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience.
- (4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player.
- (7) All the appropriate information is displayed to the user at all times while the game is running (base health, score, cash & battle medal counts). This is all important information to the player and displaying this at all times ensures the player has all the information needed for the game.

After reviewing the solution currently, the stakeholders had this to say:

- William Thorne - “All the buttons work and I’ve had zero issues starting and running the game. The pieces move smoothly and I like how different colours are used for different units with their individual speeds, the base health also works smoothly with how the pieces damage it. A nice addition would be to add health to the units to see the difference between pieces. Also an end screen after the base health reaches zero with an option to restart/reset the game can help make the game flow. The only issue I’ve had is to restart the game I have to close and open the tab (going back to the menu then into arcade continues the previous game).”

- Dominic Keyworth - “Overall really good introduction, the game was very clearly signposted and made it easy to figure out what to press to get the ball rolling. Very user intuitive. I liked the tank barrel design that you put on the side with it firing.
  Good parts
  -great colour chime
  - intuitive menu which was easy to use and navigate
  -excellent graphics with the artwork for the tank firing
  Could be improved:
  - more exciting logo instead of basic branding
  - clearer references to military on the home page
  - a back button when entering the games would be helpful instead of needing to reload the game.”

- Sam Russell - “1 thing I like about the game is the interactive menu. Although not fully complete, there is great potential for the relevant options available. Something I would change about the game currently is the light blue options on the menu, as these colours clash slightly. Another thing I would change is the font used on the title screen. A more interesting font would make the game more immersive and attract more players.”

- Harvey Miller - “Make the game end when health reaches zero. Add towers and the ability to place them. Make the enemies move smoothly along the track”.

From this feedback I plan on changing the following things:
- Health under the enemies added (once the enemies can take damage)

- Change the colours on the main menu
- Change the font on the main menu
- Make the enemies move smoothly along the track
- Add a more exciting logo

I was already planning on adding the following things which were mentioned:

- An end screen after the base health reaches zero with an option to restart the game (as part of this ensuring the previous game is wiped)
- Add towers and the ability to place them

Iterative Testing & Error Remedies:

D16 - Iterative Testing:

Iteration Number.Test Number	Outcome	Evidence
1.1	Pass	https://drive.google.com/file/d/1ZDM3pRrBa8rGmz1OwnbNdZRPrY4Ykg7T/view?usp=drive_link
1.2	Fail	https://drive.google.com/file/d/1YrjnGXP-XLdRNYZ9K2326-5NgQ2fX2d6/view?usp=drive_link
1.2	Pass	https://drive.google.com/file/d/1AWpOWUre1j5TpocLuWf4-0GDGRcRQCZI/view?usp=drive_link
1.3	Pass	https://drive.google.com/file/d/1Z5nNoD0Njm93T5Nm6_6LDRi9hvn0qTiU/view?usp=drive_link
1.4	Pass	https://drive.google.com/file/d/1hNqHxDeiKNF6luNNhuJTQz0moDaTvr2p/view?usp=drive_link
1.5	Pass	https://drive.google.com/file/d/1IDoAaLtPs8QNcZJCfUjMnxDi5e72uTm5/view?usp=drive_link
1.6	Pass	https://drive.google.com/file/d/1ZSSNZbx6acGue7VKVu_P8XfDav-B-osm/view?usp=drive_link
1.7	Fail	https://drive.google.com/file/d/1-DUOheXCV-UDNayEij5OUTgH2-BIomdk/view?usp=drive_link
1.7	Fail	https://drive.google.com/file/d/1-anv0xkijDBDd5IFOrK9HZ4bfMGhVLjK/view?usp=drive_link
1.7	Pass	https://drive.google.com/file/d/1pwInlf2EtB99RA9aSPKREVuF-DhHNa1U/view?usp=drive_link

1.8	Pass	https://drive.google.com/file/d/1tJzL-XQlkl2v7LshT2Cco_07jhwgEl9R/view?usp=drive_link
1.9	Pass	https://drive.google.com/file/d/1ROvIO-ZlEFdXujuq2Fzkp1P9QbeCpQ1C/view?usp=drive_link
1.10	Pass	https://drive.google.com/file/d/1U1ACX5Nnu2jCldbmpSzs9Lmmr5bLXnAB/view?usp=drive_link
1.11	Pass	https://drive.google.com/file/d/1P3onFeCKhS8vsc49m6EHNT6dl5bDfFOl/view?usp=drive_link
1.12	Fail	https://drive.google.com/file/d/1eRi1TP95Cm8z4bdst3XnA89-LPqK__UO/view?usp=drive_link
1.12	Fail	https://drive.google.com/file/d/1VhPDn-kfGTM9zbIVZX0MTHsSsqec6pBs/view?usp=drive_link
1.12	Fail	https://drive.google.com/file/d/1nGAhliWcnksTW2AYMsbVF2ZDEdiHzIan/view?usp=drive_link
1.12	Fail	https://drive.google.com/file/d/1hyYkKVNJ44rx7PN9AYChfIvK6eI_UJz5/view?usp=drive_link
1.12	Fail	https://drive.google.com/file/d/1DPqjr4Fi4nZWDgX5OaDvqbTLBufT0zCD/view?usp=drive_link
1.12	Pass	https://drive.google.com/file/d/1Pzq0cNBi2jwQg73QTHk-u99sqjpTF_rN/view?usp=drive_link
1.13	Pass	https://drive.google.com/file/d/1hpYeBti_dqknkPK0XdP7ovKOxVup_KfL/view?usp=drive_link

D17 - Non trivial failed tests and remedies:

I ran test 1.2 and it failed. Clicking the arcade button (and changing gameState to 2) did not cause the buttons from the menu to disappear. There was no error message since this was a scope issue rather than an error. Link to video of failed test: https://drive.google.com/file/d/1YrjnGXP-XLdRNYZ9K2326-5NgQ2fX2d6/view?usp=drive_link

Although I was aware that I needed to declare the variables used to store the buttons as global variables which are null to begin with (I had done this in *setup.js*), I had used the *createButton()* function from the p5.js library in the wrong loop - I had used it in the *draw()* loop whereas I needed to use it in the *setup()* loop. Once I moved creating the buttons to the other loop, the function worked as expected.

Here is the code before:

```
function draw() {
  background(21, 191, 61);

  switch(gameState) {
    case 1:
      textFont("Helvetica"); // Creating title.
      textSize(60);
      fill(0, 0, 0);
      text("Military Tower Defence Game", 550, 150);

      image(mainMenuImage, 0, 696); // Image of tank firing shell at the bottom.

      returnToMenuButton = createButton("Menu"); // Creating buttons.
      returnToMenuButton.position(1640, 15);
      returnToMenuButton.id("returnToMenuButtonID");
      returnToMenuButton.mousePressed(returnToMenuButtonFunction);
      returnToMenuButton.hide();

      campaignButton = createButton("Campaign");
      campaignButton.position(740, 225);
      campaignButton.class("mainMenuButtonClass");
      campaignButton.mousePressed(campaignButtonFunction);

      arcadeButton = createButton("Arcade");
      arcadeButton.position(740, 325);
      arcadeButton.class("mainMenuButtonClass");
      arcadeButton.mousePressed(arcadeButtonFunction);
```

(this piece of code is run once the “Arcade” button is clicked and state is changed to 3)

```
case 3:
  campaignButton.hide();
  arcadeButton.hide();
  tutorialButton.hide();
  skillTreeButton.hide();
  settingsButton.hide();
  returnToMenuButton.show();
  break;
```

Here is the code after, with the changes highlighted:

```
function setup() {
  createCanvas(1856, 896);

  menuButtonsSetup();
}

// Draw function from the p5.js libary. This function runs 60 times per
function draw() {
  background(21, 191, 61);

  switch(gameState) {
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      returnToMenuButtonDisplay(false);

      break;

    case 2:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    case 3:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;
```

```
function menuButtonsSetup() {
    returnToMenuButton = createButton("Menu"); // Creating buttons.
    returnToMenuButton.position(1640, 15);
    returnToMenuButton.id("returnToMenuButtonID");
    returnToMenuButton.mousePressed(returnToMenuButtonFunction);

    campaignButton = createButton("Campaign");
    campaignButton.position(740, 225);
    campaignButton.class("mainMenuButtonClass");
    campaignButton.mousePressed(campaignButtonFunction);

    arcadeButton = createButton("Arcade");
    arcadeButton.position(740, 325);
    arcadeButton.class("mainMenuButtonClass");
    arcadeButton.mousePressed(arcadeButtonFunction);

    tutorialButton = createButton("Tutorial");
    tutorialButton.position(740, 425);
    tutorialButton.class("mainMenuButtonClass");
    tutorialButton.mousePressed(tutorialButtonFunction);

    skillTreeButton = createButton("Skill Tree");
    skillTreeButton.position(740, 525);
    skillTreeButton.class("mainMenuButtonClass");
    skillTreeButton.mousePressed(skillTreeButtonFunction);

    settingsButton = createButton("Settings");
    settingsButton.position(740, 625);
    settingsButton.class("mainMenuButtonClass");
    settingsButton.mousePressed(settingsButtonFunction);
}
```

```
// Shows or hides the return to menu button depending on the input
function returnToMenuButtonDisplay(display) {
    if(display == true) {
        returnToMenuButton.show()
    } else if(display == false) {
        returnToMenuButton.hide();
    }
}

// Shows or hides the menu buttons depending on the input.
function mainMenuButtonsDisplay(display) {
    if(display == true) {
        campaignButton.show();
        arcadeButton.show();
        tutorialButton.show();
        skillTreeButton.show();
        settingsButton.show();
    } else if(display == false) {
        campaignButton.hide();
        arcadeButton.hide();
        tutorialButton.hide();
        skillTreeButton.hide();
        settingsButton.hide();
    }
}
```

Note that between the fail and pass of this test I made my code more modular so *setup.js* was renamed *masterSetup.js* and *menuSetup.js* was created since I realised each button needs its own function (taking up lots of room if put in a single setup file) and I wanted the *sketch.js* to be made of mostly subroutines. Also in the *menuSetup.js* file I created two subroutines to toggle the display of the buttons since this makes it clearer what the code is doing and I would be repeating code in a lot of the other cases of the switch-case branch.

After re-running test 1.2 it passed. Link to video of passed test: https://drive.google.com/file/d/1AWpOWUre1j5TpocLuWf4-0GDGRcRQCZI/view?usp=drive_link

---

I ran test 1.7 and it failed. Clicking the arcade button did not cause a red rectangle (enemy) to appear and move across the path. There was no error message. Link to video of failed test: https://drive.google.com/file/d/1-DUOheXCV-UDNayEij5OUTgH2-BIomdk/view?usp=drive_link

At first I thought the issue was that in the second branch of the nested if statement shown below, the code tries to look up the value at *this.map[i-1][j]*. However, at this point, *i* is zero since the enemy should be at its starting position (and it starts at x=0) and the -1st index of an array obviously doesn't exist. So I added another piece of validation, that *i>1* (as shown below) however the issue remained.

```
update() {
    this.timeElapsed += deltaTime;
    let i = this.posX/CELL_SIZE;
    let j = this.posY/CELL_SIZE;
    console.log(this.timeElapsed)
    console.log(this.speed)
    if(this.timeElapsed > this.speed) {
        this.timeElapsed=0;
        if(this.map[i+1][j] == 1 && this.mapPassed[i+1][j] == 0) {
            this.mapPassed[i][j] == true;
            this.posX+=CELL_SIZE;
        }else if (i>0 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0) {
            this.mapPassed[i][j] = true;
            this.posX -= CELL_SIZE
        } else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0) {
            this.mapPassed[i][j] = true;
            this.posY += CELL_SIZE
        } else if (j>0 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1] == 0) {
            this.mapPassed[i][j] = true;
            this.posY -= CELL_SIZE
        }
    }
    fill("red");
    rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
}
```

Next (as shown above), I added two *console.log* statements, displaying the timeElapsed and speed for this object as shown. I was checking two things: first that the sketch.js file even reaches the point of running the *update* method for this object and that the if statement is ever read (since timeElapsed needs to be greater than speed). As shown below, this wasn't the issue since timeElapsed is reset to zero in the middle.

```
350
317.30000000074506
350
333.59999999962747
350
350.69999999925494
350
16.900000000037253
350
33.700000000111759
350
49.800000000074506
350
67.800000000074506
```

After this, I realised that I had called the method in the wrong place in the main switch-case statement and that the map was effectively being drawn over the enemy. After fixing this, the issue was still present but it was slightly different. The enemy moves a single square and then moves back and gets stuck. A link to the video of this failed test is shown below:

https://drive.google.com/file/d/1-anv0xkijDBDd5IFOrK9HZ4bfMGhVLjK/view?usp=drive_link

I realised that the issue was that I hadn't assigned *this.mapPassed[i][j]* as 1 correctly in the first branch of the nested if statement, as highlighted below.

```
update() {
    this.timeElapsed += deltaTime;
    let i = this.posX/CELL_SIZE;
    let j = this.posY/CELL_SIZE;
    console.log(this.mapPassed)
    if(this.timeElapsed > this.speed) {
        this.timeElapsed=0;
        if(this.map[i+1][j] == 1 && this.mapPassed[i+1][j] == 0) {
            this.mapPassed[i][j] == 1;
            this.posX+=CELL_SIZE;
        }else if (i>0 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0) {
            this.mapPassed[i][j] = 1;
            this.posX -= CELL_SIZE
        } else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0) {
            this.mapPassed[i][j] = 1;
            this.posY += CELL_SIZE
        } else if (j>0 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1] == 0) {
            this.mapPassed[i][j] = 1;
            this.posY -= CELL_SIZE
        }
    }
    fill(this.colour);
```

After retrying the test with this fix, the enemy moves across the map as expected. A link to the passed test is below:
https://drive.google.com/file/d/1pwInlf2EtB99RA9aSPKREVuF-DhHNa1U/view?usp=drive_link

---

I ran test 1.12 and it failed. The enemies did not move smoothly along the path. There was no error message. A link to the video of the failed test is below:
https://drive.google.com/file/d/1eRi1TP95Cm8z4bdst3XnA89-LPqK__UO/view?usp=drive_link

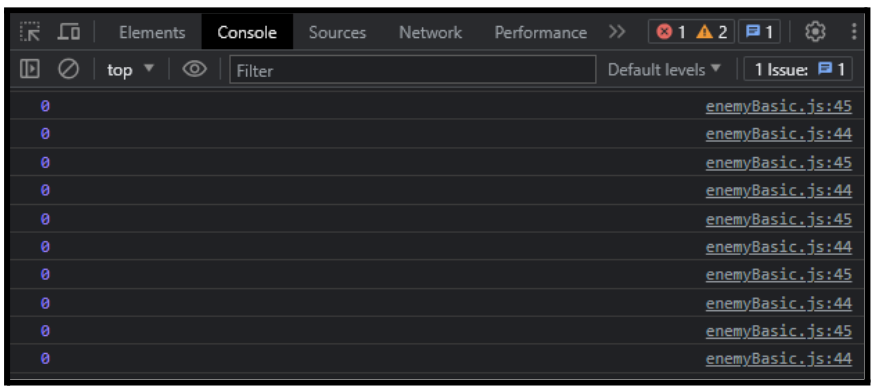
I tried changing the original code of the *update* method in the *enemyBasic* class by adding new attributes which stored the next tile (x and y) and the current position (to be displayed, x and y). I also added a new method which calculated the next tile, this is called when *posX* or *posY* is changed. This is all because the p5.js *lerp* method requires the current tile and the next one to calculate how far between them the enemy is, and this value is displayed.

```
        this.disX = lerp(this.posX, this.nextX, 1.6);
        this.disY = lerp(this.posY, this.nextY, 1.6);

        fill(this.colour);
        rect(this.disX + 10, this.disY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
    }

    nextTile() {
        let i = Math.floor(this.posX/CELL_SIZE);
        let j = Math.floor(this.posY/CELL_SIZE);
        if(this.map[i+1][j] == 1 && this.mapPassed[i+1][j] == 0) {
            this.nextX = MAP_TILE_X[i + 1];
            this.nextY = MAP_TILE_Y[j];
        } else if (i>=1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0) {
            this.nextX = MAP_TILE_X[i - 1];
            this.nextY = MAP_TILE_Y[j];
        } else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0) {
            this.nextX = MAP_TILE_X[i];
            this.nextY = MAP_TILE_Y[j + 1];
        } else if (j>=1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1] == 0) {
            this.nextX = MAP_TILE_X[i];
            this.nextY = MAP_TILE_Y[j - 1];
        }
    }
```

The above screenshot shows the bottom of the *update* method and the *nextTile* method.

Next, I removed the *nextTile* method since I realised that all the conditions are the same as in the *update* method so I could just change the value of *nextX* or *nextY* in the update method. Also I changed the way *MAP_TILE_X* and *MAP_TILE_Y* are read from since i and j are used to look up a 2d array storing the map. These changes removed the enemies from spawning all over the place and moving in random directions (as shown in the previous video). However, the enemy now moves directly upwards and gets stuck at the top. A video of this second test is shown below:
https://drive.google.com/file/d/1VhPDn-kfGTM9zbIVZX0MTHsSsqec6pBs/view?usp=drive_link

After adding *console.log(this.nextX)* and *console.log(this.nextY)*, I realised that these values are never changed from when they are declared and this is why the enemies are all moving towards the top right corner (since that is where (0,0) is located).



I also realised that *nextY* is declared as 0 when it should be declared as *MAP_TILE_Y[4]*. After changing this and changing the values of *nextX* and *nextY* as shown below (since the value only needs to be changed if *posX* is equal to *nextX* (same with y) and *nextX* would be incremented by *CELL_SIZE* every time the if statement is read from (i.e. 60 times a second)), the code still didn't run.

```
if (this.map[i + 1][j] == 1 && this.mapPassed[i + 1][j] == 0) {
  this.mapPassed[i][j] = 1;
  if(this.posX == this.nextX) {
    this.nextX += CELL_SIZE;
  }
  this.move = 0;
  console.log("Hello world")
} else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0) {
  this.mapPassed[i][j] = 1;
  if(this.posX == this.nextX) {
    this.nextX -= CELL_SIZE;
  }
  this.move = 0;
} else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0) {
  this.mapPassed[i][j] = 1;
  if(this.posY == this.nextY) {
    this.nextX += CELL_SIZE;
  }
  this.move = 1;
} else if (j >= 1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1] == 0) {
  this.mapPassed[i][j] = 1;
  if(this.posY == this.nextY) {
    this.nextY -= CELL_SIZE;
  }
  this.move = 1;
} else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) {
  if (baseHealth > 0) {
    baseHealth -= this.health;
  }
  queue.dequeue();
}
```

A link to the video of this failed test is below:
https://drive.google.com/file/d/1nGAhIiWcnksTW2AYMsbVF2ZDEdiHzIan/view?usp=drive_link
After using *console.log* a few more times, I realised *posX* is never equal to *nextX* (*posX* is 95.9999… and *nextX* is 96).

```
95.9999999967464
384
96
384
0
4
```

Next I tried calculating the distance between *posX, posY* and *nextX, nextY* using Pythagoras Theorem. As seen previously, this can never be 0 due to floating point errors with Javascript so the branches of the if statement (in the *update* method) are checked if the distance is less than 0.1. I also added a new attribute called *move* since only one of the x or y coordinates need to change at any given time and trying to change both at a time will cause diagonal movements or the enemy to become stationary. This value is changed in the loop as shown below and its value is checked in a switch-case as shown below too.

```
// Returns the distance between 2 given points
function distance(x1, x2, y1, y2) {
    return Math.sqrt((x2 - x1)^2 + (y2 - y1)^2)
}
```

```
if (this.map[i + 1][j] == 1 && this.mapPassed[i + 1][j] == 0) {
  this.mapPassed[i][j] = 1;
  if (distance(this.posX, this.posY, this.nextX, this.nextY) < 0.1) {
    this.nextX += CELL_SIZE;
    this.move = 0;
  }
} else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0) {
  this.mapPassed[i][j] = 1;
  if (distance(this.posX, this.posY, this.nextX, this.nextY) < 0.1) {
    this.nextX -= CELL_SIZE;
    this.move = 0;
  }
} else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0) {
  this.mapPassed[i][j] = 1;
  if (distance(this.posX, this.posY, this.nextX, this.nextY) < 0.1) {
    this.nextY += CELL_SIZE;
    this.move = 1;
  }
} else if (j >= 1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1] == 0) {
  this.mapPassed[i][j] = 1;
  if (distance(this.posX, this.posY, this.nextX, this.nextY) < 0.1) {
    this.nextY -= CELL_SIZE;
    this.move = 1;
  }
} else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) {
  if (baseHealth > 0) {
    baseHealth -= this.health;
  }
  queue.dequeue();
}
```

```
switch (this.move) {
  case 0:
    this.posX = lerp(this.posX, this.nextX, 0.05);
    break;
  case 1:
    this.posY = lerp(this.posY, this.nextY, 0.05);
    break;
}

fill(this.colour);
rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
```

This change improved the smoothness of the movement, however the enemy doesn't follow the path correctly; it turns one tile after it needs to at every junction. The enemies also get stuck at the very end A link to the video of the latest failed test is shown below:

https://drive.google.com/file/d/1hyYkKVNJ44rx7PN9AYChfIvK6el_UJz5/view?usp=drive_link

Next, I isolated a single enemy to try and see where the issues lied. I thought the issue could be with *mapPassed* (that the value of the tile above where the enemies are stuck is falsely changed to 1), however, a *console.log* proved this to be false. I also isolated a single direction by creating a map where the enemy should move straight right, which it did as expected. When I changed the way that *nextX* and *nextY* are declared (as *(i + 1) * CELL_SIZE*, for moving to the right), the enemy moves slightly further but still gets stuck moving upwards.

```
if (this.map[i + 1][j] === 1 && this.mapPassed[i + 1][j] === 0) {
  this.nextX = (i + 1) * CELL_SIZE;
  this.move = 0;
} else if (i >= 1 && this.map[i - 1][j] === 1 && this.mapPassed[i - 1][j] === 0) {
  this.nextX = (i - 1) * CELL_SIZE;
  this.move = 0;
} else if (this.map[i][j + 1] === 1 && this.mapPassed[i][j + 1] === 0) {
  this.nextY = (j + 1) * CELL_SIZE;
  this.move = 1;
} else if (j >= 1 && this.map[i][j - 1] === 1 && this.mapPassed[i][j - 1] === 0) {
  this.nextY = (j - 1) * CELL_SIZE;
  this.move = 1;
} else if (this.map[i + 1][j] === 2 && this.mapPassed[i + 1][j] === 0) {
  if (baseHealth > 0) {
    baseHealth -= this.health;
  }
  queue.dequeue();
}
```

```
switch (this.move) {
  case 0:
    if (Math.abs(this.nextX - this.posX) < 1) {
      this.posX = this.nextX;
    } else {
      this.posX = lerp(this.posX, this.nextX, 0.05);
    }
    break;
  case 1:
    if (Math.abs(this.nextY - this.posY) < 1) {
      this.posY = this.nextY;
    } else {
      this.posY = lerp(this.posY, this.nextY, 0.05);
    }
    break;
}

if (distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
  this.mapPassed[i][j] = 1;
}
```

A link to the video of the latest failed test is shown below:
https://drive.google.com/file/d/1DPqjr4Fi4nZWDgX5OaDvqbTLBufT0zCD/view?usp=drive_link

As shown in the video, the enemy jolts upwards slightly before moving back and settling where it was before. After using more *console.log* I discovered the issue may not have been with *mapPassed* but instead with *nextY*. This also follows my previous line of reasoning because the value of *mapPassed* is changed when the distance between the current and next position is less than 0.01, so if *nextY* is not being changed correctly then the value of *mapPassed* would be changed because the code would still see that the distance is less than the threshold and change the value of *mapPassed*.

Following this theory, I added an additional constraint to each branch of the if statement to check that the distance between the current position and next position is less than one, as shown below.

```
if (this.map[i + 1][j] == 1 && this.mapPassed[i + 1][j] == 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
  this.mapPassed[i][j] = 1;
  this.nextX = (i + 1) * CELL_SIZE;
  this.move = 0;
} else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j] == 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
  this.mapPassed[i][j] = 1;
  this.nextX = (i - 1) * CELL_SIZE;
  this.move = 0;
} else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
  this.mapPassed[i][j] = 1;
  this.nextY = (j + 1) * CELL_SIZE;
  this.move = 1;
} else if (j >= 1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1] == 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
  this.mapPassed[i][j] = 1;
  this.nextY = (j - 1) * CELL_SIZE;
  this.move = 1;
} else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) {
  if (baseHealth > 0) {
    baseHealth -= this.health;
  }
  queue.dequeue();
}
```

A copy of the passed test is below:
https://drive.google.com/file/d/1Pzq0cNBi2jwQg73QTHk-u99sqjpTF_rN/view?usp=drive_link

Although the base health doesn't update in the above video, that's just because I hadn't passed the necessary information into the enemy I was testing, when the normal system of queues is used, it works as expected.

Iteration 2 - Iterative Design, Develop, Test & Remedy, Review:

Iterative Design:

Success Criteria:

In this iteration, I will be working towards the following success criteria:

- (5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.
- (6) A projectile fired from the tower moves towards the enemy. When it hits the enemy, this collision is detected and the enemy takes damage. The projectile disappears after this. If the damage was enough to destroy the enemy, the enemy also disappears. This ensures that the game works as required for a tower defence game.
- (7) When the base health is zero, the game ends and resets. This means the user can play again if they are playing arcade or restart the level if they are playing the campaign.
- (10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.

D1 - Decomposition:

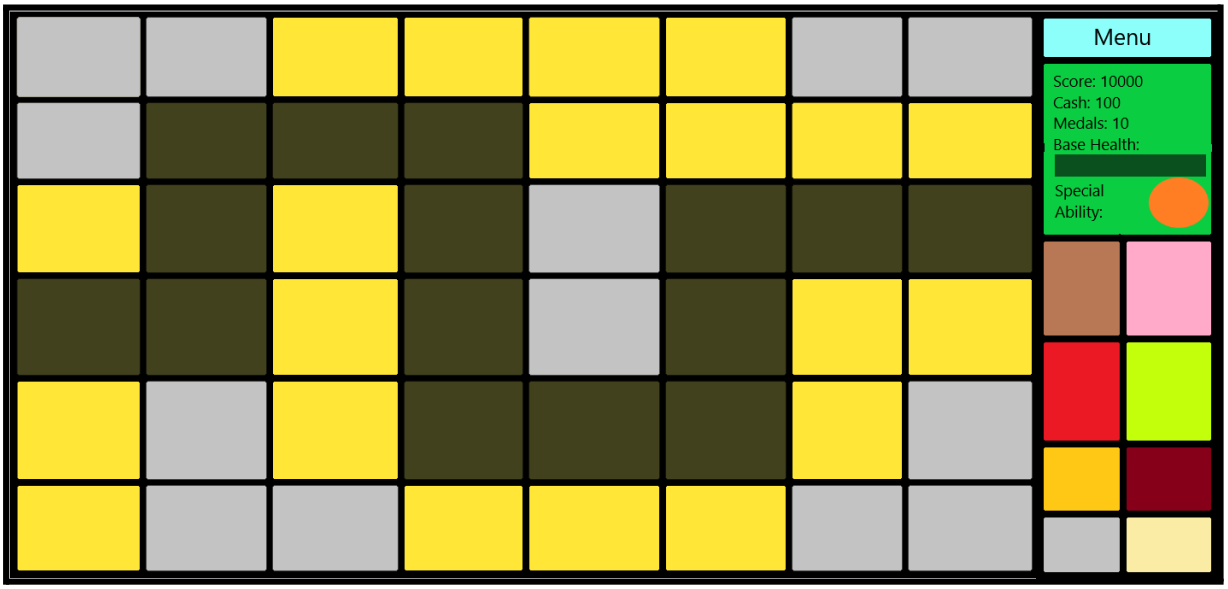


Figure 15: An idea for another map, this one based off the Gulf War

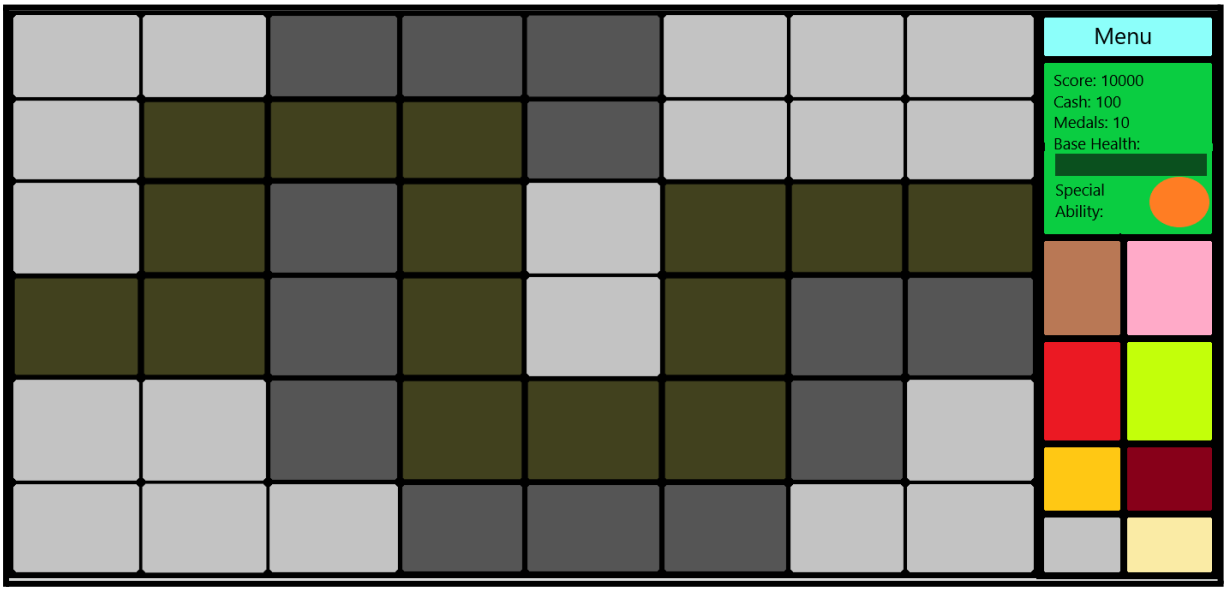


Figure 16: An idea for the third and final map, this one based off the Battle Of Stalingrad

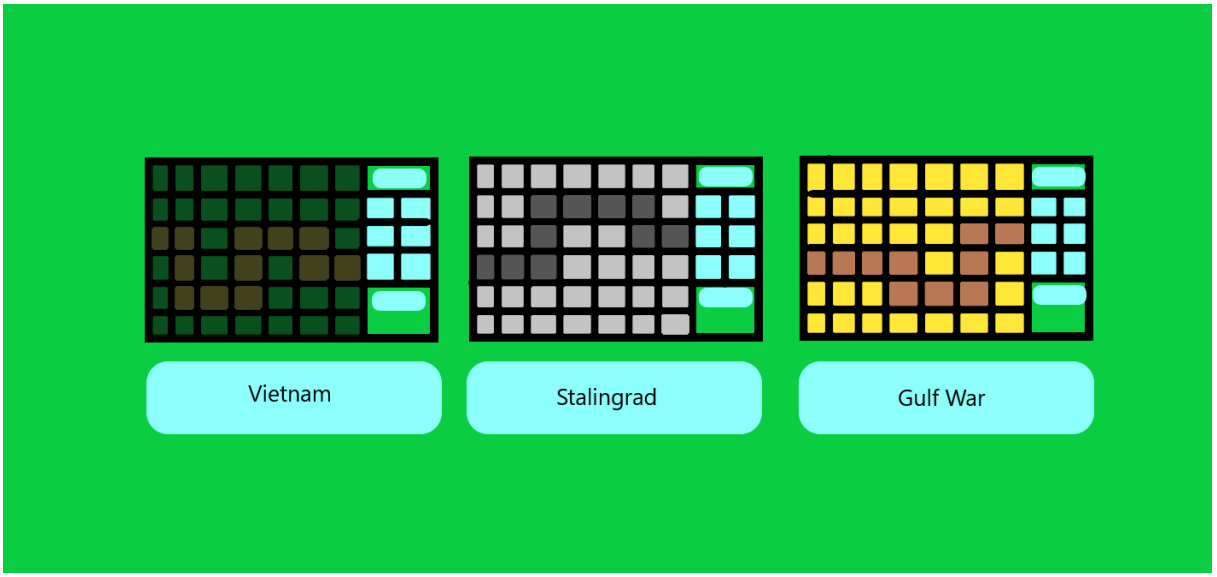


Figure 17: An idea for the map selection menu

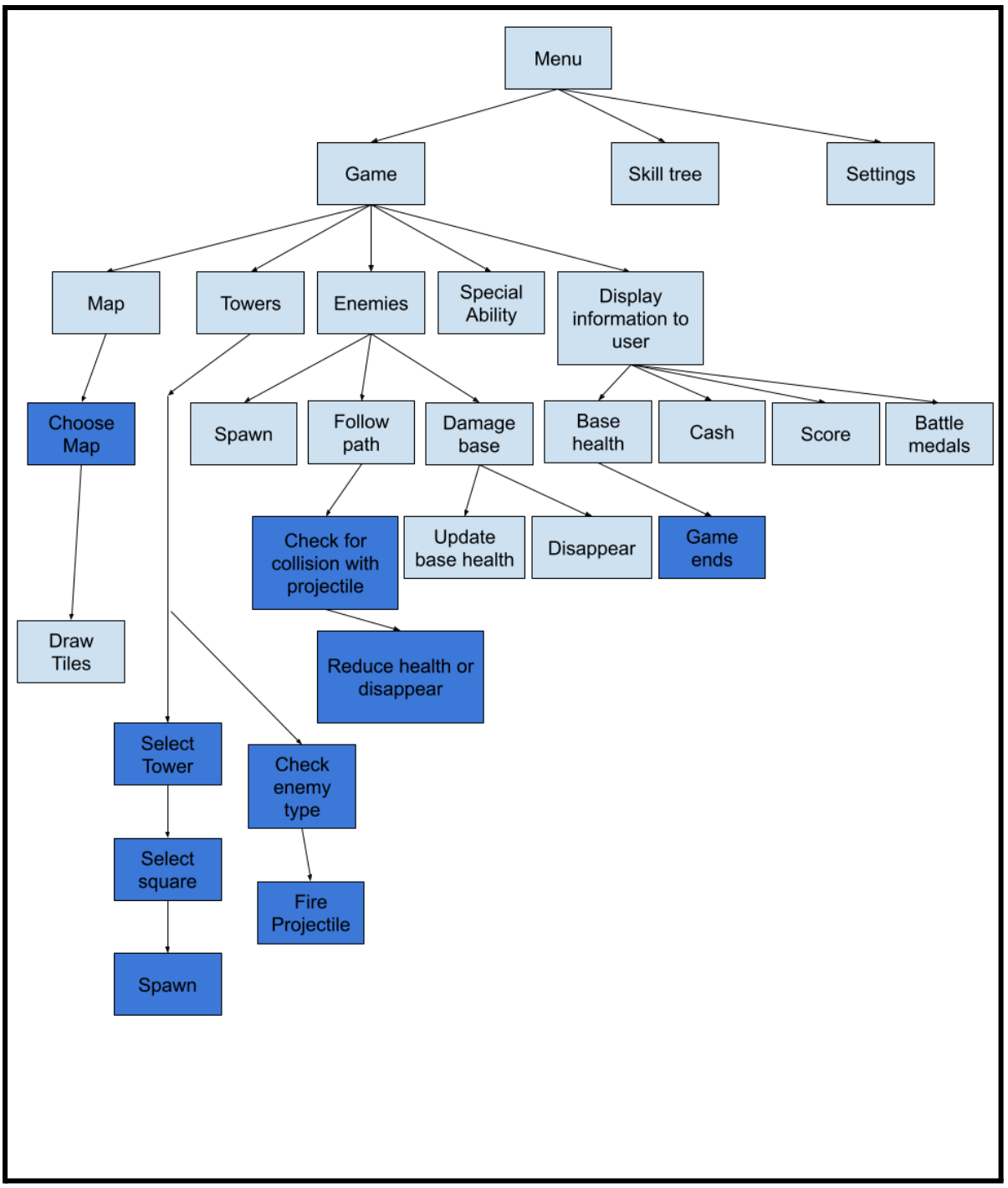


Figure 18: Structure chart for this iteration. Updates in dark blue.

Another box is placed between *Map* and *Draw Tiles* because in this iteration there are multiple maps, and the user gets to choose between them. *Towers* breaks further down into *Select Tower*, since the user needs to pick the tower they would like to place on the map. It further breaks into *Select Square* since the user needs to pick an eligible square (i.e. no obstacle, not the enemy path and no other tower already there) before the tower can be placed. This breaks into *Spawn* which is where the tower is placed (if the user has enough cash). *Towers* also splits into *Check Enemy Type* since before the tower fires a projectile it has to check it is a valid enemy type for that tower (e.g. a soldier cannot shoot down a jet). Once it has done this, it can fire the projectile and this is managed by *Fire Projectile*.

Enemies needs to be broken down slightly further since it now needs to *Check for collision with projectile* and if a projectile does hit it, its health needs to be edited and the enemy may need to disappear. This is managed by the *Reduce health or disappear* branch of the structure chart. One final branch needs to be added to the *Base Health* branch since in this iteration the game ends if the base health runs out. This is managed by the *Game ends* branch of the structure chart.

D2 - Structure of Solution:

computer-science-coursework/
├─ img/
│ ├─ **gulfwarMapImage.png**
│ ├─ mainMenuImage.png
│ ├─ **stalingradMapImage.png**
│ ├─ **vietnamMapImage.png**
├─ lib/
│ ├─ p5.js
│ ├─ p5.sound.js
├─ src/
│ ├─ classes/
│ │ ├─ enemy/
│ │ │ ├─ enemyBasic.js
│ │ │ ├─ enemyPlane.js
│ │ │ ├─ **enemyQueue.js**
│ │ │ ├─ enemyVehicle.js
│ │ ├─ **tower/**
│ │ │ ├─ **projectile.js**
│ │ │ ├─ **projectileQueue.js**
│ │ │ ├─ **towerBasic.js**
│ │ ├─ **queue.js**
│ ├─ setup/
│ │ ├─ enemySetup.js
│ │ ├─ gameSetup.js
│ │ ├─ masterSetup.js
│ │ ├─ menuSetup.js
│ │ ├─ **towerSetup.js**
│ ├─ index.html
│ ├─ jsconfig.json
│ ├─ sketch.js
│ ├─ style.css

Files in bold are new (note *enemyQueue.js* is not new, I just changed its location)

queue
+ queue[] + head + tail
+ constructor():void + enqueue(newItem):void + dequeue():object + clear():void

enemyQueue (now a subclass of queue)
(no new attributes, refer to previous iteration)
+ clear():void

enemyBasic
+ healthMax
+ display():void

enemyVehicle (subclass of enemyBasic)
+ armourMax
+ display():void

projectile
+ posX + posY + targetX + targetY + colour + speed + damage
+ constructor():void + update():void

projectileQueue (subclass of queue)
(no new attributes, refer to parent class)
+ updatePosistion():void

towerBasic
+ posX + posY + colour + type + range + damage + speed + projectiles + targetEnemy
+ constructor(posX, posY, colour, type, range, damage, speed):void + update():void + calculateDistance():object + checkEnemyType():boolean + fireProjectile():void + checkIfCollision():void

D3 - Algorithms:

cosineInterpolation(min, max, ratio) - takes two numbers and interpolates between them by the given ratio. This version uses cosine rather than a linear method to make the progressions smoother.

```
function cosineInterpolation(min, max, ratio)
  setAngles(“RADIANS”)
  ratio = (1-cos(ratio*PI))/2
  return (max - min) * ratio + min
end function
```

addScoreAndCash(enemyType) - edits the player score and cash when they kill an enemy

```
procedure addScoreAndCash(enemyType)
  playerCash += (enemyType + 1) * 1000
  playerCash =+ (enemyType + 1) * 100
```

placeTower() - lets the user place a tower if they are in the required parameters to do so (e.g. has enough cash, clicking a valid tile, etc.)

```
procedure placeTower()
  if towerPlace == true AND mouseIsPressed == true AND mouseX >= 0 AND
mouseY >= 0 AND mouseX <= MAP_WIDTH AND mouseY <= MAP_HEIGHT AND
playerCash <= towerPrice[tower] then
    tileX = floor(mouseX / CELL_SIZE)
    tileY = floor(mouseY / CELL_SIZE)
    if mapFilled[tileX][tileY] == 0 then
      if tower == 1 then
        towers.push(new towerBasic(MAP_TILE_X[tileX], MAP_TILE_Y[tileY])
```

```
      // . . . with other values
      end if
      playerCash -= towerPrice[tower]
      towerPlace = false
      mapFilled[tileX][tileY] == 1
    end if
  end if
end procedure
```

calculateDistance() - calculates whether or not an enemy is in distance of the tower to be used in the tower class.

```
function calculateDistance()
  for i=0 to enemies.length-1
    for j=enemies[i].head to enemies[i].tail-1
      enemyX = enemies[i].queue[j].posX
      enemyY = enemies[i].queue[j].posY
      if distance(this.posX, this.posY, enemyX, enemyY) < this.range then
        return type:enemies[i].type, pointer:enemies[i].queue[j].pointer
      end if
    next j
  next i
  return false
end function
```

checkIfCollision() - checks whether the projectiles have damaged the enemy to be used in the tower class.

```
procedure checkIfCollision()
  projectileX = this.projectiles.queue[this.projectiles.head].posX
  projectileY = this.projectiles.queue[this.projectiles.head].posY
  enemyX = enemies[targetEnemy.type].queue[targetEnemy.pointer].posX
  enemyY = enemies[targetEnemey.type].queue[targetEnemy.pointer].posY
  if distance(projectileX, projectileY, enemyX, enemyY) < 10 then
    this.projectiles.dequeue()
      enemies[targetEnemy.type].queue[targetEnemy.pointer].health -=
this.damage
    if enemies[targetEnemy.type].queue[targetEnemy.pointer].health < 0
then
      enemies[targetEnemy.type].dequeue()
    end if
  end if
end procedure
```

D4 - Key variables and data structures including validation:

Variable Name	Description and Justification	Validation
chosenMap[][]	Contains the map the user is currently playing. Is also used by the enemies to move.	Declared as *MAP_BLANK*. This is the default value when the user is in the menu (i.e. not playing the game) and contains a blank copy of the 2d array the same size as the maps. Its value is changed by the map buttons, brought up once you select a gamemode. These buttons assign the value of *chosenMap* to either *MAP_VIETNAM, MAP_STALINGRAD* or *MAP_GULFWAR* directly. These are all declared as constants so there is no way for their values to be changed. When the user returns to the menu, *chosenMap* is declared as *MAP_BLANK* again so they can select a new map. It is important this variable does not change to anything other than *MAP_BLANK* or one of the three maps since the program will crash when the user selects a gamemode. However, the user cannot edit this value directly and the program flow controls this variable and it is never set to anything other than these values so it works as expected.
MAP_BLANK[][]	This 2D array is the same size as the maps but every value is set as 0. Its value is compared to *chosenMap* to see if the user has chosen a map yet.	Declared as a constant 2D array. Unchanged from here.
MAP_STALINGRAD[][], MAP_GULFWAR[][]	This 2D array contains integers between 0 and 3 (inclusive) and these determine the colour and	Declared as a constant 2D array. Unchanged from here.

	picture displayed. This is achieved using a nested for loop and rectangles built into the p5.js library.	
gulfwarMapButton, stalingradMapButton, vietnamMapButton	These buttons are displayed before the player loads into either the campaign or arcade game modes. These allow the users to select their map by changing the value of *chosenMap*.	Validation is not necessary since the user cannot edit the variable themselves, they can only change *chosenMap* to a discrete value. Furthermore, the user can quickly return to the menu and change the map again with the button in the top right corner if needed.
gulfwarMapImage, stalingradMapImage, vietnamMapImage	Since the image is loaded within a function (*preload*), the variable (and therefore memory) for it needs to be declared outside so it is in the correct scope and can be displayed in another function (*draw*) so it is declared in *menuSetup.js*. Contains a screenshot of each of the maps so the user can get a view of the map before they select it	Initialised in *gameSetup.js* (for the reason described left) as a blank variable. A p5 image element is loaded into this variable in the preload loop and it is unchanged from here.
enemies[]	Replaces the old method of having one variable for each queue (and type of enemy). Each index in the array 0-9 contains an instance of the *enemyQueue* class.	Declared as instances of the *enemyQueue* class and controlled by its native methods from there.
towerPlace	Contains a boolean value saying whether or not the user wishes to place a tower. This is used when drawing the map since if the user wishes to place a tower then additional information needs to be drawn on top of the map.	Boolean value. Declared as false. If the user clicks one of the buttons to place a tower, its value is changed to true and the screen to place a tower is displayed. If the button to cancel tower placement is clicked or a tower is placed it is turned back to false. Controlled by the tower placement buttons and the cancel tower placement button. Its value cannot be changed by the user and it is managed by the control flow so no additional validation needed.

towers[]	Contains the towers the user has placed.	Declared as a blank array. Filled with an object of type *towerBasic* each time the user places a tower. The tower's positions are updated using a *for let x in y* rather than a traditional *for let x=0; x<y.length-1;x++* so validation is not really needed. This array is pushed to by the *placeTowerFunction* method so it is controlled by the program flow from here.
mapFilled[][]	2D array same size as the map. Contains a 0 if the current tile can have a tower placed on it (i.e. not a path tile, obstacle tile or already containing a tower) and a 1 for the opposite. It is used to validate the user placing a tower and to display the tiles the user can place a tower.	Declared as the same size as the maps but with each value as 0. Reset to this every time the game ends or the user returns to the menu. Filled by looping through the current *chosenMap* to see what tiles are path tiles/obstacles and amended further when the user places a tower on a specific tile.
placeCancelButton	Button used to cancel tower placement. It appears once a tower button is clicked and is not displayed at any other times. Self destructive.	Declared as a blank variable to deal with scope issues but quickly filled with a button element in the *towerButtonsSetup* method. Unchanged from here except being shown/hidden by the *placeCancelButtonDisplay* method.
soldierButton, machineGunnerButton, rpgButton, ifvButton, tankButton, shoulderAntiAirButton, surfaceAirButton, advancedSurfaceAirButton	Buttons used to place towers of a specific type. Appear on the right and when clicked bring up the tower selection screen, as well as changing *selectedTower* to their unique value.	Declared as a blank variable to deal with scope issues but quickly filled with a button element in the *towerButtonsSetup* method. Unchanged from here except being shown/hidden by the *towerButtonsDisplay* method.
selectedTower	Contains the tower the user wishes to place. Changed by the tower buttons.	Must be an integer between 0-7 (inclusive). This value is used to see which type of

		tower should be pushed to the towers array when the user places a tower, depending on the tower button they pressed. Stays between the required values because the user can only change its values using the buttons so the program control ensures it stays between these values.
towerPrices[]	Contains the price of each tower. Used to ensure the user has enough money to place a tower and subtract said money once the tower is placed.	Declared as a constant array and unchanged.
displayRange	Contains whether or not the user wants the tower ranges displayed on the screen.	Must be a boolean value. Declared as false and changed by the user clicking the button to toggle the value. Controlled by the program flow after this.
displayRangeButton	Button which the user clicks to toggle whether or not they want the tower ranges displayed.	Declared as a blank variable to deal with scope issues but quickly filled with a button element in the *towerButtonsSetup* method. Unchanged from here except being shown/hidden by the *displayRangeButtonDisplay* method.

queue
+ queue[] + head + tail
+ constructor():void + enqueue(newItem):void + dequeue():object + clear():void

Attribute Name	**Description and Justification**	**Validation**

queue[]	An array containing the enemy objects. For each enemy type (i.e. move at the same speed), the first one to spawn in will be the first to reach the base and damage it (assuming it is not destroyed) therefore a queue (FIFO) data structure is suitable. Note there has to be a new queue for each enemy type since some move faster than others so a different type can spawn after one before but overtake it.	Declared as blank. Amended by *enqueue()* and *dequeue()*.
head	Contains the head value of the queue (i.e. the first item to be removed; the first item in the queue).	Must be an integer since it is used to check the position in the queue structure. Must be above zero for the same reason. Declared as 0 and controlled by *dequeue()* from there on (i.e. incremented if an item is removed).
tail	Contains the tail value of the queue (i.e. the position of the next item to be added)	Must be an integer since it is used to check the position in the queue structure. Must be above zero for the same reason. Declared as 0 and controlled by *enqueue()* from there on (i.e. incremented if an item is added).

enemyQueue (now a subclass of queue)
(no new attributes, refer to previous iteration)
+ clear():void

enemyBasic
+ healthMax
+ display():void

Attribute Name	**Description and Justification**	**Validation**

healthMax	Contains the max amount of health the enemy has/had. Used when displaying the enemy health bar to find the percentage health remaining.	Declared in the constructor method as equal to the initial enemy health and unchanged from here.

enemyVehicle (subclass of enemyBasic)
+ armourMax
+ display():void

Attribute Name	**Description and Justification**	**Validation**
armourMax	Contains the max amount of armour the enemy has/had. Used when displaying the enemy armour bar to find the percentage armour remaining.	Declared in the constructor method as equal to the initial enemy armour and unchanged from here.

projectile
+ posX + posY + targetX + targetY + colour + speed + damage
+ constructor():void + update():void

Attribute Name	**Description and Justification**	**Validation**
posX, posY	Contains the current x and y positions of the projectile.	Must be a number. When the projectile is created by a tower, the towers current x and y are passed into the constructor method so the projectile starts where the tower is. Changed by cosine interpolation but clearly this does not result in the value being changed from a number so works as

		expected.
targetX, targetY	Contains the target x and y positions the projectile is heading towards (i.e. the position of the enemy).	Must be a number. When the projectile is created by a tower, this means an enemy is in range so the enemy position is extracted and this is passed into the constructor method of the class. Unchanged from here.
colour	Contains the colour of the projectile.	p5.js colour element. The same colour as the tower shooting it so is passed into the constructor as this value and unchanged from here.
speed	Contains the speed of the projectile. This value is then used with the cosine interpolation method to make the projectile move.	Passed into the cosine interpolation method so validation unneeded due to the repeating and symmetrical nature of cosine. The cosine interpolation method uses the linear interpolation method where a requirement that the ratio is between 0 and 1 is required, however there is validation in this method to ensure this is true.
damage	Contains the damage the projectile will deal to the enemy.	Must be a number. When a tower is made, its type determines the damage each projectile does and this is passed into the projectile so the program flow ensures this is true.

projectileQueue (subclass of queue)
(no new attributes, refer to parent class)
+ updatePosistion():void

towerBasic
+ posX + posY

+ colour + type + range + damage + speed + projectiles + targetEnemy
+ constructor(posX, posY, colour, type, range, damage, speed):void + update():void + calculateDistance():object + checkEnemyType():boolean + fireProjectile():void + checkIfCollision():void

Attribute Name	**Description and Justification**	**Validation**
posX, posY	Contains the current (and permanent) x and y position values of the tower. These are chosen when the user clicks on the tile they wish to place the tower on.	Must be a number. This value is set by the user clicking on the tile they wish to place the tower, and then their mouse coordinates are extracted. These are then fed into *MAP_TILE_X* or *MAP_TILE_Y* to work out the exact x and y positions of the tower, which is then unchanged.
colour	The colour of the tower. Determined by the tower type the user selects to differentiate between different types of tower.	A string containing a CSS readable colour (e.g. “green” or “red”). Managed by the program flow depending on the tower the user selects and cannot be directly edited by the user.
type	The type of tower.	An integer between 0-7 (inclusive). Determined by the button the user clicks to place a specific type of tower and controlled by the program flow from there.
range	The range of the tower (the area which if an enemy enters, the tower will target said enemy).	Must be a number. Depends on the type of tower. Controlled by the program flow depending on the tower the user selects and unchanged from there.

damage	The damage the tower's projectiles will do.	Must be a number. Depends on the type of tower. Controlled by the program flow depending on the tower the user selects and unchanged from there.
speed	The speed of the projectiles once fired.	Must be a number between 0-1 (inclusive). Depends on the type of tower. Controlled by the program flow depending on the tower the user selects and unchanged from there.
projectiles	The queue structure to hold the projectiles.	An instance of the *projectileQueue* class and controlled by its native methods from there.
targetEnemy	The current enemy the tower is targeting.	Undefined at first. Once an enemy is worked out to be in range of the tower, its type and pointer are extracted and stored in this object. This object is used to work out the coordinates of the enemy for the projectile to head towards.

D5 - Usability Features:

- As of this iteration, the load time is still pretty much instant (although slightly slower due to more being loaded in the *preload* function). Loading between menus and clicking buttons is still instant.
- The user is given a screenshot of each map when they select a map to allow them to see the path the enemies take, the colours and anything else they wish to see from the map before they select it. This allows the user to preview the map without having to view them all one by one and go back to the one they wish too.
- The user can quickly restart the game with the same map if they lose, or they can return to the menu to change gamemode/map by confirming or cancelling when prompted. This allows the user to quickly retry a level (in later iterations) or start another round of arcade.
- Each type of tower is coloured differently. This ensures the user can differentiate between them. Since only certain types of towers can hit certain types of enemies, these rules are generally also colour coordinated (e.g. an orange tower can hit an orange enemy and all the enemies worse than it) but there are less types of tower than enemies so these colours are slightly off.
- The user selects a tower from clearly displayed buttons on the right. This brings up a screen which highlights the tiles the user can place a tower in green and the tiles they

can't (e.g. path tile, obstacle, already has a tower on it) in red. This ensures the user knows exactly where they can place a tower.

- The price of each tower is clearly displayed on the button used to press it. This quickly lets the user know if they have enough cash to place a tower and possibly gives them a reason why they cannot place a tower.
- The user is given (by a button displayed at all times while they play the game) the option to toggle on and off whether they want to see the range of the towers. This allows them to see areas any of their towers may not cover without the need to constantly have the screen flooded with circles showing the range at all times. The circles are the same colour as the tower so the user can quickly see which range is which tower and they don't have to spend time trying to tell which tower is at the centre of which circle.
- The enemy health (and armour if it has any) are displayed as a health bar underneath the enemy. This is a quick and effective way for the player to see how much health an enemy has left and therefore tell them whether the enemy is going to do much damage to their base.

D6 - Sample Data and Iteration Tests

Iteration Number.Test Number	**Success Criteria**	**Test Details**	**Expected Outcome**
2.1	(10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.	1. Run index.html with the live server Visual Studio Code extension 2. Press the "Arcade" button	The arcade section of the game loads with the Stalingrad themed map. The enemies follow the new shape of the path.
2.2	(10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this	1. Run index.html with the live server Visual Studio Code extension 2. Press the "Arcade" button	The arcade section of the game loads with the Gulf War themed map. The enemies follow the new shape of the path.

	feature can only be used once per level.		
2.3	(10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade button” 3. Press the “Vietnam” button 4. Press the “Menu” button 5. Press the “Stalingrad” button 6. Press the “Menu” button 7. Press the “Gulf War” button	The map selection menu pops up with images of each of the maps above a button which takes you to that map. It is possible to return to the menu and select a different map.
2.4	(7) When the base health is zero, the game ends and resets. This means the user can play again if they are playing arcade or restart the level if they are playing the campaign.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Vietnam” button 4. Wait for the enemies to destroy the base 5. Press “Ok” when prompted	The game should restart once the base is destroyed once you click “Ok”. All the enemies should be cleared. The score and base health should reset.
2.5	(7) When the base health is zero, the game ends and resets. This means the user can play again if they are playing arcade or restart the level if they are playing the campaign.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Vietnam” button 4. Wait for the enemies to destroy the base 5. Press “Cancel” when prompted 6. Press the “Arcade” button 7. Press the “Stalingrad” button	The game should return to the menu once the base is destroyed and you click “Cancel”. Once you click the buttons to return to the menu and click the other map, the base health, score and enemies should all be reset.
2.6	(6) A projectile fired from the tower moves towards	1. Run index.html with the live server	A tower should be displayed automatically. Its

	the enemy. When it hits the enemy, this collision is detected and the enemy takes damage. The projectile disappears after this. If the damage was enough to destroy the enemy, the enemy also disappears. This ensures that the game works as required for a tower defence game.	Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button	range should be displayed and when an enemy enters into this zone it should start firing projectiles towards the enemy.
2.7	(6) A projectile fired from the tower moves towards the enemy. When it hits the enemy, this collision is detected and the enemy takes damage. The projectile disappears after this. If the damage was enough to destroy the enemy, the enemy also disappears. This ensures that the game works as required for a tower defence game.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button	A tower should be displayed automatically. Its range should be displayed and when an enemy enters into this zone it should start firing projectiles towards the enemy. When the projectiles hit the enemy, they disappear.
2.8	(5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press the “Soldier” button	Once the “Soldier” button is pressed, a screen should appear highlighting where the user can and can’t place a tower.
2.9	(5) The user can click a tower they want to place from the banner on the	1. Run index.html with the live server Visual Studio Code extension	The user should be able to place each type of tower. They shouldn’t be able to place a tower on any of the

	right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.	2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press a variety of tower buttons and place them 5. Press a tower button and press cancel	red tiles and where they place a tower should become a red tile. The cancel button should disappear once they place a tower, but should work if they click a tower button but don’t want to place a tower.
2.10	(5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press the “Soldier” button and place it	The tower should be able to shoot the first couple of enemies but not any more. This is because the towers can only shoot enemies of a similar tier (and to keep realism, a soldier shouldn’t be able to shoot down a plane alone).
2.11	(4) The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player. Not a target success criteria for this iteration but stakeholders requested I add this feature.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press the “Soldier” button and place it 5. Press the “RPG” button and place it	Once an enemy is damaged (but not before), a health bar should appear showing their health remaining. If the enemy has armour, their armour has to be destroyed before their health and this is displayed (as a gradient from dark to light blue instead of green to red) in the same style as the health bar and once the armour is destroyed, the normal health bar appears.

2.12	(5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press the “Soldier” button and place it 5. Press the “RPG” button and place it 6. Press the “Toggle tower ranges button” 7. Press the “Toggle tower ranges button”	The tower ranges should not be shown when the towers are first placed. Once the button is clicked, a circle appears which is the same colour as the enemy displaying the tower range. Once the button is clicked for the second time, the circles disappear again.

Iterative Development:

D8 - Iterative Development

Iteration 2 - Creating multiple maps and letting the user choose between them

The first area of functionality that needs to be set up is the second and third map. This should be relatively simple since all the code is there from previous iterations so the only thing that needs to be changed in the two new maps is the colours and the layout of the path. Creating multiple maps works towards success criteria 10. I replaced the old *drawMapVietnam* method and renamed it *drawMap*. This looks at the new variable *chosenMap* (which stores the user's chosen map) and draws specifically that. This is done by an if statement and the *JSON.stringify* method (since the value of *chosenMap* and the individual maps are compared and arrays cannot be directly compared in javascript).

```
// Draws the map including the lines and filling in the tiles for the Vietnam
themed map.
function drawMap() {

  // Drawing the tiles.
  stroke("black");
+

  for(let i=0; i<MAP_WIDTH+CELL_SIZE; i+=CELL_SIZE) {
      line(i, 0, i, MAP_HEIGHT);
      if(i<MAP_HEIGHT+CELL_SIZE) {
          line(0, i, MAP_WIDTH, i);
      }
```

```
  }

  // Filling in the tiles needed.
  if(JSON.stringify(chosenMap)==JSON.stringify(MAP_VIETNAM)) {
    for(let i=0; i<MAP_VIETNAM.length; i++) {
      for(let j=0; j<MAP_VIETNAM[0].length; j++) {
        if(MAP_VIETNAM[i][j] == 0) {
          fill(3, 38, 11);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if(MAP_VIETNAM[i][j] == 1){
          fill(66, 66, 32);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if(MAP_VIETNAM[i][j] == 2){
          fill("black");
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if(MAP_VIETNAM[i][j] == 3){
          fill(69, 69, 69);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        }
      }
    }
//. . . repeat previous if statement with other maps and colours
```

To let the user choose between them, I made a variable called *chosenMap*, which is set as equal to *MAP_BLANK* (a 2d array the same size as all the maps but all zeros). All the enemies now follow *chosenMap* when moving to make sure they follow the path of that specific map and its value is changed by buttons which the user clicks after clicking a gamemode. These are displayed alongside a screenshot of each map once the user selects a gamemode. This is done by adding a condition to the *sketch.js* file: since *chosenMap* is always blank if the user hasn't picked a map, the map selection screen is brought up if this is the case, else the game runs normally. Finally, I made sure clicking the button to return to menu resets *chosenMap* as blank so the user can select a new map.

```
// Loads the arcade game mode.
    case 3:
      if(JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK) ){
        mapButtonsDisplay(true);
        mainMenuButtonsDisplay(false);
        returnToMenuButtonDisplay(true);
      } else{
```

```
        mapButtonsDisplay(false);
        mainMenuButtonsDisplay(false);
        returnToMenuButtonDisplay(true);
        drawMap();
        displayInformation();
        displayBaseHealth(baseHealth, baseHealthMax);
        enemySpawn();
        enemyUpdate();
      }
      break;
```

Iteration 2 - Making the game end and reset when the base health is zero

The first step was adding a method to the enemy queue which resets the array, tail and head. Next, I added a method to run this on each enemy object in one master method, called *enemyClear* and similar to *enemySpawn* or *enemyUpdate* from the previous iteration. Next, I made a new method *checkBaseHealth*: that if the base health was less than or equal to zero, *enemyClear* is run as well as resetting the base health, player cash and player score. The array containing the towers the user has placed is also reset. *mapFilled* is reset (to clear any values added because of a tower placed) and the user is then prompted to either replay the same map or return to the menu. If they choose to restart, everything is set up for this already and it just happens. If they choose to return to the menu, *chosenMap* and *mapFilled* are set as blank again (so the user can choose another map if they wish since *mapFilled* has to be blank to reach that part of the if statement) and *gameState* is set to 1, returning the user to the menu.

```
if (health < 0) {
    enemyClear();
    baseHealth = baseHealthMax;
    mapFilled = [
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
```

```
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      ];

    for (let i = 0; i < mapFilled.length; i++) {
      for (let j = 0; j < mapFilled[0].length; j++) {
        if (chosenMap[i][j] == 1 || chosenMap[i][j] == 2 || chosenMap[i][j] ==
3) {
          mapFilled[i][j] = 1;
        }
      }
    }
    placeCancelButtonFunction();
    towers.length = 0;
    let tempText = "The game is over. Your score was " +
playerScore.toString() + ". Press 'Ok' to restart or press 'Cancel' to return
to the menu and/or select another map.";
    playerScore = 0;
    playerCash = 500;
    baseHealth = baseHealthMax;
    if (!(confirm(tempText))) {
      chosenMap = [
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        ];
      mapFilled = [
        [0,0,0,0,0,0,0,0,0],
```

```
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        ];
      gameState = 1;
    }
  }
```

Iteration 2 - Creating the first tower and projectiles

Next, I need to create the first tower using a class called *towerBasic*. The towers are going to be circles to differentiate them from the enemies (which are squares). I might make pixel art at some point but it seems like a waste of time now. After creating a very basic tower (displaying a circle of a specific colour at a specific x and y using a class), I wanted to test calculating when the enemy is in range of the tower. So I isolated a single enemy type (by changing the *let j = enemies[i].head* to *let j = enemies[0].head* since this is only the red enemies) and placed the tower in the middle of the map (so the enemy goes in, out and back into the towers range) to test calculating this. I changed the way all the enemies are managed, adding them all to a single array called *enemies* with each position representing a different enemy type and deleting the previous individual queues (e.g. *enemySoldier, enemyTank*). This way I can update all the enemies at once. Calculating whether the enemies are in range or not is relatively simple, I just loop through all the enemies individually and using their position relative to the tower position, a calculation can be made using this distance and the range of the tower.

```
// Checks if an enemy is in range of the tower
calculateDistance() {
        for (let i=0; i<1; i++){
            for (let j=enemies[i].head; j<enemies[i].tail; j++) {
                let enemyX = enemies[i].queue[j].posX + 48;
                let enemyY = enemies[i].queue[j].posY + 48;
```

```
                if (distance(this.posX, this.posY, enemyX, enemyY) <
this.range){
                    return {
                        type: enemies[i].type,
                        pointer: enemies[i].queue[j].pointer
                    }
                }
            }
        }
        return false;
    }
```

As for managing the projectiles, once again a queue is suitable since the first one to be fired will be the first one to hit the enemy. I changed the previous *enemyQueue* to make it a subclass of a new class called *queue*, containing common methods between the two queue classes. Overriding is also used in the *enemyQueue* class. I made another subclass of *queue* called *projectileQueue*. Each tower will manage its own projectiles through this class and the *projectile* class. Pushing the projectiles to the queue and firing them worked as expected, although they also move by linear interpolation and the limitations of this method are much clearer than the enemies moving (the first jump the projectile makes is significantly more than the enemies due to the greater speed of the projectile). Therefore, after researching alternatives, I decided to change the linear interpolation method to cosine interpolation to hopefully make the movement smoother. This is done by using cosine to change the ratio passed into the linear interpolation function (i.e. when the distance between the points is large, the ratio is small so the object doesn't move in a massive jump and when the distance between the points is small, the ratio is large so the object moves in larger jumps, which are blended in since the distance between the points is smaller).

```
// Cosine interpolation using linear interpolation to make the animation
smoother.
function cosineInterpolate(min, max, ratio) {
    angleMode(RADIANS);
    return linearInterpolate(min, max, (1-cos(ratio*PI))/2)
}
```

Next, I need to find a way to make the projectiles go towards the enemy, instead of going towards where the enemy was. My current code is below but it didn't work and I was getting bored of working on the tower specifically so I stopped there and came back to it later (and fixed the problem).
This works fine although it goes to the first registered enemy position.

```
// Fires a projectile at the target enemy.
    fireProjectile(){
        this.projectiles.enqueue(new projectile(this.posX + 48, this.posY +
48, enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posX,
```

```
enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posY,
this.colour, this.speed, this.damage));
    }
```

This doesn't work; the projectile is never in the required distance to dequeue (this is from the *enemyProjectile* class hence targetEnemy passed as a parameter rather than using *this.targetEnemy*).

```
// Checks if any projectiles in the queue (i.e. the first) have hit the enemy.
    checkIfCollision() {
        if(distance(this.queue[this.head].posX, this.queue[this.head].posY,
enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posX,

enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posY) < 50) {
                this.dequeue();
            }
        console.log(distance(this.queue[this.head].posX,
this.queue[this.head].posY,
enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posX,

enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posY));
    }
```

This still didn't work and I wanted to move on from the projectiles so I put this section to the side for now and moved on to a different success criteria.

<u>Iteration 2 - Creating the buttons the user clicks to place a tower and highlighting the tiles they can place it on</u>

I'm confident making buttons by now so I used my usual method (declaring all the variables where the buttons are stored outside the p5.js *setup* loop to keep scope and making a function which is called once in the p5.js *setup* function to make these variables a (HTML) button element and set their position, (HTML) class, etc.). After making these buttons and positioning them correctly (and making a function to toggle their display on and off), I moved on to highlighting the tiles where the user can place a tower. I made it so that clicking any one of the tower buttons turns a variable called *towerPlace* to true (as well as specifically changing the value of a variable called *selectedTower* so we can tell which tower the user wants to place). Within the function *drawMap*, there is a condition that if *towerPlace* is true, then the following code is run. The fourth parameter in the *color* function (taking RGB values) is the alpha since this screen is displayed on top of the map but we still want the map to be seen so I need it to be slightly transparent.

```
// If the user wishes to place a tower, this screen is displayed on top of the
map.
  if(towerPlace){
    let colourFill;
    for(let i=0; i<mapFilled.length; i++) {
```

```
      for(let j=0; j<mapFilled[0].length; j++) {
        if(mapFilled[i][j] == 0) {
          colourFill = color(67, 198, 11, 97);
        } else {
          colourFill = color(212, 0, 0, 98);
        }
        fill(colourFill);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);
      }
    }
  }
```

mapFilled is a 2D array the same size as the maps which marks the tiles where a user cannot place a tile with a 1. Its value is first changed when the user picks a map (and reset to blank if the user returns to the menu) as shown below.

```
// When the Stalingrad themed map is clicked in the menu, this is changed to
the user's chosen map.
function stalingradMapButtonFunction() {
  chosenMap = MAP_STALINGRAD;

  for(let i=0; i<mapFilled.length; i++) {
    for(let j=0; j<mapFilled[0].length; j++) {
      if(chosenMap[i][j] == 1 || chosenMap[i][j] == 2 || chosenMap[i][j] ==
3){
        mapFilled[i][j] = 1;
      }
    }
  }
}
```

Finally, I added a button to cancel this screen (which only appears if this screen is on). This button changes *towerPlace* to false so the screen turns off. I don't need to change/reset the value of *selectedTower* since nothing is ever done with this value unless *towerPlace* is true and its value is simply changed by the tower buttons if the user decides to select a different tower if they clicked a different tower than before.

Iteration 2 - Letting the user place a tower

Next, I need to make it so that pressing a tower button and clicking on one of the valid tiles places a tower. All the towers are stored in the *towers* array, a simple 1D array. I made a function *placeTowerFunction* (which is run in the p5.js *draw* loop) which manages the user placing the towers.

```
// Place's a tower at the x and y the user clicks.
function placeTowerFunction() {
```

```
    if(towerPlace && mouseIsPressed && mouseX >= MAP_TILE_X[0] && mouseX <=
MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <= MAP_TILE_Y[9]
        && playerCash >= TOWER_PRICES[selectedTower]) {
        let tileX = Math.floor(mouseX/CELL_SIZE);
        let tileY = Math.floor(mouseY/CELL_SIZE);
        if(mapFilled[tileX][tileY]==0) {
            switch(selectedTower) {
                case 0:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "red", 0, 200, 10, 10, 0.62));
                    break;

                case 1:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "yellow", 1, 220, 10, 10, 0.66));
                    break;
                //. . . with other values between
                case 7:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "lime", 7, 340, 10, 10, 0.9));
                    break;
            }
            playerCash -= TOWER_PRICES[selectedTower];
            towerPlace = false;
            mapFilled[tileX][tileY] = 1;
            placeCancelButtonDisplay(false);
        }
    }
}
```

mousePressed is managed by the p5.js library and is true if the user currently has the mouse clicked (including held) down. *mouseX* and *mouseY* contain the current position of the mouse cursor (relative to the canvas). If *towerPlace* is true (so the user has clicked a tower button and changed the value of *selectedTower* to the tower they want) and the mouse is being clicked somewhere on the game screen (i.e. not on one of the other buttons) then the code will run (assuming the user has enough money). The tile they are clicking on is then found by dividing the current mouse cursor coordinates by the cell size and rounding this number down to the nearest integer (since the tile coordinates are in the top left corner). If *mapFilled* has this index as 0 then the user is allowed to place a tower there and this happens. The tower the user wants is pushed to the tower array, the player cash is reduced, the tower placement screen is turned off, the index where they placed a tower is turned to 1 in *mapFilled* (to ensure they can't stack towers above each other) and the cancel tower placement button is hidden.

Iteration 2 - Finalising the towers and projectiles

Following on from my last attempt at completing this success criteria, I started by moving the *checkIfCollision* method from the *projectileQueue* class to the *towerBasic* class. This was purely for modular purposes since having this function which needs the target enemy passed into it (because it is outside the (*towerBasic*) class where the target enemy is stored) seemed a bit unnecessary. Doing this seemed to fix the problem of the projectiles being left as a trail behind the enemy rather than hitting it and disappearing. I honestly don't know why this is, my assumption is due to a lack of memory and that passing a variable back and forth between methods was causing a slight delay meaning when the projectiles reached the enemy, the fact there was a collision between them was ignored (since this is checked in a different class) but the projectiles still move towards the position of the enemy. This might also be true because when there are fewer enemies on screen it works fine (less memory needed) but once faster enemies start to appear it starts to break down (their x and y are changing more frequently meaning slightly more memory may be needed). To deal with this slightly better, I increased the distance the projectiles hit an enemy from if the enemy type moves at a speed greater than 0.17 (I decided this value by testing which speeds cause the projectiles to be left as a trail).

```
// Checks if any projectiles in the queue (i.e. the first) have hit the enemy.
    checkIfCollision() {
        let type = this.targetEnemy.type;
        let pointer = this.targetEnemy.pointer;
        let projX =  this.projectiles.queue[this.projectiles.head].posX;
        let projY = this.projectiles.queue[this.projectiles.head].posY;
        let enemyX = enemies[type].queue[pointer].posX;
        let enemyY = enemies[type].queue[pointer].posY;

        if (type <= 4 && distance(projX, projY, enemyX, enemyY) < 100) {
            this.projectiles.dequeue();
            if (type >= 2 && enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (distance(projX, projY, enemyX, enemyY) < 130) {
            this.projectiles.dequeue();
```

```
            if (enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }
        }
    }
```

Iteration 2 - Adding an enemy health/armour bar and a button to toggle the tower ranges on and off

The final features in this iteration are adding a health/armour bar below each enemy (if they are damaged. This was very simple since I'd already done a similar thing with the base health so I copied the code almost exactly for the health and for the armour, changed the colours from green/red to dark/light blue. I added variables to store the max health and armour for each enemy since I only want the health bar to be displayed if the enemy has taken some damage (from the maximum value). I also added a method called *display* to *enemyBasic* (which is overridden in *enemyVehicle* since the armour bar needs to be displayed as well) since the old *update* function was doing too much for one method so I removed all the parts of displaying the enemy from this and added them to the new method.

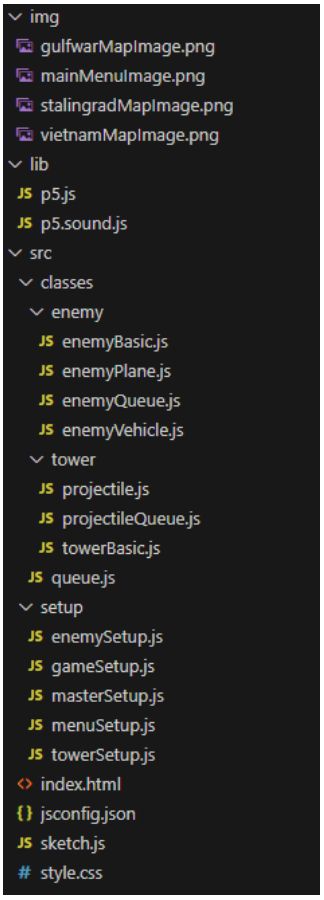
```
// If the enemies still have health or armour, it is displayed. This overrides the parent method because a blue bar needs to be displayed for vehicles to show their armour.
    display() {
      if (this.health > 0) {
        if (this.armour > 0 && this.armour < this.armourMax) {
            let percentage = this.armour / this.armourMax;
            let start = color(95, 201, 209);
            let end = color(0, 18, 132);
            let gradientColour = lerpColor(start, end, percentage); // Colours the armour bar a percentage between dark blue and light blue depending on the armour remaining.
            let barWidth = map(percentage, 0, 1, 0, 76); // Makes the size of the health bar proportional to the health remaining.
            fill(gradientColour);
            noStroke();
```

```
            rect(this.posX + 10, this.posY + 91, barWidth, 5);

        } else if (this.health < this.healthMax) {
            let percentage = this.health / this.healthMax;
            let start = color(255, 0, 0);
            let end = color(0, 255, 0);
            let gradientColour = lerpColor(start, end, percentage); // Colours
the health bar a percentage between red and green depending on the health
remaining.

            let barWidth = map(percentage, 0, 1, 0, 76); // Makes the size of
the health bar proportional to the health remaining.

            fill(gradientColour);
            noStroke();
            rect(this.posX + 10, this.posY + 91, barWidth, 5);
      }
      fill(this.colour);
      rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
    }
  }
```

The final feature of this iteration is adding a button to toggle the tower ranges on and off. Before adding this all the tower ranges were displayed on the screen permanently which gets obnoxious when you start placing more and more towers. However, the user still may want to see the range of their towers to make sure they have all spots covered. So I added a button which they click to turn the ranges on and off (which flips the value of a variable called *displayRange* between true and false). This variable is then checked when the towers are displayed and if it is true then the ranges are displayed too.

D9 - Structure and modularity:

The screenshot below is of the directory containing my code so far:



My code is still very modular and object-oriented in structure, as shown below:

```
switch(gameState) {

    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      mapButtonsDisplay(false);
      returnToMenuButtonDisplay(false);
      placeCancelButtonDisplay(false);
      displayRangeButtonDisplay(false);
      towerButtonDisplay(false);
      break;

    // Loads the campaign game mode.
    case 2:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      break;

    // Loads the arcade game mode.
```

```
  case 3:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
    } else{
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemySpawn();
      enemyUpdate();
      towerUpdate();
    }
    break;

  // Loads the tutorial.
  case 4:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);

    break;

  // Loads the skill tree.
  case 5:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);

    break;

  // Loads the settings.
  case 6:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);

    break;
```

```
    default:
      gameState = 1;
  }
```

Below are all the classes made/edited in this iteration:

queue
+ queue[] + head + tail
+ constructor():void + enqueue(newItem):void + dequeue():object + clear():void

enemyQueue (now a subclass of queue)
(no new attributes, refer to previous iteration)
+ clear():void

enemyBasic
+ healthMax
+ display():void

enemyVehicle (subclass of enemyBasic)
+ armourMax
+ display():void

projectile
+ posX + posY + targetX + targetY + colour + speed + damage
+ constructor():void + update():void

projectileQueue (subclass of queue)
(no new attributes, refer to parent class)
+ updatePosistion():void

towerBasic
+ posX + posY + colour + type + range + damage + speed + projectiles + targetEnemy
+ constructor(posX, posY, colour, type, range, damage, speed):void + update():void + calculateDistance():object + checkEnemyType():boolean + fireProjectile():void + checkIfCollision():void

D10 - Code annotation and comments:

sketch.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on 14/06/23. This is the main file which combines code from other files to run the program.

"use strict";

// Preload function from the p5.js library. Contains images which need to be loaded before the main menu appears.
function preload() {
  mainMenuImage = loadImage("./../img/mainMenuImage.png");
  vietnamMapImage = loadImage("./../img/vietnamMapImage.png");
  stalingradMapImage = loadImage("./../img/stalingradMapImage.png");
  gulfwarMapImage = loadImage("./../img/gulfwarMapImage.png");
}
```

```
// Setup function from the p5.js library. This function is run once at the
start of the program and never again, so it is used for creating the canvas,
etc.
function setup() {
  createCanvas(1856, 864);
  menuButtonsSetup();
  mapButtonsSetup();
  towerButtonsSetup();
}

// Draw function from the p5.js library. This function runs 60 times per
second and is used for animation (i.e. updating object positions, drawing map,
etc.).
function draw() {
  background(21, 191, 61);
  switch(gameState) {

    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      mapButtonsDisplay(false);
      returnToMenuButtonDisplay(false);
      placeCancelButtonDisplay(false);
      displayRangeButtonDisplay(false);
      towerButtonDisplay(false);
      break;

    // Loads the campaign game mode.
    case 2:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      break;

    // Loads the arcade game mode.
    case 3:
      if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
        mapButtonsDisplay(true);
        mainMenuButtonsDisplay(false);
        returnToMenuButtonDisplay(true);
      } else{
        mapButtonsDisplay(false);
```

```
        mainMenuButtonsDisplay(false);
        displayRangeButtonDisplay(true);
        returnToMenuButtonDisplay(true);
        towerButtonDisplay(true);
        drawMap();
        displayInformation();
        displayBaseHealth(baseHealth, baseHealthMax);
        placeTowerFunction();
        enemySpawn();
        enemyUpdate();
        towerUpdate();
      }
      break;

    // Loads the tutorial.
    case 4:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    // Loads the skill tree.
    case 5:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    // Loads the settings.
    case 6:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    default:
      gameState = 1;
  }
}
```

towerBasic.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on 22/08/23. This file contains the base class for the towers.

"use strict";

class towerBasic {

    // Constructor method for the class.
    constructor(posX, posY, colour, type, range, damage, speed) {
        this.posX = posX;
        this.posY = posY;
        this.colour = colour;
        this.type = type; // Type of tower.
        this.range = range;
        this.damage = damage;
        this.speed = speed;
        this.projectiles = new projectileQueue(); // Queue to store and manage the towers projectiles.
        this.targetEnemy; // Stores the type and pointer of the enemy, if one is in range.
        this.time = 0; // Stores the time since the last projectile hit an enemy.
    }

    // Displays the tower (which doesn't move) and calls the method to move the projectiles. Also calls the method to check whether an enemy is in range of the tower.
    update() {
        this.time +=deltaTime;

        // If an enemy is in range, information to find its position is stored in targetEnemy and a projectile is fired towards this position.
        if (this.calculateDistance() != false) {
            this.targetEnemy = this.calculateDistance();
            if (this.checkEnemyType()) {
                this.fireProjectile();
            }
        } else {
            this.projectiles.clear();
        }
```

```
        // If a projectile has been fired, its position is updated and whether
or not it has hit the enemy is checked.
        if (this.projectiles.queue.length != 0) {
            this.projectiles.updatePosistion();
            this.checkIfCollision();
        }

        fill(this.colour);
        circle(this.posX + 48, this.posY + 48, 76);

        // If the user wants, the tower ranges are displayed.
        if (displayRange) {
            noFill();
            stroke(this.colour);
            circle(this.posX + 48, this.posY + 48, this.range*2);
        }
    }

    // Checks if an enemy is in range of the tower.
    calculateDistance() {
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = enemies[i].queue[j].posX;
                let enemyY = enemies[i].queue[j].posY;
                if (distance(this.posX, this.posY, enemyX, enemyY) <
this.range){
                    return {
                        type: enemies[i].type,
                        pointer: enemies[i].queue[j].pointer
                    }
                }
            }
        }
        this.projectiles.clear();
        return false;
    }

    // Check if the tower can shoot at the given enemy type
    checkEnemyType() {
        if(this.type <= 1 && this.targetEnemy.type <= 1) {
            return true;
```

```
        } else if((this.type >= 2 && this.type <= 4) && this.targetEnemy.type
<= 4) {
            return true;
        } else if(this.type == 5 && this.targetEnemy.type <= 7) {
            return true;
        } else if(this.type == 6 && this.targetEnemy.type != 9) {
            return true;
        } else if(this.type == 7) {
            return true;
        } else {
            return false;
        }
    }

    // Fires a projectile at the target enemy.
    fireProjectile(){
        this.projectiles.enqueue(new projectile(this.posX, this.posY,
enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posX,

enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posY,
this.colour, this.speed, this.damage));
    }

    // Checks if any projectiles in the queue (i.e. the first) have hit the
enemy.
    checkIfCollision() {
        let type = this.targetEnemy.type;
        let pointer = this.targetEnemy.pointer;
        let projX;
        let projY;
        if (this.projectiles.queue[this.projectiles.head]) {
            projX =  this.projectiles.queue[this.projectiles.head].posX;
            projY = this.projectiles.queue[this.projectiles.head].posY;
        }
        let enemyX = enemies[type].queue[pointer].posX;
        let enemyY = enemies[type].queue[pointer].posY;

        if (type <= 4 && distance(projX, projY, enemyX, enemyY) < 100) {
            this.time = 0;
            this.projectiles.dequeue();
            if (type >= 2 && enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
```

```
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (distance(projX, projY, enemyX, enemyY) < 150) {
            this.time = 0;
            this.projectiles.dequeue();
            if (enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        }
    }
}
```

towerSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
23/07/23. This file contains the setup information for the towers.

"use strict";

let towers = [];

let mapFilled = [
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
```

```
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    ];


const TOWER_PRICES = [100, 250, 400, 500, 750, 900, 1200, 1500];

let placeCancelButton, displayRangeButon;

let soldierButton, machineGunnerButton, rpgButton, ifvButton, tankButton,
shoulderAntiAirButton, surfaceAirButton, advancedSurfaceAirButton;

let selectedTower;

let displayRange = false;

// Declares all the buttons the user click to place a specific type of tower.
function towerButtonsSetup() {
    placeCancelButton = createButton("Cancel");
    placeCancelButton.position(1640, 720);
    placeCancelButton.class("selectTowerButtonClass");
    placeCancelButton.mousePressed(placeCancelButtonFunction);

    soldierButton = createButton("Soldier <br> 💵100");
    soldierButton.position(1640, 300);
    soldierButton.class("towerButtonClass");
    soldierButton.mousePressed(() => towerButtonFunction(0));

    machineGunnerButton = createButton("Machine Gunner <br> 💵250");
```

```
    machineGunnerButton.position(1740, 300);
    machineGunnerButton.class("towerButtonClass");
    machineGunnerButton.mousePressed(() => towerButtonFunction(1));

    rpgButton = createButton("RPG <br> 💵400");
    rpgButton.position(1640, 400);
    rpgButton.class("towerButtonClass");
    rpgButton.mousePressed(() => towerButtonFunction(2));

    ifvButton = createButton("IFV <br> 💵500");
    ifvButton.position(1740, 400);
    ifvButton.class("towerButtonClass");
    ifvButton.mousePressed(() => towerButtonFunction(3));

    tankButton = createButton("Tank <br> 💵750");
    tankButton.position(1640, 500);
    tankButton.class("towerButtonClass");
    tankButton.mousePressed(() => towerButtonFunction(4));

    shoulderAntiAirButton = createButton("MANPADS <br> 💵900");
    shoulderAntiAirButton.position(1740, 500);
    shoulderAntiAirButton.class("towerButtonClass");
    shoulderAntiAirButton.mousePressed(() => towerButtonFunction(5));

    surfaceAirButton = createButton("Surface to Air Missile <br> 💵1200");
    surfaceAirButton.position(1640, 600);
    surfaceAirButton.class("towerButtonClass");
    surfaceAirButton.mousePressed(() => towerButtonFunction(6));

    advancedSurfaceAirButton = createButton("Advanced Surface to Air Missile
<br> 💵1500");
    advancedSurfaceAirButton.position(1740, 600);
    advancedSurfaceAirButton.class("towerButtonClass");
    advancedSurfaceAirButton.mousePressed(() => towerButtonFunction(7));

    displayRangeButon = createButton("Toggle tower ranges");
    displayRangeButon.position(1640, 780);
    displayRangeButon.class("selectTowerButtonClass");
    displayRangeButon.mousePressed(displayRangeButtonFunction);
}

// Places a tower at the x and y the user click.
```

```
function placeTowerFunction() {
    if (towerPlace && mouseIsPressed && mouseX >= MAP_TILE_X[0] && mouseX <=
MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <= MAP_TILE_Y[9]
        && playerCash >= TOWER_PRICES[selectedTower]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (mapFilled[tileX][tileY] == 0) {
            switch(selectedTower) {
                case 0:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "red", 0, 200, 2, 0.2)); // Soldier.
                    break;

                case 1:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "yellow", 1, 220, 2, 0.2)); // Machine gunner.
                    break;

                case 2:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "gray", 2, 240, 3, 0.19)); // RPG.
                    break;

                case 3:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "blue", 3, 260, 3, 0.23)); // IFV.
                    break;

                case 4:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "green", 4, 280, 4, 0.24)); // Tank.
                    break;

                case 5:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "orange", 5, 300, 4, 0.25)); // MANPADS.
                    break;

                case 6:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "purple", 6, 320, 5, 0.26)); // Surface Air Missile.
                    break;
```

```
                case 7:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "black", 7, 340, 6, 0.27)); // Advanced Surface Air
Missile.
                    break;
            }
            playerCash -= TOWER_PRICES[selectedTower];
            towerPlace = false;
            mapFilled[tileX][tileY] = 1;
            placeCancelButtonDisplay(false);
        }
    }
}

// Function called if the user cancels the tower place.
function placeCancelButtonFunction() {
    towerPlace = false;
    placeCancelButtonDisplay(false);
}

// Show or hides the tower place buttons depending on the input.
function towerButtonDisplay(display) {
    if (display) {
        soldierButton.show();
        machineGunnerButton.show();
        rpgButton.show();
        ifvButton.show();
        tankButton.show();
        shoulderAntiAirButton.show();
        surfaceAirButton.show();
        advancedSurfaceAirButton.show();

    } else {
        soldierButton.hide();
        machineGunnerButton.hide();
        rpgButton.hide();
        ifvButton.hide();
        tankButton.hide();
        shoulderAntiAirButton.hide();
        surfaceAirButton.hide();
        advancedSurfaceAirButton.hide();
```

```
    }
}

// Shows or hides the tower place cancel button depending on the input.
function placeCancelButtonDisplay(display) {
    if (display) {
        placeCancelButton.show();
    } else {
        placeCancelButton.hide();
    }
}

// Shows or hides the toggle tower range button depending on the input.
function displayRangeButtonDisplay(display) {
    if (display) {
        displayRangeButon.show();
    } else {
        displayRangeButon.hide();
    }
}

// Function called when one of the tower buttons is clicked to select that
tower to be placed.
function towerButtonFunction(tower) {
    towerButtonDisplay(false);
    placeCancelButtonDisplay(true);
    towerPlace = true;
    selectedTower = tower;
}

// Function called when the toggle tower range button is clicked.
function displayRangeButtonFunction() {
    if (displayRange || towers.length == 0) {
        displayRange = false;
    } else {
        displayRange = true;
    }
}

//Updates each of the towers the user has placed.
function towerUpdate() {
    for (let tower of towers) {
```

```
        tower.update();
    }
}
```

Above are a few examples of code annotation and comments, every file has similar annotation to the screenshots above.

D11 - Naming:

Variables	Data Structures and Objects	Subroutines
• towerPlace • selectedTower • displayRange	• MAP_STALINGRAD • MAP_GULFWAR • MAP_BLANK • chosenMap • vietnamMapButton • stalingradMapButton • gulfwarMapButton • vietnamMapImage • stalingradMapImage • gulfwarMapImage • enemies • towers • mapFilled • TOWER_PRICES • placeCancelButton • displayRangeButton • soldierButton • machineGunnerButton • rpgButton • ifvButton • tankButton • shoulderAntiAirButton • surfaceAirButton • advancedSurfaceAirButton	• drawMap() • mapButtonsSetup() • mapButtonsDisplay(display) • vietnamMapButtonFunction() • stalingradMapButtonFunction() • gulfwarMapButtonFunction() • enemyClear() • checkBaseHealth() • cosineInterpolate(min, max, ratio) • clearArray(array) • towerButtonsSetup() • placeTowerFunction() • placeCancelButtonFunction() • towerButtonDisplay(display) • placeCancelButtonDisplay(display) • displayRangeButtonDisplay(display) • towerButtonFunction(tower) • displayRangeButtonFunction() • towerUpdate() • addScoreAndCash(type)

queue
+ queue[] + head + tail
+ constructor():void + enqueue(newItem):void + dequeue():object + clear():void

enemyQueue (now a subclass of queue)

(no new attributes, refer to previous iteration)
+ clear():void

enemyBasic
+ healthMax
+ display():void

enemyVehicle (subclass of enemyBasic)
+ armourMax
+ display():void

projectile
+ posX + posY + targetX + targetY + colour + speed + damage
+ constructor():void + update():void

projectileQueue (subclass of queue)
(no new attributes, refer to parent class)
+ updatePosistion():void

towerBasic
+ posX + posY + colour + type + range + damage + speed + projectiles + targetEnemy

+ constructor(posX, posY, colour, type, range, damage, speed):void
+ update():void
+ calculateDistance():object
+ checkEnemyType():boolean
+ fireProjectile():void
+ checkIfCollision():void

D12 - Evidence of validation:

- In the main loop of the *sketch.js* file, the value of *chosenMap* is compared to *MAP_BLANK*. If their values are the same, this means the user has not selected a map so the map selection menu is brought up. If not, this means *chosenMap* is equal to one of the maps and the game loads.

```
// If the user hasn't picked a map, the map selection screen is brought up else the game loads
if(JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK) ){
        mapButtonsDisplay(true);
        mainMenuButtonsDisplay(false);
        returnToMenuButtonDisplay(true);
      } else {
        mapButtonsDisplay(false);
        mainMenuButtonsDisplay(false);
        returnToMenuButtonDisplay(true);
        drawMap();
        displayInformation();
        displayBaseHealth(baseHealth, baseHealthMax);
        enemySpawn();
        enemyUpdate();
      }
```

- In the function *cosineInterpolate*, trigonometry is used to ensure interpolation occurs smoothly and to make sure this works as expected the angle mode is changed to radians

```
// Cosine interpolation using linear interpolation to make the animation smoother.
function cosineInterpolate(min, max, ratio) {
    angleMode(RADIANS);
    return linearInterpolate(min, max, (1-cos(ratio*PI))/2)
}
```

- In the *towerUpdate* method (where each tower is displayed), the for loop is only run if the user has placed a tower. Since it is declared as blank, there would be issues if we

tried to use a traditional for loop since it would have to start at i=0 (which doesn't exist in a blank array) so a for…of… loop is used instead

```
//Updates each of the towers the user has placed.
function towerUpdate() {
    for(let tower of towers) {
        tower.update();
    }
}
```

- In the function *placeTowerFunction()* (where a user clicks the tile they wish to place a tower on), validation is used to ensure that *towerPlace* is on (a variable turned on by the user clicking the button to place a tower), the mouse is currently pressed down and the mouse is on the area of the screen where the tiles are (and not still hovering over the button to select a tower, for example). Furthermore, we need to check that the tile they have clicked is valid (isn't a path tile; isn't already populated by a tower etc.) so the tile is looked up in the *mapFilled* array to ensure this is valid (which manages all this).

```
// Place a tower at the x and y the user clicks.
function placeTowerFunction() {
    if (towerPlace && mouseIsPressed && mouseX >= MAP_TILE_X[0] &&
mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <=
MAP_TILE_Y[9]
        && playerCash >= TOWER_PRICES[selectedTower]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (mapFilled[tileX][tileY] == 0) {
            switch(selectedTower) {
                ...
            }
        }
    }
}
```

- In the function *displayRangeButtonFunction*, as well as checking if display range is true or not (since this needs to be toggled), an additional check that the user has placed at least once tower is used to ensure there are no undefined errors.

```
// Function called when the toggle tower range button is clicked.
function displayRangeButtonFunction() {
    if (displayRange || towers.length == 0) {
        displayRange = false;
    } else {
        displayRange = true;
    }
}
```

- In the *enemyVehicle* class (and the *enemyBasic* class excluding anything involving armour), a health or armour bar is displayed below the enemies once they have taken damage. If the enemy health is over zero then the enemy (circle) is displayed. If the armour has been damaged and is below the maximum, as well as checking it is over 0 (since it can be in the negatives meaning it shouldn't be damaged/displayed anymore but this branch would still be checked since this is still less than the max and therefore the health bar would never be displayed).

```
// If the enemies still have health or armour, it is displayed. This
overrides the parent method because a blue bar needs to be displayed for
vehicles to show their armour.
    display() {
      if (this.health > 0) {
        if (this.armour > 0 && this.armour < this.armourMax) {
          . . .
        } else if (this.health < this.healthMax) {
          . . .
      }
      fill(this.colour);
      rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE -
20);
    }
  }
```

- In the *towerBasic* class, within the *update* method, the first thing that's checked is whether there is an enemy in range of the tower. If there is, then this information is stored. Without this validation, we would get many undefined errors throughout the code (since *targetEnemy* is used in various other places). After this, *checkEnemyType* is run which checks the type of *targetEnemy* and validates whether the tower can hit it or not (since only certain types of towers can hit certain types of enemy).

```
// If an enemy is in range, information to find its position is stored
in targetEnemy and a projectile is fired towards this position.
        if (this.calculateDistance() != false) {
            this.targetEnemy = this.calculateDistance();
            if (this.checkEnemyType()) {
                this.fireProjectile();
            }
        } else {
            this.projectiles.clear();
        }
```

- In the *checkIfCollision* method within the *towerBasic* class, the first check is whether the enemy type is less than or equal to 4 since beyond this enemy type, the enemies

move slightly faster so the distance the projectile needs to be to them is less. After this, another check needs to be done to check whether the enemy has armour (the type is greater than 1) and that this armour hasn't been destroyed. If so, the armour is damaged, otherwise the health is damaged. Finally, if this hit destroys the enemy then the first enemy in the queue is removed.

```
        if (type <= 4 && distance(projX, projY, enemyX, enemyY) < 100) {
            this.projectiles.dequeue();
            if (type >= 2 && enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is
cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (distance(projX, projY, enemyX, enemyY) < 130) {
            . . .
        }
```

D13 - Prototypes:

The video below shows the solution at the end of this iteration:
https://drive.google.com/file/d/1MGW1YvF5h60dSv-Bi8v9mCOqvGCYkuvV/view?usp=drive_link

D14 - Review:

In this iteration, I completed the following success criteria.

- (5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.
- (6) A projectile fired from the tower moves towards the enemy. When it hits the enemy, this collision is detected and the enemy takes damage. The projectile disappears after this. If the damage was enough to destroy the enemy, the enemy

also disappears. This ensures that the game works as required for a tower defence game.
- (7) When the base health is zero, the game ends and resets. This means the user can play again if they are playing arcade or restart the level if they are playing the campaign.
- (10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.

Taking in feedback from the previous iteration, I added the following features:
- Health under the enemies
- Change the colours on the main menu
- Make the enemies move smoothly along the track

After reviewing the solution currently, the stakeholders had this to say:

- William Thorne - “All the buttons work correctly and the game runs smoothly from buying and placing turrets and the game ending when you take enough damage. The turret and cash system works although It’s not clear which turrets can damage which units. I like that you can quickly start another game. It would be nice to add an ending to each map, possibly to unlock the next level. I like that the background changes colour when I click to buy an item as it lets me know it’s working, although it would be better if I can see which item I clicked on last as it’s the same for all. Also I haven’t had any issues with the game not working or crashing.”

- Dominic Keyworth - “Once again, the game looks great and it is clear that you have used our feedback in a constructive way. The menu buttons are brilliant!
  What went well:
  - brilliant design of maps, the colours and paths really showcased the warzone theme.
  - inclusion of a menu button
  - great theme specific naming of defences
  - still clear user interface
  - clear evidence of theme knowledge with map namings
  What could be improved
  -Still think the home page could be improved with added detail
  - No indication of the differences in the colourings of the attackers? Is one stronger?
  -Soldiers don't always work”

- Sam Russell - “Good:
  - Toggle tower range is a good feature, allowing me to see how far each tower reaches
  - Seeing the bullets makes the game more interactive, that you can see what the tower is doing
  Feedback:
  - Towers target the newest target in range, meaning that the targets get damaged but often not completely destroyed
  - Would be good if the select tower showed what colour tower you're selecting.”

- Harvey Miller - “I like the fact that you can see the ranges of the towers and each enemy gives an amount of cash depending on how hard they are to kill
  Bugs :
  - If you select tower ranges to be visible and try to drag a tower onto the map the tower select and cancel buttons disappear and the other two buttons (menu and toggle tower ranges) don't work when clicked
  - I am assuming that it is a feature that only some towers can shoot some types of enemies but this should be better illustrated to the player as to which towers can hit which types”

From this feedback I plan on changing the following things:

- Make it clear which turrets can damage which units.
- Highlight which tower the user is about to place when they click to place one.

Iterative Testing & Error Remedies:

D16 - Iterative Testing:

Iteration Number.Test Number	**Outcome**	**Evidence**
2.1	Fail	https://drive.google.com/file/d/10WEW7tP5-PA1MvpSf91FzTk1ZFFXlNWg/view?usp=drive_link
2.1	Pass	https://drive.google.com/file/d/1lpGAotl4asXEKwU2px--JqwrfqieHcAb/view?usp=drive_link
2.2	Pass	https://drive.google.com/file/d/1GxMEVmQ3cehlMIyNhJGMwBWIk8r2cAGe/view?usp=drive_link
2.3	Pass	https://drive.google.com/file/d/1_5JYMcxQN4_rU4qUSTjdb_I5mIKLIFBS/view?usp=drive_link
2.4	Pass	https://drive.google.com/file/d/1sCA8nEsP5fr_UnopguMo_AuqwIUq4Mkj/view?usp=drive_link (Note I sped up the enemies so they destroy the base quicker)
2.5	Pass	https://drive.google.com/file/d/1k0hCKm2psDV541HUe89u1pqEz5CZ9qwG/view?usp=drive_link (Note I sped up the enemies so they destroy the base quicker)
2.6	Pass	https://drive.google.com/file/d/1ryCEKOqX4qN9q3UZ2hPDJVzorSUXLLTx/view?usp=drive_link
2.7	Fail	https://drive.google.com/file/d/1ryCEKOqX4qN9q3UZ2hPDJVzorSUXLLTx/view?usp=drive_link

2.8	Pass	https://drive.google.com/file/d/1v1UZMqlXVPVfMiN91_J7A07J9fZnrZsR/view?usp=drive_link
2.9	Pass	https://drive.google.com/file/d/10TcVWHi8X0mtWRuGoPOn_PzPY2Dvp5a9/view?usp=drive_link
2.10	Pass	https://drive.google.com/file/d/17u0CBgZJh9sQ4kNvKWOQIRrAtTkyjxdW/view?usp=drive_link
2.7	Fail	https://drive.google.com/file/d/1ZTPVgic1Lgcw4SgVN965-9-DSU7WUf8X/view?usp=drive_link (skip to about 1 minute to see the bug, it takes longer to appear than before because the bug is slightly fixed)
2.7	Pass	https://drive.google.com/file/d/1lzCtyJd4j1-8XsRxxXOZmlQiET_yxiG1/view?usp=drive_link
2.11	Pass	https://drive.google.com/file/d/1bHbw6vmS2DiI28YPIHS2V2fBN0oa973Y/view?usp=drive_link
2.12	Pass	https://drive.google.com/file/d/10Qza35nfF8Myo0Sj4zkoD7t14nRUTrhG/view?usp=drive_link

D17 - Non trivial failed tests and remedies:

I ran test 2.1 and it failed. Clicking the “Arcade” button did not bring up the Stalingrad-themed map despite *chosenMap* being set as *MAP_STALINGRAD*. There was no error message. A link to the video of the failed test is below: https://drive.google.com/file/d/10WEW7tP5-PA1MvpSf91FzTk1ZFFXlNWg/view?usp=drive_link

My first instinct was the issue was because of the nature of the switch-case branching and the way Javascript arrays are compared. Switch cases use the strict comparison (===) which means the type of data must also be the same (“0”==0 is true in javascript but “0”===0 is false). Arrays also follow this pattern so when comparing the map arrays with the switch case (strict comparison), they are seen as false. Following this thought process, I changed the switch-case to an if statement and after looking up a method that compares arrays as I want it to, I used the *JSON.stringify* method since comparing arrays in JSON works as I want it to.

```
// Filling in the tiles needed
  if(JSON.stringify(chosenMap)==JSON.stringify(MAP_VIETNAM)) {
    for(let i=0; i<MAP_VIETNAM.length; i++) {
      for(let j=0; j<MAP_VIETNAM[0].length; j++) {
        if(MAP_VIETNAM[i][j] == 0) {
          fill(3, 38, 11);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if(MAP_VIETNAM[i][j] == 1){
```

```
          fill(66, 66, 32);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if(MAP_VIETNAM[i][j] == 2){
          fill("black");
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if(MAP_VIETNAM[i][j] == 3){
          fill(69, 69, 69);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        }
      }
    }
//. . . with other maps
```

This worked as expected. A link to the video of the new passed test is below:
https://drive.google.com/file/d/1lpGAotl4asXEKwU2px--JqwrfqieHcAb/view?usp=drive_link

---

I ran test 2.7 and it failed. When the projectiles hit the enemy, the projectiles sometimes didn't disappear or. There was no error message. A link to the video of the failed test is below:
https://drive.google.com/file/d/1ryCEKOqX4qN9q3UZ2hPDJVzorSUXLLTx/view?usp=drive_link

After using various *console.log*, of all the attributes involved in calculating whether or not a projectile has hit the target enemy, they all seemed to be working as expected. So instead of assuming this was a logic error, I started to think it could be an issue with memory since a lot of for loops and objects are involved in this process. The first thing I did was move the method *checkIfCollision* (which checks if the first projectile in the queue hits the enemy) from the *projectileQueue* class to the *towerBasic* class, not initially because I thought it would improve the memory issues (it might slightly since this class is in a different file and it may speed up access times) but because I thought this made more sense from a modular point of view since I was having to pass information from a different class into this method which used no information from the class it was in, so I was better off just moving into that other class. However, as well as making my code modular, this slightly fixed the issue and the projectiles worked slightly better than before (still not perfect though). A link to the video of the new test is below:
https://drive.google.com/file/d/1ZTPVgic1Lgcw4SgVN965-9-DSU7WUf8X/view?usp=drive_link (skip to about 1 minute to see the bug, it takes longer to appear than before because the bug is slightly fixed).
After using some more *console.log* I found that every so often the target enemy position becomes undefined, again I assumed due to a memory issue since the code works fine when there is only one or two enemies on screen but it starts to struggle as more appear. I fixed

this simply by checking whether the enemy position exists at the start of the function *checkIfCollision* before moving the projectile position towards the enemy position (since I assume this is why the issue happens in the video above, the enemy position becomes undefined for one frame meaning the projectile heads towards the previous enemy position and never hits the enemy). After adding this, the problem is fixed and the projectiles damage the enemy before disappearing as expected.

```
// Checks if any projectiles in the queue (i.e. the first) have hit the enemy.
    checkIfCollision() {
        let type = this.targetEnemy.type;
        let pointer = this.targetEnemy.pointer;
        let projX;
        let projY;
        if (this.projectiles.queue[this.projectiles.head]) { // This line
checks the enemy position isn't undefined
            projX =  this.projectiles.queue[this.projectiles.head].posX;
            projY = this.projectiles.queue[this.projectiles.head].posY;
        }
        let enemyX = enemies[type].queue[pointer].posX;
        let enemyY = enemies[type].queue[pointer].posY;
        . . .
    }
```

A link to the video of the passed test is below:
https://drive.google.com/file/d/1IzCtyJd4j1-8XsRxxXOZmlQiET_yxiG1/view?usp=drive_link

Iteration 3 - Iterative Design, Develop, Test & Remedy, Review:

Iterative Design:

Success Criteria:

In this iteration, I will be working towards the following success criteria:

- (2) There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players.
- (3) There is an arcade game mode. The user selects one of the maps and plays endlessly with increasing difficulty over time until they lose their base. This gamemode is aimed at casual players.
- (5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are

coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.

- (9) The user can click the upgrade button in the bottom left corner. If they subsequently click a tile with a tower on, they are prompted to click one of two options which to upgrade. If they click the upgrade button again, upgrade mode turns off. This allows for an easy way for the user to upgrade towers.
- (10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.
- (11) The user can access the skill tree from the main menu. They can spend their battle medals and purchase universal upgrades. The user can therefore customise their towers to suit their playstyle, making the game more enjoyable.

Note that success criteria 5 is mostly completed, I'm just going to highlight the tower the user is currently placing by changing the colour of the button. Furthermore, success criteria 10 is partially completed already. However, in this iteration the special ability needs to be added.

D1 - Decomposition:

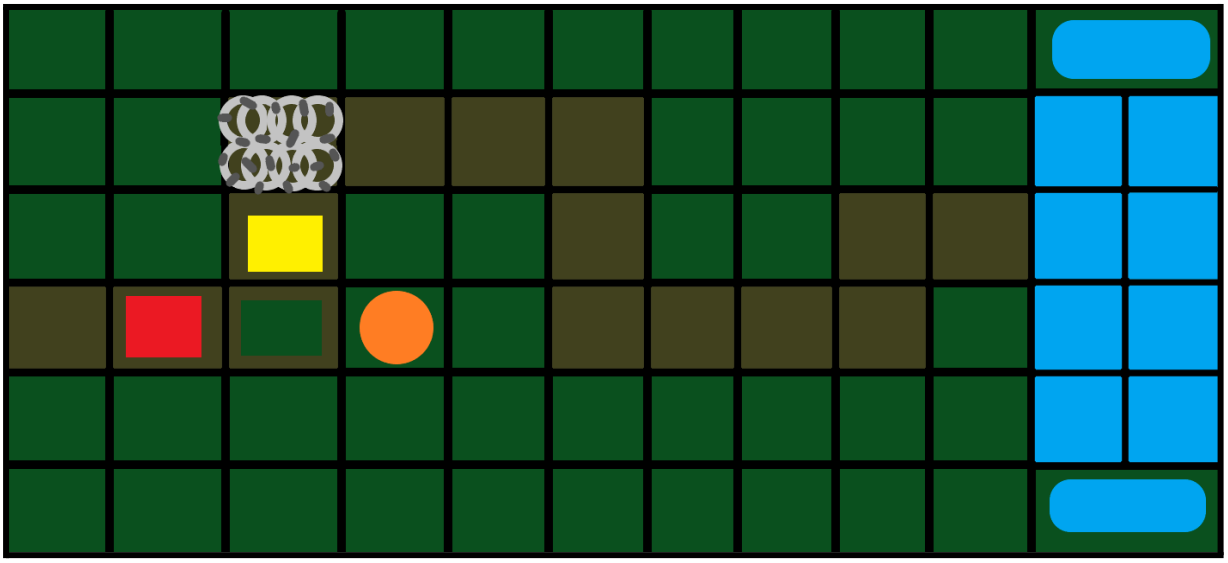


Figure 19: An idea for the special ability on the "Vietnam" map

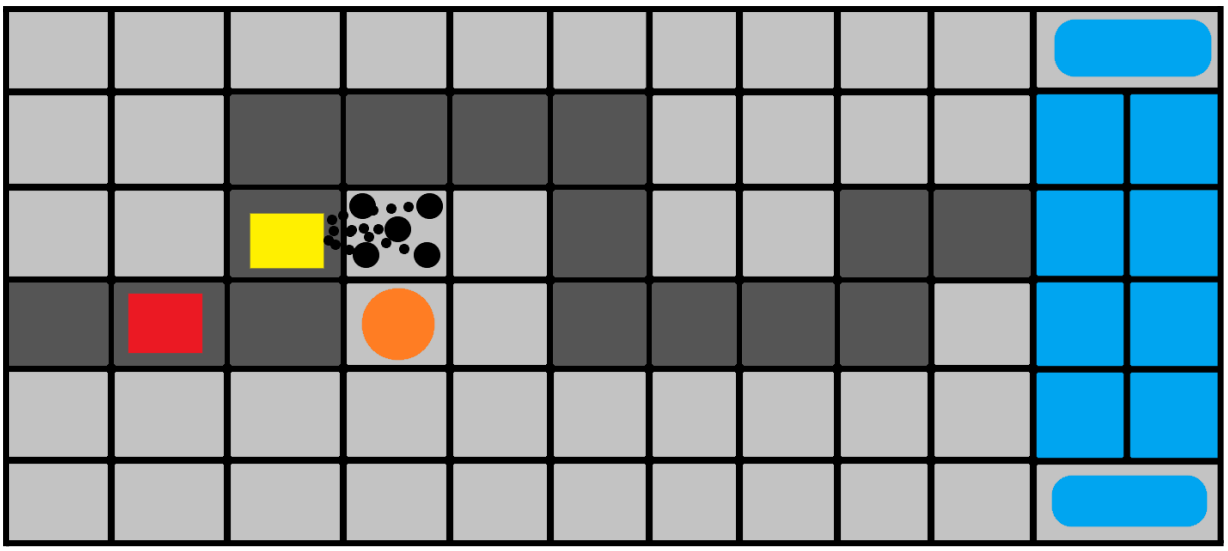


Figure 20: An idea for the special ability on the “Stalingrad” map

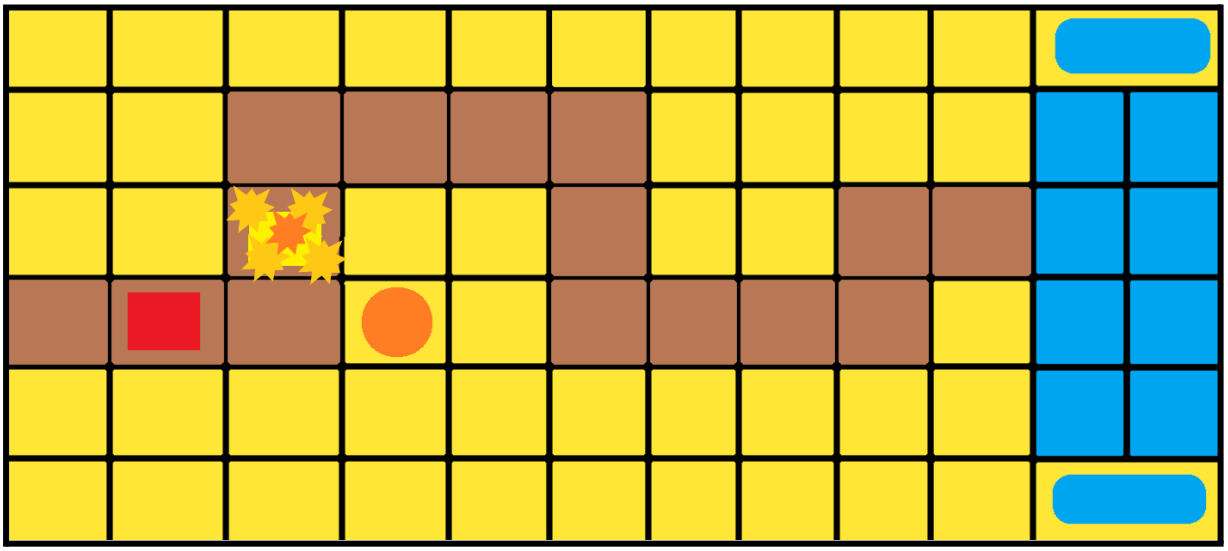


Figure 21: An idea for the special ability on the “Gulf War” map

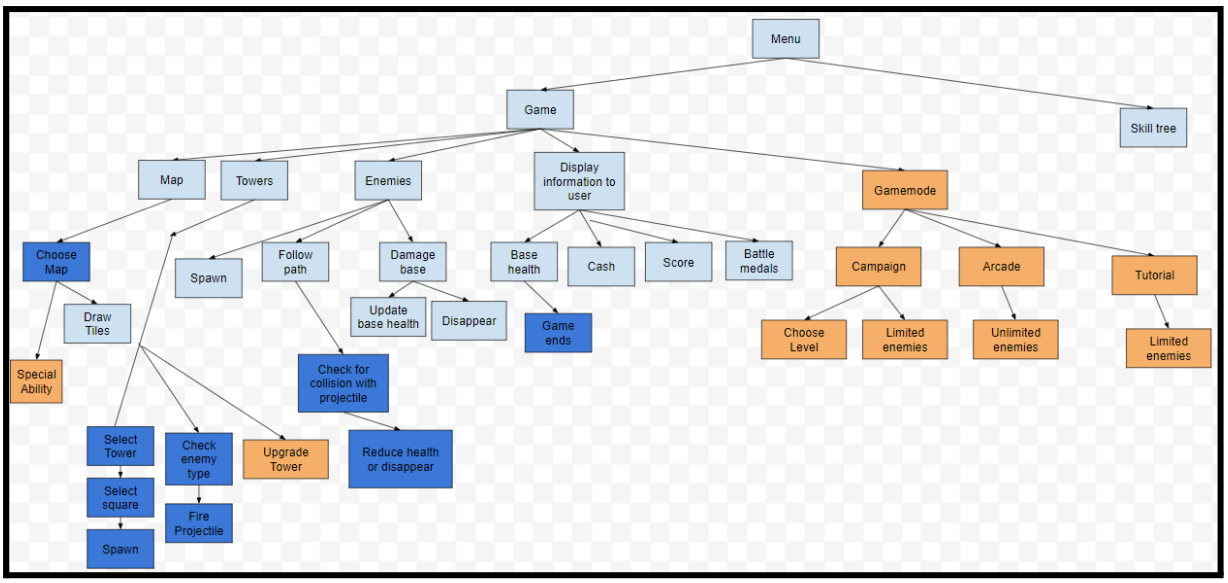


Figure 22: Structure chart for this iteration. Updates are in orange.

Chosen Map has another branch coming off it, *Special Ability*. Each map has a unique ability which can be activated once per game to defend the base. *Gamemode* branches off from *Game* since the user needs to pick the gamemode they want to play. This splits into *Campaign, Arcade* and *Tutorial* since these are the gamemodes the user can pick from. *Campaign* breaks further into *Choose Level* and *Limited Enemies* since the user needs to pick the level they want to play as well as limiting the number of enemies that spawn depending on the difficulty of that level. *Tutorial* also has *Limited enemies* coming off it for the same reason. *Arcade* has *Unlimited Enemies* on it since the gamemode itself is endless and therefore the enemies need to be the same. *Upgrade Tower* is a new branch coming off towers since the user has the option to upgrade individual towers. Finally, *Special Ability* branches off map since each map has its own special ability in this iteration.

D2 - Structure of Solution:

```
computer-science-coursework/
├── img/
│   ├── gulfwarMapImage.png
│   ├── mainMenuImage.png
│   ├── stalingradMapImage.png
│   ├── vietnamMapImage.png
├── lib/
│   ├── p5.js
│   ├── p5.sound.js
├── src/
│   ├── classes/
│   │   ├── enemy/
│   │   │   ├── enemyBasic.js
│   │   │   ├── enemyPlane.js
│   │   │   ├── enemyQueue.js
│   │   │   ├── enemyVehicle.js
```

```
│   │   ├── tower/
│   │   │   ├── projectile.js
│   │   │   ├── projectileQueue.js
│   │   │   ├── towerBasic.js
│   │   ├── level.js
│   │   ├── queue.js
│   ├── setup/
│   │   ├── campaignSetup.js
│   │   ├── enemySetup.js
│   │   ├── gameSetup.js
│   │   ├── masterSetup.js
│   │   ├── menuSetup.js
│   │   ├── skillTreeSetup.js
│   │   ├── specialAbilitySetup.js
│   │   ├── towerSetup.js
│   ├── index.html
│   ├── jsconfig.json
│   ├── sketch.js
│   ├── style.css
```

Files in bold are new.

level
+ rating + baseHealth + soldier + specialForces + truck + apc + ifv + tank + helicopter + fighterJet + bomber + stealthBomber
+ constructor(baseHealth, soldier, specialForces, truck, apc, ifv, tank, helicopter, fighterJet, bomber, stealthBomber):void + setEnemies():void + returnEnemyType(type):Object of type enemyBasic + returnNoOfEnemies(enemy):integer

D3 - Algorithms:

showTrap() - places the trap at the user's mouse position

```
// Places the trap
function showTrap() {
   if (placeTrap == true && allowSpecialAbility == true && mouseIsPressed == true && mouseX >= 0 && mouseX <= MAP_WIDTH && mouseY >= 0 && mouseY <= MAP_HEIGHT) {
        let tileX = Math.floor(mouseX / CELL_SIZE)
        let tileY = Math.floor(mouseY / CELL_SIZE)
        if (chosenMap[tileX][tileY] == enemyPathTile) {
            TRAP.placed = true
            TRAP.posX = Math.floor(mouseX / CELL_SIZE)
            TRAP.posY = Math.floor(mouseY / CELL_SIZE)
            specialAbilityDisappear(5) // make trap disappear after 5 seconds
            placeTrap = false
        }
    }

    if(TRAP.placed == true) {
        drawTrap(); // Draws the trap as a series of circles
    }
}
```

showSpecialForces() - places the special forces on the current tile the user wants

```
// Places the special forces.
function showSpecialForces() {
    if (towerPlace == true && allowAbility == true && mouseIsPressed == true && mouseX >= 0 && mouseX <= MAP_WIDTH && mouseY >= 0 && mouseY <= MAP_HEIGHT) {
        let tileX = Math.floor(mouseX / CELL_SIZE)
        let tileY = Math.floor(mouseY / CELL_SIZE)
        if (mapFilled[tileX][tileY] == 0) {
            specialForces = new towerBasic(tileX, tileY, "black", 8, 240, 2, 0.2)
            allowAbility = false
            towerPlace = false
            mapFilled[tileX][tileY] = 1
            specialAbilityDisappear(15) // makes the special forces disappear after 15 seconds
        }
    }
}
```

D4 - Key variables and data structures including validation:

Variable Name	**Description and Justification**	**Validation**
projectileUpgrade, damageUpgrade, baseHealthUpgrade, enemySpeedUpgrade, enemyArmourUpgrade, specialAbilityUpgrade	Contain the current tier the user has upgraded to in the skill tree.This variable is then passed into the *upgradePercent* function to work out the exact decimal value used for the upgrade.	Must be an integer between 0 and 5 (inclusive). Declared as 0. Only ever edited by the *skillTreeButtonsFunction* which has validation inside it to not change its value if it is above 5.
projectileUpgradeButton, damageUpgradeButton, baseHealthUpgradeButton, enemySpeedUpgradeButton, enemyArmourUpgradeButton, specialAbiltiyUpgradeButton	These buttons are displayed when the user loads the skill tree. These allow the users to spend their battle medals on permanent upgrades.	Validation is not necessary since the user cannot edit the variable themselves, they can only run the function contained within the button which has its own validation. Furthermore, the user can quickly return to the menu and change the map again with the button in the top right corner if needed.
arcade1, arcade2, arcade3	Checks whether the score was above the threshold for the previous frame to increase the enemy spawn rate and therefore the enemy spawn rates have already been changed. Otherwise, the spawn rates are changed (to the same value) every frame which wastes memory.	Must be a boolean value. Declared as false. Controlled by the program flow from here, changed to true when the score threshold is met and never changed back.
levelOneButton, levelTwoButton, levelThreeButton	These buttons are displayed when the user loads the level selection menu. These allow the user to select the level they want.	Validation is not necessary since the user cannot edit the variable themselves, they can only run the function contained within the button which has its own validation.
levels[][]	Contains information about each level including the base health and the	Declared as a blank 2d array. In the lines immediately after filled with each level and unchanged

	number of enemies of each type	from here.
chosenMapInt	Contains an integer value to represent the current map (since *chosenMap* is a 2D array and a copy of the map in question). Changed at the same times as *chosenMap* and only really used to look up the correct level in the 2D array *levels*	Must be an integer between -1 and 2 (inclusive). Declared as -1 which is the default value. Once the user picks a map in the map selection menu it is changed to that specific value. Reset to -1 once the user returns to the menu.
chosenLevel	Contains the users current level which is used to look up the correct level in the 2D array containing the levels along with *chosenMapInt*	Must be an integer between -1 and 2 (inclusive). Declared as -1 which is the default value. Once the user picks a level in the level selection menu it is changed to that specific value. Reset to -1 once the user returns to the menu.
specialAbilityButton	Contains the button which the user clicks to activate the special ability for that map.	Validation is not necessary since the user cannot edit the variable themselves, they can only run the function contained within the button which has its own validation.
allowAbility	Contains a boolean value determining whether the user can use the special ability or not (i.e. if they have already used it this false)	Must be a boolean value. Declared as true and once the user has used the ability it is turned to false. Reset to true if the user returns to the menu or the game ends to ensure they can use the ability the next game.
placeTrap	Contains a boolean value determining whether or not the user wishes to place the trap. If true, it brings up the screen to place the trap.	Must be a boolean value. Declared as false and once the user has clicked the button to use the special ability it becomes true. Once the user has actually placed the trap, it returns to false.
TRAP	An object containing three values. One is a boolean containing whether or not	The boolean is declared as false and changed to true once the user has placed the trap. It is returned back

	the trap is currently placed. The other two contain the coordinates of the trap.	to false after 5 seconds (to make the trap disappear again). The coordinates are similar, declared as 0 (for both) although the values are unused until the user has changed them (by placing the trap).
trapFrameCount	An integer containing the number of frames passed since the trap was placed. Used to ensure the trap is only placed for the set number of seconds.	Declared as 0. Once the user places the trap, set equal to *frameCount* (a p5.js variable containing the number of frames since the program started). Validation is then used to check that once there is a difference of 300 between the two (5 seconds), the trap disappears.
specialForces	An array containing five objects of the *towerBasic* class.	Declared as a blank array. Once the user clicks the special ability button and clicks a valid tile, it is filled with five instances of the *towerBasic* class. After 15 seconds, it is cleared again.
specialForcesFrameCount	An integer containing the number of frames passed since the trap was placed. Used to ensure the special forces are only placed for the set number of seconds.	Must be an integer. Declared as 0. Once the user places the special forces, set equal to *frameCount* (a p5.js variable containing the number of frames since the program started). Validation is then used to check that once there is a difference of 900 between the two (15 seconds), the special forces disappear.
airStrikeCheck	Contains a boolean value determining whether or not the air strike has been called. If true, the animation is displayed.	Must be a boolean value. Declared as false and once the user has placed the air strike it becomes true. Once 2 seconds have passed, the animation disappears and it returns to true.
airStrikeX, airStrikeY	Contains an integer value determining the airstrike coordinates (calculated by	Must be an integer. Declared as 0. Set to the *mouseX* and *mouseY*

	the user's mouse position when they place the air strike). Used in the air strike animation.	positions once the user places the air strike.
airStrikeFrameCount	An integer containing the number of frames passed since the air strike was placed. Used to ensure the air strike animation is only displayed for 2 seconds.	Must be an integer. Declared as 0. Once the user places the air strike, it is set equal to *frameCount* (a p5.js variable containing the number of frames since the program started). Validation is then used to check that once there is a difference of 100 between the two (just under 2 seconds), the special forces disappear.

level
+ rating + baseHealth + soldier + specialForces + truck + apc + ifv + tank + helicopter + fighterJet + bomber + stealthBomber
+ constructor(baseHealth, soldier, specialForces, truck, apc, ifv, tank, helicopter, fighterJet, bomber, stealthBomber):void + setEnemies():void + returnEnemyType(type):Object of type enemyBasic + returnNoOfEnemies(enemy):integer

Attribute Name	**Description and Justification**	**Validation**
rating	Contains the users current progress on the level as a rating out of 3.	Must be an integer between 0-3 (inclusive). Declared as 0. Changed at the end of each level depending on how well the user did.
baseHealth	Contains the value the base	Must be an integer. Other

	health should be set to for a given level.	validation controlled by the *setBaseHealth* method.
soldier, specialForces, truck, apc, ifv, tank, helicopter, fighterJet, bomber, stealthBomber	Contains the number of each enemy type there should be in a given level.	Must be an integer. Other validation controlled by the *setEnemies* method.

D5 - Usability Features:

- Even using the p5.js *preload* function, this screen is barely noticeable and navigating between menus is instant. This improves overall user experience and gives the user a good first impression of the game. This is implemented by minimal use of the *preload* function and by keeping the code modular.
- At the stakeholders request, I added a feature which highlights the current tower being placed. This helps the user keep track of the tower they are currently placing.
- As with placing towers, when placing the special ability a screen appears highlighting which tiles the user can and can't place it on. This ensures the user knows how to use the special ability (since placing the trap, for example, is different to placing a tower since it has to be placed on the enemy path).
- Once the special ability is used, the button changes to grey. This ensures the user is aware they cannot use the special ability again for that game.
- In the skill tree, the user is told their current upgrade level and the cost to upgrade. This means they can compare upgrading different traits and consider which would benefit them most.

D6 - Sample Data and Iteration Tests

Iteration Number.Test Number	**Success Criteria**	**Test Details**	**Expected Outcome**
3.1	(11) The user can access the skill tree from the main menu. They can spend their battle medals and purchase universal upgrades. The user can therefore customise their towers to suit their playstyle, making the game more enjoyable.	1. Run index.html with the live server Visual Studio Code extension 2. Press the "Skill Tree" button 3. Press the "Upgrade Base Health" button	The user spends some of their battle medals. The new upgrade value is displayed and the cost for the next upgrade is also displayed.
3.2	(5) The user can click a tower they want to place	1. Run index.html with the live server Visual Studio Code	The current tower selected should be highlighted in green. When the cancel

	from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.	extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press the “Soldier” button 5. Press the “Machine Gunner” button 6. Press the “Cancel” button 7. Press the “Soldier Button” 8. Place the tower	button is pressed or the tower is placed, the highlight turns off.
3.3	(3) There is an arcade game mode. The user selects one of the maps and plays endlessly with increasing difficulty over time until they lose their base. This gamemode is aimed at casual players.	1. Set the value of player score to 0 2. Run index.html with the live server Visual Studio Code extension 3. Press the “Arcade” button 4. Press the “Stalingrad” button 5. Watch the enemies spawn 6. Set the value of player score to 10000000 7. Run index.html with the live server Visual Studio Code extension 8. Press the “Arcade” button 9. Press the “Stalingrad” button 10. Watch the enemies spawn	When the player score is low (i.e. 0) the enemies spawn slowly but when it is higher (i.e. 1000000) they spawn faster. This increases the difficulty of the arcade gamemode over time
3.4	(2) There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Campaign” button 3. Press the “Stalingrad” button 4. Press the “Level One” button 5. Watch the enemies spawn	Only a few enemies of each type should spawn.

3.5	(10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Gulf War” button 4. Press the “Call airstrike” button 5. Click somewhere with enemies	Once clicked, the button should turn green to show that it has been pressed. The enemies within a 2 tile radius of the mouse should be destroyed and an explosion effect should appear here, before shortly disappearing. The “Call Airstrike” button should turn light grey and be unable to be used again
3.6	(10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Stalingrad” button 4. Press the “Call Special Forces” button 5. Place as you would with a regular tower	Once clicked, the button should turn green to show that it has been pressed. 5 smaller towers should appear on a single tile. The “Call Special Forces” button should turn light grey and be unable to be used again. After 15 seconds, the special forces disappear.
3.7	(10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.	1. Run index.html with the live server Visual Studio Code extension 2. Press the “Arcade” button 3. Press the “Vietnam” button 4. Press the “Place trap” button 5. Place somewhere on the enemy path.	Once clicked, the button should turn green to show that it has been pressed.A trap should appear on the screen which the enemies are stuck in..The “Place Trap” button should turn light grey and be unable to be used again. After 5 seconds, the trap disappears and the enemies move again.

Iterative Development:

D8 - Iterative Development

Iteration 3 - Creating the buttons and variables for the skill tree

The first thing I did is create the buttons that the user clicks to upgrade when using the skill tree. This was simple, since I've done it multiple times by now and once I'd got the layout of all the buttons, I created the variables actually storing the upgrade tier. I displayed the current upgrade percent next to the button, as well as the cost, in battle medals, to upgrade it. Next, I made the function which runs when the buttons are clicked, which after some slight issues as explained in D17 - Non trivial failed tests and remedies:, I finalised as this:

```
// Edits the variable depending on the upgrade passed.
function skillTreeButtonsFunction(upgrade, type) {
    if(upgrade < 5 && playerBattleMedals >= upgrade + 1) {
        playerBattleMedals -= upgrade + 1;
        switch(type) {
            case 0:
                projectileUpgrade++;
                break;

            case 1:
                damageUpgrade++;
                break;

            case 2:
                baseHealthUpgrade++;
                break;

            case 3:
                enemySpeedUgrade++;
                break;

            case 4:
                enemyArmourUpgrade++;
                break;

            case 5:
                specialAbilityUpgrade++;
                break;
        }
    }
}
```

Finally, I need to apply the upgrades to make them actually do something (except for the special ability upgrade which I'll add later since it's not in the game yet). I made a function which converts the upgrade variable (since this is a tier between 0-5 inclusive not the actual value applied to the upgrade) to the decimal applied:

```
// Returns the percentage upgrade depending on the tier of the upgrade
variable passed.
function upgradePercent(upgrade) {
    switch(upgrade) {
        case 0:
            return 0;
            break;

        case 1:
            return 0.02;
            break;

        case 2:
            return 0.05;
            break;

        case 3:
            return 0.08;
            break;

        case 4:
            return 0.12;
            break;

        case 5:
            return 0.18;
            break;
    }
}
```

I also made a procedure to change the base health depending on the given input (and applying the upgrade):

```
// Resets the base health relative to the upgraded value
function setBaseHealth(health) {
  baseHealth = health * (1 + upgradePercent(baseHealthUpgrade));
  baseHealthMax = health * (1 + upgradePercent(baseHealthUpgrade));
}
```

The other upgrades were easy to apply since I just added (or subtracted) the decimal value returned from the *upgradePercent* function.

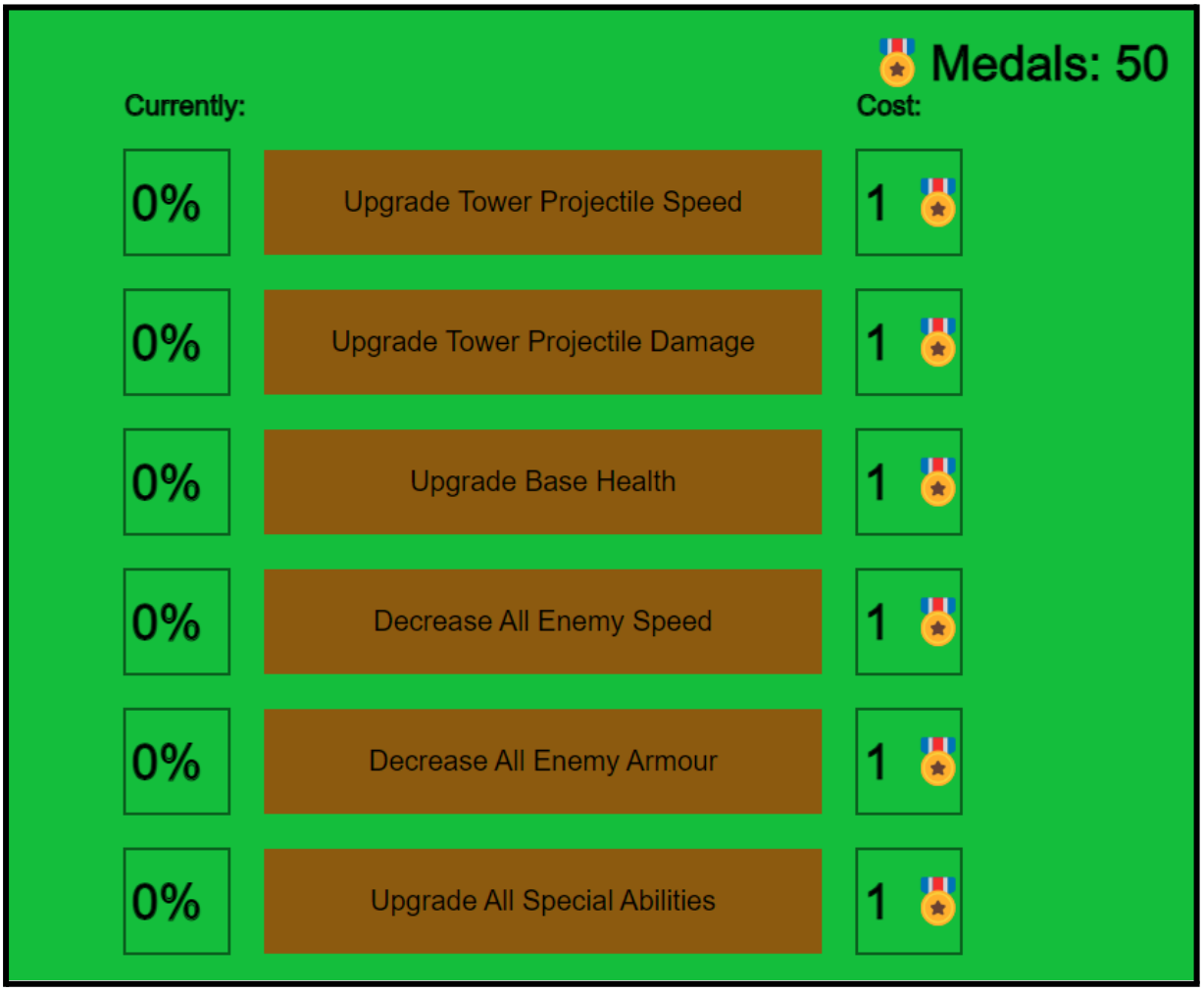


Figure 23: A screenshot of the skill tree.

Iteration 3 - Highlighting button of the tower selected for placing

Next, at the stakeholders request, I added a feature which highlights the button clicked when placing a tower. This was once again relatively simple since there is a p5.js method for changing the CSS feature of an element. I made a procedure which takes two arguments: whether or not we are changing the button colour to green or back to the default aqua (boolean) and the button we want to change.

```
// Changes the colour of the button the user is currently clicking
function changeTowerColour(change, tower) {
    if(change) {
        switch (tower) {
            case 0:
                soldierButton.style("background-color", "green");
                break;

            case 1:
```

```
                machineGunnerButton.style("background-color", "green");
                break;

            case 2:
                rpgButton.style("background-color", "green");
                break;

            case 3:
                ifvButton.style("background-color", "green");
                break;

            case 4:
                tankButton.style("background-color", "green");
                break;

            case 5:
                shoulderAntiAirButton.style("background-color", "green");
                break;

            case 6:
                surfaceAirButton.style("background-color", "green");
                break;

            case 7:
                advancedSurfaceAirButton.style("background-color", "green");
                break;
        }
    } else {
        soldierButton.style("background-color", "#3fcaca");
        machineGunnerButton.style("background-color", "#3fcaca");
        rpgButton.style("background-color", "#3fcaca");
        ifvButton.style("background-color", "#3fcaca");
        tankButton.style("background-color", "#3fcaca");
        shoulderAntiAirButton.style("background-color", "#3fcaca");
        surfaceAirButton.style("background-color", "#3fcaca");
        advancedSurfaceAirButton.style("background-color", "#3fcaca");
    }
}
```

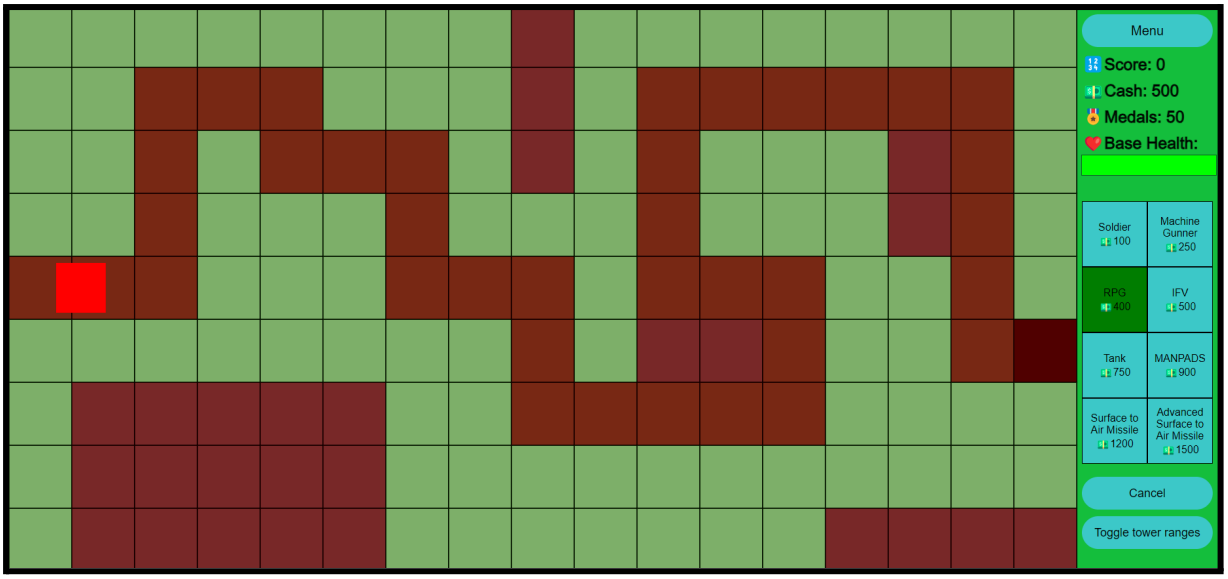


Figure 24: A screenshot of the button now highlighted

Iteration 3 - Finalising the arcade gamemode

Third, I finalised the arcade game mode which was very simple since this is what I've been developing this whole time (added to the simple nature of the game mode, it's endless). The only thing I wanted to change from what the gamemode already is, is to increase the enemy spawn rate over time. After making a function (which I'll also use for the campaign gamemode), I made a series of simple if-statements and for loops to edit the spawn rates. The final thing I added was 3 variables which check whether the score was above the threshold for the previous frame and therefore the enemy spawn rates have already been changed. Otherwise, the spawn rates are changed (to the same value) every frame which wastes memory.

```
/ Sets up the enemies depending on the input (for each gamemode)
function enemySetup(gamemode) {
    switch(gamemode) {
        case 0:
            // Campaign gamemode
            break;

        case 1:
            if(playerScore > 1000000 && arcade3 == false) {
                arcade3 = true;
                enemies[0].spawnRate = 2000;
                enemies[1].spawnRate = 5000;
                enemies[2].spawnRate = 8000;
                enemies[3].spawnRate = 8000;
                enemies[4].spawnRate = 10000;
                enemies[5].spawnRate = 15000;
                enemies[6].spawnRate = 20000;
```

```
                enemies[7].spawnRate = 25000;
                enemies[8].spawnRate = 25000;
                enemies[9].spawnRate = 30000;
            } else if (playerScore > 500000 && arcade2 == false) {
                enemies[0].spawnRate = 5000;
                enemies[1].spawnRate = 8000;
                enemies[2].spawnRate = 8000;
                enemies[3].spawnRate = 10000;
                enemies[4].spawnRate = 15000;
                enemies[5].spawnRate = 20000;
                enemies[6].spawnRate = 30000;
                enemies[7].spawnRate = 40000;
                enemies[8].spawnRate = 50000;
                enemies[9].spawnRate = 50000;
            } else if (playerScore > 100000 && arcade1 == false)  {
                enemies[0].spawnRate = 5000;
                enemies[1].spawnRate = 5000;
                enemies[2].spawnRate = 8000;
                enemies[3].spawnRate = 10000;
                enemies[4].spawnRate = 15000;
                enemies[5].spawnRate = 20000;
                enemies[6].spawnRate = 30000;
                enemies[7].spawnRate = 40000;
                enemies[8].spawnRate = 50000
                enemies[9].spawnRate = 60000;
            }
            break;
    }
}
```

Iteration 3 - Making the campaign gamemode

Firstly, I considered the ways I needed to change what I already had to make a campaign style gamemode. I needed to create buttons to load each level so I started with that. After this, I needed a way to display the users current attempt at a level as a rating out of 3 so I started by making a 2d array (one dimension is the map in question, one is the level) but I quickly realised that it would be smarter to just include this as part of the level class (which I was already planning on making to control the enemies spawning). After displaying the rating under each level, I moved onto controlling the number of enemies for each level.

```
// Displays the users previous best attempts at each level.
function displayStarRatings() {
    for (let i = 0; i<3; i++) {
        let rating = levels[chosenMapInt][i].rating;
        if (rating == 0) {
```

```
            text("⚫", 145 + (200 * i), 240)
            text("⚫", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 1) {
            text("🟡", 145 + (200 * i), 240)
            text("⚫", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 2) {
            text("🟡", 145 + (200 * i), 240)
            text("🟡", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 3) {
            text("🟡", 145 + (200 * i), 240)
            text("🟡", 195 + (200 * i), 240)
            text("🟡", 245 + (200 * i), 240)
        }
    }
}
```

I created a class called level which contains the number the base health should be and how many of each enemy type should spawn. Once the user selects a level, the array containing the enemies has its lengths set to the number of each enemy type. It is then filled with a given number of the enemy type.

```
// Sets the length of the enemy arrays so only that many spawn
    setEnemies() {
        enemies[0].queue.length = this.soldier;
        enemies[1].queue.length = this.specialForces;
        enemies[2].queue.length = this.truck;
        enemies[3].queue.length = this.apc;
        enemies[4].queue.length = this.ifv;
        enemies[5].queue.length = this.tank;
        enemies[6].queue.length = this.helicopter;
        enemies[7].queue.length = this.fighterJet;
        enemies[8].queue.length = this.bomber;
        enemies[9].queue.length = this.stealthBomber;

        for(let i = 0; i < 10; i++) {
            for(let j = 0; j < enemies[i].queue.length; j++) {
                enemies[i].queue[j] = this.returnEnemyType(i);
            }
        }
    }
```

```
    // Returns the enemy type to fill the queue
    returnEnemyType(type) {
        switch (type) {
            case 0:
                return new enemyBasic(200, 0.12, chosenMap, "red", this.tail,
5); // Soldier.
                break;

            case 1:
                return new enemyBasic(300, 0.13, chosenMap, "yellow",
this.tail, 5); // Special forces.
                break;

            case 2:
                return new enemyVehicle(400, 0.14, chosenMap, "gray",
this.tail, 2, 100); // Truck.
                break;

            case 3:
                return new enemyVehicle(500, 0.15, chosenMap, "blue",
this.tail, 2, 125); // APC.
                break;

            case 4:
                return new enemyVehicle(300, 0.16, chosenMap, "green",
this.tail, 2, 150); // IFV.
                break;

            case 5:
                return new enemyVehicle(350, 0.17, chosenMap, "orange",
this.tail, 1, 200); // Tank.
                break;

            case 6:
                return new enemyPlane(400, 0.18, chosenMap, "purple",
this.tail, 1, 400, false); // Helicopter.
                break;

            case 7:
                return new enemyPlane(450, 0.19, chosenMap, "lime", this.tail,
1, 500, false); // Fighter jet.
```

```
                break;

            case 8:
                return new enemyPlane(450, 0.2, chosenMap, "white", this.tail,
1, 600, false); // Bomber.
                break;

            case 9:
                return new enemyPlane(500, 0.2, chosenMap, "black", this.tail,
1, 750, true); // Stealth bomber.
                break;
        }
    }
```

Now that the enemy array contains only the enemies for that level and no more, we can work on slowly spawning them. Looping through the array, validation is used to check that if the spawn rate has been met and that the tail is still less than the number of enemies there should be (found by returning said value from the instance of the levels class for that level), the tail is incremented (meaning the enemy will appear since the update function loops from head to tail).

```
// Updates the enemies for the campaign gamemode
function enemyUpdateCampaign() {
    for (let i = 0; i < 10; i++) {
        for (let j = 0; j < enemies[i].queue.length; j++) {
            enemies[i].timeElapsed += deltaTime;
            if(enemies[i].timeElapsed > enemies[i].spawnRate &&
enemies[i].tail < levels[chosenMapInt][chosenLevel].returnNoOfEnemies(i)) {
                enemies[i].tail++;
                enemies[i].timeElapsed = 0;
            }
        }
    }
    enemyUpdate();
}
```

This spawns the required number of enemies but as soon as a tower targets an enemy, the game crashes.

Iteration 3 - Making the trap special ability

Next, I need to create all the special abilities. Starting by creating a universal button to place the special ability (since the function can be altered depending on the map so the same button can be used). After setting these up in the usual way, I started on the trap itself. I split the process into three functions: one for placing the trap, one for making the trap affect the enemies and one for drawing it. Placing the trap works in the same way as placing a tower

and the same validations are used. If the user has pressed the button to place the ability, hasn't already used it and clicks on a valid tile (this time a path tile in comparison to a normal tile when placing a tower) then the trap is placed at the tile they click, the variable to control the trap being displayed is changed to true and the timer till the trap disappears start.

```
// Places the trap.
function showTrap() {
   if (placeTrap && allowAbility && mouseIsPressed && mouseX >= MAP_TILE_X[0]
&& mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <=
MAP_TILE_Y[9]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (chosenMap[tileX][tileY] == 1) {
            TRAP.placed = true;
            TRAP.posX = Math.floor(mouseX / CELL_SIZE);
            TRAP.posY = Math.floor(mouseY / CELL_SIZE);
            trapFrameCount = frameCount;
            changeTowerColour(true, 9);
            placeTrap = false;
        }
    }

}
```

Making the trap affect the enemies is similar to making the projectiles damage the enemy. It loops through the array containing the enemies and checks if any are on the same tile as the trap. If so, the given enemy's speed is set to zero (to give the effect of falling into the trap). This function also ensures the trap is only used for 5 seconds by calculating 300 frames since the trap was placed and calls the function to draw the trap.

```
// Uses the trap.
function trapFunction() {
    if (TRAP.placed && allowAbility) {
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);
                if(enemyX - TRAP.posX == 0 && enemyY - TRAP.posY == 0) {
                    enemies[i].queue[j].speed = 0;
                }
            }
        }
    }
```

```
    if (TRAP.placed && allowAbility && frameCount - trapFrameCount > 300) {
        TRAP.placed = false;
        allowAbility = false;
    }

    if(TRAP.placed) {
        drawTrap();
    }
}
```

Drawing the trap is just a combination of circles based off the tile the trap is placed.

```
// Displays the trap on screen
function drawTrap() {
    for(let i = 0; i < 88; i+=22) {
        for(let j = 0; j < 88; j+=22) {
            fill("gray");
            circle(TRAP.posX*CELL_SIZE + 15 + i, TRAP.posY*CELL_SIZE + 15 + j,
20);
            fill(3, 38, 11);
            circle(TRAP.posX*CELL_SIZE + 15 + i, TRAP.posY*CELL_SIZE + 15 + j,
10);
        }
    }
}
```

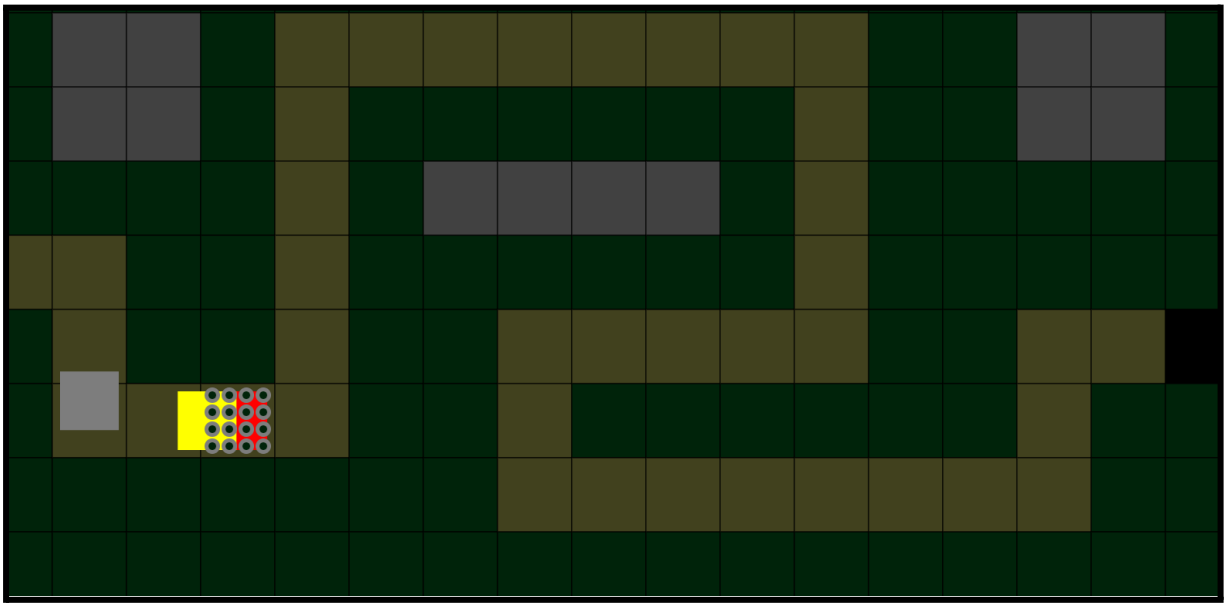


Figure 25: An enemy stuck in the trap

Iteration 3 - Making the special forces special ability

In the Stalingrad map, the special ability the user can place is some special forces towers which are there for a limited time and quickly destroy enemies. Creating the special forces is slightly easier than creating the trap since they are just towers (which a class already exists for) so it is split into two functions - one for placing the special forces and one for updating them frame by frame. After validating that the tile the user wants to place the special forces on is valid, an array containing the special forces is filled with five (smaller) towers on that one tile. At the same time, a timer for fifteen seconds starts to ensure the special forces disappear after this time.

```
// Places the special forces.
function showSpecialForces() {
    if (towerPlace && allowAbility && mouseIsPressed && mouseX >=
MAP_TILE_X[0] && mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY
<= MAP_TILE_Y[9]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (mapFilled[tileX][tileY] == 0) {
            specialForces[0] = new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "black", 8, 240, 2, 0.2);
            specialForces[1] = new towerBasic(MAP_TILE_X[tileX] + 20,
MAP_TILE_Y[tileY] + 20, "black", 8, 240, 2, 0.2);
            specialForces[2] = new towerBasic(MAP_TILE_X[tileX] + 20,
MAP_TILE_Y[tileY] - 20, "black", 8, 240, 2, 0.2);
            specialForces[3] = new towerBasic(MAP_TILE_X[tileX] - 20,
MAP_TILE_Y[tileY] + 20, "black", 8, 240, 2, 0.2);
            specialForces[4] = new towerBasic(MAP_TILE_X[tileX] - 20,
MAP_TILE_Y[tileY] - 20, "black", 8, 240, 2, 0.2);
            allowAbility = false;
            changeTowerColour(true, 9);
            towerPlace = false;
            mapFilled[tileX][tileY] = 1;
            specialForcesFrameCount = frameCount;
        }
    }

    if (specialForces.length > 0 && frameCount - specialForcesFrameCount >
900) {
        let x = specialForces[0].posX / CELL_SIZE;
        let y = specialForces[0].posY / CELL_SIZE;
        mapFilled[x][y] = 0;
        specialForces.length = 0;
    }
}
```

Secondly, we need to loop through the array and update the special forces frame by frame so a second function is used for this.

```
// Displays the special forces once placed.
function specialForcesUpdate() {
    for (let tower of specialForces) {
        tower.update();
    }
}
```

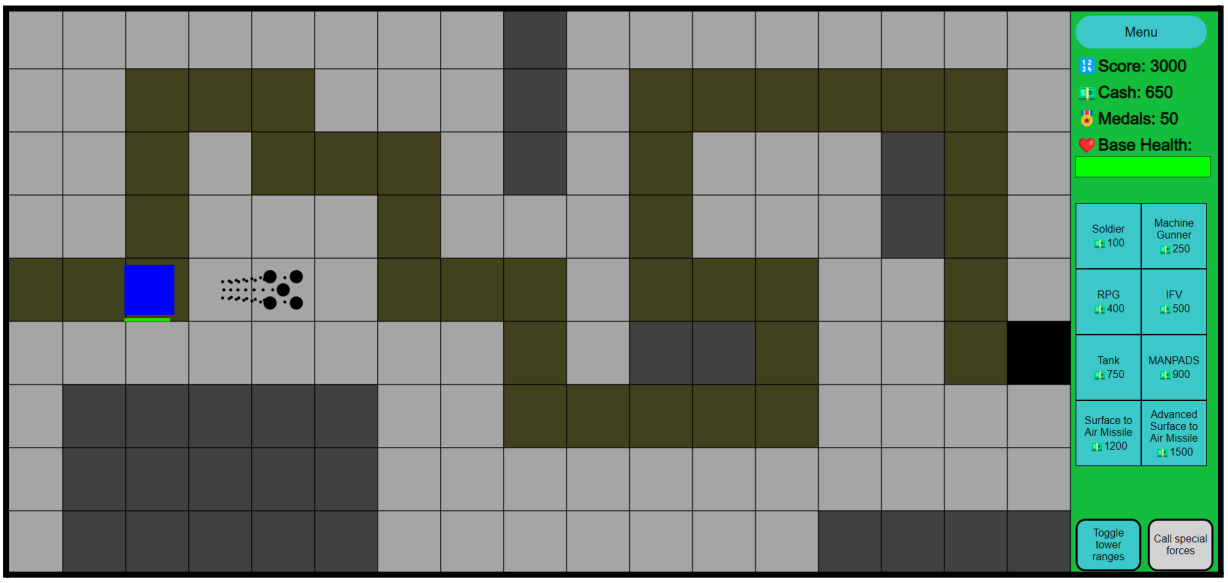


Figure 26: The special forces placed and destroying an enemy

Iteration 3 - Making the airstrike special ability

Finally, on the Gulf War map, the special ability that needs to be made is an airstrike. This will completely wipe out any enemies in a certain radius. This is again quite simple and split into only two functions, one for placing the airstrike and one for displaying it on the screen. When placing the airstrike, first it is validated that the user is on the game screen (and not on the side menu for example) and once they click, the coordinates of their mouse are used as the airstrike coordinates. First the enemy array is looped through to check if any enemies are in a 2 tile radius of the airstrike (if so, their health is set to zero and they disappear).

```
// The special ability for the Gulf War map.
function gulfWarSpecialAbility(){
    if (allowAbility && mouseIsPressed && mouseX >= MAP_TILE_X[0] && mouseX <=
MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <= MAP_TILE_Y[9] ) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);

        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);
```

```
                if(Math.abs(enemyX - tileX) < 2 && Math.abs(enemyY - tileY) <
2) {
                    airStrikeFrameCount = frameCount;
                    airstrikeCheck = true;
                    enemies[i].queue[j].health -= 600;
                    allowAbility = false;
                    changeTowerColour(true, 9);
                }
            }
        }
    }
}
```

The second function displays the airstrike and is a series of red/orange/yellow circles based on where the user clicked. The first thing that is checked is whether or not the coordinates where the user clicked are saved. When I initially made this function, I based the circle positions off the *mouseX* and *mouseY* variables from the *p5.js* library but quickly realised if the user moved their mouse while the animation was displayed, the animation would move with it. So after the coordinates where the user initially clicked are displayed, the airstrike animation is shown for just over 3 seconds before disappearing.

```
// Displays an explosion on the screen.
function showAirstrike() {
    if(frameCount - airStrikeFrameCount < 100 && airstrikeCheck &&
!(airstrikeCheckCoordCheck)) {
        airstrikeX = mouseX;
        airstrikeY = mouseY;
        airstrikeCheckCoordCheck = true;
        fill("orange");
        circle(airstrikeX + 10, airstrikeY + 10, 60);
        circle(airstrikeX - 10, airstrikeY - 10, 80);
        circle(airstrikeX + 30, airstrikeY + 30, 40);
        circle(airstrikeX, airstrikeY, 50);
        fill("red");
        circle(airstrikeX + 15, airstrikeY + 10, 30);
        circle(airstrikeX - 10, airstrikeY - 10, 60);
        circle(airstrikeX - 65, airstrikeY + 25, 30);
        circle(airstrikeX - 45, airstrikeY + 45, 30);
        circle(airstrikeX, airstrikeY, 30);
        fill("yellow");
        circle(airstrikeX + 5, airstrikeY + 20, 50);
        circle(airstrikeX + 20, airstrikeY - 40, 40);
        circle(airstrikeX - 45, airstrikeY - 35, 25);
```

```
        circle(airstrikeX - 20, airstrikeY + 30, 10);
    } else if(frameCount - airStrikeFrameCount < 100 && airstrikeCheck) {
        fill("orange");
        circle(airstrikeX + 10, airstrikeY + 10, 60);
        circle(airstrikeX - 10, airstrikeY - 10, 80);
        circle(airstrikeX + 30, airstrikeY + 30, 40);
        circle(airstrikeX, airstrikeY, 50);
        fill("red");
        circle(airstrikeX + 15, airstrikeY + 10, 30);
        circle(airstrikeX - 10, airstrikeY - 10, 60);
        circle(airstrikeX - 65, airstrikeY + 25, 30);
        circle(airstrikeX - 45, airstrikeY + 45, 30);
        circle(airstrikeX, airstrikeY, 30);
        fill("yellow");
        circle(airstrikeX + 5, airstrikeY + 20, 50);
        circle(airstrikeX + 20, airstrikeY - 40, 40);
        circle(airstrikeX - 45, airstrikeY - 35, 25);
        circle(airstrikeX - 20, airstrikeY + 30, 10);
    } else {
        airstrikeCheck = false;
        airStrikeFrameCount = 0;
    }
}
```

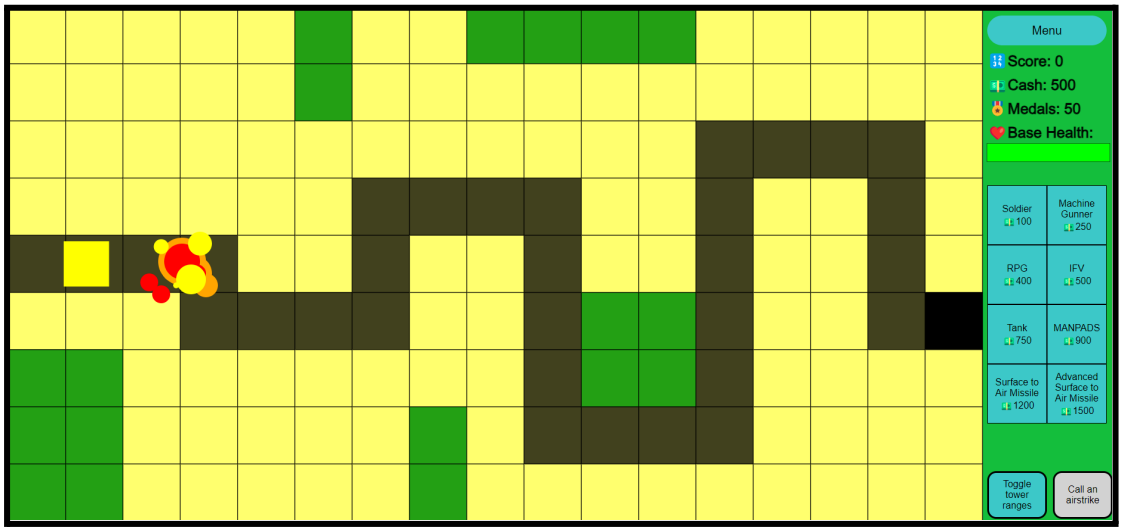
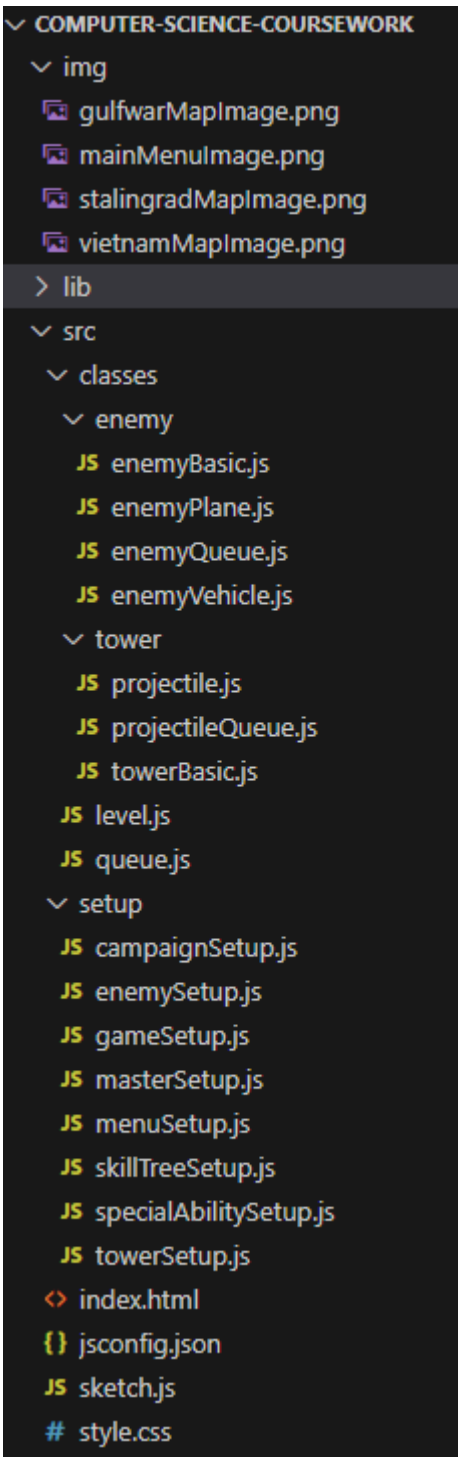


Figure 27: The airstrike animation on the screen

D9 - Structure and modularity:

The screenshot below is of the directory containing my code so far:



My code is still very modular and object-oriented in structure, as shown below:

```
switch(gameState) {

    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      mapButtonsDisplay(false);
      returnToMenuButtonDisplay(false);
```

```
    placeCancelButtonDisplay(false);
    displayRangeButtonDisplay(false);
    towerButtonDisplay(false);
    skillTreeButtonsDisplay(false);
    campaignButtonsDisplay(false);
    specialAbilityButtonDisplay(false);
    break;

  // Loads the campaign game mode.
  case 2:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
    } else if (chosenLevel == -1) {
      mapButtonsDisplay(false);
      campaignButtonsDisplay(true);
      returnToMenuButtonDisplay(true);
      setBaseHealth(1000);
    } else{
      specialAbilityButton.html(returnSpecialAbilityName());
      returnSpecialAbilityName();
      campaignButtonsDisplay(false);
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      specialAbilityButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      checkBaseHealth();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemyUpdate();
      towerUpdate();
      specialForcesUpdate();
      enemySetup(0);
      enemyUpdateCampaign();
      showSpecialForces();
      showAirstrike();
      showTrap();
```

```
      trapFunction();
    }
    break;

  // Loads the arcade game mode.
  case 3:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      setBaseHealth(1000);
    } else{
      specialAbilityButton.html(returnSpecialAbilityName());
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      specialAbilityButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      checkBaseHealth();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemySpawn();
      enemyUpdate();
      towerUpdate();
      specialForcesUpdate();
      enemySetup(1);
      showSpecialForces();
      showAirstrike();
      showTrap();
      trapFunction();
    }
    break;

  // Loads the tutorial.
  case 4:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);

    break;
```

```
  // Loads the skill tree.
  case 5:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);
    skillTreeButtonsDisplay(true);

    break;

  // Loads the settings.
  case 6:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);

    break;

  default:
    gameState = 1;
}
```

Below are all the classes made/edited in this iteration:

level
+ rating + baseHealth + soldier + specialForces + truck + apc + ifv + tank + helicopter + fighterJet + bomber + stealthBomber
+ constructor(baseHealth, soldier, specialForces, truck, apc, ifv, tank, helicopter, fighterJet, bomber, stealthBomber):void + setEnemies():void + returnEnemyType(type):Object of type enemyBasic + returnNoOfEnemies(enemy):integer

D10 - Code annotation and comments:

sketch.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
14/06/23. This is the main file which combines code from other files to run
the program.

"use strict";

// Preload function from the p5.js library. Contains images which need to be
loaded before the main menu appears.
function preload() {
  mainMenuImage = loadImage("./../img/mainMenuImage.png");
  vietnamMapImage = loadImage("./../img/vietnamMapImage.png");
  stalingradMapImage = loadImage("./../img/stalingradMapImage.png");
  gulfwarMapImage = loadImage("./../img/gulfwarMapImage.png");
}

// Setup function from the p5.js library. This function is run once at the
start of the program and never again, so it is used for creating the canvas,
etc.
function setup() {
  createCanvas(1856, 864);
  menuButtonsSetup();
  mapButtonsSetup();
  towerButtonsSetup();
  skillTreeButtonsSetup();
  campaignButtonsSetup();
  specialAbilityButtonSetup();
}

// Draw function from the p5.js library. This function runs 60 times per
second and is used for animation (i.e. updating object positions, drawing map,
etc.).
function draw() {
  background(21, 191, 61);
  switch(gameState) {

    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
```

```
    mapButtonsDisplay(false);
    returnToMenuButtonDisplay(false);
    placeCancelButtonDisplay(false);
    displayRangeButtonDisplay(false);
    towerButtonDisplay(false);
    skillTreeButtonsDisplay(false);
    campaignButtonsDisplay(false);
    specialAbilityButtonDisplay(false);
    break;

  // Loads the campaign game mode.
  case 2:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
    } else if (chosenLevel == -1) {
      mapButtonsDisplay(false);
      campaignButtonsDisplay(true);
      returnToMenuButtonDisplay(true);
      setBaseHealth(1000);
    } else{
      specialAbilityButton.html(returnSpecialAbilityName());
      returnSpecialAbilityName();
      campaignButtonsDisplay(false);
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      specialAbilityButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      checkBaseHealth();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemyUpdate();
      towerUpdate();
      specialForcesUpdate();
      enemySetup(0);
      enemyUpdateCampaign();
      showSpecialForces();
```

```
      showAirstrike();
      showTrap();
      trapFunction();
    }
    break;

  // Loads the arcade game mode.
  case 3:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      setBaseHealth(1000);
    } else{
      specialAbilityButton.html(returnSpecialAbilityName());
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      specialAbilityButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      checkBaseHealth();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemySpawn();
      enemyUpdate();
      towerUpdate();
      specialForcesUpdate();
      enemySetup(1);
      showSpecialForces();
      showAirstrike();
      showTrap();
      trapFunction();
    }
    break;

  // Loads the tutorial.
  case 4:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);
```

```
      break;

    // Loads the skill tree.
    case 5:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      skillTreeButtonsDisplay(true);

      break;

    // Loads the settings.
    case 6:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    default:
      gameState = 1;
  }
}
```

specialAbilitySetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
17/11/23. This file contains the setup information for the special abilities.

"use strict";

let specialAbilityButton;

let allowAbility = true;

let placeTrap = false;
const TRAP = {
    posX : 0,
    posY : 0,
    placed : false
};
let trapFrameCount = 0;

let specialForces = [];
```

```
let specialForcesFrameCount = 0;

let airStrikeFrameCount = 0;
let airstrikeCheck = false;
let airstrikeX = 0, airstrikeY = 0;
let airstrikeCheckCoordCheck = false;

// Sets up the button to call the special ability.
function specialAbilityButtonSetup() {
    specialAbilityButton = createButton(" ");
    specialAbilityButton.position(1750, 780);
    specialAbilityButton.class("towerRangeSpecialAbility");
    specialAbilityButton.mousePressed(() =>
specialAbilityButtonFunction(chosenMapInt));
}

// Shows or hides the special ability buttons depending on the input.
function specialAbilityButtonDisplay(display) {
    if (display) {
        specialAbilityButton.show();
    } else {
        specialAbilityButton.hide();
    }
}

// Function called when the special ability button is clicked.
function specialAbilityButtonFunction(map) {
    if(map == 0 && allowAbility) {
        changeTowerColour(true, 8);
        vietnamSpecialAbility();
    } else if (map == 1 && allowAbility) {
        changeTowerColour(true, 8);
        stalingradSpecialAbility();
    } else if (map == 2 && allowAbility) {
        changeTowerColour(true, 8);
        gulfWarSpecialAbility();
    }
}

// The special ability for the Vietnam Map.
function vietnamSpecialAbility() {
    placeTrap = true;
```

```
}

// Places the trap.
function showTrap() {
   if (placeTrap && allowAbility && mouseIsPressed && mouseX >= MAP_TILE_X[0]
&& mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <=
MAP_TILE_Y[9]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (chosenMap[tileX][tileY] == 1) {
            TRAP.placed = true;
            TRAP.posX = Math.floor(mouseX / CELL_SIZE);
            TRAP.posY = Math.floor(mouseY / CELL_SIZE);
            trapFrameCount = frameCount;
            changeTowerColour(true, 9);
            placeTrap = false;
        }
    }

    if(TRAP.placed) {
        drawTrap();
    }
}

// Uses the trap.
function trapFunction() {
    if (TRAP.placed && allowAbility) {
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);
                if(enemyX - TRAP.posX == 0 && enemyY - TRAP.posY == 0) {
                    enemies[i].queue[j].speed = 0;
                }
            }
        }
    }

    if (TRAP.placed && allowAbility && frameCount - trapFrameCount > 300) {
        TRAP.placed = false;
        allowAbility = false;
    }
```

```
}

// Displays the trap on screen
function drawTrap() {
    for(let i = 0; i < 88; i+=22) {
        for(let j = 0; j < 88; j+=22) {
            fill("gray");
            circle(TRAP.posX*CELL_SIZE + 15 + i, TRAP.posY*CELL_SIZE + 15 + j,
20);
            fill(3, 38, 11);
            circle(TRAP.posX*CELL_SIZE + 15 + i, TRAP.posY*CELL_SIZE + 15 + j,
10);
        }
    }
}

// The special ability for the Stalingrad map.
function stalingradSpecialAbility() {
    towerPlace = true;
}

// Places the special forces.
function showSpecialForces() {
    if (towerPlace && allowAbility && mouseIsPressed && mouseX >=
MAP_TILE_X[0] && mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY
<= MAP_TILE_Y[9]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (mapFilled[tileX][tileY] == 0) {
            specialForces[0] = new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "black", 8, 240, 2, 0.2);
            specialForces[1] = new towerBasic(MAP_TILE_X[tileX] + 20,
MAP_TILE_Y[tileY] + 20, "black", 8, 240, 2, 0.2);
            specialForces[2] = new towerBasic(MAP_TILE_X[tileX] + 20,
MAP_TILE_Y[tileY] - 20, "black", 8, 240, 2, 0.2);
            specialForces[3] = new towerBasic(MAP_TILE_X[tileX] - 20,
MAP_TILE_Y[tileY] + 20, "black", 8, 240, 2, 0.2);
            specialForces[4] = new towerBasic(MAP_TILE_X[tileX] - 20,
MAP_TILE_Y[tileY] - 20, "black", 8, 240, 2, 0.2);
            allowAbility = false;
            changeTowerColour(true, 9);
            towerPlace = false;
```

```
            mapFilled[tileX][tileY] = 1;
            specialForcesFrameCount = frameCount;
        }
    }

    if (specialForces.length > 0 && frameCount - specialForcesFrameCount >
900) {
        let x = specialForces[0].posX / CELL_SIZE;
        let y = specialForces[0].posY / CELL_SIZE;
        mapFilled[x][y] = 0;
        specialForces.length = 0;
    }
}

// Displays the special forces once placed.
function specialForcesUpdate() {
    for (let tower of specialForces) {
        tower.update();
    }
}

// The special ability for the Gulf War map.
function gulfWarSpecialAbility(){
    if (allowAbility && mouseIsPressed && mouseX >= MAP_TILE_X[0] && mouseX <=
MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <= MAP_TILE_Y[9] ) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);

        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);

                if(Math.abs(enemyX - tileX) < 2 && Math.abs(enemyY - tileY) <
2) {
                    airStrikeFrameCount = frameCount;
                    airstrikeCheck = true;
                    enemies[i].queue[j].health -= 600;
                    allowAbility = false;
                    changeTowerColour(true, 9);
                }
            }
```

```
        }
    }
}

// Displays an explosion on the screen.
function showAirstrike() {
    if(frameCount - airStrikeFrameCount < 100 && airstrikeCheck &&
!(airstrikeCheckCoordCheck)) {
        airstrikeX = mouseX;
        airstrikeY = mouseY;
        airstrikeCheckCoordCheck = true;
        fill("orange");
        circle(airstrikeX + 10, airstrikeY + 10, 60);
        circle(airstrikeX - 10, airstrikeY - 10, 80);
        circle(airstrikeX + 30, airstrikeY + 30, 40);
        circle(airstrikeX, airstrikeY, 50);
        fill("red");
        circle(airstrikeX + 15, airstrikeY + 10, 30);
        circle(airstrikeX - 10, airstrikeY - 10, 60);
        circle(airstrikeX - 65, airstrikeY + 25, 30);
        circle(airstrikeX - 45, airstrikeY + 45, 30);
        circle(airstrikeX, airstrikeY, 30);
        fill("yellow");
        circle(airstrikeX + 5, airstrikeY + 20, 50);
        circle(airstrikeX + 20, airstrikeY - 40, 40);
        circle(airstrikeX - 45, airstrikeY - 35, 25);
        circle(airstrikeX - 20, airstrikeY + 30, 10);
    } else if(frameCount - airStrikeFrameCount < 100 && airstrikeCheck) {
        fill("orange");
        circle(airstrikeX + 10, airstrikeY + 10, 60);
        circle(airstrikeX - 10, airstrikeY - 10, 80);
        circle(airstrikeX + 30, airstrikeY + 30, 40);
        circle(airstrikeX, airstrikeY, 50);
        fill("red");
        circle(airstrikeX + 15, airstrikeY + 10, 30);
        circle(airstrikeX - 10, airstrikeY - 10, 60);
        circle(airstrikeX - 65, airstrikeY + 25, 30);
        circle(airstrikeX - 45, airstrikeY + 45, 30);
        circle(airstrikeX, airstrikeY, 30);
        fill("yellow");
        circle(airstrikeX + 5, airstrikeY + 20, 50);
        circle(airstrikeX + 20, airstrikeY - 40, 40);
```

```
        circle(airstrikeX - 45, airstrikeY - 35, 25);
        circle(airstrikeX - 20, airstrikeY + 30, 10);
    } else {
        airstrikeCheck = false;
        airStrikeFrameCount = 0;
    }
}

// Changes the text displayed on the special ability button depending on the
map.
function returnSpecialAbilityName() {
    switch(chosenMapInt) {
        case 0:
            return "Place trap";
            break;

        case 1:
            return "Call special forces";
            break;

        case 2:
            return "Call an airstrike";
            break;
    }
}
```

campaignSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
10/11/23. This contains the setup for the campaign gamemode.

"use strict";

let chosenLevel = -1;

let campaignCheck = false;

let levelOneButton, levelTwoButton, levelThreeButton;

let levels = [[0, 0, 0], [0, 0, 0], [0, 0, 0]];
levels[0][0] = new level(800, 5, 3, 2, 1, 0, 0, 0, 0, 0, 0);
levels[0][1] = new level(900, 8, 6, 4, 3, 3, 1, 0, 0, 0, 0);
levels[0][2] = new level(1000, 12, 8, 8, 6, 4, 2, 1, 0, 0, 0);
```

```
levels[1][0] = new level(800, 5, 3, 2, 1, 0, 0, 0, 0, 0, 0);
levels[1][1] = new level(900, 12, 8, 4, 3, 3, 1, 0, 0, 0, 0);
levels[1][2] = new level(1000, 15, 10, 5, 5, 3, 2, 1, 0, 0, 0);
levels[2][0] = new level(800, 5, 3, 2, 1, 0, 0, 0, 0, 0, 0);
levels[2][1] = new level(900, 7, 8, 10, 5, 2, 1, 0, 0, 0, 0);
levels[2][2] = new level(1000, 8, 10, 12, 8, 3, 1, 2, 0, 0, 0);

// Declares all the buttons seen on the skill tree screen.
function campaignButtonsSetup() {
    levelOneButton = createButton("Level One");
    levelOneButton.position(140, 125);
    levelOneButton.class("campaignButtonClass");
    levelOneButton.mousePressed(() => campaignButtonsFunction(0));

    levelTwoButton = createButton("Level Two");
    levelTwoButton.position(340, 125);
    levelTwoButton.class("campaignButtonClass");
    levelTwoButton.mousePressed(() => campaignButtonsFunction(1));

    levelThreeButton = createButton("Level Three");
    levelThreeButton.position(540, 125);
    levelThreeButton.class("campaignButtonClass");
    levelThreeButton.mousePressed(() => campaignButtonsFunction(2));
}

// Shows or hides the campaign buttons depending on the input.
function campaignButtonsDisplay(display) {
    if (display) {
        levelOneButton.show();
        levelTwoButton.show();
        levelThreeButton.show();
        displayStarRatings();
    } else {
        levelOneButton.hide();
        levelTwoButton.hide();
        levelThreeButton.hide();
    }
}

// Displays the users previous best attempts at each level.
function displayStarRatings() {
    for (let i = 0; i<3; i++) {
```

```
        let rating = levels[chosenMapInt][i].rating;
        if (rating == 0) {
            text("⚫", 145 + (200 * i), 240)
            text("⚫", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 1) {
            text("🟡", 145 + (200 * i), 240)
            text("⚫", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 2) {
            text("🟡", 145 + (200 * i), 240)
            text("🟡", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 3) {
            text("🟡", 145 + (200 * i), 240)
            text("🟡", 195 + (200 * i), 240)
            text("🟡", 245 + (200 * i), 240)
        }
    }
}

// Changes the value of chosenMap depending on the button the user clicks
function campaignButtonsFunction(level) {
    chosenLevel = level;
}

// Updates the enemies for the campaign gamemode
function enemyUpdateCampaign() {
    for (let i = 0; i < 10; i++) {
        for (let j = 0; j < enemies[i].queue.length; j++) {
            enemies[i].timeElapsed += deltaTime;
            if(enemies[i].timeElapsed > enemies[i].spawnRate &&
enemies[i].tail < levels[chosenMapInt][chosenLevel].returnNoOfEnemies(i)) {
                enemies[i].tail++;
                enemies[i].timeElapsed = 0;
            }
        }
    }
    enemyUpdate();
}
```

Above are a few examples of code annotation and comments, every file has similar annotation to the screenshots above.

D11 - Naming:

Variables	Data Structures and Objects	Subroutines
• projectileUpgrade • damageUpgrade • baseHealthUpgrade • enemySpeedUpgrade • enemyArmourUpgrade • specialAbilityUpgrade • arcade1 • arcade2 • arcade3 • chosenMapInt • chosenLevel • campaignCheck • allowAbility • placeTrap • trapFrameCount • specialForcesFrameCount • airStrikeFrameCount • airStrikeCheck • airStrikeX • airStrikeY • airStrikeCoordCheck	• projectileUpgradeButton • damageUpgradeButton • baseHealthUpgradeButton • enemySpeedUpgradeButton • enemyArmourUpgradeButton • specialAbiltiyUpgradeButton • levelOneButton • levelTwoButton • levelThreeButton • levels • specialAbilityButton • TRAP • specialForces	• skillTreeButtonsSetup() • skillTreeButtonsDisplay(display) • displayInformationSkillTree() • upgradePercent(upgrade) • upgradeCost(upgrade) • skillTreeButtonsFunction(upgrade, type) • changeTowerColour(change, tower) • setBaseHealth(health) • enemySetup(gamemode) • campaignButtonsSetup() • campaignButtonsDisplay(display) • displayStarRatings() • campaignsButtonFunction(level) • enemyUpdateCampaign()

D13 - Prototypes:

The video below shows the solution at the end of this iteration:
https://drive.google.com/file/d/1KFEPYEJCzvMpVK4Ksoqq-b9_s_3es0t9/view?usp=drive_link

D14 - Review:

In this iteration, I completed the following success criteria:

- (2) There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players.
- (3) There is an arcade game mode. The user selects one of the maps and plays endlessly with increasing difficulty over time until they lose their base. This gamemode is aimed at casual players.
- (5) The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the

available squares where the user can place towers. This system makes playing the game easier for the user.

- (10) There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level.
- (11) The user can access the skill tree from the main menu. They can spend their battle medals and purchase universal upgrades. The user can therefore customise their towers to suit their playstyle, making the game more enjoyable.
-

Taking in feedback from the previous iteration, I added the following features:

- Add an indicator of which tower the user is placing (by highlighting it green)

After reviewing the solution currently, the stakeholders had this to say:

- William Thorne - “All the buttons work correctly and the game runs smoothly from buying and placing turrets and the game ending when you take enough damage. I like how you added the green to highlight the tower I am placing. The campaign gamemode doesn’t work, it crashes the game as soon as an enemy is hit. The special abilities all work fine and the skill tree is easy to use. There is still no tutorial or settings.

- Dominic Keyworth - “The game appears mostly completed. There is enough to enjoy the game currently so just removing the sections that are incomplete would not be an issue.
  What went well:
  - historical accuracy of special abilities.
  - special abilities work well
  - wide variety of options in the skill tree
  - still clear user interface
  - clear evidence of theme knowledge with map namings
  What could be improved
  - add detail the home page
  - Still no indication of the differences in the colourings of the attackers? Is one stronger? Was this meant to be in the tutorial?”

- Sam Russell - “Good:
  - Skill tree is intuitive
  - Special abilities work well
  - You added the tower to be selected highlighted
  Feedback:
  - Campaign gamemode crashes the game
  - Tutorial and settings empty”

- Harvey Miller - “I like the fact that each map has a different ability
  Bugs :
  - If you place a tower before using the special forces ability it repeats the tower you previously placed

- Still no indication as to which towers target which enemies”

Iterative Testing & Error Remedies:

D16 - Iterative Testing:

Iteration Number.Test Number	Outcome	Evidence
3.1	Fail	https://drive.google.com/file/d/1MshVeSwHwoD6ZzarsisXSPkOUUDoaRhU/view?usp=drive_link
3.1	Pass	https://drive.google.com/file/d/1WC6d5ew8PgxWsZXQXzfo1Y66QTImiW31/view?usp=drive_link
3.2	Pass	https://drive.google.com/file/d/1OgE2MVr0K6ie9aMM9L8mBCyGwlHzQCqm/view?usp=drive_link
3.3	Pass	https://drive.google.com/file/d/1Y8Ibh4Gvoe4OBy8-ilQGknmO6FnJGNMI/view?usp=drive_link
3.4		
3.5	Pass	https://drive.google.com/file/d/1fAyKYmlDtncL100vgSOtDoIp46iwfXiQ/view?usp=drive_link
3.6	Pass	https://drive.google.com/file/d/1vobLHSh6IKjvQbA88AKGFF3op1w2PqHu/view?usp=drive_link
3.7	Fail	https://drive.google.com/file/d/19t1V0poJsbbHhlKWjDYRNw8up7XW7Ufc/view?usp=drive_link
3.7	Pass	https://drive.google.com/file/d/1I4xwSH1syaCS3DgTgETuks2CIHQFm2KY/view?usp=drive_link

D17 - Non trivial failed tests and remedies:

I ran test 3.1 and it failed. Clicking the “Upgrade Base Health” button did not upgrade the base health but instead upgraded the projectile speed, which also was upgraded past its limit. There was the following error message “🌸 p5.js says: text() was expecting String|Object|Array|Number|Boolean for the first parameter, received an empty variable instead. (on line 77 in skillTreeSetup.js)” which is to do with upgrading the projectile speed past its limit.

. A link to the video of the failed test is below:

https://drive.google.com/file/d/1MshVeSwHwoD6ZzarsisXSPkOUUDoaRhU/view?usp=drive_link

```
// Edits the variable depending on the upgrade passed.
function skillTreeButtonsFunction(upgrade) {
    if(upgrade < 4) {
        switch(upgrade) {
            case 0:
                projectileUpgrade++;
                break;

            case 1:
                damageUpgrade++;
                break;

            case 2:
                baseHealthUpgrade++;
                break;

            case 3:
                enemySpeedUgrade++;
                break;

            case 4:
                enemyArmourUpgrade++;
                break;

            case 5:
                specialAbilityUpgrade++;
                break;
        }
    }
}
```

Above is the current version of the upgrade function. The upgrade variables are numbers 1-5 inclusive and this determines the tier of the upgrade applied. The first thing I took note of was that all the upgrades are currently tier 0 and that they all upgrade *projectileUpgrade* which is the first case checked (0) so I assumed the issue lies here. After changing the function to take a second parameter, which specifies which upgrade variable needs to be incremented, the function worked as expected. An easier solution for this would obviously be to give each button its own function which purely edits that but I wanted to keep the code slightly shorter. Finally, I made it so the user spends battle medals when they buy an upgrade which I just forgot to add earlier.

```
// Edits the variable depending on the upgrade passed.
function skillTreeButtonsFunction(upgrade, type) {
    if(upgrade < 5) {
        playerBattleMedals -= upgrade + 1;
        switch(type) {
            case 0:
                projectileUpgrade++;
                break;

            case 1:
                damageUpgrade++;
                break;

            case 2:
                baseHealthUpgrade++;
                break;

            case 3:
                enemySpeedUgrade++;
                break;

            case 4:
                enemyArmourUpgrade++;
                break;

            case 5:
                specialAbilityUpgrade++;
                break;
        }
    }
}
```

This worked as expected. A link to the video of the new passed test is below:
https://drive.google.com/file/d/1WC6d5ew8PgxWsZXQXzfo1Y66QTImiW31/view?usp=drive_link

---

I ran test 3.7 and it failed. The enemies walking through the trap did not cause them to stop. There was no error message. A link to the video of the failed test is below:
https://drive.google.com/file/d/19t1V0poJsbbHhlKWjDYRNw8up7XW7Ufc/view?usp=drive_link

After placing various console.logs throughout the process of placing a trap (which is split into multiple functions to keep code modular), I found the issue was with the following function:

```
// Uses the trap
function trapFunction() {
    if (TRAP.placed && allowAbility) {
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);

                if(enemyX - TRAP.posX == 0 && enemyY - TRAP.posY == 0) {
                    enemies[i].queue[j].speed = 0;
                }
            }
        }
    }

    if (frameCount - trapFrameCount > 300) {
        TRAP.placed = false;
        allowAbility = false;
    }
}
```

After isolating only the bottom section of code (which makes the special ability stop after 5 seconds), this ran perfectly so the problem lies in the top section. Running a few more tests, I discovered that the trap behaves differently depending on whether or not it is placed whilst enemies lie on the screen. The opposite also seemed to be true though. Temporarily, I decided to split the two statements into two functions although in doing this I realised the problem. The second if statement doesn't take into account whether the trap is actually placed yet, so as long as the game has run for 300 frames (since *trapFrameCount* is declared as zero and changed when the trap is placed) this statement runs and the trap is turned off (so clicking the button to place the trap also turns the trap off). After adding additional conditions to check that the trap is placed, it works as expected.

```
if (TRAP.placed && allowAbility && frameCount - trapFrameCount > 300) {
        TRAP.placed = false;
        allowAbility = false;
    }
```

A link to the video of the new test is below:
https://drive.google.com/file/d/1I4xwSH1syaCS3DgTgETuks2CIHQFm2KY/view?usp=drive_link

Post Development Testing:

D7 - Post Development Testing

Functionality

Test Reference	Link to testing video	Tester's comment on expected outcome
Scenario 1 Tester - Stakeholder Success Criteria: 2 - There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players. Test Details: ● Open https://computer-science-coursework.jamesbaldwxn.repl.co/ ● Press the “Campaign” button ● Press the “Stalingrad” button ● Press the “Level One” button Expected Outcome The user can play the campaign gamemode. Once the game ends (assuming they beat the level), the rating is updated		
Scenario 2 Tester - Stakeholder Success Criteria: 3 - There is an arcade game mode. The user selects one of the maps and plays endlessly with increasing difficulty over time until they lose their base. This gamemode is aimed at casual players. Test Details: ● Open https://computer-science-coursework.jamesbaldwxn.repl.co/ ● Press the “Arcade” button ● Press the “Stalingrad” button Expected Outcome The user can play the arcade gamemode.		

Robustness

Test Reference	Link to testing video	Tester's comment on expected outcome
Scenario 1 Tester - Stakeholder		

Success Criteria: 5 - The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user. Test Details: ● Open https://computer-science-coursework.jamesbaldwxn.repl.co/ ● Press the “Arcade” button ● Press the “Stalingrad” button Expected Outcome The user cannot break the game by placing towers on top of eachother, placing them on the enemies etc.		
Scenario 2 Tester - Stakeholder Success Criteria: 2 - There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players. Test Details: ● Open https://computer-science-coursework.jamesbaldwxn.repl.co/ ● Press the “Campaign” button ● Press the “Stalingrad” button ● Press the “Level One” button Expected Outcome The user can play the campaign gamemode.		

Usability

Test Reference	**Link to testing video**	**Tester's comment on expected outcome**
Scenario 1 Tester - Stakeholder Success Criteria: 5 - The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user.		

Test Details: ● Open https://computer-science-coursework.jamesbaldwxn.repl.co/ ● Press the “Arcade” button ● Press the “Stalingrad” button Expected Outcome Once the user clicks the button to place a tower, a screen appears showing them where they can and can’t place a tower. The tower they clicked is now highlighted on the left hand side of the screen.		
Scenario 1 Tester - Myself Success Criteria: 5 - The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user. Test Details: ● Open https://computer-science-coursework.jamesbaldwxn.repl.co/ ● Press the “Arcade” button ● Press the “Stalingrad” button ● Wait for enemies to appear ● Go into chrome dev tools and load lighthouse Expected Outcome The page loads fast with little issues.	https://drive.google.com/file/d/14qFhooQAAzPuWU0HWrVFvIB_FitctEer/view?usp=drive_link	The page appears a bit less efficient than I expected, I thought it might be something to do with replit but I repeated the test with a local host and the results were similar.

Evaluation:

E3 & E4 - Assessment of Success Criteria:

1. The user is greeted by a main menu. This is where they can choose between campaign, arcade, tutorial, skill tree or access the settings. Having the menu and all game options in a central location provides for a better user experience. **FULLY MET**

All the tests involving this success criteria are below:

Result	Test	Expected Outcome
Pass	https://drive.google.com/file/d/1ZDM3p	The main menu appears with the title,

	RrBa8rGmz1OwnbNdZRPrY4Ykg7T/view?usp=drive_link	buttons and image at the bottom
Fail	https://drive.google.com/file/d/1YrjnGXP-XLdRNYZ9K2326-5NgQ2fX2d6/view?usp=drive_link	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear.
Pass	https://drive.google.com/file/d/1AWpOWUre1j5TpocLuWf4-0GDGRcRQCZI/view?usp=drive_link	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear.
Pass	https://drive.google.com/file/d/1Z5nNoD0Njm93T5Nm6_6LDRi9hvn0qTiU/view?usp=drive_link	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear. After the “Menu” button is clicked, a confirmation prompt appears. Once “Ok” is clicked, the main menu screen appears again, including the buttons redisplaying.
Pass	https://drive.google.com/file/d/1hNqHxDeiKNF6luNNhuJTQz0moDaTvr2p/view?usp=drive_link	The arcade game mode loads and the menu button appears in the top right corner. The buttons from the menu disappear. After the “Menu” button is clicked, a confirmation prompt appears. Once “Cancel” is clicked, nothing changes and the arcade screen is still displayed.
Pass	https://drive.google.com/file/d/1IDoAaLtPs8QNcZJCfUjMnxDi5e72uTm5/view?usp=drive_link	The skill tree loads and the menu button appears in the top right corner. The buttons from the menu disappear. After the “Menu” button is clicked, there is no prompt and the main menu immediately loads.

The videos show that this feature is fully met because when the game first opens, the menu appears. It has options to take you to different parts of the game and always return to the menu using the button in the top right. Some of the buttons lead to an empty screen but that is due to other success criteria not being met rather than this one.

Perhaps currently it would make more sense to remove the campaign, tutorial or settings sections currently since these either don’t work or are empty but after further development these sections would be filled so removing them is unnecessary.

---

2. There is a campaign game mode. This takes the user in chronological order through the maps with multiple levels per map. This gamemode is aimed at competitive players. **PARTIALLY MET**

The levels work as expected and the enemies are correctly limited to a reduced number (compared to unlimited in arcade). However, as soon as you place a tower (and an enemy comes in range of it), the game crashes. I couldn't resolve this issue.

This would clearly be the first thing I'd resolve in future development since this is one of the two main game modes/parts of the game.

---

3. There is an arcade game mode. The user selects one of the maps and plays endlessly with increasing difficulty over time until they lose their base. This gamemode is aimed at casual players. **FULLY MET**

All the tests involving this success criteria are below:

Result	**Test**	**Expected Outcome**
Pass	https://drive.google.com/file/d/1Y8Ibh4Gvoe4OBy8-iIQGknmO6FnJGNMI/view?usp=drive_link	When the player score is low (i.e. 0) the enemies spawn slowly but when it is higher (i.e. 1000000) they spawn faster. This increases the difficulty of the arcade gamemode over time

This video shows the arcade gamemode working as expected because unlimited enemies spawn and over time the difficulty increases (once the score is over the thresholds shown).

In the future, I may change the thresholds to reduce the enemy spawn rate to balance the game more if it is too easy/hard and I may start spawning only specific enemies (i.e. once the score reaches a certain point only the hardest enemy types spawn since the easier ones aren't going to do anything).

---

4. The enemy follows the path highlighted towards the base. Once it reaches the base, it deals damage to the base and disappears. There are multiple enemy types with different sprites, speed and health. Multiple enemy types increases challenge for the player. **FULLY MET**

All the tests involving this success criteria are below:

Result	**Test**	**Expected Outcome**
Fail	https://drive.google.com/file/d/1-DUOheXCV-UDNayEij5OUTgH2-BIomdk/view?usp=drive_link	The arcade game mode loads and a single red rectangle (enemy) should move along the path.
Fail	https://drive.google.com/file/d/1-anv0xkijDBDd5IFOrK9HZ4bfMGhVLjK/view?usp=drive_link	The arcade game mode loads and a single red rectangle (enemy) should move along the path.
Pass	https://drive.google.com/file/d/1pwInlf2EtB99RA9aSPKREVuF-	The arcade game mode loads and a single red rectangle

	DhHNa1U/view?usp=drive_link	(enemy) should move along the path.
Pass	https://drive.google.com/file/d/1tJzL-XQlkl2v7LshT2Cco_07jhwgEl9R/view?usp=drive_link	The arcade game mode loads and multiple red rectangles (enemy) should spawn and move along the path.
Pass	https://drive.google.com/file/d/1U1ACX5Nnu2jCldbmpSzs9Lmmr5bLXnAB/view?usp=drive_link	The arcade game mode loads and on the right hand side there is a rectangle displaying the current base health as a gradient between red and green. When an enemy hits the base, it deals its current health as damage to the base and disappears. The base health is updated.
Pass	https://drive.google.com/file/d/1P3onFeCKhS8vsc49m6EHNT6dl5bDfFOl/view?usp=drive_link	The arcade game mode loads and enemies of different colours and speeds start to move down the path.
Fail	https://drive.google.com/file/d/1eRi1TP95Cm8z4bdst3XnA89-LPqK__UO/view?usp=drive_link	The arcade game mode loads and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
Fail	https://drive.google.com/file/d/1VhPDn-kfGTM9zbIVZX0MTHsSsqec6pBs/view?usp=drive_link	The arcade game mode loads and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
Fail	https://drive.google.com/file/d/1nGAhIiWcnksTW2AYMsbVF2ZDEdiHzIan/view?usp=drive_link	The arcade game mode loads and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
Fail	https://drive.google.com/file/d/1hyYkKVNJ44rx7PN9AYChfIvK6el_UJz5/view?usp=drive_link	The arcade game mode loads and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
Fail	https://drive.google.com/file/d/1DPqjr4Fi4nZWDgX5OaDvqbTLBufT0zCD/view?usp=drive_link	The arcade game mode loads and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
Pass	https://drive.google.com/file/d/1	The arcade game mode loads

	Pzq0cNBi2jwQg73QTHk-u99sqjpTF_rN/view?usp=drive_link	and one type of enemy (red) moves continuously across the path (instead of moving in cell size jumps as before).
Pass	https://drive.google.com/file/d/1hpYeBti_dqknkPK0XdP7ovKOxVup_KfL/view?usp=drive_link	The arcade game mode loads and there are multiple enemies with different colours, speeds and spawn rates.

The videos show the enemies spawning, moving towards the base and different types of enemy which when put together completes the success criteria.

In future development I may add more enemy types (to increase difficulty if balance changes are needed) or change the speeds/spawn rates of current enemies.

---

5. The user can click a tower they want to place from the banner on the right of the screen and click the tile they want to place it on. The tiles they cannot place it on are coloured red (including obstacles and tiles with towers already on). This highlights the available squares where the user can place towers. This system makes playing the game easier for the user. **FULLY MET**

All the tests involving this success criteria are below:

Result	**Test**	**Expected Outcome**
Pass	https://drive.google.com/file/d/1v1UZMqlXVPVfMiN91_J7A07J9fZnrZsR/view?usp=drive_link	Once the “Soldier” button is pressed, a screen should appear highlighting where the user can and can’t place a tower.
Pass	https://drive.google.com/file/d/10TcVWHi8X0mtWRuGoPOn_PzPY2Dvp5a9/view?usp=drive_link	The user should be able to place each type of tower. They shouldn’t be able to place a tower on any of the red tiles and where they place a tower should become a red tile. The cancel button should disappear once they place a tower, but should work if they click a tower button but don’t want to place a tower.
Pass	https://drive.google.com/file/d/17u0CBgZJh9sQ4kNvKWOQIRrAtTkyjxdW/view?usp=drive_link	The tower should be able to shoot the first couple of enemies but not any more. This is because the towers can only shoot enemies of a similar tier (and to keep realism, a soldier shouldn’t be able to shoot down a plane alone).
Pass	https://drive.google.com/file/d/10Qza35nfF8Myo0Sj4zkoD7t14nRUTrhG/view?usp=drive_link	The tower ranges should not be shown when the towers are first placed. Once the button is clicked, a circle appears which is the same colour as the enemy displaying the tower range. Once the button is clicked for the second time, the circles disappear again.
Pass	https://drive.google.co	The current tower selected should be highlighted in green.

	m/file/d/1OgE2MVr0K6ie9aMM9L8mBCyGwlHzQCqm/view?usp=drive_link	When the cancel button is pressed or the tower is placed, the highlight turns off.

The videos show the towers being placed and shooting at enemies (the other part of the success criteria relates to usability).

In the future, I may add new towers or balance the current ones if needed.

---

6. A projectile fired from the tower moves towards the enemy. When it hits the enemy, this collision is detected and the enemy takes damage. The projectile disappears after this. If the damage was enough to destroy the enemy, the enemy also disappears. This ensures that the game works as required for a tower defence game. **FULLY MET**

All the tests involving this success criteria are below:

Result	**Test**	**Expected Outcome**
Pass	https://drive.google.com/file/d/1ryCEKOqX4qN9q3UZ2hPDJVzorSUXLLTx/view?usp=drive_link	A tower should be displayed automatically. Its range should be displayed and when an enemy enters into this zone it should start firing projectiles towards the enemy.
Fail	https://drive.google.com/file/d/1ryCEKOqX4qN9q3UZ2hPDJVzorSUXLLTx/view?usp=drive_link	A tower should be displayed automatically. Its range should be displayed and when an enemy enters into this zone it should start firing projectiles towards the enemy. When the projectiles hit the enemy, they disappear.
Fail	https://drive.google.com/file/d/1ZTPVgic1Lgcw4SgVN965-9-DSU7WUf8X/view?usp=drive_link (skip to about 1 minute to see the bug, it takes longer to appear than before because the bug is slightly fixed)	A tower should be displayed automatically. Its range should be displayed and when an enemy enters into this zone it should start firing projectiles towards the enemy. When the projectiles hit the enemy, they disappear.
Pass	https://drive.google.com/file/d/1lzCtyJd4j1-8XsRxxXOZmlQiET_yxiG1/view?usp=drive_link	A tower should be displayed automatically. Its range should be displayed and when an enemy enters into this zone it should start firing projectiles towards the enemy. When the projectiles hit the enemy, they disappear.

The videos show the projectiles being fired at the enemy and slowly destroying it (and disappearing once it has hit the enemy). Once the enemy has no health, it disappears. This meets the success criteria.

---

7. When the base health is zero, the game ends and resets. This means the user can play again if they are playing arcade or restart the level if they are playing the campaign. **PARTIALLY MET**

All the tests involving this success criteria are below:

Result	Test	Expected Outcome
Pass	https://drive.google.com/file/d/1sCA8nEsP5fr_UnopguMo_AuqwIUq4Mkj/view?usp=drive_link (Note I sped up the enemies so they destroy the base quicker)	The game should restart once the base is destroyed once you click “Ok”. All the enemies should be cleared. The score and base health should reset.
Pass	https://drive.google.com/file/d/1k0hCKm2psDV541HUe89u1pqEz5CZ9qwG/view?usp=drive_link (Note I sped up the enemies so they destroy the base quicker)	The game should return to the menu once the base is destroyed and you click “Cancel”. Once you click the buttons to return to the menu and click the other map, the base health, score and enemies should all be reset.

The game ends when the base health reaches zero and everything resets but I forgot to add battle medals so the user can buy things in the skill tree.

---

8. All the appropriate information is displayed to the user at all times while the game is running (base health, score, cash & battle medal counts). This is all important information to the player and displaying this at all times ensures the player has all the information needed for the game. **FULLY MET**

All the tests involving this success criteria are below:

Result	Test	Expected Outcome
Pass	https://drive.google.com/file/d/1ZSSNZbx6acGue7VKVu_P8XfDav-B-osm/view?usp=drive_link	The arcade game mode loads and the user is greeted with a blank copy of the map (there are no enemies yet). The user’s score, cash and battle medal counts are displayed in the top right corner, underneath the menu button.

Pass	https://drive.google.com/file/d/1ROvIO-ZlEFdXujuq2Fzkp1P9QbeCpQ1C/view?usp=drive_link	The arcade game mode loads and on the right hand side there is a rectangle displaying the current base health as a gradient between red and green. This should steadily go down over time, to show off the gradient.

---

9. The user can click the upgrade button in the bottom left corner. If they subsequently click a tile with a tower on, they are prompted to click one of two options which to upgrade. If they click the upgrade button again, upgrade mode turns off. This allows for an easy way for the user to upgrade towers. **NOT MET**

There were no tests towards this success criteria. I think if I was to implement it I would have to fundamentally change the way the towers were made (although I could try just calculating the nearest tower when the user clicks on the screen to upgrade the tower by looping through the tower array, doubt it would work very well though). Plus, the user can already upgrade all towers permanently in the skill tree so implementing a feature to upgrade individual towers would be a similar thing.

---

10. There are multiple maps. Each map has a different special ability. The ability can be used once per level (in campaign) and then it is turned off. This ensures that the game is unique and increases challenge for the player since this feature can only be used once per level. **PARTIALLY MET**

All the tests involving this success criteria are below:

Result	**Test**	**Expected Outcome**
Fail	https://drive.google.com/file/d/10WEW7tP5-PA1MvpSf91FzTk1ZFFXlNWg/view?usp=drive_link	The arcade section of the game loads with the Stalingrad themed map. The enemies follow the new shape of the path.
Pass	https://drive.google.com/file/d/1lpGAotl4asXEKwU2px--JqwrfqieHcAb/view?usp=drive_link	The arcade section of the game loads with the Stalingrad themed map. The enemies follow the new shape of the path.
Pass	https://drive.google.com/file/d/1GxMEVmQ3cehlMIyNhJGMwBWIk8r2cAGe/view?usp=drive_link	The arcade section of the game loads with the Gulf War themed map. The enemies follow the new shape of the path.
Pass	https://drive.google.com/file/d/1_5JYMcxQN4_rU4qUSTjdb_I5mIKLIFBS/view?usp=drive_link	The map selection menu pops up with images of each of the maps above a button which takes you to that map. It is possible to return to the menu and select a different map.

Pass	https://drive.google.com/file/d/1fAyKYmlDtncL100vgSOtDoIp46iwfXiQ/view?usp=drive_link	Once clicked, the button should turn green to show that it has been pressed. The enemies within a 2 tile radius of the mouse should be destroyed and an explosion effect should appear here, before shortly disappearing. The "Call Airstrike" button should turn light grey and be unable to be used again
Pass	https://drive.google.com/file/d/1vobLHSh6IKjvQbA88AKGFF3op1w2PqHu/view?usp=drive_link	Once clicked, the button should turn green to show that it has been pressed. 5 smaller towers should appear on a single tile. The "Call Special Forces" button should turn light grey and be unable to be used again. After 15 seconds, the special forces disappear.
Fail	https://drive.google.com/file/d/19t1V0poJsbbHhlKWjDYRNw8up7XW7Ufc/view?usp=drive_link	Once clicked, the button should turn green to show that it has been pressed.A trap should appear on the screen which the enemies are stuck in..The "Place Trap" button should turn light grey and be unable to be used again. After 5 seconds, the trap disappears and the enemies move again.
Pass	https://drive.google.com/file/d/1I4xwSH1syaCS3DgTgETuks2CIHQFm2KY/view?usp=drive_link	Once clicked, the button should turn green to show that it has been pressed.A trap should appear on the screen which the enemies are stuck in..The "Place Trap" button should turn light grey and be unable to be used again. After 5 seconds, the trap disappears and the enemies move again.

---

11. The user can access the skill tree from the main menu. They can spend their battle medals and purchase universal upgrades. The user can therefore customise their towers to suit their playstyle, making the game more enjoyable. **PARTIALLY MET**

All the tests involving this success criteria are below:

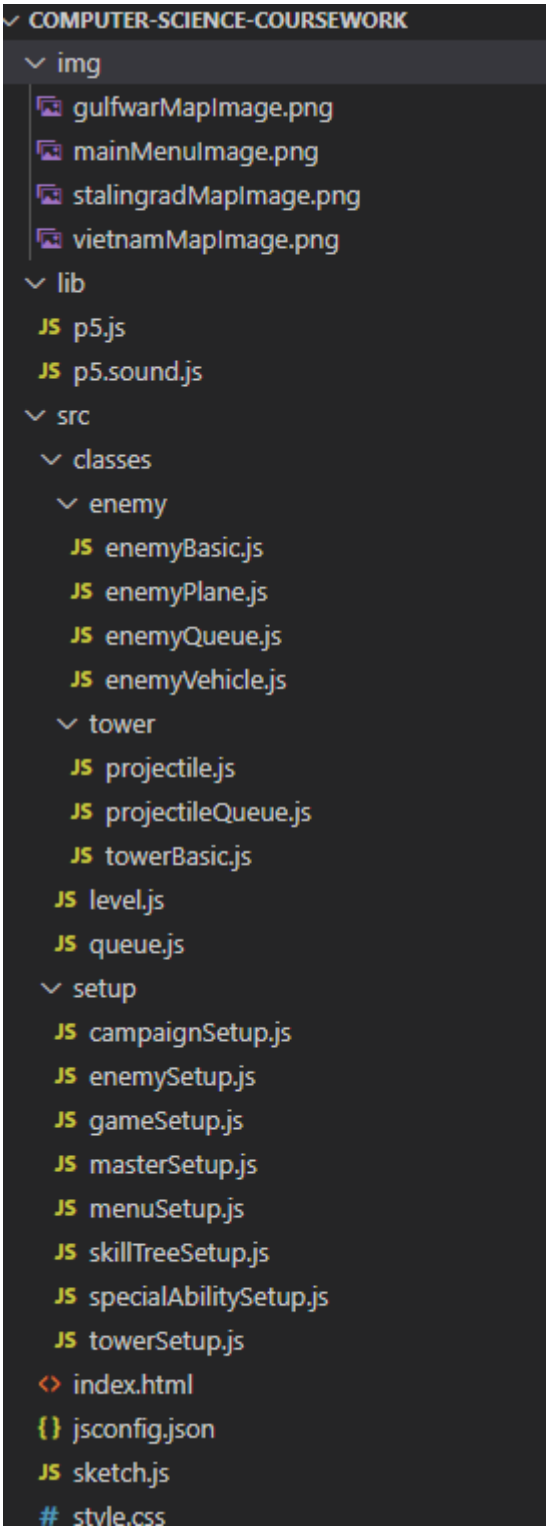
Result	**Test**	**Expected Outcome**
Fail	https://drive.google.com/file/d/1MshVeSwHwoD6ZzarsisXSPkOUUDoaRhU/view?usp=drive_link	The user spends some of their battle medals. The new upgrade value is displayed and the cost for the next upgrade is also displayed.
Pass	https://drive.google.com/file/d/1WC6d5ew8PgxWsZXQXzfo1Y66QTImiW31/view?usp=drive_link	The user spends some of their battle medals. The new upgrade value is displayed and the cost for the next upgrade is also displayed.

E5 & E6 - Assessment of Usability Features:

E7 - Maintenance and limitations:

Adding new levels to the campaign gamemode would be very easy, I'd just need to make a new button, add a new instance of a level to the levels array and do the same to the ratings array.
In the same way, adding a new enemy type or tower type would be easy due to the object oriented nature of the code although perhaps slightly harder since I'd have to edit all the loops to fit this new length of the array (perhaps in hindsight I should've made a variable containing the number of enemy types for example and loop from 0 to the number).
The first thing that I would change with further development would be to fix the campaign gamemode since this is the first button the user sees and the game crashes if they place a tower in it.
Overall, adding new features that are similar to previous would be very achievable due to the object oriented nature of my code (proven in the last iteration since when adding the special abilities I often used very similar code to the code used for the enemies or towers).

Appendix A:



enemyBasic.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
19/06/23. This file contains the base class for the enemy, used for soldiers
and special forces.
```

```
"use strict";

class enemyBasic {

  // Constructor method for the class.
  constructor(health, speed, map, colour, pointer, changeOfDirection) {
    this.posX = MAP_TILE_X[0];
    this.posY = MAP_TILE_Y[4];
    this.nextX = MAP_TILE_X[1];
    this.nextY = MAP_TILE_Y[4];
    this.health = health;
    this.healthMax = health;
    this.speed = speed * (1 - upgradePercent(enemySpeedUgrade)); // The rate
at which the enemy moves to the text square.
    this.colour = colour;
    this.map = map; // Current map.
    this.mapPassed = [ // Tracks where the enemy has already been on the map
so it doesn't go back on itself.
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0],
    ];
    this.pointer = pointer; // Position in the queue.
    this.move = false; // Contains the direction which the enemy is moving
(false is left or right, true is up or down).
    this.changeOfDirection = changeOfDirection;
  }
```

```
  // Updates the enemies position each frame.
  update(queue) {
    let i = Math.floor(this.posX / CELL_SIZE);
    let j = Math.floor(this.posY / CELL_SIZE);

    // Checks if adjacent tiles are path tiles and need to be moved onto.
    if (this.map[i + 1][j] == 1 && this.mapPassed[i + 1][j] == 0 &&
distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
      this.mapPassed[i][j] = 1;
      this.nextX = (i + 1) * CELL_SIZE;
      this.move = false;

    } else if (i >= 1 && this.map[i - 1][j] == 1 && this.mapPassed[i - 1][j]
== 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
      this.mapPassed[i][j] = 1;
      this.nextX = (i - 1) * CELL_SIZE;
      this.move = false;

    } else if (this.map[i][j + 1] == 1 && this.mapPassed[i][j + 1] == 0 &&
distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
      this.mapPassed[i][j] = 1;
      this.nextY = (j + 1) * CELL_SIZE;
      this.move = true;

    } else if (j >= 1 && this.map[i][j - 1] == 1 && this.mapPassed[i][j - 1]
== 0 && distance(this.posX, this.posY, this.nextX, this.nextY) < 1) {
      this.mapPassed[i][j] = 1;
      this.nextY = (j - 1) * CELL_SIZE;
      this.move = true;

    } else if (this.map[i + 1][j] == 2 && this.mapPassed[i + 1][j] == 0) { //
If the next tile is the user base then the enemy damages it and disappears.
      if (baseHealth > 0) {
        baseHealth -= this.health;
      }
      queue.dequeue();
    }

    // Checks whether the x or y direction needs to be changed and changes it,
as well as checking if the enemy has moved to the next tile.
    switch (this.move) {
      case false:
```

```
        if (Math.abs(this.nextX - this.posX) < this.changeOfDirection) {
          this.posX = this.nextX;
        } else {
          this.posX = cosineInterpolate(this.posX, this.nextX, this.speed);
        }
        break;

      case true:
        if (Math.abs(this.nextY - this.posY) < this.changeOfDirection) {
          this.posY = this.nextY;
        } else {
          this.posY = cosineInterpolate(this.posY, this.nextY, this.speed);
        }
        break;
    }

    // If the end of the current tile is reached, it is marked as passed.
    if (distance(this.posX, this.posY, this.nextX, this.nextY) <
this.changeOfDirection) {
      this.mapPassed[i][j] = 1;
    }
    this.display();
  }

  // If the enemies still has health, it is displayed.
  display() {
    if (this.health > 0) {
      if (this.health < this.healthMax) {
        let percentage = this.health / this.healthMax;
        let start = color(255, 0, 0);
        let end = color(0, 255, 0);
        let gradientColour = lerpColor(start, end, percentage); // Colours the
health bar a percentage between red and green depending on the health
remaining.
        let barWidth = map(percentage, 0, 1, 0, 76); // Makes the size of the
health bar proportional to the health remaining.
        fill(gradientColour);
        noStroke();
        rect(this.posX + 10, this.posY + 91, barWidth, 5);
      }

      fill(this.colour);
```

```
      rect(this.posX + 10, this.posY + 10, CELL_SIZE - 20, CELL_SIZE - 20);
    }
  }
}
```

enemyPlane.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
27/06/23. This file contains the class for enemy planes and helicopters.

"use strict";

class enemyPlane extends enemyVehicle {

    // Constructor method for the class.
    constructor(health, speed, map, colour, pointer, changeOfDirection,
armour, stealth) {
        super(health, speed, map, colour, pointer, changeOfDirection, armour);
        this.flight = true;
        this.stealth = stealth;
    }
}
```

enemyQueue.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/06/23. This file contains the queue data structure used for the enemies.

"use strict";

class enemyQueue extends queue {

    // Constructor method for the class.
    constructor(spawnRate, type) {
        super();
        this.timeElapsed = 0; // Time elapsed since the last time an enemy was
spawned.
        this.spawnRate = spawnRate; // The rate at which enemies spawn. The
lower the value, the more frequently they spawn.
        this.type = type; // The type of enemy.
    }

    // Clears the queue. Overrides the parent clear method.
    clear() {
```

```
        this.queue = [];
        this.head = 0;
        this.tail = 0;
        this.timeElapsed = 0;
    }

    // Checks the enemy type to see what needs to be displayed.
    checkType() {
        switch (this.type) {
            case 0:
                this.enqueue(new enemyBasic(200, 0.12, chosenMap, "red",
this.tail, 5)); // Soldier.
                break;

            case 1:
                this.enqueue(new enemyBasic(300, 0.13, chosenMap, "yellow",
this.tail, 5)); // Special forces.
                break;

            case 2:
                this.enqueue(new enemyVehicle(400, 0.14, chosenMap, "gray",
this.tail, 2, 100)); // Truck.
                break;

            case 3:
                this.enqueue(new enemyVehicle(500, 0.15, chosenMap, "blue",
this.tail, 2, 125)); // APC.
                break;

            case 4:
                this.enqueue(new enemyVehicle(300, 0.16, chosenMap, "green",
this.tail, 2, 150)); // IFV.
                break;

            case 5:
                this.enqueue(new enemyVehicle(350, 0.17, chosenMap, "orange",
this.tail, 1, 200)); // Tank.
                break;

            case 6:
                this.enqueue(new enemyPlane(400, 0.18, chosenMap, "purple",
this.tail, 1, 400, false)); // Helicopter.
```

```
                break;

            case 7:
                this.enqueue(new enemyPlane(450, 0.19, chosenMap, "lime",
this.tail, 1, 500, false)); // Fighter jet.
                break;

            case 8:
                this.enqueue(new enemyPlane(450, 0.2, chosenMap, "white",
this.tail, 1, 600, false)); // Bomber.
                break;

            case 9:
                this.enqueue(new enemyPlane(500, 0.2, chosenMap, "black",
this.tail, 1, 750, true)); // Stealth bomber.
                break;
        }
    }

    // For each item in the queue, its position is updated (using the update
method from the enemy class).
    updatePosistion() {
        for (let i = this.head; i < this.tail; i++) {
                this.queue[i].update(this);
        }
     }

    // Spawns new enemies.
    spawn() {
        this.timeElapsed += deltaTime;
        if (this.timeElapsed > this.spawnRate) {
          this.checkType();
          this.timeElapsed = 0;
        }
    }
}
```

enemyVehicle.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/06/23. This file contains the queue data structure used for the enemies.

"use strict";
```

```
class enemyQueue extends queue {

    // Constructor method for the class.
    constructor(spawnRate, type) {
        super();
        this.timeElapsed = 0; // Time elapsed since the last time an enemy was
spawned.
        this.spawnRate = spawnRate; // The rate at which enemies spawn. The
lower the value, the more frequently they spawn.
        this.type = type; // The type of enemy.
    }

    // Clears the queue. Overrides the parent clear method.
    clear() {
        this.queue = [];
        this.head = 0;
        this.tail = 0;
        this.timeElapsed = 0;
    }

    // Checks the enemy type to see what needs to be displayed.
    checkType() {
        switch (this.type) {
            case 0:
                this.enqueue(new enemyBasic(200, 0.12, chosenMap, "red",
this.tail, 5)); // Soldier.
                break;

            case 1:
                this.enqueue(new enemyBasic(300, 0.13, chosenMap, "yellow",
this.tail, 5)); // Special forces.
                break;

            case 2:
                this.enqueue(new enemyVehicle(400, 0.14, chosenMap, "gray",
this.tail, 2, 100)); // Truck.
                break;

            case 3:
                this.enqueue(new enemyVehicle(500, 0.15, chosenMap, "blue",
this.tail, 2, 125)); // APC.
```

```
                break;

            case 4:
                this.enqueue(new enemyVehicle(300, 0.16, chosenMap, "green",
this.tail, 2, 150)); // IFV.
                break;

            case 5:
                this.enqueue(new enemyVehicle(350, 0.17, chosenMap, "orange",
this.tail, 1, 200)); // Tank.
                break;

            case 6:
                this.enqueue(new enemyPlane(400, 0.18, chosenMap, "purple",
this.tail, 1, 400, false)); // Helicopter.
                break;

            case 7:
                this.enqueue(new enemyPlane(450, 0.19, chosenMap, "lime",
this.tail, 1, 500, false)); // Fighter jet.
                break;

            case 8:
                this.enqueue(new enemyPlane(450, 0.2, chosenMap, "white",
this.tail, 1, 600, false)); // Bomber.
                break;

            case 9:
                this.enqueue(new enemyPlane(500, 0.2, chosenMap, "black",
this.tail, 1, 750, true)); // Stealth bomber.
                break;
        }
    }

    // For each item in the queue, its position is updated (using the update
method from the enemy class).
    updatePosistion() {
        for (let i = this.head; i < this.tail; i++) {
                this.queue[i].update(this);
        }
     }
```

```
    // Spawns new enemies.
    spawn() {
        this.timeElapsed += deltaTime;
        if (this.timeElapsed > this.spawnRate) {
          this.checkType();
          this.timeElapsed = 0;
        }
    }
}
```

projectile.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/08/23. This file contains the class for projectiles.

"use strict";

class projectile {

    // Constructor method for the class.
    constructor(posX, posY, targetX, targetY, colour, speed, damage) {
        this.posX = posX;
        this.posY = posY;
        this.targetX = targetX;
        this.targetY = targetY;
        this.colour = colour;
        this.speed = speed;
        this.damage = damage;
    }

    // Updates the posistion of the projectile using cosine interpolation.
    update() {
        this.posX = cosineInterpolate(this.posX, this.targetX, 0.18 -
(upgradePercent(projectileUpgrade) / 6));
        this.posY = cosineInterpolate(this.posY, this.targetY, 0.18 -
(upgradePercent(projectileUpgrade) / 6));
        fill(this.colour);
        circle(this.posX + 48, this.posY + 48, 5);
    }
}
```

projectileQueue.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/08/23. This file contains the class for a queue structure.

"use strict";

class projectileQueue extends queue {

    // Constructor method for the class. No new attributes needed and no
attributes passed into queue class hence it looks a bit empty.
    constructor() {
        super();
    }

    // Updates the posistion of each projectile in the queue.
    updatePosistion() {
        for (let i = this.head; i < this.tail; i++) {
            this.queue[i].update();
        }
    }
}
```

towerBasic.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/08/23. This file contains the base class for the towers.

"use strict";

class towerBasic {

    // Constructor method for the class.
    constructor(posX, posY, colour, type, range, damage, speed) {
        this.posX = posX;
        this.posY = posY;
        this.colour = colour;
        this.type = type; // Type of tower.
        this.range = range;
        this.damage = damage  * (1 + upgradePercent(damageUpgrade));
        this.speed = speed;
        this.projectiles = new projectileQueue(); // Queue to store and manage
the towers projectiles.
        this.targetEnemy; // Stores the type and pointer of the enemy, if one
is in range.
```

```
        this.time = 0; // Stores the time since the last projectile hit an
enemy.
    }

    // Displays the tower (which doesn't move) and calls the method to move
the projectiles. Also calls the method to check whether an enemy is in range
of the tower.
    update() {

        this.time +=deltaTime;

        // If an enemy is in range, information to find its position is stored
in targetEnemy and a projectile is fired towards this position.
        if (this.calculateDistance() != false) {
            this.targetEnemy = this.calculateDistance();
            if (this.checkEnemyType()) {
                this.fireProjectile();
            }
        } else {
            this.projectiles.clear();
        }

        // If a projecile has been fired, its posistion is updated and whether
or not it has hit the enemy is checked.
        if (this.projectiles.queue.length != 0) {
            this.projectiles.updatePosistion();
            this.checkIfCollision();
        }

        if (this.type == 8) {
            fill(this.colour);
            circle(this.posX + 48, this.posY + 48, 20);
        } else {
            fill(this.colour);
            circle(this.posX + 48, this.posY + 48, 76);
        }

        // If the user wants, the tower ranges are displayed.
        if (displayRange) {
            noFill();
            stroke(this.colour);
            circle(this.posX + 48, this.posY + 48, this.range*2);
```

```
        }
    }

    // Checks if an enemy is in range of the tower.
    calculateDistance() {
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = enemies[i].queue[j].posX;
                let enemyY = enemies[i].queue[j].posY;
                if (distance(this.posX, this.posY, enemyX, enemyY) <
this.range){
                    return {
                        type: enemies[i].type,
                        pointer: enemies[i].queue[j].pointer
                    }
                }
            }
        }
        this.projectiles.clear();
        return false;
    }

    // Check if the tower can shoot at the given enemy type.
    checkEnemyType() {
        if (this.type <= 1 && this.targetEnemy.type <= 1) {
            return true;
        } else if (this.type >= 2 && this.type <= 4 && this.targetEnemy.type
<= 4) {
            return true;
        } else if (this.type == 5  || this.type == 8 && this.targetEnemy.type
<= 7) {
            return true;
        } else if (this.type == 6 && this.targetEnemy.type != 9) {
            return true;
        } else if (this.type == 7) {
            return true;
        } else {
            return false;
        }
    }

    // Fires a projectile at the target enemy.
```

```
    fireProjectile(){
        this.projectiles.enqueue(new projectile(this.posX, this.posY,
enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posX,

enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posY,
this.colour, this.speed, this.damage));
    }

    // Checks if any projectiles in the queue (i.e. the first) have hit the
enemy.
    checkIfCollision() {
        let type = this.targetEnemy.type;
        let pointer = this.targetEnemy.pointer;
        let projX;
        let projY;
        if (this.projectiles.queue[this.projectiles.head]) {
            projX =  this.projectiles.queue[this.projectiles.head].posX;
            projY = this.projectiles.queue[this.projectiles.head].posY;
        }
        let enemyX = enemies[type].queue[pointer].posX;
        let enemyY = enemies[type].queue[pointer].posY;

        if (type <= 4 && distance(projX, projY, enemyX, enemyY) < 100) {
            this.time = 0;
            this.projectiles.dequeue();
            if (type >= 2 && enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (distance(projX, projY, enemyX, enemyY) < 130) {
            this.time = 0;
            this.projectiles.dequeue();
            if (enemies[type].queue[pointer].armour > 0) {
```

```
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (this.time > 50) {
            this.time = 0;
            this.projectiles.dequeue();
                if (enemies[type].queue[pointer].armour > 0) {
                    enemies[type].queue[pointer].armour -= this.damage;
                } else {
                    enemies[type].queue[pointer].health -= this.damage;
                }

                // If this projectile destroys the enemy, the enemy is
cleared.
                if (enemies[type].queue[pointer].health < 0) {
                    enemies[type].dequeue();
                    this.projectiles.clear();
                    addScoreAndCash(type);
                }
        }
    }
}
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/08/23. This file contains the base class for the towers.

"use strict";

class towerBasic {

    // Constructor method for the class.
    constructor(posX, posY, colour, type, range, damage, speed) {
        this.posX = posX;
        this.posY = posY;
```

```
        this.colour = colour;
        this.type = type; // Type of tower.
        this.range = range;
        this.damage = damage  * (1 + upgradePercent(damageUpgrade));
        this.speed = speed;
        this.projectiles = new projectileQueue(); // Queue to store and manage
the towers projectiles.
        this.targetEnemy; // Stores the type and pointer of the enemy, if one
is in range.
        this.time = 0; // Stores the time since the last projectile hit an
enemy.
    }

    // Displays the tower (which doesn't move) and calls the method to move
the projectiles. Also calls the method to check whether an enemy is in range
of the tower.
    update() {

        this.time +=deltaTime;

        // If an enemy is in range, information to find its position is stored
in targetEnemy and a projectile is fired towards this position.
        if (this.calculateDistance() != false) {
            this.targetEnemy = this.calculateDistance();
            if (this.checkEnemyType()) {
                this.fireProjectile();
            }
        } else {
            this.projectiles.clear();
        }

        // If a projecile has been fired, its posistion is updated and whether
or not it has hit the enemy is checked.
        if (this.projectiles.queue.length != 0) {
            this.projectiles.updatePosistion();
            this.checkIfCollision();
        }

        if (this.type == 8) {
            fill(this.colour);
            circle(this.posX + 48, this.posY + 48, 20);
        } else {
```

```
            fill(this.colour);
            circle(this.posX + 48, this.posY + 48, 76);
        }

        // If the user wants, the tower ranges are displayed.
        if (displayRange) {
            noFill();
            stroke(this.colour);
            circle(this.posX + 48, this.posY + 48, this.range*2);
        }
    }

    // Checks if an enemy is in range of the tower.
    calculateDistance() {
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = enemies[i].queue[j].posX;
                let enemyY = enemies[i].queue[j].posY;
                if (distance(this.posX, this.posY, enemyX, enemyY) <
this.range){
                    return {
                        type: enemies[i].type,
                        pointer: enemies[i].queue[j].pointer
                    }
                }
            }
        }
        this.projectiles.clear();
        return false;
    }

    // Check if the tower can shoot at the given enemy type.
    checkEnemyType() {
        if (this.type <= 1 && this.targetEnemy.type <= 1) {
            return true;
        } else if (this.type >= 2 && this.type <= 4 && this.targetEnemy.type
<= 4) {
            return true;
        } else if (this.type == 5  || this.type == 8 && this.targetEnemy.type
<= 7) {
            return true;
        } else if (this.type == 6 && this.targetEnemy.type != 9) {
```

```
            return true;
        } else if (this.type == 7) {
            return true;
        } else {
            return false;
        }
    }

    // Fires a projectile at the target enemy.
    fireProjectile(){
        this.projectiles.enqueue(new projectile(this.posX, this.posY,
enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posX,

enemies[this.targetEnemy.type].queue[this.targetEnemy.pointer].posY,
this.colour, this.speed, this.damage));
    }

    // Checks if any projectiles in the queue (i.e. the first) have hit the
enemy.
    checkIfCollision() {
        let type = this.targetEnemy.type;
        let pointer = this.targetEnemy.pointer;
        let projX;
        let projY;
        if (this.projectiles.queue[this.projectiles.head]) {
            projX =  this.projectiles.queue[this.projectiles.head].posX;
            projY = this.projectiles.queue[this.projectiles.head].posY;
        }
        let enemyX = enemies[type].queue[pointer].posX;
        let enemyY = enemies[type].queue[pointer].posY;

        if (type <= 4 && distance(projX, projY, enemyX, enemyY) < 100) {
            this.time = 0;
            this.projectiles.dequeue();
            if (type >= 2 && enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
```

```
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (distance(projX, projY, enemyX, enemyY) < 130) {
            this.time = 0;
            this.projectiles.dequeue();
            if (enemies[type].queue[pointer].armour > 0) {
                enemies[type].queue[pointer].armour -= this.damage;
            } else {
                enemies[type].queue[pointer].health -= this.damage;
            }

            // If this projectile destroys the enemy, the enemy is cleared.
            if (enemies[type].queue[pointer].health < 0) {
                enemies[type].dequeue();
                this.projectiles.clear();
                addScoreAndCash(type);
            }

        } else if (this.time > 50) {
            this.time = 0;
            this.projectiles.dequeue();
                if (enemies[type].queue[pointer].armour > 0) {
                    enemies[type].queue[pointer].armour -= this.damage;
                } else {
                    enemies[type].queue[pointer].health -= this.damage;
                }

                // If this projectile destroys the enemy, the enemy is
cleared.
                if (enemies[type].queue[pointer].health < 0) {
                    enemies[type].dequeue();
                    this.projectiles.clear();
                    addScoreAndCash(type);
                }
        }
    }
}
```

level.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
10/11/23. This file contains the class for a campaign level.

"use strict";

class level {

    // Constructor for the class
    constructor(baseHealth, soldier, specialForces, truck, apc, ifv, tank,
helicopter, fighterJet, bomber, stealthBomber) {
        this.rating = 0;
        this.baseHealth = baseHealth;
        this.soldier = soldier;
        this.specialForces = specialForces;
        this.truck = truck;
        this.apc = apc;
        this.ifv = ifv;
        this.tank = tank;
        this.helicopter = helicopter;
        this.fighterJet = fighterJet;
        this.bomber = bomber;
        this.stealthBomber = stealthBomber;
    }

    // Sets the length of the enemy arrays so only that many spawn
    setEnemies() {
        enemies[0].queue.length = this.soldier;
        enemies[1].queue.length = this.specialForces;
        enemies[2].queue.length = this.truck;
        enemies[3].queue.length = this.apc;
        enemies[4].queue.length = this.ifv;
        enemies[5].queue.length = this.tank;
        enemies[6].queue.length = this.helicopter;
        enemies[7].queue.length = this.fighterJet;
        enemies[8].queue.length = this.bomber;
        enemies[9].queue.length = this.stealthBomber;

        for(let i = 0; i < 10; i++) {
            for(let j = 0; j < enemies[i].queue.length; j++) {
                enemies[i].queue[j] = this.returnEnemyType(i);
            }
        }
```

```
    }

    // Returns the enemy type to fill the queue
    returnEnemyType(type) {
        switch (type) {
            case 0:
                return new enemyBasic(200, 0.12, chosenMap, "red", this.tail,
5); // Soldier.
                break;

            case 1:
                return new enemyBasic(300, 0.13, chosenMap, "yellow",
this.tail, 5); // Special forces.
                break;

            case 2:
                return new enemyVehicle(400, 0.14, chosenMap, "gray",
this.tail, 2, 100); // Truck.
                break;

            case 3:
                return new enemyVehicle(500, 0.15, chosenMap, "blue",
this.tail, 2, 125); // APC.
                break;

            case 4:
                return new enemyVehicle(300, 0.16, chosenMap, "green",
this.tail, 2, 150); // IFV.
                break;

            case 5:
                return new enemyVehicle(350, 0.17, chosenMap, "orange",
this.tail, 1, 200); // Tank.
                break;

            case 6:
                return new enemyPlane(400, 0.18, chosenMap, "purple",
this.tail, 1, 400, false); // Helicopter.
                break;

            case 7:
```

```
                return new enemyPlane(450, 0.19, chosenMap, "lime", this.tail,
1, 500, false); // Fighter jet.
                break;

            case 8:
                return new enemyPlane(450, 0.2, chosenMap, "white", this.tail,
1, 600, false); // Bomber.
                break;

            case 9:
                return new enemyPlane(500, 0.2, chosenMap, "black", this.tail,
1, 750, true); // Stealth bomber.
                break;
        }
    }

    returnNoOfEnemies(enemy) {
        switch (enemy) {
            case 0:
                return this.soldier;
                break;

            case 1:
                return this.specialForces;
                break;

            case 2:
                return this.truck;
                break;

            case 3:
                return this.apc;
                break;

            case 4:
                return this.ifv;
                break;

            case 5:
                return this.tank;
                break;
```

```
            case 6:
                return this.helicopter;
                break;

            case 7:
                return this.fighterJet;
                break;

            case 8:
                return this.bomber;
                break;

            case 9:
                return this.stealthBomber;
                break;
        }
    }
}
```

queue.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
22/08/23. This file contains the class for a queue structure.

"use strict";

class queue {

    // Constructor for the class.
    constructor() {
        this.queue = [];
        this.head = 0;
        this.tail = 0;
    }

    // Pushes a given item to the queue.
    enqueue(newItem) {
        if (this.tail!=this.queue.length) {
            return false;
        } else {
            this.queue[this.tail] = newItem;
            this.tail++;
        }
```

```
    }

    // Removes the item at the start of the queue.
    dequeue() {
        if (this.head==this.tail) {
            return false;
        } else {
            let item = this.queue[this.head];
            this.head++;
            return item;
        }
    }

    // Clears the queue.
    clear() {
        this.queue = [];
        this.head = 0;
        this.tail = 0;
    }
}
```

campaignSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
10/11/23. This contains the setup for the campaign gamemode.

"use strict";

let chosenLevel = -1;

let campaignCheck = false;

let levelOneButton, levelTwoButton, levelThreeButton;

let levels = [[0, 0, 0], [0, 0, 0], [0, 0, 0]];
levels[0][0] = new level(800, 5, 3, 2, 1, 0, 0, 0, 0, 0, 0);
levels[0][1] = new level(900, 8, 6, 4, 3, 3, 1, 0, 0, 0, 0);
levels[0][2] = new level(1000, 12, 8, 8, 6, 4, 2, 1, 0, 0, 0);
levels[1][0] = new level(800, 5, 3, 2, 1, 0, 0, 0, 0, 0, 0);
levels[1][1] = new level(900, 12, 8, 4, 3, 3, 1, 0, 0, 0, 0);
levels[1][2] = new level(1000, 15, 10, 5, 5, 3, 2, 1, 0, 0, 0);
levels[2][0] = new level(800, 5, 3, 2, 1, 0, 0, 0, 0, 0, 0);
levels[2][1] = new level(900, 7, 8, 10, 5, 2, 1, 0, 0, 0, 0);
```

```
levels[2][2] = new level(1000, 8, 10, 12, 8, 3, 1, 2, 0, 0, 0);

// Declares all the buttons seen on the skill tree screen.
function campaignButtonsSetup() {
    levelOneButton = createButton("Level One");
    levelOneButton.position(140, 125);
    levelOneButton.class("campaignButtonClass");
    levelOneButton.mousePressed(() => campaignButtonsFunction(0));

    levelTwoButton = createButton("Level Two");
    levelTwoButton.position(340, 125);
    levelTwoButton.class("campaignButtonClass");
    levelTwoButton.mousePressed(() => campaignButtonsFunction(1));

    levelThreeButton = createButton("Level Three");
    levelThreeButton.position(540, 125);
    levelThreeButton.class("campaignButtonClass");
    levelThreeButton.mousePressed(() => campaignButtonsFunction(2));
}

// Shows or hides the campaign buttons depending on the input.
function campaignButtonsDisplay(display) {
    if (display) {
        levelOneButton.show();
        levelTwoButton.show();
        levelThreeButton.show();
        displayStarRatings();
    } else {
        levelOneButton.hide();
        levelTwoButton.hide();
        levelThreeButton.hide();
    }
}

// Displays the users previous best attempts at each level.
function displayStarRatings() {
    for (let i = 0; i<3; i++) {
        let rating = levels[chosenMapInt][i].rating;
        if (rating == 0) {
            text("⚫", 145 + (200 * i), 240)
            text("⚫", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
```

```
        } else if (rating == 1) {
            text("🟡", 145 + (200 * i), 240)
            text("⚫", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 2) {
            text("🟡", 145 + (200 * i), 240)
            text("🟡", 195 + (200 * i), 240)
            text("⚫", 245 + (200 * i), 240)
        } else if (rating == 3) {
            text("🟡", 145 + (200 * i), 240)
            text("🟡", 195 + (200 * i), 240)
            text("🟡", 245 + (200 * i), 240)
        }
    }
}

// Changes the value of chosenMap depending on the button the user clicks
function campaignButtonsFunction(level) {
    chosenLevel = level;
}

// Updates the enemies for the campaign gamemode
function enemyUpdateCampaign() {
    for (let i = 0; i < 10; i++) {
        for (let j = 0; j < enemies[i].queue.length; j++) {
            enemies[i].timeElapsed += deltaTime;
            if(enemies[i].timeElapsed > enemies[i].spawnRate &&
enemies[i].tail < levels[chosenMapInt][chosenLevel].returnNoOfEnemies(i)) {
                enemies[i].tail++;
                enemies[i].timeElapsed = 0;
            }
        }
    }
    enemyUpdate();
}
```

enemySetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
23/06/23. This file contains the setup information (queues) for the enemies.

"use strict";
```

```
let enemies = [];
enemies[0] = new enemyQueue(15000, 0);
enemies[1] = new enemyQueue(18000, 1);
enemies[2] = new enemyQueue(20000, 2);
enemies[3] = new enemyQueue(25000, 3);
enemies[4] = new enemyQueue(30000, 4);
enemies[5] = new enemyQueue(30000, 5);
enemies[6] = new enemyQueue(50000, 6);
enemies[7] = new enemyQueue(60000, 7);
enemies[8] = new enemyQueue(70000, 8);
enemies[9] = new enemyQueue(80000, 9);

let arcade1 = false, arcade2 = false, arcade3 = false;

// Spawns all types of enemy in one function to keep modularity.
function enemySpawn() {
    for (let i = 0; i < 10; i++) {
        enemies[i].spawn();
    }
}


// Updates all types of enemy in one function to keep modularity.
function enemyUpdate() {
    for (let i = 0; i < 10; i++) {
        enemies[i].updatePosistion();
    }
}

// Clears all the enemies on screen.
function enemyClear() {
    for (let i = 0; i < 10; i++) {
        enemies[i].clear();
    }
}

// Sets up the enemies depending on the input (for each gamemode)
function enemySetup(gamemode) {
    switch (gamemode) {
        case 0:
            if (!(campaignCheck)) {
                campaignCheck = true;
```

```
            let userLevel = levels[chosenMapInt][chosenLevel];
            userLevel.setEnemies();
        }
        break;

    case 1:
        if(playerScore > 1000000 && !(arcade3)) {
            arcade3 = true;
            enemies[0].spawnRate = 2000;
            enemies[1].spawnRate = 5000;
            enemies[2].spawnRate = 8000;
            enemies[3].spawnRate = 8000;
            enemies[4].spawnRate = 10000;
            enemies[5].spawnRate = 15000;
            enemies[6].spawnRate = 20000;
            enemies[7].spawnRate = 25000;
            enemies[8].spawnRate = 25000;
            enemies[9].spawnRate = 30000;
        } else if (playerScore > 500000 && !(arcade2)) {
            enemies[0].spawnRate = 5000;
            enemies[1].spawnRate = 8000;
            enemies[2].spawnRate = 8000;
            enemies[3].spawnRate = 10000;
            enemies[4].spawnRate = 15000;
            enemies[5].spawnRate = 20000;
            enemies[6].spawnRate = 30000;
            enemies[7].spawnRate = 40000;
            enemies[8].spawnRate = 50000;
            enemies[9].spawnRate = 50000;
        } else if (playerScore > 100000 && !(arcade1))  {
            enemies[0].spawnRate = 5000;
            enemies[1].spawnRate = 5000;
            enemies[2].spawnRate = 8000;
            enemies[3].spawnRate = 10000;
            enemies[4].spawnRate = 15000;
            enemies[5].spawnRate = 20000;
            enemies[6].spawnRate = 30000;
            enemies[7].spawnRate = 40000;
            enemies[8].spawnRate = 50000
            enemies[9].spawnRate = 60000;
        }
        break;
```

```
    }
}
```

gameSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on 19/06/23. This file contains all the variables and functions needed to draw the map.

"use strict";

const CELL_SIZE = 96;
const MAP_HEIGHT = 864;
const MAP_WIDTH = 1632;

const MAP_TILE_X = [];
const MAP_TILE_Y = [];

// Fills the arrays declared above with the x and y posistions where each tile starts to make referencing them easier and clearer.
for (let i = 0; i < 18; i++) {
    MAP_TILE_X[i] = i * CELL_SIZE;
    if (i < 10) {
        MAP_TILE_Y[i] = i * CELL_SIZE;
    }
}

const MAP_VIETNAM = [
  [0,0,0,0,1,0,0,0,0],
  [0,3,3,0,1,1,1,0,0],
  [0,3,3,0,0,0,1,0,0],
  [0,0,0,0,0,0,1,0,0],
  [0,1,1,1,1,1,1,0,0],
  [0,1,0,0,0,0,0,0,0],
  [0,1,0,3,0,0,0,0,0],
  [0,1,0,3,0,1,1,1,0],
  [0,1,0,3,0,1,0,1,0],
  [0,1,0,3,0,1,0,1,0],
  [0,1,0,0,0,1,0,1,0],
  [0,1,1,1,1,1,0,1,0],
  [0,0,0,0,0,0,0,1,0],
  [0,0,0,0,0,0,0,1,0],
  [0,3,3,0,0,1,1,1,0],
```

```
  [0,3,3,0,0,1,0,0,0],
  [0,0,0,0,0,2,0,0,0],
];

const MAP_STALINGRAD = [
  [0,0,0,0,1,0,0,0,0],
  [0,0,0,0,1,0,3,3,3],
  [0,1,1,1,1,0,3,3,3],
  [0,1,0,0,0,0,3,3,3],
  [0,1,1,0,0,0,3,3,3],
  [0,0,1,0,0,0,3,3,3],
  [0,0,1,1,1,0,0,0,0],
  [0,0,0,0,1,0,0,0,0],
  [3,3,3,0,1,1,1,0,0],
  [0,0,0,0,0,0,1,0,0],
  [0,1,1,1,1,3,1,0,0],
  [0,1,0,0,1,3,1,0,0],
  [0,1,0,0,1,1,1,0,0],
  [0,1,0,0,0,0,0,0,3],
  [0,1,3,3,0,0,0,0,3],
  [0,1,1,1,1,1,0,0,3],
  [0,0,0,0,0,2,0,0,3],
];

const MAP_GULFWAR = [
  [0,0,0,0,1,0,3,3,3],
  [0,0,0,0,1,0,3,3,3],
  [0,0,0,0,1,0,0,0,0],
  [0,0,0,0,1,1,0,0,0],
  [0,0,0,0,0,1,0,0,0],
  [3,3,0,0,0,1,0,0,0],
  [0,0,0,1,1,1,0,0,0],
  [0,0,0,1,0,0,0,3,3],
  [3,0,0,1,0,0,0,0,0],
  [3,0,0,1,1,1,1,1,0],
  [3,0,0,0,0,3,3,1,0],
  [3,0,0,0,0,3,3,1,0],
  [0,0,1,1,1,1,1,1,0],
  [0,0,1,0,0,0,0,0,0],
  [0,0,1,0,0,0,0,0,0],
  [0,0,1,1,1,1,0,0,0],
  [0,0,0,0,0,2,0,0,0],
```

```
];

const MAP_BLANK = [
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
];

let chosenMap = [
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0],
  ];
```

```
// Draws the map including the lines and filling in the tiles for the map.
function drawMap() {

  // Drawing the lines to create the tiles.
  stroke("black");
  for (let i = 0; i < MAP_WIDTH+CELL_SIZE; i+=CELL_SIZE) {
    line(i, 0, i, MAP_HEIGHT);
    if (i < MAP_HEIGHT+CELL_SIZE) {
        line(0, i, MAP_WIDTH, i);
    }
  }

  // Filling in the tiles needed.
  if (JSON.stringify(chosenMap)==JSON.stringify(MAP_VIETNAM)) {
    for (let i = 0; i < MAP_VIETNAM.length; i++) {
      for (let j = 0; j < MAP_VIETNAM[0].length; j++) {
        if (MAP_VIETNAM[i][j] == 0) {
          fill(3, 38, 11);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if (MAP_VIETNAM[i][j] == 1) {
          fill(66, 66, 32);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if (MAP_VIETNAM[i][j] == 2) {
          fill("black");
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        } else if (MAP_VIETNAM[i][j] == 3) {
          fill(69, 69, 69);
          rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

        }
      }
    }

  } else if (JSON.stringify(chosenMap)==JSON.stringify(MAP_STALINGRAD)) {
    for (let i = 0; i < MAP_STALINGRAD.length; i++) {
      for (let j = 0; j < MAP_STALINGRAD[0].length; j++) {
        if (MAP_STALINGRAD[i][j] == 0) {
          fill(168, 168, 168);
```

```
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      } else if (MAP_STALINGRAD[i][j] == 1) {
        fill(66, 66, 32);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      } else if (MAP_STALINGRAD[i][j] == 2) {
        fill("black");
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      } else if (MAP_STALINGRAD[i][j] == 3) {
        fill(69, 69, 69);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      }
    }
  }

} else if (JSON.stringify(chosenMap)==JSON.stringify(MAP_GULFWAR)) {
  for (let i = 0; i < MAP_GULFWAR.length; i++) {
    for( let j = 0; j < MAP_GULFWAR[0].length; j++) {
      if (MAP_GULFWAR[i][j] == 0) {
        fill(255, 255, 111);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      } else if (MAP_GULFWAR[i][j] == 1) {
        fill(66, 66, 32);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      } else if (MAP_GULFWAR[i][j] == 2) {
        fill("black");
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      } else if (MAP_GULFWAR[i][j] == 3) {
        fill(36, 162, 22);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);

      }
    }
  }
}
```

```
  // If the user wishes to place a tower, this screen is displayed on top of
the map.
  if (towerPlace) {
    let colourFill;
    for (let i = 0; i < mapFilled.length; i++) {
      for (let j = 0; j < mapFilled[0].length; j++) {
        if (mapFilled[i][j] == 0) {
          colourFill = color(67, 198, 11, 97);
        } else {
          colourFill = color(212, 0, 0, 98);
        }
        fill(colourFill);
        rect(MAP_TILE_X[i], MAP_TILE_Y[j], CELL_SIZE, CELL_SIZE);
      }
    }
  }

  if(chosenMapInt == 2) {
    gulfWarSpecialAbility();
  }
}

// Displays the user score, cash and battle medals.
function displayInformation() {
  fill("black");
  textSize(25);
  text("🔢Score: " + playerScore, 1640, 100);
  text("💵Cash: " + playerCash, 1640, 140);
  text("🎖️Medals: " + playerBattleMedals, 1640, 180);
}

// Displays the base health as a gradient and ends the game if the health is
zero.
function displayBaseHealth(health, max) {
  text("❤️Base Health:", 1640, 220);

  let percentage = health/max; // Percentage of original health remaining.
  let start = color(255, 0, 0);
  let end = color(0, 255, 0);
  let gradientColour = lerpColor(start, end, percentage); // Colours the
health bar a percentage between red and green depending on the health
remaining.
```

```
  let barWidth = map(percentage, 0, 1, 0, 205); // Makes the size of the
health bar proportional to the health remaining.

  stroke("black");
  rect(1640, 230, 205, 30);

  if (health > 0) {
    fill(gradientColour);
    noStroke();
    rect(1640, 230, barWidth, 30);
  }
}

let vietnamMapButton, stalingradMapButton, gulfwarMapButton;

let vietnamMapImage, stalingradMapImage, gulfwarMapImage;

// Declares all the buttons which control the map selection.
function mapButtonsSetup() {
  vietnamMapButton = createButton("Vietnam");
  vietnamMapButton.position(200, 550);
  vietnamMapButton.class("mainMenuButtonClass");
  vietnamMapButton.mousePressed(vietnamMapButtonFunction);

  stalingradMapButton = createButton("Stalingrad");
  stalingradMapButton.position(700, 550);
  stalingradMapButton.class("mainMenuButtonClass");
  stalingradMapButton.mousePressed(stalingradMapButtonFunction);

  gulfwarMapButton = createButton("Gulf War");
  gulfwarMapButton.position(1200, 550);
  gulfwarMapButton.class("mainMenuButtonClass");
  gulfwarMapButton.mousePressed(gulfwarMapButtonFunction);
}

// Toggles the map selection buttons on and off depending on the input.
function mapButtonsDisplay(display) {
  if (display == true) {
    textSize(30);
    fill("black");
    text("Choose your map:", 770, 100);
    vietnamMapButton.show();
```

```
    stalingradMapButton.show();
    gulfwarMapButton.show();
    image(vietnamMapImage, 200, 200, 400, 300);
    image(stalingradMapImage, 700, 200, 400, 300);
    image(gulfwarMapImage, 1200 , 200, 400, 300);
  } else if (display == false) {
    vietnamMapButton.hide();
    stalingradMapButton.hide();
    gulfwarMapButton.hide();
  }
}

// When the Vietnam themed map is clicked in the menu, this is changed to the
users chosen map.
function vietnamMapButtonFunction() {
  chosenMap = MAP_VIETNAM;
  chosenMapInt = 0;

  for (let i = 0; i < mapFilled.length; i++) {
    for (let j = 0; j < mapFilled[0].length; j++) {
      if (chosenMap[i][j] == 1 || chosenMap[i][j] == 2 || chosenMap[i][j] ==
3) {
        mapFilled[i][j] = 1;
      }
    }
  }
}

// When the Stalingrad themed map is clicked in the menu, this is changed to
the users chosen map.
function stalingradMapButtonFunction() {
  chosenMap = MAP_STALINGRAD;
  chosenMapInt = 1;

  for (let i = 0; i < mapFilled.length; i++) {
    for (let j = 0; j < mapFilled[0].length; j++) {
      if (chosenMap[i][j] == 1 || chosenMap[i][j] == 2 || chosenMap[i][j] ==
3) {
        mapFilled[i][j] = 1;
      }
    }
  }
```

```
}

// When the Gulf War themed map is clicked in the menu, this is changed to the
user's chosen map.
function gulfwarMapButtonFunction() {
  chosenMap = MAP_GULFWAR;
  chosenMapInt = 2;

  for (let i = 0; i < mapFilled.length; i++) {
    for (let j = 0; j < mapFilled[0].length; j++) {
      if (chosenMap[i][j] == 1 || chosenMap[i][j] == 2 || chosenMap[i][j] ==
3) {
        mapFilled[i][j] = 1;
      }
    }
  }
}

// If an enemy is killed, the player score is edited accordingly using the
function below
function addScoreAndCash(type) {
  playerScore += (type + 1) * 500;
  playerCash += (type + 1) * 25;
}

// Checks if the base still has health and resets the game if it doesn't
function checkBaseHealth() {
  if (baseHealth < 0) {
    enemyClear();
    baseHealth = baseHealthMax;
    mapFilled = [
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
```

```
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0],
      ];

    for (let i = 0; i < mapFilled.length; i++) {
      for (let j = 0; j < mapFilled[0].length; j++) {
        if (chosenMap[i][j] == 1 || chosenMap[i][j] == 2 || chosenMap[i][j] ==
3) {
          mapFilled[i][j] = 1;
        }
      }
    }
    placeCancelButtonFunction();
    changeTowerColour(false, 0);
    changeTowerColour(true, 10);
    TRAP.placed = false;
    towers.length = 0;
    specialForces.length = 0;
    allowAbility = true;
    airstrikeCheck = false;
    airstrikeCheckCoordCheck = false;
    campaignCheck = false;
    let tempText = "The game is over. Your score was " +
playerScore.toString() + ". Press 'Ok' to restart or press 'Cancel' to return
to the menu and/or select another map.";
    playerBattleMedals += Math.floor(playerScore / 50000);
    playerScore = 0;
    playerCash = 500;
    baseHealth = baseHealthMax;
    if (!(confirm(tempText))) {
      chosenMap = [
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0],
```

```
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        ];
      mapFilled = [
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0],
        ];
      gameState = 1;
      chosenMapInt = -1;
      chosenLevel = -1;
    }
  }
}

// Resets the base health relative to the upgraded value
function setBaseHealth(health) {
  baseHealth = health * (1 + upgradePercent(baseHealthUpgrade));
  baseHealthMax = health * (1 + upgradePercent(baseHealthUpgrade));
}
```

masterSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
14/06/23. This file sets up all the most important global variables and
subroutines.

"use strict";

let gameState = 1;

let playerScore = 0;
let playerCash = 500;
let playerBattleMedals = 50;

let baseHealthMax = 1000;
let baseHealth = baseHealthMax;

let towerPlace = false;

let chosenMapInt;

// Returns a random integer between a min (inclusive) and max (exclusive)
using the Javascript built-in Math library.
function randomInteger(min, max) {
    return Math.random() * (max - min) + min;
}

// Returns the distance between 2 given points.
function distance(x1, y1, x2, y2) {
    return Math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2);
}

// Linear interpolates between a given number using the third parameter.
function linearInterpolate(min, max, ratio) {
    ratio = Math.max(0, Math.min(ratio, 1)); // Ensures ratio is between 0 and
1 to allow ease of multiplication.
    return (max - min) * ratio + min;
}

// Cosine interpolation using linear interpolation to make the animation
smoother.
function cosineInterpolate(min, max, ratio) {
    angleMode(RADIANS);
```

```
    return linearInterpolate(min, max, (1 - cos(ratio*PI)) / 2);
}
```

menuSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
17/06/23. This file sets up all the variables and functions needed for the
main menu

"use strict";

let returnToMenuButton, campaignButton, arcadeButton, tutorialButton,
skillTreeButton, settingsButton;

let mainMenuImage;

// Displays the title and image on the main menu.
function titleAndImageSetup() {

    // Creating title.
    textFont("Helvetica");
    textSize(60);
    fill(0, 0, 0);
    text("Military Tower Defence Game", 550, 150);

    image(mainMenuImage, 0, 686); // Image of tank firing shell at the bottom.
}

// Declares all the buttons seen on the main menu screen.
function menuButtonsSetup() {
    returnToMenuButton = createButton("Menu");
    returnToMenuButton.position(1640, 15);
    returnToMenuButton.id("returnToMenuButtonID");
    returnToMenuButton.mousePressed(returnToMenuButtonFunction);

    campaignButton = createButton("Campaign");
    campaignButton.position(740, 225);
    campaignButton.class("mainMenuButtonClass");
    campaignButton.mousePressed(campaignButtonFunction);

    arcadeButton = createButton("Arcade");
    arcadeButton.position(740, 325);
    arcadeButton.class("mainMenuButtonClass");
```

```
    arcadeButton.mousePressed(arcadeButtonFunction);

    tutorialButton = createButton("Tutorial");
    tutorialButton.position(740, 425);
    tutorialButton.class("mainMenuButtonClass");
    tutorialButton.mousePressed(tutorialButtonFunction);

    skillTreeButton = createButton("Skill Tree");
    skillTreeButton.position(740, 525);
    skillTreeButton.class("mainMenuButtonClass");
    skillTreeButton.mousePressed(skillTreeButtonFunction);

    settingsButton = createButton("Settings");
    settingsButton.position(740, 625);
    settingsButton.class("mainMenuButtonClass");
    settingsButton.mousePressed(settingsButtonFunction);
}

// Shows or hides the return to menu button depending on the input
function returnToMenuButtonDisplay(display) {
    if (display) {
        returnToMenuButton.show();
    } else {
        returnToMenuButton.hide();
    }
}

// Shows or hides the menu buttons depending on the input.
function mainMenuButtonsDisplay(display) {
    if (display) {
        campaignButton.show();
        arcadeButton.show();
        tutorialButton.show();
        skillTreeButton.show();
        settingsButton.show();
    } else {
        campaignButton.hide();
        arcadeButton.hide();
        tutorialButton.hide();
        skillTreeButton.hide();
        settingsButton.hide();
    }
```

```
}

// When the menu button is clicked on any part of the game, the menu loads
(because gameState is changed to 1).
function returnToMenuButtonFunction() {
    if ((gameState == 2 || gameState == 3 || gameState == 4) && confirm("Are
you sure you want to return to menu?")) {
        enemyClear();
        chosenMap = [
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            ];
        mapFilled = [
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
```

```
            [0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0],
            ];
        allowAbility = true;
        towers.length = 0;
        specialForces.length = 0;
        towerPlace = false;
        baseHealth = baseHealthMax;
        chosenMapInt = -1;
        chosenLevel = -1;
        campaignCheck = false;
        placeCancelButtonFunction();
        changeTowerColour(false, 0);
        changeTowerColour(true, 10);
        TRAP.placed = false;
        playerBattleMedals += Math.floor(playerScore / 50000);
        playerScore = 0;
        playerCash = 500;
        gameState = 1;
    } else if (gameState == 5 || gameState == 6) {
        gameState = 1;
    }
}

// When the campaign button is clicked on the main menu, the campaign loads
(because gameState is changed to 2).
function campaignButtonFunction() {
    gameState = 2;
}

// When the arcade button is clicked on the main menu, the arcade loads
(because gameState is changed to 3).
function arcadeButtonFunction() {
    gameState = 3;
}

// When the tutorial button is clicked on the main menu, the tutorial loads
(because gameState is changed to 4).
function tutorialButtonFunction() {
    gameState = 4;
}
```

```
// When the skill tree button is clicked on the main menu, the skill tree
loads (because gameState is changed to 5).
function skillTreeButtonFunction() {
    gameState = 5;
}

// When the settings button is clicked on the main menu, the settings load
(because gameState is changed to 6).
function settingsButtonFunction() {
    gameState = 6;
}
```

skillTreeSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
26/10/23. This file sets up the "Skill Tree" option in the menu.

"use strict";

let projectileUpgrade = 0, damageUpgrade = 0, baseHealthUpgrade = 0,
enemySpeedUgrade = 0, enemyArmourUpgrade = 0, specialAbilityUpgrade = 0;

let projectileUpgradeButton, damageUpgradeButton, baseHealthUpgradeButton,
enemySpeedUpgradeButton, enemyArmourUpgradeButton,
specialAbilityUpgradeButton;

// Declares all the buttons seen on the skill tree screen.
function skillTreeButtonsSetup() {
    projectileUpgradeButton = createButton("Upgrade Tower Projectile Speed");
    projectileUpgradeButton.position(740, 125);
    projectileUpgradeButton.class("skillTreeButtonClass");
    projectileUpgradeButton.mousePressed(() =>
skillTreeButtonsFunction(projectileUpgrade, 0));

    damageUpgradeButton = createButton("Upgrade Tower Projectile Damage");
    damageUpgradeButton.position(740, 225);
    damageUpgradeButton.class("skillTreeButtonClass");
    damageUpgradeButton.mousePressed(() =>
skillTreeButtonsFunction(damageUpgrade, 1));

    baseHealthUpgradeButton = createButton("Upgrade Base Health");
    baseHealthUpgradeButton.position(740, 325);
```

```
    baseHealthUpgradeButton.class("skillTreeButtonClass");
    baseHealthUpgradeButton.mousePressed(() =>
skillTreeButtonsFunction(baseHealthUpgrade, 2));

    enemySpeedUpgradeButton = createButton("Decrease All Enemy Speed");
    enemySpeedUpgradeButton.position(740, 425);
    enemySpeedUpgradeButton.class("skillTreeButtonClass");
    enemySpeedUpgradeButton.mousePressed(() =>
skillTreeButtonsFunction(enemySpeedUgrade, 3));

    enemyArmourUpgradeButton = createButton("Decrease All Enemy Armour");
    enemyArmourUpgradeButton.position(740, 525);
    enemyArmourUpgradeButton.class("skillTreeButtonClass");
    enemyArmourUpgradeButton.mousePressed(() =>
skillTreeButtonsFunction(enemyArmourUpgrade, 4));

    specialAbilityUpgradeButton = createButton("Upgrade All Special
Abilities");
    specialAbilityUpgradeButton.position(740, 625);
    specialAbilityUpgradeButton.class("skillTreeButtonClass");
    specialAbilityUpgradeButton.mousePressed(() =>
skillTreeButtonsFunction(specialAbilityUpgrade, 5));
}

// Shows or hides the skill tree buttons depending on the input.
function skillTreeButtonsDisplay(display) {
    if (display) {
        projectileUpgradeButton.show();
        damageUpgradeButton.show();
        baseHealthUpgradeButton.show();
        enemySpeedUpgradeButton.show();
        enemyArmourUpgradeButton.show();
        specialAbilityUpgradeButton.show();
        displayInformationSkillTree();
    } else {
        projectileUpgradeButton.hide();
        damageUpgradeButton.hide();
        baseHealthUpgradeButton.hide();
        enemySpeedUpgradeButton.hide();
        enemyArmourUpgradeButton.hide();
        specialAbilityUpgradeButton.hide();
    }
```

```
}

function displayInformationSkillTree() {
    noFill();
    stroke("black");
    for(let i=0; i<6; i++) {
        rect(640, 125 + 100*i, 75, 75);
        rect(1165, 125 + 100*i, 75, 75);
    }
    fill("black");
    textSize(35);
    text(100*upgradePercent(projectileUpgrade) + "%", 645, 175);
    text(100*upgradePercent(damageUpgrade) + "%", 645, 275);
    text(100*upgradePercent(baseHealthUpgrade) + "%", 645, 375);
    text(100*upgradePercent(enemySpeedUgrade) + "%", 645, 475);
    text(100*upgradePercent(enemyArmourUpgrade) + "%", 645, 575);
    text(100*upgradePercent(specialAbilityUpgrade) + "%", 645, 675);
    text(upgradeCost(projectileUpgrade), 1170, 175);
    text(upgradeCost(damageUpgrade), 1170, 275);
    text(upgradeCost(baseHealthUpgrade), 1170, 375);
    text(upgradeCost(enemySpeedUgrade), 1170, 475);
    text(upgradeCost(enemyArmourUpgrade), 1170, 575);
    text(upgradeCost(specialAbilityUpgrade), 1170, 675);
    text("🎖️Medals: " + playerBattleMedals, 1170, 75);
    textSize(20);
    text("Currently: ", 640, 100);
    text("Cost: ", 1165, 100);
}

// Returns the percentage upgrade depending on the tier of the upgrade
variable passed.
function upgradePercent(upgrade) {
    switch(upgrade) {
        case 0:
            return 0;
            break;

        case 1:
            return 0.02;
            break;

        case 2:
```

```
            return 0.05;
            break;

        case 3:
            return 0.08;
            break;

        case 4:
            return 0.12;
            break;

        case 5:
            return 0.18;
            break;
    }
}

// Returns the cost to upgrade depending on the tier of the upgrade varaible
passed.
function upgradeCost(upgrade) {
    switch(upgrade) {
        case 0:
            return "1 🎖️";
            break;

        case 1:
            return "2 🎖️";
            break;

        case 2:
            return "3 🎖️";
            break;

        case 3:
            return "4 🎖️";
            break;

        case 4:
            return "5 🎖️";
            break;

        case 5:
```

```
            return "Max";
            break;
    }
}

// Edits the variable depending on the upgrade passed.
function skillTreeButtonsFunction(upgrade, type) {
    if(upgrade < 5 && playerBattleMedals >= upgrade + 1) {
        playerBattleMedals -= upgrade + 1;
        switch(type) {
            case 0:
                projectileUpgrade++;
                break;

            case 1:
                damageUpgrade++;
                break;

            case 2:
                baseHealthUpgrade++;
                break;

            case 3:
                enemySpeedUgrade++;
                break;

            case 4:
                enemyArmourUpgrade++;
                break;

            case 5:
                specialAbilityUpgrade++;
                break;
        }
    }
}
```

specialAbilitySetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
17/11/23. This file contains the setup information for the special abilities.
```

```
"use strict";

let specialAbilityButton;

let allowAbility = true;

let placeTrap = false;
const TRAP = {
    posX : 0,
    posY : 0,
    placed : false
};
let trapFrameCount = 0;

let specialForces = [];
let specialForcesFrameCount = 0;

let airStrikeFrameCount = 0;
let airstrikeCheck = false;
let airstrikeX = 0, airstrikeY = 0;
let airstrikeCheckCoordCheck = false;

// Sets up the button to call the special ability.
function specialAbilityButtonSetup() {
    specialAbilityButton = createButton(" ");
    specialAbilityButton.position(1750, 780);
    specialAbilityButton.class("towerRangeSpecialAbility");
    specialAbilityButton.mousePressed(() =>
specialAbilityButtonFunction(chosenMapInt));
}

// Shows or hides the special ability buttons depending on the input.
function specialAbilityButtonDisplay(display) {
    if (display) {
        specialAbilityButton.show();
    } else {
        specialAbilityButton.hide();
    }
}

// Function called when the special ability button is clicked.
function specialAbilityButtonFunction(map) {
```

```
    if(map == 0 && allowAbility) {
        changeTowerColour(true, 8);
        vietnamSpecialAbility();
    } else if (map == 1 && allowAbility) {
        changeTowerColour(true, 8);
        stalingradSpecialAbility();
    } else if (map == 2 && allowAbility) {
        changeTowerColour(true, 8);
        gulfWarSpecialAbility();
    }
}

// The special ability for the Vietnam Map.
function vietnamSpecialAbility() {
    placeTrap = true;
}

// Places the trap.
function showTrap() {
   if (placeTrap && allowAbility && mouseIsPressed && mouseX >= MAP_TILE_X[0]
&& mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <=
MAP_TILE_Y[9]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (chosenMap[tileX][tileY] == 1) {
            TRAP.placed = true;
            TRAP.posX = Math.floor(mouseX / CELL_SIZE);
            TRAP.posY = Math.floor(mouseY / CELL_SIZE);
            trapFrameCount = frameCount;
            changeTowerColour(true, 9);
            placeTrap = false;
        }
    }

    if(TRAP.placed) {
        drawTrap();
    }
}

// Uses the trap.
function trapFunction() {
    if (TRAP.placed && allowAbility) {
```

```
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);
                if(enemyX - TRAP.posX == 0 && enemyY - TRAP.posY == 0) {
                    enemies[i].queue[j].speed = 0;
                }
            }
        }
    }

    if (TRAP.placed && allowAbility && frameCount - trapFrameCount > 300) {
        TRAP.placed = false;
        allowAbility = false;
    }
}

// Displays the trap on screen
function drawTrap() {
    for(let i = 0; i < 88; i+=22) {
        for(let j = 0; j < 88; j+=22) {
            fill("gray");
            circle(TRAP.posX*CELL_SIZE + 15 + i, TRAP.posY*CELL_SIZE + 15 + j,
20);
            fill(3, 38, 11);
            circle(TRAP.posX*CELL_SIZE + 15 + i, TRAP.posY*CELL_SIZE + 15 + j,
10);
        }
    }
}

// The special ability for the Stalingrad map.
function stalingradSpecialAbility() {
    towerPlace = true;
}

// Places the special forces.
function showSpecialForces() {
    if (towerPlace && allowAbility && mouseIsPressed && mouseX >=
MAP_TILE_X[0] && mouseX <= MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY
<= MAP_TILE_Y[9]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
```

```
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (mapFilled[tileX][tileY] == 0) {
            specialForces[0] = new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "black", 8, 240, 2, 0.2);
            specialForces[1] = new towerBasic(MAP_TILE_X[tileX] + 20,
MAP_TILE_Y[tileY] + 20, "black", 8, 240, 2, 0.2);
            specialForces[2] = new towerBasic(MAP_TILE_X[tileX] + 20,
MAP_TILE_Y[tileY] - 20, "black", 8, 240, 2, 0.2);
            specialForces[3] = new towerBasic(MAP_TILE_X[tileX] - 20,
MAP_TILE_Y[tileY] + 20, "black", 8, 240, 2, 0.2);
            specialForces[4] = new towerBasic(MAP_TILE_X[tileX] - 20,
MAP_TILE_Y[tileY] - 20, "black", 8, 240, 2, 0.2);
            allowAbility = false;
            changeTowerColour(true, 9);
            towerPlace = false;
            mapFilled[tileX][tileY] = 1;
            specialForcesFrameCount = frameCount;
        }
    }

    if (specialForces.length > 0 && frameCount - specialForcesFrameCount >
900) {
        let x = specialForces[0].posX / CELL_SIZE;
        let y = specialForces[0].posY / CELL_SIZE;
        mapFilled[x][y] = 0;
        specialForces.length = 0;
    }
}

// Displays the special forces once placed.
function specialForcesUpdate() {
    for (let tower of specialForces) {
        tower.update();
    }
}

// The special ability for the Gulf War map.
function gulfWarSpecialAbility(){
    if (allowAbility && mouseIsPressed && mouseX >= MAP_TILE_X[0] && mouseX <=
MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <= MAP_TILE_Y[9] ) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
```

```
        for (let i = 0; i < 10; i++){
            for (let j = enemies[i].head; j < enemies[i].tail; j++) {
                let enemyX = Math.floor(enemies[i].queue[j].posX/CELL_SIZE);
                let enemyY = Math.floor(enemies[i].queue[j].posY/CELL_SIZE);

                if(Math.abs(enemyX - tileX) < 2 && Math.abs(enemyY - tileY) <
2) {
                    airStrikeFrameCount = frameCount;
                    airstrikeCheck = true;
                    enemies[i].queue[j].health -= 600;
                    allowAbility = false;
                    changeTowerColour(true, 9);
                }
            }
        }
    }
}

// Displays an explosion on the screen.
function showAirstrike() {
    if(frameCount - airStrikeFrameCount < 100 && airstrikeCheck &&
!(airstrikeCheckCoordCheck)) {
        airstrikeX = mouseX;
        airstrikeY = mouseY;
        airstrikeCheckCoordCheck = true;
        fill("orange");
        circle(airstrikeX + 10, airstrikeY + 10, 60);
        circle(airstrikeX - 10, airstrikeY - 10, 80);
        circle(airstrikeX + 30, airstrikeY + 30, 40);
        circle(airstrikeX, airstrikeY, 50);
        fill("red");
        circle(airstrikeX + 15, airstrikeY + 10, 30);
        circle(airstrikeX - 10, airstrikeY - 10, 60);
        circle(airstrikeX - 65, airstrikeY + 25, 30);
        circle(airstrikeX - 45, airstrikeY + 45, 30);
        circle(airstrikeX, airstrikeY, 30);
        fill("yellow");
        circle(airstrikeX + 5, airstrikeY + 20, 50);
        circle(airstrikeX + 20, airstrikeY - 40, 40);
        circle(airstrikeX - 45, airstrikeY - 35, 25);
        circle(airstrikeX - 20, airstrikeY + 30, 10);
```

```
    } else if(frameCount - airStrikeFrameCount < 100 && airstrikeCheck) {
        fill("orange");
        circle(airstrikeX + 10, airstrikeY + 10, 60);
        circle(airstrikeX - 10, airstrikeY - 10, 80);
        circle(airstrikeX + 30, airstrikeY + 30, 40);
        circle(airstrikeX, airstrikeY, 50);
        fill("red");
        circle(airstrikeX + 15, airstrikeY + 10, 30);
        circle(airstrikeX - 10, airstrikeY - 10, 60);
        circle(airstrikeX - 65, airstrikeY + 25, 30);
        circle(airstrikeX - 45, airstrikeY + 45, 30);
        circle(airstrikeX, airstrikeY, 30);
        fill("yellow");
        circle(airstrikeX + 5, airstrikeY + 20, 50);
        circle(airstrikeX + 20, airstrikeY - 40, 40);
        circle(airstrikeX - 45, airstrikeY - 35, 25);
        circle(airstrikeX - 20, airstrikeY + 30, 10);
    } else {
        airstrikeCheck = false;
        airStrikeFrameCount = 0;
    }
}

// Changes the text displayed on the special ability button depending on the
map.
function returnSpecialAbilityName() {
    switch(chosenMapInt) {
        case 0:
            return "Place trap";
            break;

        case 1:
            return "Call special forces";
            break;

        case 2:
            return "Call an airstrike";
            break;
    }
}
```

towerSetup.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
23/07/23. This file contains the setup information for the towers.

"use strict";

let towers = [];

let mapFilled = [
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0],
    ];


const TOWER_PRICES = [100, 250, 400, 500, 750, 900, 1200, 1500];

let placeCancelButton, displayRangeButon;

let soldierButton, machineGunnerButton, rpgButton, ifvButton, tankButton,
shoulderAntiAirButton, surfaceAirButton, advancedSurfaceAirButton;

let selectedTower;

let displayRange = false;

// Declares all the buttons the user click to place a specific type of tower.
function towerButtonsSetup() {
    placeCancelButton = createButton("Cancel");
```

```
    placeCancelButton.position(1640, 720);
    placeCancelButton.class("selectTowerButtonClass");
    placeCancelButton.mousePressed(placeCancelButtonFunction);

    soldierButton = createButton("Soldier <br> 💵100");
    soldierButton.position(1640, 300);
    soldierButton.class("towerButtonClass");
    soldierButton.mousePressed(() => towerButtonFunction(0));

    machineGunnerButton = createButton("Machine Gunner <br> 💵250");
    machineGunnerButton.position(1740, 300);
    machineGunnerButton.class("towerButtonClass");
    machineGunnerButton.mousePressed(() => towerButtonFunction(1));

    rpgButton = createButton("RPG <br> 💵400");
    rpgButton.position(1640, 400);
    rpgButton.class("towerButtonClass");
    rpgButton.mousePressed(() => towerButtonFunction(2));

    ifvButton = createButton("IFV <br> 💵500");
    ifvButton.position(1740, 400);
    ifvButton.class("towerButtonClass");
    ifvButton.mousePressed(() => towerButtonFunction(3));

    tankButton = createButton("Tank <br> 💵750");
    tankButton.position(1640, 500);
    tankButton.class("towerButtonClass");
    tankButton.mousePressed(() => towerButtonFunction(4));

    shoulderAntiAirButton = createButton("MANPADS <br> 💵900");
    shoulderAntiAirButton.position(1740, 500);
    shoulderAntiAirButton.class("towerButtonClass");
    shoulderAntiAirButton.mousePressed(() => towerButtonFunction(5));

    surfaceAirButton = createButton("Surface to Air Missile <br> 💵1200");
    surfaceAirButton.position(1640, 600);
    surfaceAirButton.class("towerButtonClass");
    surfaceAirButton.mousePressed(() => towerButtonFunction(6));

    advancedSurfaceAirButton = createButton("Advanced Surface to Air Missile
<br> 💵1500");
    advancedSurfaceAirButton.position(1740, 600);
```

```
    advancedSurfaceAirButton.class("towerButtonClass");
    advancedSurfaceAirButton.mousePressed(() => towerButtonFunction(7));

    displayRangeButon = createButton("Toggle tower ranges");
    displayRangeButon.position(1640, 780);
    displayRangeButon.class("towerRangeSpecialAbility");
    displayRangeButon.mousePressed(displayRangeButtonFunction);
}

// Place's a tower at the x and y the user clicks.
function placeTowerFunction() {
    if (towerPlace && mouseIsPressed && mouseX >= MAP_TILE_X[0] && mouseX <=
MAP_TILE_X[17] && mouseY >= MAP_TILE_Y[0] && mouseY <= MAP_TILE_Y[9]
        && playerCash >= TOWER_PRICES[selectedTower]) {
        let tileX = Math.floor(mouseX / CELL_SIZE);
        let tileY = Math.floor(mouseY / CELL_SIZE);
        if (mapFilled[tileX][tileY] == 0) {
            switch(selectedTower) {
                case 0:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "red", 0, 200, 2, 0.2)); // Soldier.
                    break;

                case 1:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "yellow", 1, 220, 2, 0.2)); // Machine gunner.
                    break;

                case 2:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "gray", 2, 240, 3, 0.19)); // RPG.
                    break;

                case 3:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "blue", 3, 260, 3, 0.23)); // IFV.
                    break;

                case 4:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "green", 4, 280, 4, 0.24)); // Tank.
                    break;
```

```
                case 5:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "orange", 5, 300, 4, 0.25)); // MANPADS.
                    break;

                case 6:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "purple", 6, 320, 5, 0.26)); // Surface Air Missile.
                    break;

                case 7:
                    towers.push(new towerBasic(MAP_TILE_X[tileX],
MAP_TILE_Y[tileY], "black", 7, 340, 6, 0.27)); // Advanced Surface Air
Missile.
                    break;
            }
            playerCash -= TOWER_PRICES[selectedTower];
            towerPlace = false;
            mapFilled[tileX][tileY] = 1;
            placeCancelButtonDisplay(false);
            changeTowerColour(false, 0);
        }
    }
}

// Function called if the user cancels the tower place.
function placeCancelButtonFunction() {
    towerPlace = false;
    changeTowerColour(false, 0)
    placeCancelButtonDisplay(false);
}

// Show or hides the tower place buttons depending on the input.
function towerButtonDisplay(display) {
    if (display) {
        soldierButton.show();
        machineGunnerButton.show();
        rpgButton.show();
        ifvButton.show();
        tankButton.show();
        shoulderAntiAirButton.show();
```

```
        surfaceAirButton.show();
        advancedSurfaceAirButton.show();

    } else {
        soldierButton.hide();
        machineGunnerButton.hide();
        rpgButton.hide();
        ifvButton.hide();
        tankButton.hide();
        shoulderAntiAirButton.hide();
        surfaceAirButton.hide();
        advancedSurfaceAirButton.hide();
    }
}

// Shows or hides the tower place cancel button depending on the input.
function placeCancelButtonDisplay(display) {
    if (display) {
        placeCancelButton.show();
    } else {
        placeCancelButton.hide();
    }
}

// Shows or hides the toggle tower range button depending on the input.
function displayRangeButtonDisplay(display) {
    if (display) {
        displayRangeButon.show();
    } else {
        displayRangeButon.hide();
    }
}

// Function called when one of the tower buttons is clicked to select that
tower to be placed.
function towerButtonFunction(tower) {
    towerButtonDisplay(false);
    placeCancelButtonDisplay(true);
    changeTowerColour(false, 0);
    towerPlace = true;
    selectedTower = tower;
    changeTowerColour(true, tower);
```

```
}

// Function called when the toggle tower range button is clicked.
function displayRangeButtonFunction() {
    if (displayRange || towers.length == 0) {
        displayRange = false;
    } else {
        displayRange = true;
    }
}

//Updates each of the towers the user has placed.
function towerUpdate() {
    for (let tower of towers) {
        tower.update();
    }
}

// Changes the colour of the button the user is currently clicking
function changeTowerColour(change, tower) {
    if (change) {
        switch (tower) {
            case 0:
                soldierButton.style("background-color", "green");
                break;

            case 1:
                machineGunnerButton.style("background-color", "green");
                break;

            case 2:
                rpgButton.style("background-color", "green");
                break;

            case 3:
                ifvButton.style("background-color", "green");
                break;

            case 4:
                tankButton.style("background-color", "green");
                break;
```

```
            case 5:
                shoulderAntiAirButton.style("background-color", "green");
                break;

            case 6:
                surfaceAirButton.style("background-color", "green");
                break;

            case 7:
                advancedSurfaceAirButton.style("background-color", "green");
                break;

            case 8:
                specialAbilityButton.style("background-color", "green");
                break;

            case 9:
                specialAbilityButton.style("background-color", "lightgrey");
                break;

            case 10:
                specialAbilityButton.style("background-color", "#3fcaca");
                break;
        }
    } else {
        soldierButton.style("background-color", "#3fcaca");
        machineGunnerButton.style("background-color", "#3fcaca");
        rpgButton.style("background-color", "#3fcaca");
        ifvButton.style("background-color", "#3fcaca");
        tankButton.style("background-color", "#3fcaca");
        shoulderAntiAirButton.style("background-color", "#3fcaca");
        surfaceAirButton.style("background-color", "#3fcaca");
        advancedSurfaceAirButton.style("background-color", "#3fcaca");
    }
}
```

index.html

```
<!-- Written automatically by the p5.vscode extension by Sam Lavigne and
edited by James Baldwin for my Computer Science NEA Coursework.
Started on 14/06/23. This is the program's HTML file which describes the
structure of the web page. -->
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Military Tower Defence Game</title>

    <link rel="stylesheet" type="text/css" href="style.css">

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.3.1/p5.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.3.1/addons/p5.sound.js"></
script>
  </head>

  <body>
    <script src="classes/queue.js"></script>
    <script src="classes/level.js"></script>
    <script src="classes/enemy/enemyBasic.js"></script>
    <script src="classes/enemy/enemyVehicle.js"></script>
    <script src="classes/enemy/enemyPlane.js"></script>
    <script src="classes/enemy/enemyQueue.js"></script>
    <script src="classes/tower/towerBasic.js"></script>
    <script src="classes/tower/projectile.js"></script>
    <script src="classes/tower/projectileQueue.js"></script>
    <script src="setup/masterSetup.js"></script>
    <script src="setup/menuSetup.js"></script>
    <script src="setup/gameSetup.js"></script>
    <script src="setup/enemySetup.js"></script>
    <script src="setup/towerSetup.js"></script>
    <script src="setup/skillTreeSetup.js"></script>
    <script src="setup/campaignSetup.js"></script>
    <script src="setup/specialAbilitySetup.js"></script>
    <script src="sketch.js"></script>
  </body>
</html>
```

sketch.js

```
// Written by James Baldwin for my Computer Science NEA Coursework. Started on
14/06/23. This is the main file which combines code from other files to run
the program.

"use strict";

// Preload function from the p5.js library. Contains images which need to be
loaded before the main menu appears.
function preload() {
  mainMenuImage = loadImage("./../img/mainMenuImage.png");
  vietnamMapImage = loadImage("./../img/vietnamMapImage.png");
  stalingradMapImage = loadImage("./../img/stalingradMapImage.png");
  gulfwarMapImage = loadImage("./../img/gulfwarMapImage.png");
}

// Setup function from the p5.js library. This function is run once at the
start of the program and never again, so it is used for creating the canvas,
etc.
function setup() {
  createCanvas(1856, 864);
  menuButtonsSetup();
  mapButtonsSetup();
  towerButtonsSetup();
  skillTreeButtonsSetup();
  campaignButtonsSetup();
  specialAbilityButtonSetup();
}

// Draw function from the p5.js library. This function runs 60 times per
second and is used for animation (i.e. updating object positions, drawing map,
etc.).
function draw() {
  background(21, 191, 61);
  switch(gameState) {

    // Loads the menu.
    case 1:
      titleAndImageSetup();
      mainMenuButtonsDisplay(true);
      mapButtonsDisplay(false);
      returnToMenuButtonDisplay(false);
      placeCancelButtonDisplay(false);
```

```
    displayRangeButtonDisplay(false);
    towerButtonDisplay(false);
    skillTreeButtonsDisplay(false);
    campaignButtonsDisplay(false);
    specialAbilityButtonDisplay(false);
    break;

  // Loads the campaign game mode.
  case 2:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
    } else if (chosenLevel == -1) {
      mapButtonsDisplay(false);
      campaignButtonsDisplay(true);
      returnToMenuButtonDisplay(true);
      setBaseHealth(1000);
    } else{
      specialAbilityButton.html(returnSpecialAbilityName());
      returnSpecialAbilityName();
      campaignButtonsDisplay(false);
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      specialAbilityButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      checkBaseHealth();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemyUpdate();
      towerUpdate();
      specialForcesUpdate();
      enemySetup(0);
      enemyUpdateCampaign();
      showSpecialForces();
      showAirstrike();
      showTrap();
      trapFunction();
```

```
    }
    break;

  // Loads the arcade game mode.
  case 3:
    if (JSON.stringify(chosenMap) == JSON.stringify(MAP_BLANK)) {
      mapButtonsDisplay(true);
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      setBaseHealth(1000);
    } else{
      specialAbilityButton.html(returnSpecialAbilityName());
      mapButtonsDisplay(false);
      mainMenuButtonsDisplay(false);
      displayRangeButtonDisplay(true);
      returnToMenuButtonDisplay(true);
      specialAbilityButtonDisplay(true);
      towerButtonDisplay(true);
      drawMap();
      displayInformation();
      checkBaseHealth();
      displayBaseHealth(baseHealth, baseHealthMax);
      placeTowerFunction();
      enemySpawn();
      enemyUpdate();
      towerUpdate();
      specialForcesUpdate();
      enemySetup(1);
      showSpecialForces();
      showAirstrike();
      showTrap();
      trapFunction();
    }
    break;

  // Loads the tutorial.
  case 4:
    mainMenuButtonsDisplay(false);
    returnToMenuButtonDisplay(true);

    break;
```

```
    // Loads the skill tree.
    case 5:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);
      skillTreeButtonsDisplay(true);

      break;

    // Loads the settings.
    case 6:
      mainMenuButtonsDisplay(false);
      returnToMenuButtonDisplay(true);

      break;

    default:
      gameState = 1;
  }
}
```

style.css

```
/* Written automatically by the p5.vscode extension by Sam Lavigne and edited
by James Baldwin for my Computer Science NEA Coursework.
Started on 14/06/23. This is the program's CSS file and contains all the
styling for the HTML and buttons. */

html, body {
  margin: 0;
  padding: 0;
}

canvas {
  display: block;
}

#returnToMenuButtonID {
  height: 50px;
  width: 200px;
  font-size: 20px;
  border: none;
  border-radius: 25px;
  padding: 10px;
```

```
  background: rgb(63, 202, 202);
}

.mainMenuButtonClass {
  height: 75px;
  width: 400px;
  font-size: 20px;
  border: none;
  border-radius: 25px;
  padding: 20px;
  background: rgb(63, 202, 202);
}

.selectTowerButtonClass {
  height: 50px;
  width: 200px;
  font-size: 18px;
  border: none;
  border-radius: 25px;
  padding: 10px;
  background: rgb(63, 202, 202);
}

.towerButtonClass {
  height: 100px;
  width: 100px;
  font-size: 16px;
  border: solid;
  border-width: 1px;
  background: rgb(63, 202, 202);
}

.towerRangeSpecialAbility {
  height: 80px;
  width: 100px;
  font-size: 16px;
  border: solid;
  border-radius: 15px;
  background: rgb(63, 202, 202);
}

.skillTreeButtonClass {
```

```
  height: 75px;
  width: 400px;
  font-size: 20px;
  border: none;
  padding: 20px;
  background: rgb(141, 91, 15);
}

.campaignButtonClass {
  height: 80px;
  width: 150px;
  font-size: 20px;
  border: none;
  padding: 20px;
  background: rgb(201, 87, 191);
}
```

Bibliography:

Figures:

1. Google (2022), *Send help as I have discovered Bloons TD 6*. Available at: https://www.gamingonlinux.com/2022/01/send-help-as-i-have-discovered-bloons-td-6/ (Accessed 19th April 2023).
2. Google (2022), *If all primary monkeys had a 4th upgrade path!* Available at: https://www.reddit.com/r/btd6/comments/rdl0d9/if_all_primary_monkeys_had_a_4th_upgrade_path/ (Accessed 19th April 2023).
3. Google (2021), *Intermediate Maps*. Available at: https://interfaceingame.com/screenshots/bloons-td-6-intermediate-maps/ (Accessed 19th April 2023).
4. Google (2013), *'Plants vs. Zombies' sequel shambling toward release in July'*. Available at: https://www.nbcnews.com/tech/tech-news/plants-vs-zombies-sequel-shambling-toward-release-july-flna6c9801258 (Accessed 19th April 2023).
5. Google (2021), *Plants vs Zombies Walkthrough Guide*. Available at: https://ayumilove.net/plants-vs-zombies-walkthrough-guide/ (Accessed 19th April 2023).
6. Google (2015), *Popular Tower Defense Game 'Toy Defense' Now Available For Download In Windows Phone Store*. Available at: https://mspoweruser.com/popular-tower-defense-game-toy-defense-now-available-for-download-in-windows-phone-store/ (Accessed 19th April 2023).
7. Google (2015), *The Best Games Ever: Toy Defense*. Available at: http://pcgamescreens.blogspot.com/2015/11/toy-defense-best-games-ever.html (Accessed 19th April 2023).

8. Google (2015), *The Best Games Ever: Toy Defense*. Available at: http://pcgamescreens.blogspot.com/2015/11/toy-defense-best-games-ever.html (Accessed 25th April 2023).
9. Created by myself in Paint 3D.
10. Created by myself in Paint 3D.
11. Created by myself in Google Drawings.
12. A screenshot of my project, which was written by myself.
13. A screenshot of my project, which was written by myself.
14. A screenshot of my project, which was written by myself.
15. Created by myself in Paint 3D.
16. Created by myself in Paint 3D.
17. Created by myself in Paint 3D.
18. Created by myself in Google Drawings.
19. Created by myself in Paint 3D.
20. Created by myself in Paint 3D.
21. Created by myself in Paint 3D.
22. Created by myself in Google Drawings.
23. A screenshot of my project, which was written by myself.
24. A screenshot of my project, which was written by myself.
25. A screenshot of my project, which was written by myself.
26. A screenshot of my project, which was written by myself.
27. A screenshot of my project, which was written by myself.