

Computer Games Development

CW208

TDD

Year IV

Darragh McKernan

C00271651

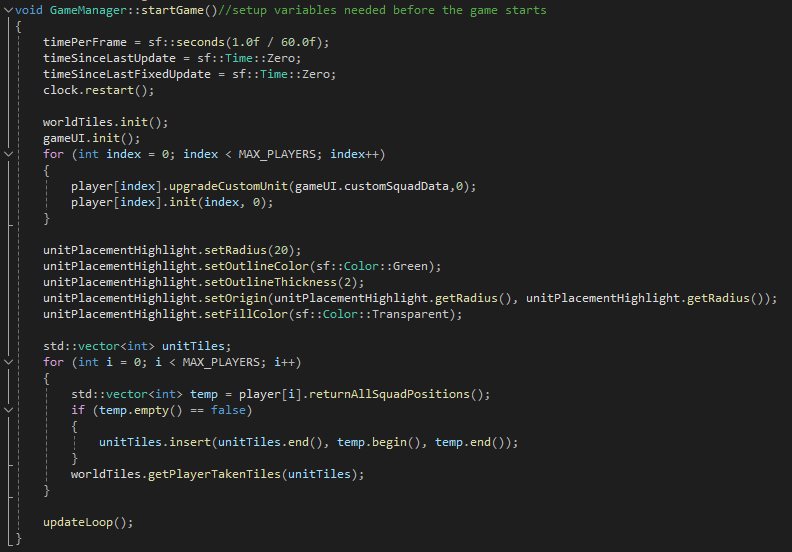
|  |  |
| --- | --- |
|  |  |
| 11/05/2025 | |

**Technical Design**

**Purpose:**  
This document will detail the technical details and design decisions of the War of Attrition game. It will cover the software architecture, algorithms, code structure.  
  
**Tools and Technologies:**  
War of Attrition is built in C++ using SFML for graphics and was made using Visual Studio 2022 as the IDE.  
  
**Software Architecture:**  
The architecture of the game follows an object oriented design pattern based around a core GameManager class that links all of the gameplay systems together.

UML Class Diagram A diagram of a diagram

AI-generated content may be incorrect.  
  
**Game Classes Descriptions:**  
**GameManager**  
The core loop of the game, all other class are linked together and communicate using the GameManager.  
Core Functions:

void startGame() – used to create instances of all of the classes the game needs, sets up all of the other classes and begins the core game loop.  


void updateLoop() – the main loop that runs the entire, sets up a deltaTime update for itself and all other classes to use. Features a 60 frames per second fixed update loop for timers and other interactions that do not need to be ran as quickly as possible. Calls all the other functions needed for the game as seen below.  
A computer screen shot of a program code

AI-generated content may be incorrect.

void updatePlayers(sf::Time& t\_deltaTime) – cycles through all players and only updates the function that passes the information of the hovered unit to the data readout and the function that spins the helicopter blades. Other than that only 1 player gets to be updated per loop and that is the player whose turn it currently is. This will go through everything the player needs in order to complete their turn, from simply calling the players update to handling finding the requested path and giving it back to the player. This is possibly the most important function as we handle everything player related here. Collisions are the most expensive part to process about this function but since the game is turn based we will only need to do this once per turn and over the span of 1 pass. As soon as the players turn has ended and all of their units have stopped moving we can then check all of the players units against all of the opponents units and if there is a collision we can deal damage and then after that handle any units that have died. After a players turn is ended we reset that player and then change the current turn to the next player that still has living units.   
A screen shot of a computer code

AI-generated content may be incorrect.

void display(sf::RenderWindow& t\_window) – this function will go through and call the render function and pass in a reference to the sf::RenderWindow so all of the players and the world tiles can draw all of the sprites that make up the game. We also get the mouse position relative to the world in here so that it can be used elsewhere in the game.  
A screen shot of a computer code

AI-generated content may be incorrect.

void displayHUD(sf::RenderWindow& t\_window,sf::View& t\_fixedWindow) – this function will render ever UI element in the game and display it to the sf::ViewPort so that the UI is static in relation to the viewport and will not move around when the camera does. This function will also render the splash screens and call on gameUI to render the rest of the UI elements that it responsible for.  
A screen shot of a computer

AI-generated content may be incorrect.

void menuInteractions() – this function links the menus in gameUI to the rest of the classes in the game. We check for menus being open or changes to values that are from menus here and use the variables to modify other classes. If the player has opened a menu for creating a unit and has selected a unit to be created we stop the player from being able to interact with their units until the menu has been closed to avoid any misclicks. If the gameUI tells us the player has requested a new unit and has specified the type we can create the unit for the player here.  
A computer screen shot of a computer program

AI-generated content may be incorrect.

**Player**  
Handles all of the data and functions required for each player. For every player in the game there will be one instance of the Player class. The player handles all of the units and towers that the player owns.

A diagram of a computer

AI-generated content may be incorrect.

Core Functions:

init()

Used to create the players first 4 units and set them into the world. It will behave slightly differently depending on the team so that units are spaced out and are correctly assigned to the right team.  
A computer screen with white text

AI-generated content may be incorrect.  
  
void update(sf::Time& t\_deltaTime)

Handles everything to do with the players units. If the player has already selected a unit we let it be the formation leader and can then add other units to the formation as well. Otherwise we have the player pick a target and we send a request out to the TileGrid to generate us a path so we can then give it to the unit that requested it. We will constantly update units in here and wait for the player to end their turn which then lets the units and formation begin moving towards their targets. We then wait until all units have reached their targets and then we can swap to the next player.  
A computer code on a black background

AI-generated content may be incorrect.  
  
void updateMovingTexures(sf::Time& t\_deltaTime)

This is a simple function that will run regardless of if it is this players turn or not. It is used to allow the helicopter units propeller to always spin and allows for the explosions to work even if this unit isn’t active.  
A computer screen shot of a program code

AI-generated content may be incorrect.  
  
void dealDamageToUnit (int t\_unit, int t\_damage)

Used to apply damage to the specified unit after it collided with an enemy. Here there is a random chance for the attack to do more or less damage to add some variance to the game and to have the game not have the same boring outcome from fights.  
A computer screen shot of a code

AI-generated content may be incorrect.  
  
void givePathToFormation(std::vector<int> t\_path)

Will pass the path that was generated by the TileGrid into the formation if it had requested it. This allows the formation leader to have a path that it will follow to reach it end goal and so will dictate the overall movement of the followers of the formation too.  
A screen shot of a computer code

AI-generated content may be incorrect.  
  
void checkForDeadSquads()

At the end of each players turn we loop through all of the squads so that we can check for any units with a health below 0 and if they are found we will first instantiate several explosion particle effects around the position of the squad before then adding the squad to the deletion list. After this we then loop through the deletion list and remove the squads from back to from so that when we remove a squad we don’t then remove the wrong one when the squads further in the list fill into its place.  
A computer screen shot of a program code

AI-generated content may be incorrect.

**Squad**  
This class is used to manage the individual units that the player can have. They are extremely modular and are highly customisable. The Squad also has multiple behaviours that it can swap through in order to get to its target point.

A diagram of a project

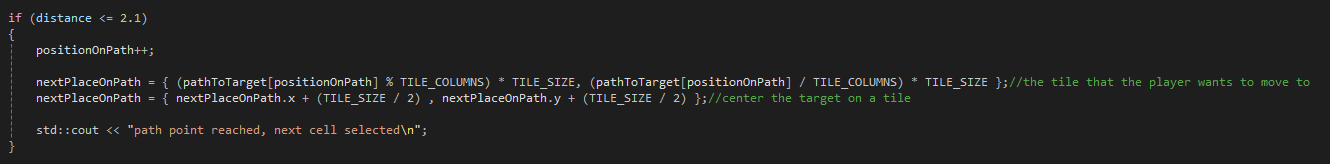
AI-generated content may be incorrect.

Core Functions:

void init(sf::Vector2f t\_startingPos, int t\_teamNum, int t\_unitType)

Used to set up all the values in the units squadData data structure so that it can be stored by itself and wont be effected by external changes such as custom units getting modified. This function also sets up all the units colliders. The unit has the troopContainer which is the collider used for movement and for checking if this unit is occupying a tile as this collider is smaller than a world tile. The verticalCollider and horizontalCollider are exclusively responsible for checking if this unit has collided another unit. If there is a collision this is how we know if the unit needs to take damage. These are customisable so that certain units can have longer engagement range than others. The reason it is 2 rectangular colliders is so that it can only attack in a NESW direction and not in a radius which would not be an intended behaviour.A computer screen with white text

AI-generated content may be incorrect.  
  
void update(sf::Time t\_deltaTime)

This is only used if the unit is not in a formation and will have the unit follow the path that was given to it after it had been selected and the target had been selected, after they were set the unit would have been given its path that it could follow. The unit will move from 1 cell to the next and would keep going until it reaches the center of that cell then will move again.  


void updateFormationUnit (sf::Vector2f t\_formationPosition, sf::Time& t\_deltaTime)

The function for moving units that are in a formation, all of their behaviours are updated from here.  
A computer screen shot of a code

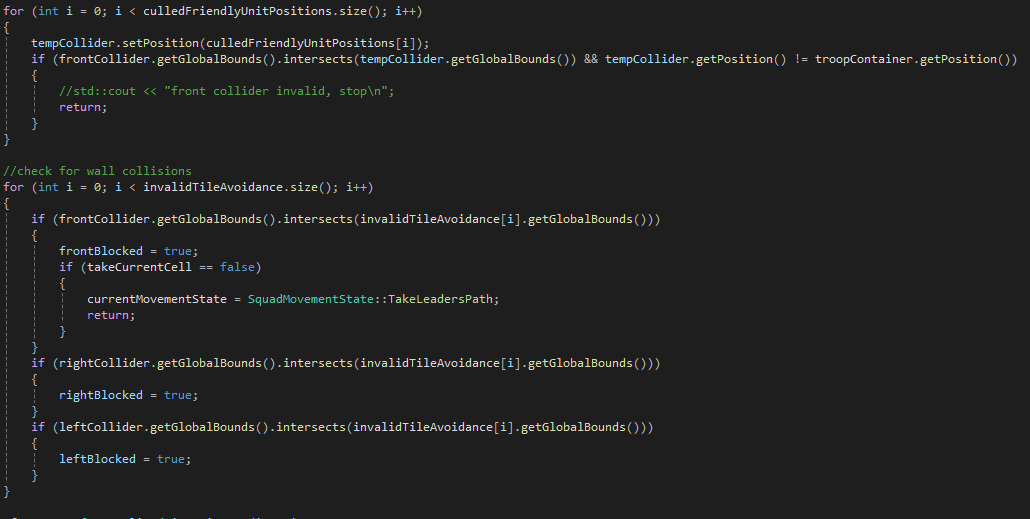
AI-generated content may be incorrect.

void moveToFormation(sf::Vector2f t\_formationPosition,sf::Time t\_deltaTime)

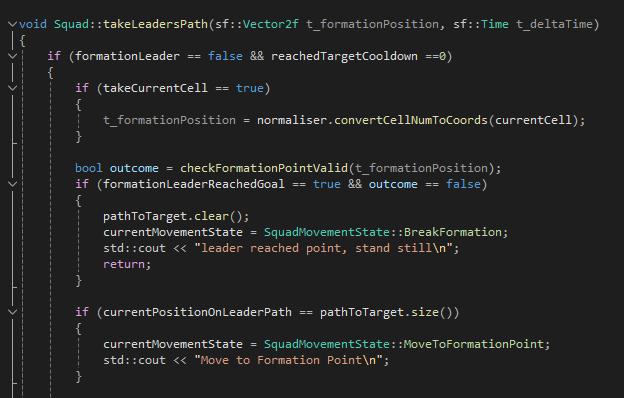
This function will move the unit to its specified position in the formation, it is the default formation behaviour. It will first check that the point it is trying to move to is still valid and if it is not it will set itself to automatically follow the path that the leader is taking. If the point is valid it will simply move to the point that the formation has specified this unit should be going to. If it has reached its position and is the leader we can now report to all of the other units that the leader has reached the end and they can change behaviour if needed. If we are not the leader and have reached the point that probably means the leader is turning or has had to stop for some reason and we should be prepared to move again so do not immediately say that we have also reached our goal unless we know the leader has.  
A computer screen shot of a program code

AI-generated content may be incorrect.

void steerAroundObstacle(sf::Vector2f t\_formationPosition, sf::Time t\_deltaTime)

This function will attempt to move to the specified formation will steering around the terrain and obstacles in the world. The unit will also take into account the positions of friendly unit but can add enemies to the world obstacles as they cannot move during the players turn. If the target cell is not free and the leader has reached its goal that means the unit has to speed up and just pick somewhere to go so we swap to the BreakFormation behaviour. If the cell is taken and we do not have to stop and take the current cell we are on we can then simply follow the leaders path until something else can be done. If none of this is the case we go ahead with steering and set up some bools and temporary colliders in front and to the side of the unit. We force the colliders into a cardinal direction so if we end up turning at and angle in a corridor we don’t end up with both colliders at an angle touching both walls. We get all friendly positions and then we simply check what if any colliders are hitting something. If the front collider is hitting a friendly unit that is moving we simply stop in place until there is no longer a collision and we can then proceed as normal. If it’s the left collider we cannot move left and if it’s the right we cannot move right. If the front collider has hit something we follow the leaders path in case we have hit a dead end. If there are no collisions we can simply move towards the specified point while still checking for collisions.  


void takeLeadersPath(sf::Vector2f t\_formationPosition, sf::Time t\_deltaTime)

This function will have the unit follow the path that was given to the formation as we know that this path is navigable since the TileGrid generated the path before the end turn state was started. If we have decided that there is no better option but to just take our current cell and stay put then the bool takeCurrentCell will be true and if that’s the case we will set the t\_formation position to our own normalised cell so that we can ensure we get centered on it.If our target is not free and the leader has reached its goal the unit has to then break formation and take a nearby cell as our target so the round can end. If we have reached the end of the leaders path we also must change behaviour as we cannot take the formation leaders target cell so we attempt to take our original formation point again and if that wont work we will simply end up on the behaviour for breaking formation. Assuming we can simply follow the leaders path we will behave very similarly to the standard units update. We move from cell to cell and if our distance is close enough we lock into the cell center and then start moving to the next cell. If we reach the center we then check again if we can move to our formation position and if it is free we swap back to the SteerAroundObstacle behaviour so we can then transition to the follow formation point behaviour. 

void breakFormation(sf::Vector2f t\_formationPosition, sf::Time t\_deltaTime)

This function is designed to just break out of the formation altogether and just find a nearby cell to move to so we can end the turn, it’s a last ditch redundancy. The first time we hit this function in this turn we have a 30% chance to just move to the cell directly in front of us by using the SteerAroundObstacle behaviour and giving it the position of the cell in front of us. Otherwise the unit will resend a request for a path to be generated and given to it, it will also select one a cell within a 3\*3 grid of itself with itself as the center cell and if the cell is valid and isn’t the current cell we can then wait for the path to be given back to us. After receiving the path we simply follow the path in the same way as was done for the other path following functions by moving from cell to cell 1 and a time and picking the next cell once getting close enough.

A computer screen shot of a program code

AI-generated content may be incorrect.

bool checkFormationPointValid(sf::Vector2f t\_formationPosition)

Used to check if the specified position overlaps with any of the tiles that were flagged as invalid because they were water or had another unit on them. This is done easily by just moving a collider around to all of the invalid positions and checking if the position that was passed in is inside the collider at any point.

A screen shot of a computer code

AI-generated content may be incorrect.

**Formation**  
Player can have instances of the formation manager and it is used to create a group for the units to be placed together. The units can be added into the formation and the formation will give back the information that the units will need to move such as target position and whether or not the leader has reached the end goal.

A diagram of a company

AI-generated content may be incorrect.

Core Functions:

void update(sf::Time t\_deltaTime)

This is used to simply move the formation along the path that it was given when the leader was set and the leaders target was set. The same as the other path following scripts this one moves from point to point slowly and will wait when it gets to a cell and will slowly turn so that it is not as disruptive to the overall formation when compared to just snapping to the correct direction. The formation moves completely independently of the leader. The formation leader is also technically a follower as it simply moves to the leader position as it is told by the formation.

A computer code on a black background

AI-generated content may be incorrect.

int getPositionInFormation()

This function is used to assign units their numbers for their position as they are added to the formation. Every time the player selects a unit they are assigned a new number and the unit then has to remember its position in the formation so that it can request the coordinates it needs to move to while following.

A computer screen shot of a black screen

AI-generated content may be incorrect.

sf::Vector2f getFormationPosition(int t\_posInFormation)

This function will return the position the unit needs to move to when it has provided its position in the formation. This is done by taking the leaders point and getting its position and rotation first. Then we set its offset depending on which position it is, this is simply a distance on the x and y axis that we need to be away from the leader. We can also specify the x and y spread to bring the formation closer or further together. We can then calculate the backwards position using the offset and then by applying the angle to the offset to get the actual in world coordinate the unit will have to move to. Then the function can just return this point.

A computer screen with white text

AI-generated content may be incorrect.

**TileGrid**  
This class manages the entire map and contains all of the WorldTiles that are used to represent the whole world. The TileGrid manager is used to get coordinates from cell positions and also modifies the WorldTiles so that we can know what ones are taken and what kind of cell they are. The TileGrid will also create paths between any specified tiles by looping through the WorldTiles. The world editor is also handled in here and if a tile has been changed it will be automatically given its updated texture in here.

A diagram of a network

AI-generated content may be incorrect.

Core Functions:

void init()

Sets up all of tiles that make up the entire map, first by setting the texture from the spritesheet then creating a tile in all of the positions they are needed so that the map can be filled in. The tiles that need to be impassable can be loaded in from the wallsData that just stores the index of which cells need to be changed to impassable.

A black screen with white text

AI-generated content may be incorrect.

void hightlightTiles(bool t\_valid)

Sets a coloured highlight on the most recently hovered over cell to show whether or not that cell is within the range of the currently selected unit. This is crucial for visually showing the player how far they are allowed to move their unit.

A computer screen shot of text

AI-generated content may be incorrect.

void TileGrid::findTargetedTile()

This function waits until the player clicks on a tile and sets it as the target after checking it is valid a valid tile to move to.

A computer screen shot of text

AI-generated content may be incorrect.

void updateTileType(int t\_type)

This is used for the level editor, it first sets the currently hovered tile to the type that is passed into the function. It then clears all the saved invalid tiles and repopulates it with the changes. After this it calls on the updateTileTexture so that the current tile and the 8 immediate neighbours can update their textures, we specify that the depth is 0 because this code is going to be called recursively and we only want to check the immediate neighbours though this number could be changed if we wanted it to check even further from the starting tile.

A computer screen shot of a program code

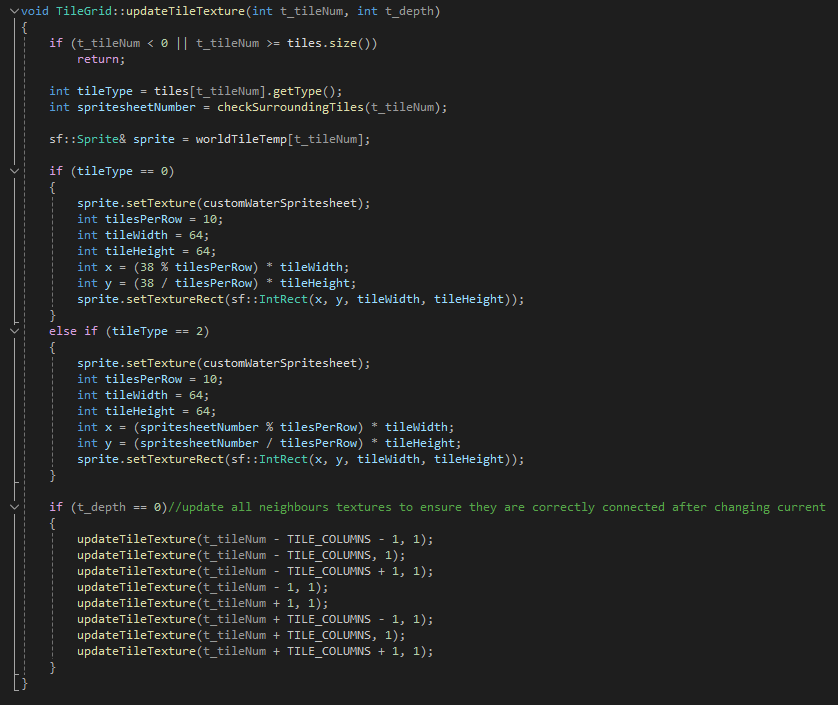
AI-generated content may be incorrect.

std::vector<int> getPathToTarget(sf::Vector2f t\_startPos, sf::Vector2f t\_targetPos)

This code uses BFS to search through all of the tiles starting from the specified start position and will search until it finds the target position. It specifically uses Manhattan distance and not Euclidian distance as I wanted to avoid the path clipping through cells at a corner, it also just visually looked better when it followed the cells going up or down and not at diagonals. After finding the target cell it then gets the path back to the start of the search using the cameFrom vector and working backwards. It then saves the path as a std::vector so that we can send that to the formation or unit that needs it. It also sets the tiles to blue so we can visually see the path that it has chosen. A computer screen with text on it

AI-generated content may be incorrect.

void updateTileTexture(int t\_tileNum, int t\_depth)  
This code automatically checks the tiles neighbours and sets the texture accordingly. The first line is a check to ensure the tile we want to check does exist because if we check a wall tile originally its neighbours would be out of bounds. We then get the spritesheetNumber from the checkSurroundingTiles function which will act as a key and tell us exactly where in the spritesheet the texture we need to set it. Then we set up the tilesheet so that if its just grass we can set it to the basic grass texture but if not we need to be able to load in the correct water tile based on spritesheetNumber. Then we call the current function again but this time we pass in a 1 for depth to signify that this is no longer the first run of it. It is currently set up to stop after this but the number can be simply increased to any depth, this would be needlessly inefficient as when a tile is changed it should only effect its immediate neighbours.



int checkSurroundingTiles(int t\_tileNum)

This function is used to check the neighbours of the specified cell and determine what texture should be applied to the current cell. First we check the current cell is valid and also check its type so we know what we have to look for with the neighbouring cells. We then use a lambda that will use the 2 inputted numbers as positional offsets from the center cell and return a true if the cell we check is within the bounds and is the same type as the tile we are checking. Using this lambda to set the values for the neighbours we can use these to tell what cells are important and need to be used to set the texture correctly. Then we use lots of checks and compare if the cells match up with what a texture requires. If the tiles of N,E,S,W, are all taken we know that the current tile has to be a crossroads texture and so we return the number that corresponds to that on the spritesheet. If at the end none of the neighbours matched we can set the texture to the singular water sprite as we know it has no connection points to the tiles around it. A screen shot of a computer

AI-generated content may be incorrect.

**Tile**  
This class simply contains the shape of each of the cells, it is used for highlighting specific cells such as if a move is in valid range or not which needs a red or blue highlight. It can also be used to visually show the paths that have been selected by highlighting the tiles that consist of the path. This class is entirely managed by the TileGrid class.

A diagram of a project

AI-generated content may be incorrect.

Core Functions:

void setTarget(bool t\_active)

This is used to show if the tile has already been set as a target or not.

A computer screen shot of a code

AI-generated content may be incorrect.

void setType(int t\_type)

Used to set the type of tile that this tile represents, originally was for programmer art but now is used for collision and pathfinding.

A computer screen shot of a program code

AI-generated content may be incorrect.

void Tile::setHighlight(bool t\_valid)

This class is used to set the current tile to blue or red depending on if its within the players range or not.

A screen shot of a computer code

AI-generated content may be incorrect.

**Explosion**  
This class is a lightweight modular particle system that is used for when a unit is destroyed. It is highly customisable and has a lot of random variance when it is created. When created it randomises most of its variables so that the explosions are not uniform. This class is easy to add to and modify and currently only holds an explosion effect but if other spritesheets were provided they would easily work too.

A diagram of a computer program

AI-generated content may be incorrect.

Core Functions:

Explosion(sf::Vector2f t\_spawnPos)

This constructor just takes in the position it has to spawn at and then randomises most of its variables so that the explosions are not identical and to add some visual variety. The totalLifetime is randomly set to between 0.6 and 1 second and also has a random start delay between 0 and 0.6 seconds. The scale is then also randomised between 0.25 and 1

A computer screen shot of a program code

AI-generated content may be incorrect.

void update(float deltaTime)

This function will wait until the delay is 0 and then begin its lifetime timer and will set its own texture on the spritesheet depending on how long has passed since it began and how long its total lifetime is. A computer screen shot of a program code

AI-generated content may be incorrect.

void draw(sf::RenderWindow& window)

This class simply draws the explosion sprite but will only do so once it has started the effect and has not yet ran out of time on it s totalLifetime timer.

A computer code on a black background

AI-generated content may be incorrect.

**UI**This class handles the placement and rendering of all of the UI elements. It also handles the button clicks and most of the key press interactions with the game.  
A diagram of a program

AI-generated content may be incorrect.

Core Functions:

void fixedUpdate()

This is only used for updating the sprites used to represent the what sprite the custom unit has currently set.

A screenshot of a computer program

AI-generated content may be incorrect.

void handleMenuInteractions()

This function handles all of the player interactions between the buttons and the mouse. It is also responsible for what menus and UI elements get displayed and at what time. The create unit menu and create tower menu are the main interactions and each have several buttons on their respective menus. The custom unit upgrade menu is also rendered and interacted with here and all of the changes must then be sent to the player so they have the values of the custom unit. A computer screen shot of a program

AI-generated content may be incorrect.

void updateUnitDataDisplay(SquadData t\_squadData)

This function takes the mouse position and places the squad data readout on the cursor while restricting it to the bounds of the screen. The text for all of the stats is also set here and the squad data must be passed in when this is called to the data can be displayed.  
A computer code with many small text

AI-generated content may be incorrect.

**Normaliser**

This virtual class is used across the game to help convert values from coordinates to tile numbers and vice versa, it’s a collection of functions that are constantly used.

Core Functions:virtual sf::Vector2f normalizeToTileCenter(const sf::Vector2f& position) const

This function takes in a position and normalises it to the center of the cell to ensure objects like units are perfectly centered on their cells.

A screen shot of a computer program

AI-generated content may be incorrect.

virtual sf::Vector2f convertCellNumToCoords(const int t\_cellNum) const

This function takes in a cell number and converts it into the coords are the center of the cell, this is used when we go from the path a unit has to follow in cell numbers to actual positions that the unit can move to.

A screen shot of a computer program

AI-generated content may be incorrect.

virtual int convertCoordsToCellNum(const sf::Vector2f& t\_coords) const

This function takes in a set of coordinates and returns the number of the cell that is at the coordinates. This is useful for going from mouse position to the cell that the mouse would be currently hovering over.

A screen shot of a computer code

AI-generated content may be incorrect.