**SHadez Computer science project analysis**

Sebastian Roffey

HSFC

Computer Science

Contents

[Problem Identification 3](#_Toc31749480)

[Main Problem 3](#_Toc31749481)

[Computational methods to be used for the project 5](#_Toc31749482)

[Abstraction 5](#_Toc31749483)

[Decomposition 5](#_Toc31749484)

[Pattern recognition 5](#_Toc31749485)

[Algorithms 6](#_Toc31749486)

[Stakeholders 7](#_Toc31749487)

[Research 8](#_Toc31749488)

[Background of the original games 8](#_Toc31749489)

[GUI 10](#_Toc31749490)

[Title screen (Shadez) 10](#_Toc31749491)

[Map Screen (Shadez II) 11](#_Toc31749492)

[Gameplay Screen 12](#_Toc31749493)

[Mechanics of Shadez 13](#_Toc31749494)

[Unit Combat 13](#_Toc31749495)

[Game Progress 13](#_Toc31749496)

[Research Questionnaire Completed By Stakeholders 13](#_Toc31749497)

[Analysis of the research questionnaire 16](#_Toc31749498)

[Success Criteria 17](#_Toc31749499)

[Hardware requirements 19](#_Toc31749500)

[Software Requirements 19](#_Toc31749501)

[Iteration 1 20](#_Toc31749502)

[Discussion with Stakeholders Regarding Iteration Priorities 20](#_Toc31749503)

[Post development usability test plan 20](#_Toc31749504)

[Proposed Solution and Purpose of Iteration 1 23](#_Toc31749505)

[Requirements for iteration 1 (Success criteria) 23](#_Toc31749506)

[Decomposition and Sub Task Time Allocation 25](#_Toc31749507)

[Design 27](#_Toc31749508)

[User interface design 27](#_Toc31749509)

[Usability Features. 30](#_Toc31749510)

[Landscape and system drawing (With Pseudocode) 31](#_Toc31749511)

[Unit Class 33](#_Toc31749512)

[Unit specific class 34](#_Toc31749513)

[Spawn Unit Procedure 35](#_Toc31749514)

[New Location Procedure 37](#_Toc31749515)

[Take Damage Function (Development of efficient unit death) 41](#_Toc31749516)

[Game Model Class 44](#_Toc31749517)

[Collision range function 45](#_Toc31749518)

[Collision Execute Procedure. 46](#_Toc31749519)

[Unit instance reuse 50](#_Toc31749520)

[Check unit locations procedure 53](#_Toc31749521)

[Iteration 1 Testing 55](#_Toc31749522)

# Problem Identification

## Main Problem

The aim of this project is to create a replica of the online strategy game Shadez using an object orientated programming approach. Shadez is a combat themed single player game where the player controls an army full of units which are purchased with in-game money, to defeat an enemy army in waves (stages). To defeat the enemy, the player’s base health must not be depleted by the end of the battle, this is achieved by preventing enemy units from crossing into the player’s base. If an enemy unit crosses into the player’s base, the amount of the unit’s health dictates the amount of base health deducted from the player. Similarly, if the player’s units cross into the enemy’s base, the player is rewarded with extra in-game money which will further help defeating any further waves. 

The game interface will first consist of a menu screen, containing options and help facilities for players to see. I also plan to allow players to load their progress from a text file, to unlock levels and continue from where they left off previously. I also plan to create a rough story line behind the game, to make the game more interesting. My stakeholders have played the original game and have stated it became quite repetitive in its further stages; therefore a story line should make the game more engaging. Branching of the menu will be a map with level names and background information to brief the player on battles which are currently unlocked, and as levels are completed, more difficult levels are unlocked, progressing the story. The player will eventually be able to unlock the final level and complete it to finish the game.

The battle interface consists of a landscape window, which is where all of the graphics of the units fighting will occur. Units will progress along the landscape until becoming within a certain proximity of an enemy unit (Their engagement distance), where they will pause and will fire at the enemy units until the enemy’s unit health is depleted. When an enemy unit is destroyed, the player is rewarded with money. Below the landscape window is the player’s controls. There are buttons to buy each available unit, with their price and health displayed. When the mouse is over the button, small text will appear at the bottom of the screen describing the features of the unit. The player must have adequate money to buy the unit. Also displayed at the bottom of the landscape window is a base health bar, stage number and money amount. There is also a pause button and I have not yet decided to do a battlefield overview, with the possibility to scroll across the battlefield using a preview window.

This game will be created in Visual Studio 2017 in C Sharp. The project lends itself well to object orientated programming as units can be integrated as objects, with each unit’s health and stats encoded as attributes.

## Computational methods to be used for the project

### Abstraction

It is important that I focus on information that is relevant to my project whilst completing it, and to ignore irrelevant information that won’t impact the quality of the final program. This will prevent me from wasting time on unneeded elements of the program. So, for example, I will ignore attributes of units that won’t affect their performance in battle, such as armour thickness or men per unit. These stats will not be needed or used within the mechanics of the game and therefore would waste time if I chose to incorporate them. I also will ignore excessive detail on the menu and map screen, as the focus of this project is the gameplay of my game.

### Decomposition

This is a computational method I will use a lot of in my project. Due to the size and nature of the game I am creating, I plan to decompose the game into small programming tasks which I can tackle one at a time. This will allow me to approach the large task with more structure, and during each programming session, I will know what to complete to be on track for finishing each iteration of the game. Decomposition will also allow me to order programming tasks to create a playable version of the game sooner, even in its most basic stages, for my stakeholders to provide feedback on. So, for example, Instead of coding the whole landscape and interactions of units with each other and the landscape all at once, I will separate the task into small sub tasks, consisting of coding the landscape, spawning units, moving units across the landscape, proximity detection between units and unit interactions.

Without decomposition, the sheer size of the project would be far to intimidating and complex to code. I would be overwhelmed with the task without being able to decompose it, as decomposition allows me to organise and recognise better what is needed within the program. It also will allow me to write and present each aspect of the program in a more organised way due to the manageable sized chucks tasks are broken down into.

### Pattern recognition

Programmers must be able to recognise patterns within code, as doing so results in an opportunity to create more efficient code through using functions and procedures. Recognising patterns while coding the game Shadez will make my code shorter and will reduce the amount of code needed to be written, as copying out similar code increases the chance of errors being made and increases the overall program size. After playing the game a lot myself I have already identified some patterns in the game. Unit movement is repeated by every unit, enemy or friendly, the only difference between units is their speed and direction. Therefore, I plan to make a procedure which will move each unit in the correct manner, taking in parameters which dictate the nature of movement. These will consist of; proximity (If another unit is in range of attack, resulting in no movement), speed (A multiplier) and direction (friendly units will move from left to right on the landscape, enemies will move in the opposite way) .

### Algorithms

The whole processes and mechanics behind my game will all be executed algorithms strung together within the classes and the main program. The step by step sequencing of code will be the basic construct of which the whole game runs around. Alongside decomposition, each sub task which is decomposed, I will create an algorithm to solve each problem. I will create classes which design objects using algorithms to create their stats and procedures which can be called upon in the main program. Altogether, every algorithm will go towards making the game work fluidly. I will be using a lot of the PaintEvent procedures within my algorithms to graphically produce landscapes specific to each level using an array of dictated points read in by an algorithm.

# Stakeholders

I have chosen three people to be stakeholders for my project - Ollie Wilkinson(17), Rose Addison(16) and Mark Lamando(32). Being a strategy game, Shadez requires the player to learn and understand the logic and mechanics of the game in order to win, making it not very suitable for younger audiences who probably won’t understand the strategies to win. Ollie and Rose are about the age of the audience this game targets, their feedback should reflect this and will help me develop the game to be more appealing for youth players. I have made sure to include both genders as although the game is army themed, I as the Developer want to not assume females will not want the play the game, even if it is army/combat themed. Rose is very interested in strategy games and plays them often in her free time. Her experience will be very helpful as she can directly compare my game in development to other strategy games she plays(ed) and can give me some good advice to further improve my game.

Ollie is in my computer science class and has learnt to code C Sharp with me. Therefore, Ollie is a very good stakeholder as he has a good understanding of object orientated programming in C Sharp, so he will be able to give me useful tips and advice whilst the game is in development.

Mark Lamando does not play computer games, so I am going to ensure within the game there is plenty of help facilities and instructions which are easy to understand for him to quickly get the hang of playing the game. This will allow him to be better able to understand and enjoy the game. His older age will provide me with feedback from a different age range and may give useful insights my other two Stakeholders may not give. My stakes holders will continually offer advice whilst I research, develop, prototype and analyse my project. They will all have a very active part in the project and their input will greatly improve the finished game. They all plan to play the game when it is finished for their own enjoyment, and their machines all have the minimum system requirements with .Net 6.4.1 already installed.

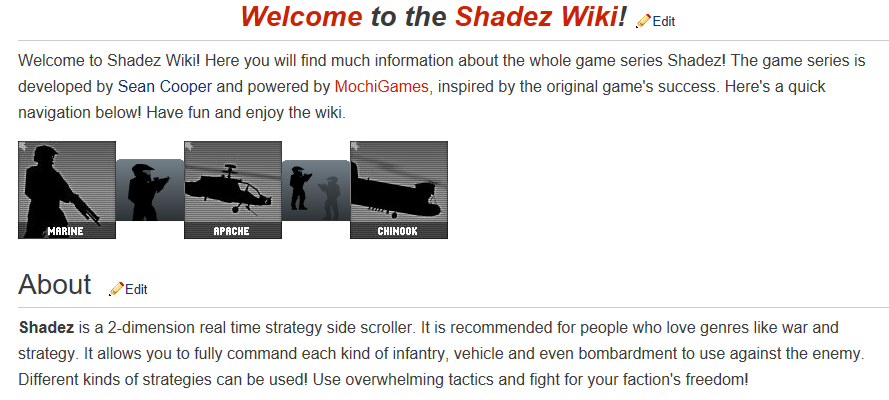
# Research

## Background of the original games

The games Shadez, Shadez II and Shadez III are all online games produced by Sean Cooper Games, hosted on websites which contain hundreds of games and are accessed by anyone’s computer which has an internet connection and the ability to run adobe flash. The first game Shadez, is simple compared to the other two sequels, lacking a story line. The game focuses on creating and controlling the players army to defeat the enemy in waves. Each level is only a starting point, as the gameplay becomes endless, only stopping when the player is defeated. This game created a concept which Sean Cooper took on and developed in his next two games, adding other aspects such as buildings and a story line to create a better game.

A screenshot of a cell phone

Description automatically generatedAll the games have online articles about stories behind the games, tips and hints for progressing in the game. The screen shot below shows the Hints and Tricks article for Shadez. Players of the game access this webpage by clicking on the game guide option in the menu. This screenshot is taken from [http://www.crazymonkeygames.com/guide/Shadez//](http://www.crazymonkeygames.com/guide/Shadez/)

The games also have wiki’s for players to read into the back story behind each game, and

https://shadez.fandom.com/wiki/Main\_Page

## GUI

I have taken some screen shots of the interface of the original game to break down and analyse, this will help me design my game to be more visually aesthetic and simple for the player to interact with. Decomposing the GUI will allow me to break down coding each element of the interface into separate tasks which over time will be completed individually per session. This way each part of the interface will be separate and can be easily worked on and changed when improving the visuals of the game in further iterations. Elements can be moved around and changed as needed based on feedback from my stakeholders.

The game consists of three parts; the title screen, map screen (not present in the original Shadez but in the Shadez Sequels there is one) and the gameplay screen.

### Title screen (Shadez)

This screen is the first thing the player will see after running the game. The title of the game with graphics are present, setting the scene and atmosphere of the game from the get-go. A menu is present with four options; Play, Game Guide, Trailer and Credits. After clicking play, the player is shown different levels they can play; The Flat Plans (Easy), The Red Wood Hill (Intermediate), Twin Peaks (Intermediate) and The Big Long War (Hard). The player interacts with the menu using the courser, and each button becomes highlighted when the courser. I plan to allow the user in my version of the game to interact with menus in the same way.

A screenshot of a cell phone

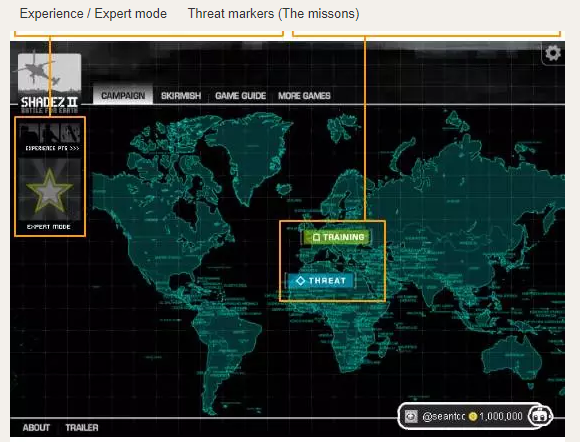
Description automatically generated

### Map Screen (Shadez II)

The map screen is the screen where the player can select missions to play. Players can see their progress and advance through the story by completing levels as they are unlocked. As suggested by my stakeholders, I plan to create a similar feature to this although the original Shadez did not have this. Therefore, a storyline can be created and implemented, making the game more interesting. The Map has a view of the world map and shows the location of each mission visually. When these levels are opened, they are themed around their location on the map. I also plan to integrate features like unit upgrade screen accessible from the map screen, and other windows displaying the players progress and money they possess for upgrades. As missions are selected, a window pops up further explaining information about the battle with a picture. There is also a back button to return to the menu, and a start button to load into the battle, taking the player to the battle screen. I plan to create a similar prosses for starting missions in my version of the game.

(Need to insert other pic of start button)

Screenshot taken from <http://www.crazymonkeygames.com/guide/Shadez-2/>



Players amount of money and username

### Gameplay Screen

This screen is where all the action and gameplay occur. Here the player oversees creating and leading their army to defeat the waves of enemies. This view consists of two main parts; The landscape window and The UI. The landscape window in the original game can be used to select units to control their actions, I don’t plan to integrate this within my version of the game within the first iterations. The UI is where the player can interact with the game with their courser. There are several buttons of which when clicked complete different actions on the battlefield. Unit buttons will spawn the shown unit onto the landscape when clicked, then the unit will move along and interact with enemies until it reaches the enemy’s base. Money is deducted from the players total whenever a unit is bought. The players health, money and stage count are also displayed in the UI. Whenever a button is highlighted by the courser, the help text located at the bottom of the UI displays text describing the action carried out if the player clicks the button.

Animations when Units are purchased, when the build time is complete, they spawn in on the landscape. Also shown is the locked unit visual.



Health Bar

Money Counter

Unit

Start, Stop and Fast Forward

Stage Counter

Map Overview

Buy Unit Button

Help Menu

Landscape Window

UI

## Mechanics of Shadez

When I create my version of Shadez, its important that the mechanics of the original game are understood so they can be implemented into my version to work like the original. By Mechanics I more specifically mean the processes the program makes as units interact with each other, and how the game model will measure the players progress through the level or reward the player for progressing within the level. These mechanics are what allows the gameplay to become meaningful as events are triggered when certain conditions are fulfilled.

### Unit Combat

As Friendly and Enemy units move along the landscape, they need to engage in combat after a certain range is reached between them. In Shadez, each type of unit has different engagement ranges. For example, a gun truck can engage a tank from a much further distance than the tank can engage it. This creates the opportunity to deploy units tactically to destroy units from a range, so therefore each unit class will have an attribute stating its combat engagement range. Attacking Units will have to deduct health from the unit under attack and will also remain stationary during attacking. I also plan to change the image of the unit to an attacking image so that the player can see the action occurring. Units will attack at different rates and with different strengths dictated by their class. For example, a marine will be able to deal damage every second, but with weak force. A Tank however will be able to deal massive damage but only every ten seconds. When a unit’s health is depleted it will de-spawn with an explosion animation. All attributes like strength and attack speed will be stored in the unit class per unit type.

### Game Progress

In the original Shadez game, the mission of the game progresses in waves, and the game is endless, until the player is defeated. I am not going incorporate this endless design into my project as I feel the missions in Shadez 2 which are not endless would make it easier for me to create a story line around. Shadez 2 however has a game storyline which I would like to replicate in my game, so players will become more invested in the story of the game. My stakeholders have reported when they played the original game that they felt that a storyline would make the game less repetitive.

## Research Questionnaire Completed By Stakeholders

Below is the questionnaire I created to find out about what my stakeholders thought my game should include.

Research Questionnaire

**Would you like the game to have a story line to play through?**

Ollie: “Yes, It will give more satisfaction when completing levels and progressing.”

Rose: “Yes, I think you can do a really funny story”

Mark: “Yes, stories are interesting”

**Would you appreciate a tutorial level at the start of the game?**

Ollie: “Yes, that would be useful for the player.”

Rose: “Yeah, makes things easier”

Mark: “Yes, it’s a good way to learn how to play the game whilst still having fun”

**Would you like the game to have music in the background?**

Ollie: “Yes, music is always good in a game.”

Rose: “Yes”

Mark: “Yes, suitable music always sets the scene better”

**Would you like to be able to spend money outside of missions to upgrade or buy new units?**

Ollie: “Yes, otherwise there is too much pressure to upgrade units during the game.”

Rose: “Yes, that’s a good idea”

Mark: “Yes, that would make the game simpler”

**Would you appreciate different backgrounds for different levels in parallel to the level location?**

Ollie: “Yes, it would make the story more integrated within the gameplay.”

Rose: “Yes, because the player will know the battle is in different places”

Mark: “Yes, that would set the scene better”

**How many levels would you recommend the game to have in the story? A) 5 B) 10 C) 15 D) 20**

Ollie: B) 10

Rose: B) 10, Quality over quantity.

Mark: A) 5, but longer levels.

**Would you like to complete missions with special challenges which reward you with extra money which are optional?**

Ollie: “No, I find it to be too frustrating as I am not very good at playing games.”

Rose: “Yes, Bad players don’t need to complete the challenges to enjoy the game.”

Mark: “I don’t think this is necessary in the first versions of the game you make.”

**Would you like controls to be using the keyboard, a mouse, or both?**

Ollie: “Just the mouse.”

Rose: ” Just the mouse to keep it simple”

Mark: “The mouse, as I would get frustrated remembering keys to use for actions.”

**Can you think of any other aspects you would like to be integrated in the game?**

Ollie: “No.”

Rose:” Score or time to complete missions, so the shorter time a level is completed, the better score.”

Mark: “No”

## Analysis of the research questionnaire

Using the answers gathered by my questionnaire, I now have an idea of the features my stakeholders will appreciate me including, and the features that they consider to be irrelevant. In conclusion of the research, I have made the decision to incorporate; a storyline in my finished product, a tutorial level, background music, different backgrounds for levels, a 10-level storyline, mouse controlled and a time counter on missions. I have discovered that using keys to interact with the game was undesired by my stakeholders, and that they would prefer the quality of the levels to be high rather than for me to focus on quantity.

It was extremely beneficial to carry out this questionnaire so I can present my ideas to my stakeholders which provide me feedback for the final design of the program. Whenever I make decisions which will affect the result of the game, I will always use advice from my stakeholders. I have created a table of stakeholder requirements based upon the questionnaire to better visualize their discovered design requirements

|  |  |
| --- | --- |
| Stakeholder Requirements | Explanation |
| Timer | Timer variable that counts time within a level in real time. |
| Enjoyable Storyline | To include a storyline within the gameplay to make the missions meaningful. |
| Playable Tutorial | Rather than a screen displaying the instructions, have a playable tutorial level which will allow for better understanding of the game. |
| Implemented Background Music | To include background music to intensify the mission. |
| Integrated Unit Upgrade Screen | Separate unit upgrade section to boost unit stats. |
| Varying Backgrounds | To implement changes of background per mission to suit the storyline. |
| 10 Level Storyline | To have 10 missions with different aspects making them unique from each other. |
| Mouse controls via Buttons | To not use the keyboard to interact with the game. |

# Success Criteria

I have laid out a table to better represent the criteria that would complete my proposed solution. There are two categories of Criteria I have considered for my proposed solution to be successful: Functional Criteria and UI criteria.

|  |  |
| --- | --- |
| **Criteria** | **Evidence Which can be tested** |
| Simple Gameplay Interface with understandable layout. | To have a suitable interface where gameplay can occur. Ask stakeholders to provide feedback whether the gameplay GUI is suitable for enjoyable gameplay. |
| Working Curve Class | Have a curve class which contains attributes of the curve points which can be accessed by the form. |
| Generated Landscape painted onto landscape window. | Visible landscape line generated in the correct position of the screen which follows dictated points form the curve class. |
| Working Unit Class | Instances of units can be created with appropriate values and procedures pass values to and from the Form. |
| Working Unit Specific Classes | Create unique unit classes with working functions and procedures to pass values to and from the Main Unit Class later on. |
| Working Unit Definition | Procedure which creates a specific unit instance within the original unit instance for attributes to be read into the original instance. |
| Correct unit Spawn position | Units can be spawned in the appropriate position depending on whether they are enemy or friendly units. |
| Calculation of new unit Coordinates | Correct new coordinate values are calculated within the unit function for units to be moved across the landscape. |
| Working Moving and Spawning Unit Procedure | Graphics are updated with new unit position. |
| Implemented Speed Variability. | Units can be assigned a speed value as a float, which should affect the rate at which they move across the landscape. For both Friendly and Enemy units. |
| Game Model Class | Create a Game Model class which will initially contain all the current locations of units spawned on the screen. |
| Working Collisions Detection of Unit Ranges | Within the Game Model class, compare the locations of units with location of asking unit and return the correct ranges of the X coordinate values in an array. |

|  |  |
| --- | --- |
| Units interact when within range. | When the Timer ticks, units within range of each other will stop moving and exchange damage values to deduct health from each other. |
| Unit Death when Health < 0. | Units will disappear from the screen when their health becomes 0, and all their activities stop. |
| Unit Instance Re-use | Game model can track the status of each unit, and unspawned dead units can be reused later in the gameplay, discarding the need of creating new instances of unit. |
| Units de-spawn when out of range of the landscape. | When the unit runs of the landscape, they de-spawn so they can be re-used. |
| Enemy Unit automatic Spawn | ‘Waves’ of single enemy units are created, which are fully automated. |
| Working Money Counter | Money Counter increases per wave and decreases when a friendly unit is spawned. |
| Win detection | When the player finishes the waves of a mission, a win screen will appear as the gameplay is stopped. |
| Suitable Welcome Form | When the game opens, A welcome form opens which asks for the player name and has a button to start the tutorial. Needs to look professional. |
| Playable Tutorial Initiation | First level to have a playable tutorial interface to teach the players the workings of the game. |
| Aesthetically Appropriate Map Screen | Easy to use Map Screen where unlocked missions are available for selection. Appears after every mission finish. |
| Seamless running of the overall program. | The program runs without bugs and provides the player with an enjoyable experience. Map screen mission buttons load correct valid missions. |
| Usability for impaired users. | Text is readable, The game can be easily understood and navigated by impaired users (Such as users with reading impairment) |

## Hardware requirements

I am creating a game in Visual studio 2017, coded in the language C Sharp. It will run within the operating system Windows 10. The system requirements for a computer to run windows 10 are:

* CPU: 1Ghz Processor
* Storage: 16GB for 32 bit, 20GB for 64 bit.
* RAM: 1GB for 32 bit, 2GB for 64 bit.
* Screen Resolution: 800x600
* Graphics: Microsoft DirectX 9
* A keyboard.
* A monitor.

Therefore, any computer systems which my game will run on will require these attributes as a minimum.

## Software Requirements

In order to run Visual Studio 2017, the computer is required to run a windows operating system. When finishing each iteration, I will export the programs into an executable file which will allow the player to run the game simply by opening the file and will not allow for alteration of code.

# Iteration 1

## Discussion with Stakeholders Regarding Iteration Priorities

## Post development usability test plan

|  |  |  |
| --- | --- | --- |
| **Mechanics to be Tested** | **Type of Test** | **Expected Results of Test** |
| Buttons Run Desired code when clicked. | BlackBox | * Breakpoint to be triggered when button triggers correct code. |
| Curve Class Set method validation. | Unit Test | * Set methods change correct class attribute values. |
| Curve Class get method validation. | Unit Test | * Get methods return correct values from class attributes. |
| Generate Correct Landscape painted onto landscape window (Draw Procedure) | BlackBox | * Visible landscape line generated in the correct position of the screen which follows dictated points form the curve class. * Look at the landscape window to see manipulation of line when points are change |
| Unit Class Set method validation. | Unit Test | * Set methods change correct class attribute values. |
| Unit Class Get method validation. | Unit Test | * Get methods return correct values from class attributes. |
| Spawn Unit Procedure Testing | Unit Test | * Correct starting location calculated (Curve start point X – Unit image width = Unit X location, Curve start point Y – Unit image height = Unit Y location * Unit spawns on the left if it is friendly, and on the right if its an enemy unit (Controlled by speed multiplier, if speed is positive the unit is friendly, negative the unit in enemy.) * Picture box instance is created with correct image * Picture box is cropped correctly to Unit Width and Hight values. * Picture box appears on the landscape in the correct position. |
| Unit Class Get method validation. | Unit Test | * Get methods return correct values from class attributes. |
| Unit Definition Procedure | Unit Test | * Passed in parameter triggers correct code (breakpoint) * Correct specific unit instance created * Correct values read into attributes from new instance. |
| New Unit Location Procedure | Unit Test | * Calculation of correct Xdifference * Calculation of correct Ydifference * Calculation of correct gradient between Xdifference and Ydifference * GradientAdd runover double calculation is added from original gradient * GradientAdd if greater than 1 increases or decreases the unit up value. * Gradient add when greater or less than 1 calibrates its value back by +- 1 * LeftAdd is calibrated +- 1 back to its original value. * Function returns correct new point. |
| Speed variability test | BlackBox | * Speed multiplier in unit specific class correctly effects the rate at which the unit picture box moves across the landscape. |
| Game Model Get Method validation | Unit Test | * Get methods return correct value when called. |
| Game Model Set Method validation | Unit Test | * Set method transfers parameter correctly into game model’s attribute value. |
| Collision Detection Testing | BlackBox | * unitLocation’s elements are all compared with the parameters unitLocation element value. * No errors caused by unitLocations becoming out of bounds. |
| Unit Interactions Testing | BlackBox | * Unit movementValid attribute is set to false when the unit is in range of an enemy unit. * Units take appropriate damage. * Unit death occurs when their health attribute becomes <= 0, picture box disappears from the landscape. * Unit movementValid is set to true when no enemy units are in range. |

## Proposed Solution and Purpose of Iteration 1

The first iteration will implement the mechanics of the game which the program will use in future iterations for the development of gameplay. At the end of this iteration, the backbones of the program will be expressed through a simplified gameplay Form which will allow testing and demonstration of key mechanics that have been created. This will allow me to implement Blackbox Testing to catch any logic errors in methods which are more easily evaluated through changes in the Form and will also allow me to demonstrate the programs current capabilities to the stakeholders. The stakeholders will be able to better analyse and understand the created mechanics as they are in a simplified environment.

### Requirements for iteration 1 (Success criteria)

Using decomposition, the solution to be completed can be split into sub tasks which can be allocated deadlines based upon their priority (aspects of the iteration stakeholders and myself consider different levels of importance). These programming tasks will be followed sequentially to formulate a correctly structured solution.

The requirements themselves will be the result of decomposition of the proposed solution. The criteria of which the solution is considered successful depends if the requirements of the iteration are met and successful. The sub tasks will be derived from further decomposition of the iteration requirements and will each influence the completion of each criteria.

#### Stakeholder Priority

I calculated the stakeholder priority of each requirement by explaining each criterion to my stakeholders in an easy to understand way which required little/no software development knowledge. Oli (He is in my computer science class so has knowledge on OOP) received a more in-depth explanation of the criterion as he would be able to provide an insight from someone with greater knowledge of the subject. I asked each stakeholder after explaining each requirement what they would rate it out of ten in level of importance. This rating will later be considered when allocating programming time to each sub-task.

|  |  |  |
| --- | --- | --- |
| **Requirement/Criteria** | **Evidence Which can be tested** | **Stakeholder Priority (Out of 10)** |
| Simple Gameplay Interface with understandable layout. | To have a suitable interface where gameplay can occur. Ask stakeholders to provide feedback whether the gameplay GUI is suitable for enjoyable gameplay. | **10**, ‘Crucial for me to be able to visually judge the progress of the iteration, and of course allow for a more polish program in the long run’ - Mark |
| Working Curve Class | Have a curve class which contains attributes of the curve points which can be accessed by the form. | **5**, ‘If the curve class does not allow points to be passed, point locations could always be hard coded into the solution’ - Oli |
| Generated Landscape painted onto landscape window. | Visible landscape line generated in the correct position of the screen which follows dictated points form the curve class. | **10**, ‘This is important as otherwise how would the game even work if there was no landscape for the units to travel across?’ -Mark |
| Working Unit Class | Instances of units can be created with appropriate values and procedures pass values to and from the Form. | **10**, ‘As I understand what classes are, the importance of the unit class working correctly must need to have a high priority because otherwise the units will have bugs working correctly.’ - Oli |
| Working Unit Specific Classes | Create unique unit classes with working functions and procedures to pass values to and from the Main Unit Class later on. | **5**, ‘It would be a nice thing to accomplish having specific units in this iteration, but a standard unit could always be set as a default to start with’ -Rose |
| Working Unit Definition | Procedure which creates a specific unit instance within the original unit instance for attributes to be read into the original instance. | **5**, see above quote. |
| Correct unit Spawn position | Units can be spawned in the appropriate position depending on whether they are enemy or friendly units. | **10**, ‘This is obviously important as the game wouldn’t work otherwise’ - Mark |
| Calculation of new unit Coordinates | Correct new coordinate values are calculated within the unit function for units to be moved across the landscape. | **10**, ‘The units would have to move across the landscape correctly for the game to be playable!’ - Rose |
| Working Moving and Spawning Unit Procedure | Graphics are updated with new unit position. | **10**, ‘The visuals would have to be correct because otherwise the game wouldn’t be developed properly’ - Rose |
| Implemented Speed Variability. | Units can be assigned a speed value as a float, which should affect the rate at which they move across the landscape. For both Friendly and Enemy units. | **4**, ‘It’s not that important because at this point, I just want to see that a unit can move correctly on the screen regardless of speed’ – Oli |
| Game Model Class | Create a Game Model class which will initially contain all the current locations of units spawned on the screen. | **8**, ‘From how you described it, this sounds like an important mechanism of the game. Obviously, It should be prioritised.’ - Mark |
| Working Collisions Detection of Unit Ranges | Within the Game Model class, compare the locations of units with location of asking unit and return the correct ranges of the X coordinate values in an array. | **8**, ‘Eventually this feature will defiantly need to be implemented in the game, but the reason I’m not rating it 10 is because other parts of the program would be more relevant at this stage’ - Oli |

|  |  |  |
| --- | --- | --- |
| Units interact when within range. | When the Timer ticks, units within range of each other will stop moving and exchange damage values to deduct health from each other. | **7**, ‘Yes it sounds important but it’s not the end of the world if you don’t complete it in the first iteration’ |
| Unit Death when Health < 0. | Units will disappear from the screen when their health becomes 0, and all their activities stop. | **8**, ‘It wouldn’t be hard to implement a method such as this so its worth the small amount of time doing’ - Oli |
| Units de-spawn when out of range of the landscape. | Game model can track the status of each unit, and unspawned dead units can be reused later in the gameplay, discarding the need of creating new instances of unit. | **5**, ‘It’s not really important at this stage’ - Mark |
| Unit Instance re-use | When the unit runs of the landscape, they de-spawn so they can be re-used. | **5**, ‘Same as before, it’s not really important in this iteration, compared to other aspects.’ - Mark |

### Decomposition and Sub Task Time Allocation

I have allocated 30 hours of programming time to complete all the requirements for this iteration to be successful. Sub tasks are a result of the main problem being separated into manageable tasks.

(Unfinished section)

## Design

### User interface design

The User will be interacting with the program during the total three moments of the game; The start-up of the game, actual gameplay and the mission selection stage (Map screen). Through each stage of playing the game, the player is viewing and interacting with the programs different Forms. The game should switch between the forms seamlessly, so the running of the game is smooth. Each form is designed for easy interaction and to have a professional appearance. I have created the diagram below to better show how the player will progress through the stages in the game.

(Success Criteria: Seamless Running of the whole program)

**Start Up Welcome Form**

**Gameplay Form**

**Map Screen Form**

#### Start Up Welcome Form

The welcoming form is required to look good as it’s the first impression the player will get of the game. It will include a title screen, welcoming paragraph with some background history of the story line, username input and a suitable background. There will be a large start button which will appear after the user enters their name which will lead onto the Gameplay Form.

**Shadez**

**Shadez**

**SHADEZ A War Game**

Enter Name Here

(Introduction Paragraph)

START

Above is my initial basic design for the Start Up Welcome Form. The light grey background colour represents the background image that I will include in the future. The screen is simple and bold, with the start button easily accessible for a quick start to the game. The introduction paragraph will introduce the story but will not be to lengthy for players to skip reading. (Stakeholder requirement: Enjoyable storyline / Success Criteria: Suitable Welcome Form) I will develop further on the paragraph later in the design of the story.

Stakeholder Feedback

#### Gameplay Form

**Shadez**

Real Time

Money Counter

Landscape Panel

Health

Wave Counter

Buy Unit (Price)

Buy Unit (Price)

Buy Unit (Price)

The gameplay form I have designed bares a lot of similarity to the gameplay screen of the original Shadez (See page 11). The layout of the landscape panel and buttons remains the same however the complexity of the controls has been simplified in parallel to the simplified gameplay. (Success criteria: Simple Gameplay Interface with understandable layout.).

The money counter is positioned in the top centre of the screen so it can be easily seen before the player buys a unit. The Real Time counter has the time currently spent playing the mission displayed and is placed in the top right of the form, so it is out of the way of the gameplay and other important counters. The Landscape panel is where all the action takes place, therefore is positioned just above the centre of the Form. It takes up the most space as it will be the centre of attention. The Health and Wave Counter are positioned below the landscape panel to the right of the form, away from the unit purchase buttons. These are apart from the money and time counters, so the player does not get confused searching for the information they need during the gameplay. The Buy Unit buttons are positioned to the left underneath the landscape panel, they will have a picture of the unit and price on them for the player to easily identify and buy the units they want during the gameplay. This will allow for a more enjoyable gaming experience for the player.

#### Map Screen

Again, the map screen will resemble a lot of similarities to that of the one used in Shadez 2, but simpler. This will allow for a more basic enjoyment and understanding of the game. I do want the deign of the form to really create an atmosphere for the game, as if the player was a real-life army commander of a futuristic army. To do this I will have an appropriate background image of which I will ask my stakeholders for advice. (Success criteria: Aesthetically Appropriate Map Screen).

**Welcome PlayerName**

**Encounter**

**The Beginnings**

**Mountain Attack**

**Missions Overview**

**Exit**

**Overseas Development**

**You Wot**

**Not Epic**

**Invasion**

**Counter Invasion**

**Australia Is not Epic**

**Indiana Jones**

You can see the background map which shows where the mission locations are. This is the Hub for all the progress made in the game. Players can see their progress through the game and decide whether to replay missions or play newly unlocked ones. This allows for simple progression through the missions of the game whilst the player understands the developing story. An exit button is located out of the way in the top left-hand corner for if they want to exit the game. Mission buttons will appear as missions become unlocked. When the mission button is clicked, a box with a start mission button will appear with an explanation to the next mission goals (development of the storyline).

An exit button is located in the mini box’s top left-hand corner to make the box disappear so that the player can reselect another mission on the map view. The title textbox is positioned at the top of the box with the explanation below. Finally, the start button I have created is large and to the centre of the box, so that players can easily start a stage and move onto gameplay.

**START**

X

Australia be squaring up tryina use force to come at big up America, America yanno is sound and that so lets nuke them. - Sarge

**Australia is not epic**

### Usability Features.

Having reading impairment myself, I understand Its crucial my game can be played and enjoyed by any user. Therefore, my program will be appealing for a larger audience of players who might not be able to play and enjoy other strategy games, as often these games are complicated and are filled with text. I have considered that not all players will appreciate lengthy chunks of text or complicated menus, so my simplified version of the original Shadez will be easier to understand and enjoy by all players.

#### Text

Small text and funky fonts create issues for some users with reading issues. I have decided to keep the text font as large and readable as possible. The Font Georgia is rated easy to read compared to usual fonts therefore I will use it for any reasonable amounts of text in my program, such as for the mission descriptions or the opening paragraph of the game in the welcoming form. This should allow players with reading impairments to read the story of the game more easily, allowing for a more stress-free enjoyable gaming experience.

#### Simple Selection

I have avoided using ‘menu layout’s in my program as this sort of layout can become complicated and hard to understand when options are all represented by text. An example of this in the context of my game would be listing all available missions in a menu instead of using a map on the map screen. This would be inappropriate for users who prefer to associate options with visuals when selecting. Therefore, I have designed the map screen in a way player can select unlocked missions from locations of a map. Players can associate the unlocked missions by their location on the map and creates easier interaction in the mission selection prosses.

I have made buttons large and easy to click for users who would normally have issues with trying to move the mouse over a small area. This allows players who have to put in effort when completing precise movements to worry less about interacting with the game, so that they can have a more enjoyable experience focusing on the gameplay.

## Algorithm Explanation

## Landscape and system drawing (With Pseudocode)

Success Criteria: Generated Landscape painted onto landscape window. Working curve class.

On the gameplay form, the landscape needs to be plotted at the start of a mission, within the landscape panel. This will be the line that units will move across. So, the production of this will occur after a basic gameplay GUI is created on the form. The panel will have to have a sensible location (for example, (50,50)) as units’ location on the form will be later processed within functions to control their movement along the landscape.

I will create a ‘Curve Class’ which will contain the attributes and methods of the landscape per mission. The Landscape will change per; therefore, the landscape plotting method should be able to plot a line based of points given for it to pass through. So these points the line passes through should be stored as the lines attributes which can be changed through point set and get methods.

(50,50)

(X,Y) Point 4

(X,Y) Point 3

(X,Y) Point 2

(X,Y) Point 1

(50,250)

The Points will be declared within the class curve. These points will be used within the class to plot the landscape as the DrawLandscape procedure is called. These points can be returned and changed through the methods of the class (get and set points methods).

Class Curve()

{

Point Point1 = new Point(X,Y)

Declaring the Point types.

‘’ ‘’

‘’ ‘’

curvePoints = new Point[] { point1, point2, point3, point4, point5, point6, point7 };

Declaring the array of Points.

}

Public Procedure setPoints(Point[] newPoints)

{

curvePoints = newPoints;

}

Public Point[] getPoints()

{

return curvePoints;

}

#### Landscape creation

In Windows Forms, using system drawing I can use the created graphics instance and the pre defined DrawLines(pen, curvePoints); method to plot the points in the curvePoints array onto the screen. When the gameplay form is refreshed, the landscape should appear on the screen. Initially this code will be ran when a temporary button is clicked on the form, therefore allowing the procedure to be tested with different point values. After the other forms of the program are finished, this code will be ran as part of loading a mission from the map screen.

public void Draw(Graphics g, Pen pen)

{

g.DrawLines(pen, curvePoints);

}

## Unit Class

Success criteria: Working unit class.

The unit class is separated into its methods and attributes. Its attributes are at the top of the class so values can be added and changed as the program grows in complexity. Then the class’s methods will be encapsulated as public functions and procedures so that they can be called by the main program.

The class should contain the attributes of the unit’s general values, such as location and spawn state. But the class will also contain unit specific attributes which will later contain values loaded from instances of new specific unit class types. For example, the class will contain a speed attribute which will remain default until the unit is defined and the specific speed to the unit type is read in. These attributes include:

None Specific:

* bool movementValid = false;
* unitLeftValue = 0;
* int unitUpValue = 0;
* double gradient = 0;
* bool spawned = false;
* int Ydifference = 0;
* int Xdifference = 0;
* double gradientAdd = 0;
* double leftAdd = 0;
* int pointVariable = 0;
* int arrayPosition = 0;

Specific to unit type:

* int unitType = 0;
* int unitHight = 0;
* int unitWidth = 0;
* double unitSpeed = 0;
* int unitRange = 0;
* int unitHealth = 0;
* int unitAttack = 0;
* bool unitAlive = false;

The class will have methods to allow the values of the attributes to be read and altered. These are the set and get methods of the class. This will allow for the virtual game model to read the status of spawned units and therefore control any interactions. The game model will later be implemented as a game class. The other methods will control how the unit interacts with the game environment. At this point in time, an instance of unit will spawn at the correct location on the landscape using a spawn and unit definition method.

### Unit specific class

Success Criteria: Working Unit Specific Classes.

General Unit Parent

Enemy Tank Specificity

Friendly Tank Specificity

Unit definition method(Enemy tank setting parameter)

Unit definition method(Friendly tank setting parameter)

Unit specific classes contain the specific attribute values of a unit. Tanks will have different attack and health values to cars, therefore the general unit class can’t contain all of the unit specific values without a unit definition method. What the unit definition method will do is create an instance of a specific unit object inside of the general unit class, depending on the parameter value passed into the procedure. The integer passed in represents what unit the object will become, causing the corresponding instance of unit specific class to be created.

Then, the general unit class will call the unit specific class’s get methods to read in the new unit’s attribute values into the general unit class’s unit specific attribute values. The instance of picturebox will have its image changed to the unit image corresponding to the procedures parameter number. In the phuedocode below, the passed in unitType value 1 will cause a friendly tank instance to be created of which its methods retrieve the friendly tanks speed into the general unit class’s speed attribute.

Success criteria: Working Unit Definition

Public void unitDefinition(int unitType)

{

If unitType == 1:

{

FreindlyTank FT = new FreindlyTank();

unitSpeed = FT.getSpeed();

‘’ All other attributes get methods’’

}

}

### Spawn Unit Procedure

Success criteria: Correct unit Spawn position

The spawn unit method will be called from the class to spawn a unit when the player buys a unit. For now, the method will be called when the program user clicks the Spawn Unit button to allow simplified testing. The unitAlive attribute will be set to true for reference, and different spawning algorithms will be called based on the speed multiplier of the unit. This is because the speed multiplier of the unit represents if the unit is friendly or enemy, A positive speed is read as a friendly unit and a negative speed is read as an enemy unit.

If unitSpeed >0 then

{

#Friendly unit spawning algorithm

}

elIf unitSpeed <0 then

{

#Enemy unit spawning algorithm

}

else

{

#Failure

}

Next, the spawning algorithm for both friendly and enemy units follow the same steps to correctly spawn the unit onto the landscape. (1) The units starting location is calculated, locating the unit at the start of the landscape. The unit image height is considered to position the image correctly on the form. (2) Picturebox instance location set to the calculated starting location. (3) The picture box instance dimensions are set to the unit image dimensions, and the background of the picturebox instance is set to transparent. (4) Picturebox instance is added to the form. (5) Unit spawned is set to true so the game model can recognize the unit is present. (6) The unit left and up attributes are set to the calculated start point.

int offset = curvePoints[curvePoints.Length-1].Y - unitHight; (1)

Point startingPoint = (curvePoints[curvePoints.Length-1].X - unitWidth, offset); (1)

Unit1.Location = startingPoint; (2)

setUnitUp(startingPoint.Y); (2)

Unit1.Visible = true; (2)

Unit1.BackColor = Color.Transparent; (3)

Unit1.Width = unitWidth; (3)

Unit1.Height = unitHight; (3)

screen.Controls.Add(Unit1); (4)

screen.lanscapeBounds.SendToBack(); (4)

screen.Refresh(); (4)

spawned = true; (5)

unitLeftValue = point1.X; (6)

unitUpValue = point1.Y; (6)

### New Location Procedure

Success Criteria: Calculation of new unit Coordinates

The new unit procedure calculates the new location of the unit picture box to move the unit by a factor of its speed across the landscape line. Because I have defined the landscape to pass through several known points, the unit can be tracked as it goes across the landscape, and therefore the two points which the unit is between can be identified. Using the location of two points between the line, the unit can be moved left by an amount of Speed, and up by an amount of calculated Gradient between the two points it is situated \* Speed.

Using Decomposition, the procedure has been constructed step by step to confront the following problems which provide a solution to returning a new calculated location point. The Below pseudocode is a design meeting the criteria of the function needed. The code is repeated for units moving in the opposite direction except the details of the gradient calculation work in reverse, and the left value becomes decreasing.

procedure newUnitLocation(Point[] curvePoints)

The procedure takes in the array of points representing the landscape so the location of each vertex of the landscape can be used for calculation. The new unitLeftValue and unitUpValue have their values changed by the procedure and are not only used to return the calculated new location, they are the units location attributes and remain changed in parallel to the units moved location on the landscape.

if (movementValid == true) (0)

{

if (unitSpeed > 0) (1)

{

//Here the program calculates the gradient of the line the

//Unit is currently on, using the points its between

if (unitLeftValue < curvePoints[1].X - (unitWidth / 2)) (2)

{

Ydifference = curvePoints[0].Y - curvePoints[1].Y; (3)

Xdifference = curvePoints[0].X - curvePoints[1].X; (3)

}

else if (unitLeftValue < curvePoints[2].X - (unitWidth / 2)) (2)

{

Ydifference = curvePoints[1].Y - curvePoints[2].Y; (3)

Xdifference = curvePoints[1].X - curvePoints[2].X; (3)

else if( ‘’ ) (2)

{

(3)

}

else (4)

{

Ydifference = 0; (4)

Xdifference = 0; (4)

}

//Gradient calculation here, using (double) for a none rounded result.

gradient = (double)Ydifference / Xdifference; (5)

gradientAdd = gradientAdd + (double)(gradient \* unitSpeed); (5)

leftAdd = leftAdd + (1 \* unitSpeed); (6)

if (gradientAdd > 1) (7)

{

unitUpValue = unitUpValue + (int)gradientAdd; (7)

gradientAdd = gradientAdd - 1; (7)

}

if (gradientAdd < -1) (7)

{

unitUpValue = unitUpValue + (int)gradientAdd; (7)

gradientAdd = gradientAdd + 1; (7)

}

//Now that the speed multiplier is creted, left add is used due to

//the limmitations of moving a unit up

//only pixle by pixle, so the unit is only moved when its total Y

//movement is greater than 1 pixle.

if (leftAdd > 1) (8)

{

unitLeftValue = unitLeftValue + (int)leftAdd; (8)

leftAdd = leftAdd - (int)leftAdd; (8)

}

if (leftAdd < -1) (8)

{

unitLeftValue = unitLeftValue + (int)leftAdd; (8)

leftAdd = leftAdd + (int)leftAdd; (8)

}

//New point which is returned is created here.

Point newPoint = new Point(unitLeftValue, unitUpValue); (9)

return newPoint; (10)

}

1. The movement of the unit should only occur if the unit is in a state of movement, therefore the movementValid attribute of the unit should be identified to be TRUE before any further code is called. The If statement is used to validate if the new location needs to be calculated.
2. The unit’s direction needs to be identified, represented by their sign. If the unit speed is positive, then the location calculation will execute the correct calculations for movement from right to left. If the unit speed is negative, then different code is executed to calculate the unit’s movement from right to left.
3. The procedure needs to compare the unit location to the points in the landscape so the correct points its in-between can be identified. A series of IF statements are used to compare all the points of the landscape to the unit location. The unit width attribute is used to make sure the unit picturebox is moving across the landscape correctly from its centre point, not from its left-hand side.
4. The gradient needs to be calculated between the two identified points. This is done by dividing the difference between the Y coordinates by the difference in X coordinates. Therefore, the X difference and Y difference of the two points are calculated here to be used in a calculation further in the procedure.
5. If the program fails to identify the points the unit is between, or the unit is out of range for whatever reason, the procedure has an else statement to catch the exception and will set the X difference and Y difference values to be 0.
6. The gradient is calculated using the X difference and Y difference and is stored as a double because the division calculation will often create a decimal value. To overcome this problem, the gradient will be stored as a double and I have created a new value called gradientAdd. ((7) Elaborates on this principal). This will allow the unit to move correctly along the landscape a pixel at a time.
7. The Horizontal movement of the unit needs to be calculated using the speed attribute of the unit. The speed attribute is stored as a decimal (double) so a variable named leftAdd is created to move the unit a pixel at a time, the same principal as using gradientAdd.
8. Gradient add is used due to the limitations of moving a unit up only pixel by pixel, so the unit is only moved when its total Y movement is greater than 1. gradientAdd keep the remainder of the previous movement up or down by factors of 1, and the remainder is carried forward each sum to move the unit upwards or downwards at the right rate. This IF statement is used to detect when gradientAdd is greater than 1 or -1 indicating the unit to be moved up or down by a pixel.
9. Due to the instance of the speed double attribute being used in the left movement calculation, the leftAdd principal is introduced just like gradientAdd is used. The program moves the unit left everytime the leftAdd value is smaller or larger than 1. This detection of size is found using an IF statement.
10. The program constructs a return point setting the X coordinate to the unitLeftValue and the Y coordinate as the unitUpValue.
11. The point instance is returned.

#### Timer Influence on unit movement

Success criteria: Working Moving Procedure (Working Moving and Spawning Unit Procedure)

The game’s action will be coordinated by a timer in the gameplay form. Per timer tick, the program will execute unit movement, location recording and collision detection. Simple IF statements check each instance of Unit’s movementValid to be TRUE, and if so, move the unit’s instance of picturebox to the new location calculated by the unit’s newLocation method. The game model is also updated so the virtual model of the unit’s location is kept update.

#### Speed Multiplier Influence on unit movement

Success Criteria: Implemented Speed Variability.

As different types of units move at different speeds, the unit speed is represented as one of the unit’s attributes. Units moving at different speeds will have different speed attributes. The design of the speed multiplier allows its value to be multiplied to the calculated gradient and unitLeft value to change the rate at which the movement moves. The speed attribute is very important in defining if the unit is an enemy or friendly unit. (See unit spawning procedure (pg32,pg33). The greater the Modulus\* of the speed value is the faster the unit will move across the landscape.

\*Mathematical context of Modulus.

### Take Damage Function (Development of efficient unit death)

Success Criteria: Unit Death when Health < 0, Units interact when within range.

This function is responsible for the taking of damage and death of units. This was implemented after the collision execute procedure when the time a unit was needed to take damage became identifiable. Units have a health attribute which should be deducted by the attack value of the attacking unit. This function also triggers the death of a unit when its health becomes less than 0. The reason the function needs to return a value is because the procedure which will call this method will require to know if the victim unit has died. The function returns a Boolean value, the victim unit is dead(TRUE) or alive (FALSE) after the damage dealt. There is further information on this in the Collision Execute Procedure subsection.

public bool takeDamage(int damage) (0)

{

unitHealth = unitHealth - damage; (1)

if (unitHealth < 0) (2)

{

//Death

Unit1.Visible = false; (3)

unitUpValue = 0; (4)

unitLeftValue = 0; (4)

movementValid = false; (5)

unitAlive = false; (6)

return true; (7)

}

return false; (8)

}

1. The function takes in the damage to be dealt to the unit; this is the attacking units attack attribute.
2. The damage Is subtracted from the unit’s current total health, and the new current health of the unit is set.
3. An If statement is used to identify if the unit is dead from the attack by checking if the units health has become less than 0.
4. The unit picturebox instance becomes invisible on the form, so the player no longer sees the unit on the landscape.
5. The units location is changed to the point(0,0) so the unit is ignored during other parts of the program and does not affect the collision detection methods of the game model.
6. The movementValid value is set to false to stop the unit from moving any further from its null point ((0,0)).
7. The unit alive value is set to false so the unit does not interact with any other unit instances in the form (any sort of interaction or movement only happens when the unitAlive attribute is true)
8. The value TRUE is returned from the function to be used in the Collision Execute procedure, representing that the unit death has occurred.
9. The value False is returned from the function as in this case, a unit death has not occurred.

#### Re Desgn of Unit Death

After some thought on the life of a unit during the game, I realised that it would be more efficient for unit death to be a procedure instead of the processes for unit death being replicated across the program. This is because the processes of unit death do not just occur when a unit is killed and will need to be used when the unit runs out of the bounds of the landscape. Therefore the take damage function will call the unit death procedure instead of completing the necessary actions itself.

public bool takeDamage(int damage)

{

unitHealth = unitHealth - damage;

if (unitHealth < 0)

{

//Death

unitDeath(Game1)

return true;

}

return false;

}

The unit death procedure takes in the game model instance as a parameter as the unit’s state needs to be updated in the game model’s data. The unit needs to be identified as dead so the game model can reuse the instance later in the gameplay.

public void unitDeath(GameModel Game1)

{

Unit1.Visible = false; (0)

unitUpValue = 0; (1)

unitLeftValue = 0; (1)

movementValid = false; (2)

unitAlive = false; (3)

Game1.setUnitState(false, getUnitArrayPosition()); (4)

gradient = 0; (5)

spawned = false; (6)

gradientAdd = 0; (7)

leftAdd = 0; (7)

}

1. The units picturebox instance is set to be invisible so it is no longer visible on the landscape. This process will be changed in the second iteration as the units will visibly explode on the landscape.
2. The unit’s location is set to (0,0) to avoid other interactions with units. (0,0) is also the default position for units before they spawn. This will prevent the collision range and execute functions from mistakenly ranging the dead unit against live units.
3. The movement valid attribute of the unit is set to false to prevent the unit from moving from its dead location (0,0).
4. The unit alive attribute is set to false to prevent the form from identifying the unit as being alive and carrying out unnecessary operations.
5. The game model is required to record the death of the unit in its unit state array, so the unit instance can be reset and reused.
6. The gradient variable is reset to 0 as after the unit instance is reused, the reset gradient value will be ready to be re-used in gradient calculations when the location procedure is called to move the reborn unit.
7. The spawned attribute is set to false as the unit is de-spawned, preventing the unit from being moved from the form, which uses this attribute to check if the new location procedure should be called.
8. The gradient and left add variables are reset to 0 which prevent a bug which prevents the new location procedure from working properly. This bug was caused by a carry over of the previous unit’s life position leftover.

## Game Model Class

Success Criteria: Game model class.

The game model class is an object which stores the status of the game to be referenced later by procedures elsewhere in the program. The class also stored the unit number which is assigned per unit in the spawning process. The class stores all the locations of the units in an array which is continually updated as units change position on the landscape and are killed or spawned. The unit’s location and unit number attribute are altered in the game model by using set and get methods.

public GameModel()

{

Point point1 = new Point(0, 0); //These are the points of which each unit’s

Point point2 = new Point(0, 0); //location will be stored as elements in

Point point3 = new Point(0, 0); //unit location.

Point point4 = new Point(0, 0);

Point point5 = new Point(0, 0);

Point point6 = new Point(0, 0);

Point point7 = new Point(0, 0);

Point point8 = new Point(0, 0);

unitLocations = new Point[] { point1, point2, point3, point4, point5, point6, point7,point8};

}

public Function Point[] getUnitLocations()

{ //Get Meathod

return unitLocations;

}

public Procedure setNewUnitLocation(int position, Point newLocation)

{ //Set Meathod

unitLocations[position] = newLocation;

}

### Collision range function

The collision range function calculates the distance between all of the units location contained in the unit Location array. The function takes in an integer representing the unit number of the unit which distances are being compared against. The function returns the distance values in the form of an array which each element position contains the corresponding unit’s distance in pixels from the compared unit location.

public int[] Function collisionRange(int unitNum) (0)

{

int difference = 0; (1)

int[] collisionRanges; (1)

collisionRanges = new int[] { 0, 0, 0, 0, 0, 0, 0, 0 }; (1)

Point empty = new Point(0, 0); (1)

for (int index = 0; index < unitLocations.Length; index++) (2)

{

if (unitLocations[index] != empty) (3)

{

difference = unitLocations[unitNum].X – unitLocations[index].X; (4)

collisionRanges[index] = difference; (5)

}

}

return collisionRanges ; (6)

}

1. The function declaration which accepts the integer parameter unitNum to be used to identify the value to be compared.
2. Variable declaration needed only for the instance of function. Difference is used to store the value of location difference calculated. collisionRanges is the array which will have its values changed to the new differences calculated and will be returned at the end of the function. Point empty is used to compare to the current locations of the units to identify units which haven’t spawned or are dead.
3. The for loop is initiated so each value in unitLocations array is compared as each element is compared in order of index in the loop.
4. If statement identifies if the element in unitlocations is unmoved, meaning the unit is dead or hasn’t spawned.
5. Comparison between the index unitLocation X value and unitNum unitlocation X value is stored in difference.
6. Difference is appended to the collisionRange array each loop so the array is built up with the differences per unit
7. CollisionRange is returned and the function is finished.

### Collision Execute Procedure.

Success criteria: Working Collisions Detection of Unit Ranges, Units interact when within range.

In the original design of the program when I was experimenting with how to use the information from collisionRange to allow units to attack each other, I made a lengthy solution to the problem by programming the solution inside of the form’s timer. This is because currently the instances of units could only be accessed inside of the form. This led to a lengthy repetitive solution of which I knew worked but was not efficient.

if(U0.getunitAlive() == true)

{

arrayOfDistances = Game1.collisionDetect(0);

//Enemy tank attacking freindly 1

if (U0.getSpeed() < 0)

{

if (arrayOfDistances[1] < U0.getUnitRange() && U1.getUnitLocation() != nul && U1.getSpeed() > 0)

{

U0.setValid(false);

if (U1.takeDamage(U0.getUnitAttack()) == true)

{

U0.setValid(true);

}

}

}

//Same prospect if Unit speed > 0 Except values are swapped for the change in direction.

The areas of the code highlighted in red show the parts of the code which are not changeable, and if I wanted all of the units to interact as intended, the code would have to be copied and reproduced several times with just the red values changed. Instead of writing out the code several times, I turned to my stakeholder Ollie Wilkinson for advice on this matter, as he is experienced in object orientated programming himself. He did not have a solution to this problem, but he did remind me it is possible to pass instances of objects as a parameter into a procedure.

This allowed me to design a much more efficient solution for executing unit interactions. By passing in the instances of Unit into a procedure within the game model, the red values in the example on the previous page could be set to the parameters of the procedure, and the procedure can be called per timer tick for each unit set as its parameter. The procedure below is my new design for the procedure, which is in a more appropriate place in the program within the game model class. This allows for the collision range function to be encapsulated privately as it is only used internally in the class.

public void collisionExecute(Unit attacker, Unit victim, Unit T0, Unit T1, Unit T2, Unit T3, Unit T4, Unit T5, Unit T6, Unit T7) (0)

{

arrayOfDistances = collisionDetect(attacker.getUnitArrayPosition()); (1)

Point nul = new Point(0, 0); (2)

if (attacker.getunitAlive() == true) (3)

{

//Attacker is Enemy

if (attacker.getSpeed() < 0) (4)

{

if (arrayOfDistances[victim.getUnitArrayPosition()] < attacker.getUnitRange() && victim.getUnitLocation() != nul && victim.getSpeed() > 0) (5)

{

attacker.setValid(false); (6)

if (victim.takeDamage(attacker.getUnitAttack()) == true) (7)

{

attacker.setValid(true); (8)

if (T0.getSpeed() < 0) (9)

{

T0.setValid(true);

}

}

}

}

//Attacker is freindly

if (attacker.getSpeed() > 0)

{

if (arrayOfDistances[victim.getUnitArrayPosition()] > attacker.getUnitRange() && victim.getUnitLocation() != nul && victim.getSpeed() < 0)

{

attacker.setValid(false);

if (victim.takeDamage(attacker.getUnitAttack()) == true)

{

attacker.setValid(true);

if (T0.getSpeed() > 0)

{

T0.setValid(true);

}

}

}

}

}

1. The procedure takes in the instances of units from the form. The first parameter passed in is the unit which is attempting the attack. The second parameter is the unit which is being attacked. The attack only occurs later in the procedure if the conditions for a successful attack are met. The rest of the units are passed in due to a problem when two or more units are attacking each other, all units move value need to be set to true if a unit on their team kills another unit otherwise a second attacking unit will never get the command to move again.
2. The array of distances is created using the local method collisionRange of which the attacker element and victim element will be compared to check if an attack is in range.
3. Point nul is created to compare to unit locations which are unspawned or dead from being mistakenly attacked.
4. This if statement checks if the attacker is alive, if an attacker is dead then an attack is not possible. This prevents dead units from interfering with alive unit.
5. The unit is identified as a friendly or enemy unit so the correct evaluation of attack can be completed. The methods of attack validation differ from friendly units to enemy units.
6. The final conditions for attack are checked in this if statement. For the attack to occur; The distance between the units has to be less than the attackers range, The victims location does not equal the Nul Point (If the unit is not off the landscape) and the victim needs to be on the opposite team of the attacker (represented by its sign of speed). If these conditions are met, the attack code is ran.
7. As the attack conditions have been met, the attacker must be commanded to become stationary during the attack.
8. The victim takes damage during the takeDamage function, and the function returns a TRUE Boolean value if the victim unit has died from the attack, therefore an if statement catches if the victim dies so…
9. …the attacker’s movement attribute is set to TRUE in this case so it can continue to move across the landscape.
10. If the victim unit dies the other unit instances are checked what team they are on, and if they are on the same side as the attacker, their movement value is set to true to prevent units freezing after a multiple unit attack as only the one unit which finishes the kill will receive the TRUE value from the takeDamage function from the victim.

### Unit instance reuse

Success Criteria: Unit Instance Re-use

The game model is responsible for tracking the statuses of units during the running of the gameplay. As well as tracking the locations of the units, the game model as well records which units are currently active in the program. The statuses of the units are recorded in an array of Booleans with each element representing a unit. If the element’s value is TRUE, then the unit is active in the gameplay. If the element’s value is FALSE, then the corresponding unit is not active in the gameplay, because it is either un-spawned or dead.

bool[] unitStates = new bool[] { false, false,…………}

Unit reuse prevents the requirement of an ‘infinite’ amount of unit instances (A large, still comprehendible amount of unit instances which cannot be predicted, therefore impossible to efficiently program), even with a limit to the number of units possible to spawn at one given time. What I mean by this is that for every new unit I want to create, a unit instance will need to be created, which for large amount of units would require a different approach to the one I have created, as each new unit will have a new instance of unit, which would then need to be taken into account in all procedures and functions in the program, which cannot cater for an unpredictable amount of unit instances. By reusing a premade set of unit instances, there can be an unlimited number of units spawned throughout the duration of the gameplay, if the number of units spawned at one time is limited to the amount of pre-made instances. I have created methods in the game model to append and read the statuses of units and identify unused unit instances to allow the seamless generation of ‘new’ units for the gameplay. I have created 10 instances of unit in the form to use. (U0,U1……..U9)

#### Next available unit function

The game model needs to search through the array of unit states to discover which unit instances are inactive when the game wants to spawn a new unit. I have used a linear search here as the array has a short length of only 10 elements. The simplicity of the code is worth more than the insignificant amount of time saved using a faster more complex method.

public int nextAvaliableUnit()

{

bool found = false; (0)

int value = -1; (1)

for (int index = 0; index < unitStates.Length; index ++) (2)

{

if(found == false) (3)

{

if (unitStates[index] == false) (4)

{

found = true; (5)

value = index; (5)

}

return value; (6)

}

}

}

1. The Boolean found is declared for use in the function, with its initial value set to false which is only set to true when an inactive unit instance is found, this allows the program to select the lowest element index of a free unit to return.
2. The value variable is declared and set to -1, as -1 is not an element in the array of unit states. The value variable is changed to the element index of the first identified inactive unit and is returned at the end of the function. Therefore, a returned value of -1 represents the absence of any free unit instances.
3. A For loop is initiated and an index variable is created for the iteration. The loop will iterate the amount of times of the length of the unit states array. This will allow each element of the array to be accessed and compared by using Index and will not run out of bounds of the array.
4. The function checks if the found variable is false, as the first inactive unit instance to be found should be returned. If this unit instance has been found, this selection will prevent any other inactive unit instances from being returned.
5. The element selected using index is checked to see if the corresponding unit instance is inactive, represented by the element being FALSE. If it is then…
6. …the found variable is set to true as an inactive instance has been found, and the variable ‘value’ is set to the ‘index’ value.
7. The value variable is returned which contains the correct value either representing the unit instance location in the array or -1 when no unit instance is found.

#### unitStates Get and Set methods

The unitStates array needs to be accessed by other classes in the program. The array can be accessed through the get and set methods in the game model. The set method takes in two parameters; state (Boolean) and index (integer). Index holds the value of the location in the array which needs to be changed, and state holds the value of which the element is being changed to.

public void setUnitState(bool state, int index)

{

unitStates[index] = state;

}

The get method still takes in a parameter as the element which needs to be returned needs to be identified. The index parameter holds the location in the array which needs to be returned.

public bool getUnitState(int index)

{

if(index == -1)

{

//error, no unit instances are inactive.

return true;

}

else

{

return unitStates[index];

}

}

### Check unit locations procedure

Success Criteria: Units de-spawn when out of range of the landscape.

The check unit locations procedure is a method in the game model called to de-spawn units when they reach the end of the landscape. The objective of a unit is to reach the other side of the landscape, which results in a reward for the unit’s team, and a deduction in the opponents base health.

This procedure will be further developed in the future development of the game, but the basics of this procedure have been created which focuses on the de-spawning of the unit. If the criteria for a unit to be considered out of range has been met, then the unit death procedure is called as well as other methods of the game model which have not been created to do with the players progress (Ex. unitKillReward).

public void checkUnitLocations(Unit U0, Unit U1, Unit U2, Unit U3, Unit U4, Unit U5, Unit U6, Unit U7, Unit U8, Unit U9) (0)

{

for (int index = 0; index < unitLocations.Length; index ++) (1)

{

if (unitLocations[index].X <= -25 | unitLocations[index].X >= 1460 ) (2)

{

switch(index) (3)

{

case 0: (4)

U0.unitDeath(this); (4)

break; (4)

case 1: (5)

U1.unitDeath(this); (5)

break; (5)

}

}

}

}

}

1. The procedure takes in the unit instances from the form so their methods can be called if necessary.
2. A For loop is initiated and an index variable is created for the iteration. The loop will iterate the amount of times of the length of the unit states array. This will allow each element of the array to be accessed and compared by using Index and will not run out of bounds of the array.
3. The If statement compares each elements X coordinate in the unit Locations array to -25 (Unit has gone past the limit on the left of the landscape) or 1460 (Unit has gone past the limit on the right of the landscape). If the units X coordinate value meets these criteria, then…
4. …for each value of index, a switch skips to the correct index which represents the unit instance, which is detected to be out of range, so the following procedures are triggered on the correct unit instance.
5. An index value of 1 causes the U1 unit instance to have its death method ran.
6. An index value of 2 causes the U2 unit instance to have its death method ran.

# Iteration 1 Testing

So far, the program is visually working as designed, but whether it will be able to cater for new unit classes with different unit attributes cannot be inferred by observing the outputs. All the major components of the program have now been created which should now be tested to ensure the program will work as intended as iteration 2 is developed.

#### Buttons Run Desired code when clicked.

A screenshot of a social media post

Description automatically generated

Expected result: Breakpoint to be triggered when button is clicked, this indicates the correct code is being ran.

Observed interaction: Correct breakpoints were triggered when each button was clicked.

The naming and references are all working correctly which result in the triggering of correct procedures in the program.

#### Curve Class Set and Get method validation.

Expected Result: Set and Get methods change and retrieve correct class attribute values. Unit test are successful.

private Point[] testPoints;

[TestMethod()]

public void setPointsTest()

{

ClassCurve curve = new ClassCurve();

Point point1 = new Point(0, 0);

Point point2 = new Point(0, 0);

Point point3 = new Point(0, 0);

Point point4 = new Point(0, 0);

Point point5 = new Point(0, 0);

Point point6 = new Point(0, 0);

Point point7 = new Point(0, 0);

testPoints = new Point[] { point1, point2, point3, point4, point5, point6, point7 };

curve.setPoints(testPoints);

Assert.AreEqual(curve.getPoints(),testPoints);

Observed outcome: Unit test passed. This shows that the curve’s point attributes have been changed to ‘empty’ (0,0) values by the set method, and that the get method correctly retrieves the new points back from the curve instance.

#### Generate Correct Landscape painted onto landscape window (Draw Procedure)

Expected result: Visible landscape line generated in the correct position of the screen which follows dictated points form the curve class, within the landscape panel.

Look at the landscape window to see manipulation of line where points are changed in the curve class during different runs of the program.

All point attributes are set to (0,0) in the curve class for the first run of the program. No landscape line should appear with these location points.

Inputted Points:

Point point1 = new Point(0, 0);

Point point2 = new Point(0, 0);

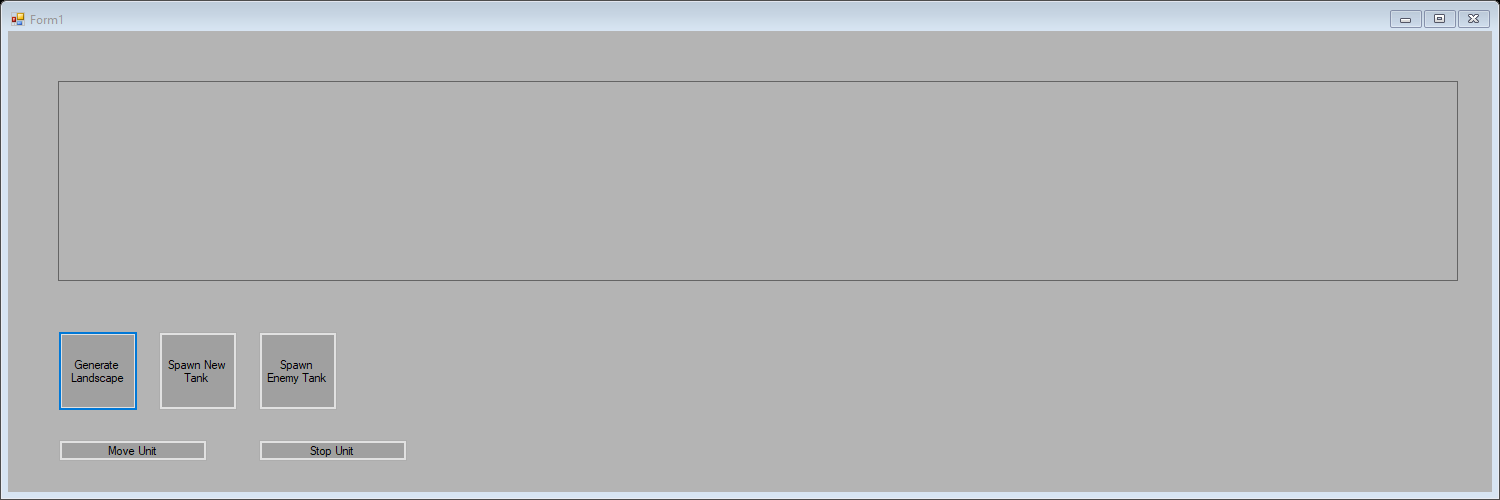
Point point3 = new Point(0, 0);

Point point4 = new Point(0, 0);

Point point5 = new Point(0, 0);

Point point6 = new Point(0, 0);

Point point7 = new Point(0, 0);

Result:

Expected outcome has occurred in this instance of testing.

All point attributes are then set to run along the bottom of the landscape window, with a constant Y coordinate of 250. A straight line on the bottom of the landscape window is expected which does not leave the bounds of the landscape panel.

Inputted Points:

Point point1 = new Point(50, 250);

Point point2 = new Point(250, 250);

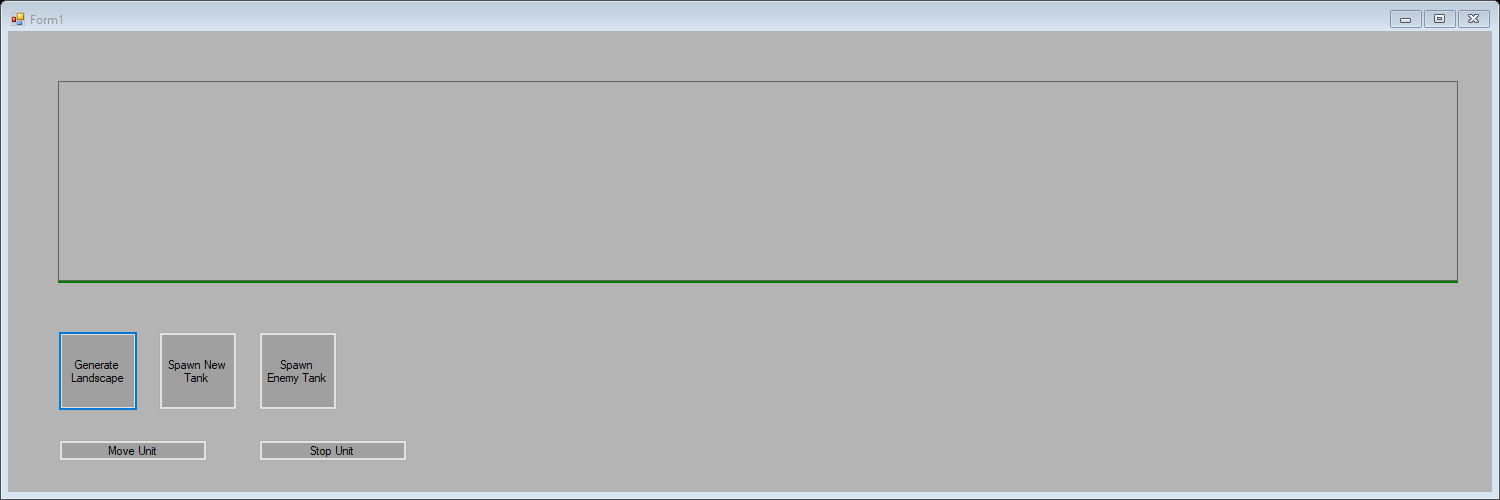
Point point3 = new Point(450, 250);

Point point4 = new Point(650, 250);

Point point5 = new Point(850, 250);

Point point6 = new Point(1250, 250);

Point point7 = new Point(1450, 250);

Result:

Expected outcome has occurred in this instance of testing.

All point attributes are set to run across the landscape panel but this time with a changing Y coordinate. A gradient is introduced which will result in the line passing from the bottom left of the landscape panel to the top then continuing to the top right-hand corner of the panel. A straight landscape line with a single vertex where the gradient becomes 0 should be observed.

Inputted points:

Point point1 = new Point(50, 250);

Point point2 = new Point(250, 210);

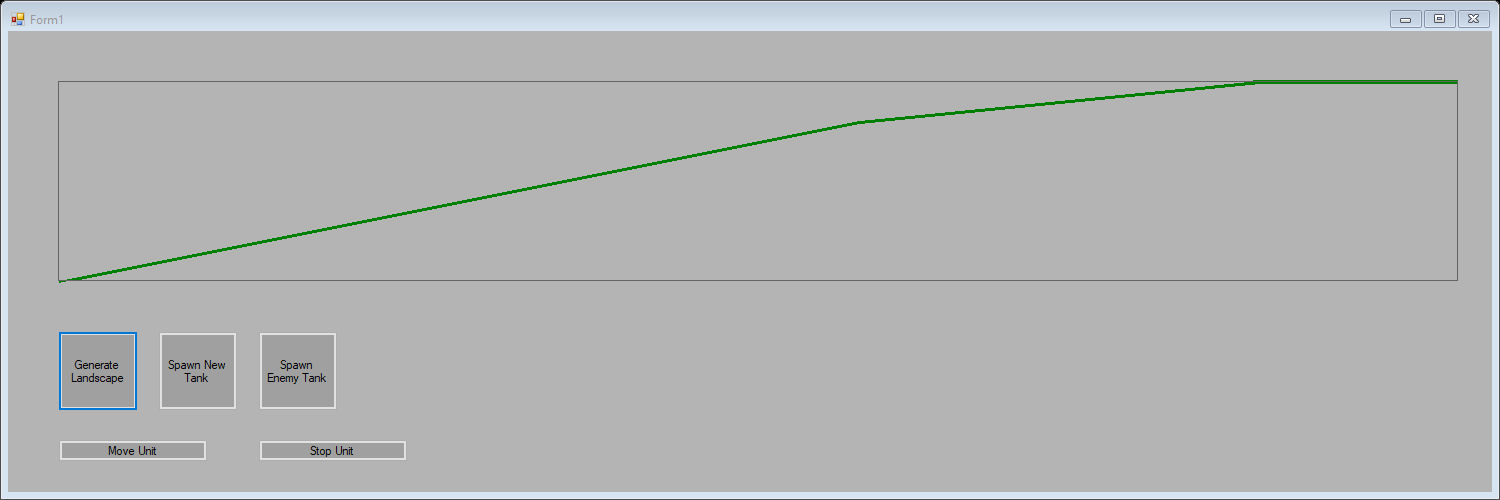
Point point3 = new Point(450, 170);

Point point4 = new Point(650, 130);

Point point5 = new Point(850, 90);

Point point6 = new Point(1250, 50);

Point point7 = new Point(1450, 50);

Result: (Failure)

Two vertexes are present indicating an issue in the logic of the landscape generation. As the line has seven-point attributes, we can identify the out of place point by figuring out the location of the landscape, which is causing the issue, and associate it with the faulty point. In this case, the faulty area is shown by the circle and corresponds to point 6.

Point 6 is found to have its X coordinate jump by 400 instead of 200 from point 5, as I mistakenly thought that I had created eight-point attributes instead of seven. This is a problem as originally the design of the landscape was to change through evenly spread point attributes, but with only seven points, these points cannot be evenly spread across the landscape as the length of the landscape (1400) is not divisible by the current number of point attributes (7)

##### Landscape Generation Debugging

This error of the point attributes could be fixed by creating an eighth point which would allow the points to be spread evenly across the landscape, but this would be a complicated task due to the fact that the number of points has already been hard coded into other methods in the game and unit class, as the landscape will never have a changing number of point attributes. Also, the fact that this error can simply be accounted for by spreading the point attributes in a different way where the spread of points isn’t necessarily even but malleable to still allow easy landscape manipulation.

The current design to spread points by an X value of 200 will be kept because it is a very divisible number which will help the program by simplifying gradient calculations, however a jump of 400 will need to occur in order for the landscape to be 1400 pixels in length. I have changed the jump in X coordinate to occur between point 6 and 7. This will make the landscape easier to manipulate as the x change isn’t between two previously unknown points in the middle of the landscape.

All point attributes are set to run across the landscape panel but this time with a changing Y coordinate. A gradient is introduced which will result in the line passing from the bottom left of the landscape panel to the top then continuing to the top right-hand corner of the panel. A straight landscape line with a single vertex where the gradient becomes 0 should be observed. This time the X values of point 6 and 7 have been changed to fit the new design of the landscape.

Inputted points:

Point point1 = new Point(50, 250);

Point point2 = new Point(250, 210);

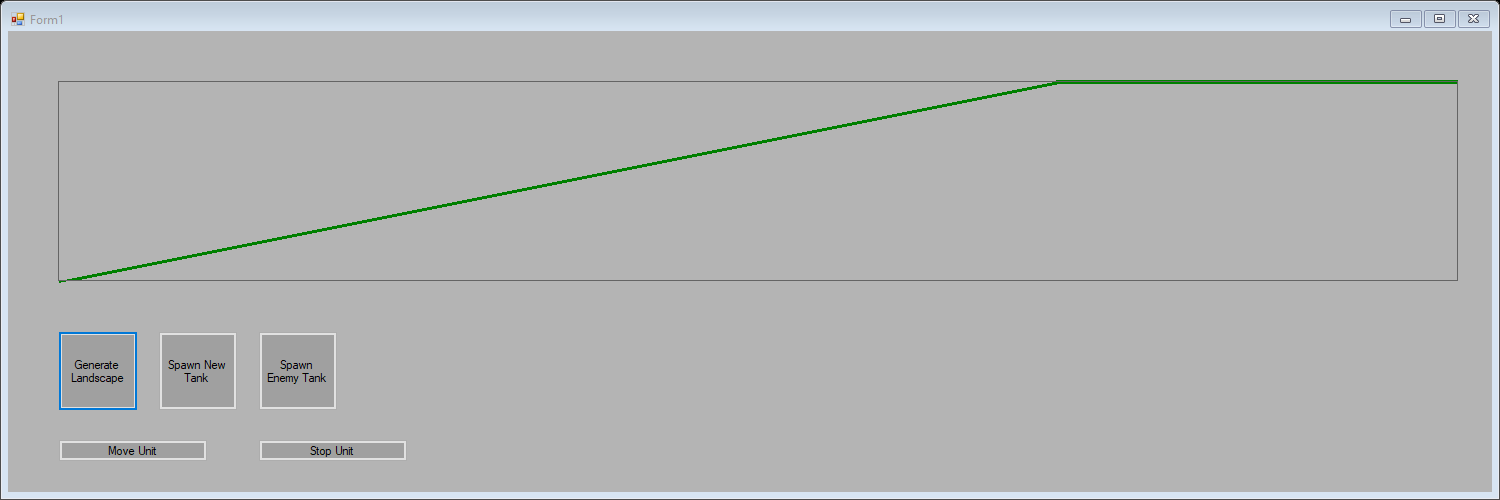
Point point3 = new Point(450, 170);

Point point4 = new Point(650, 130);

Point point5 = new Point(850, 90);

Point point6 = new Point(1050, 50);

Point point7 = new Point(1450, 50);

Result:

Expected outcome has occurred in this instance of testing.

#### Unit class Get and Set Method Validation (Including use of unit definition in some cases)

The following methods are going to be tested in the following unit tests:

* Unit movement valid get and set methods.
* Unit get height method.
* Unit get width method.
* Unit get speed method.
* Unit get unit left value method.
* Unit get unit up value method.
* Unit valid move.
* Unit get unit spawn State.
* Unit set unit up.
* Unit set and get unit array position.
* Unit get unit location.
* Unit get unit range.
* Unit get and set unit health.
* Unit get unit attack.
* Unit get and set unit alive.

These set and get methods are suitable for unit testing due to their predictable outputs. A pair of set and get method can be called to change a default attribute in the class. A test variable can be assigned from a newly set attribute using the get method in the class. A unit test on a pair of set and get methods will fail if the test variable doesn’t match the corresponding attributes changed value. This would indicate a failure of either the set or get method, which then can be further investigated, or a failure or change in value of the unit definition procedure. If attributes per unit specific class were to change in an unwanted way, or values passed through the unit definition procedure, this will cause some of the unit tests to fail, allowing an insight into what errors might be causing a problem.

I created a test class called UnitTestClass which attributes all hold known values. This class is identical to other unit specific classes in its methods and attribute types and is only used in the program through unit testing. This class allows us to test methods of unit which access values which are only set through the unit definition method, where attributes are read in from a unit specific class.

class UnitTestClass

{

//Attributes spercific to the type of unit.

double unitSpeed = 1;

int unitHealth = 0;

int unitHight = 2; //All values are unique to help

int unitWidth = 3; //catch methods reading wrong values.

int unitRange = 4;

int unitAttack = 5;

//Gets

public double getUnitSpeed()

{

return unitSpeed;

}

public int getUnitHealth() //Not used in testing.

{

return unitHealth;

}

public int getUnitHight()

{

return unitHight;

}

public int getUnitWidth()

{

return unitWidth;

}

public int getUnitRange()

{

return unitRange;

}

public int getUnitAttack()

{

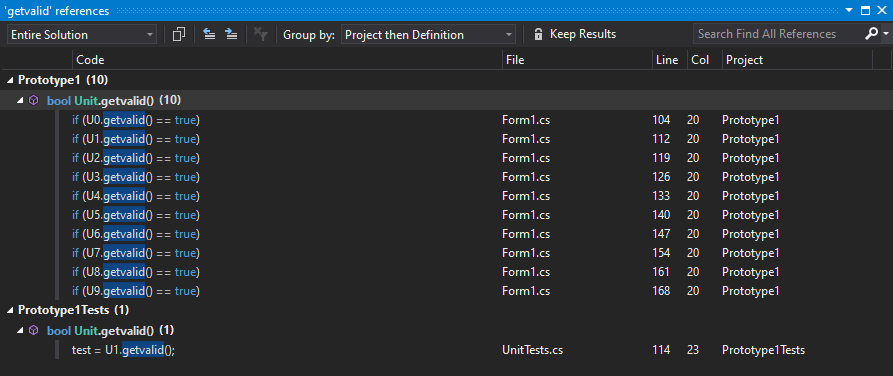
return unitAttack;

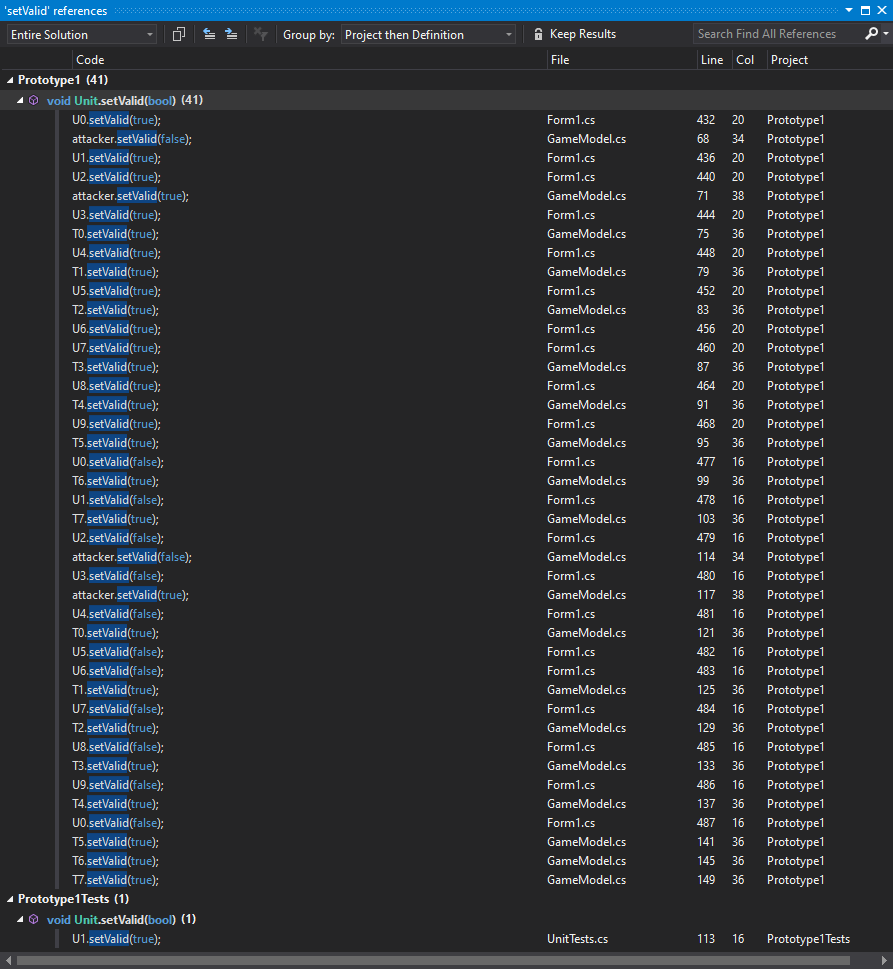
}

}

All methods are also reference checked to insure they are not being used in unneeded ways which would make the program more inefficient. This also prevents unused methods from remaining in the unit class.

* Unit movement valid get and set methods.

References of methods in the program to identify valid use of methods:



getValid references are shown above, setValid references are shown to the left.

After reading through all of the references, It is shown that the getValid method and the setValid method in the unit class are used throughout the program and play a part in key algorithms in the program which change the unit’s status. They are not used anywhere where they are unneeded.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods. Attribute default changed by assigning new value using set method, and a test variable is assigned a new value from changed attribute.

Test:

public void getAndSetvalidTest()

{

//Defult movement valid attribute is set to false

bool test = false;

Unit U1 = new Unit();

U1.setValid(true);

test = U1.getvalid();

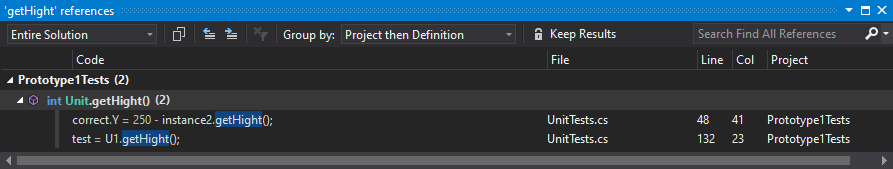
Assert.AreEqual(test, true);

}

Result: Test pass (Evidence at end of section)

* Unit get height method.

References of method in the program to identify valid use of methods:



Method is only used within the UnitTests file, indicating that the method not being called anywhere else in the program. This is unexpected but shows the importance of checking tested methods if they are being used. Although the method is currently unused, I have decided to still create a unit test for this method and keep it in the unit class. This is because this method may become useful in the next iteration and can be used with confidence if it has been tested already. If the method is never used in the final iteration of the program, It will be removed from the class for security reasons.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods (Including unit definition method). Attribute default changed by assigning new value using unit definition of the unit specific test class, and a test variable is assigned a new value from changed attribute.

Test:

public void getHeightTest() //Includes unit definition

{

//Defult Height set to 0

int test = 0;

Unit U1 = new Unit();

//No set method, height is only changed through

//unit definition, using test class

U1.unitDefinition(0);

//Height is now set to 2

test = U1.getHight();

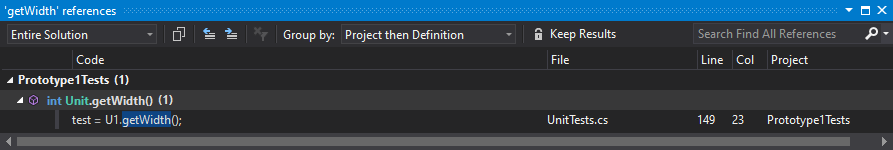
Assert.AreEqual(test, 2);

}

Result: Test pass with noted caution (Evidence at end of section)

* Unit get width method.

References of method in the program to identify valid use of methods:



The same issue is present as the getHeight method. I also have decided to test and keep this method in the class with the same caution as the method before.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods (Including unit definition method). Attribute default changed by assigning new value using unit definition of the unit specific test class, and a test variable is assigned a new value from changed attribute.

Test:

public void getWidthTest() //Includes unit definition

{

//Defult Width set to 0

int test = 0;

Unit U1 = new Unit();

//No set method, Width is only changed through

//unit definition, using test class

U1.unitDefinition(0);

//Width is now set to 3

test = U1.getWidth();

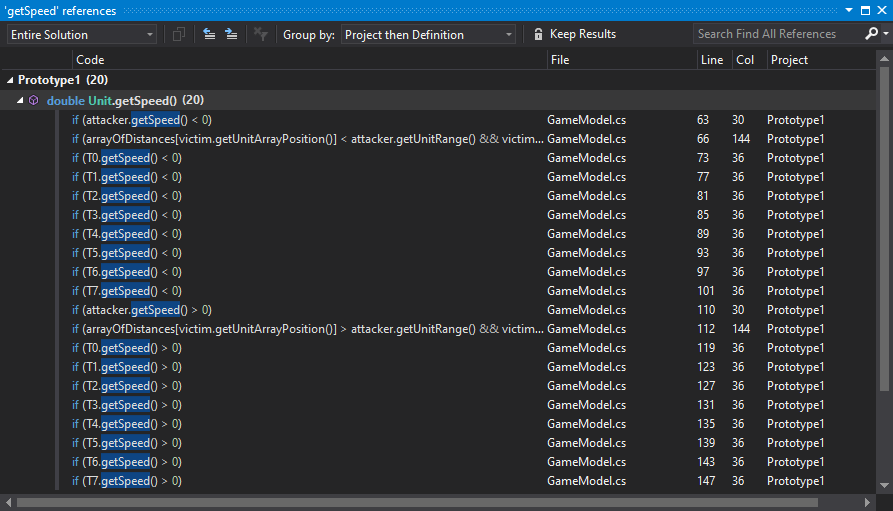
Assert.AreEqual(test, 3);

}

Result: Test pass with noted caution (Evidence at end of section)

* Unit get speed method

References of method in the program to identify valid use of methods:



It is shown that the getSpeed method in the unit class are used throughout the program and play a part in key algorithms in the GameModel collisions methods. It is not used anywhere where it is unneeded.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods (Including unit definition method). Attribute default changed by assigning new value using unit definition of the unit specific test class, and a test variable is assigned a new value from changed attribute.

Test:

public void getSpeedTest() //Includes unit definition

{

//Defult Speed set to 0

double test = 0;

Unit U1 = new Unit();

//No set method, Speed is only changed through

//unit definition

U1.unitDefinition(0);

//Speed is now set to 1 as test class

test = U1.getSpeed();

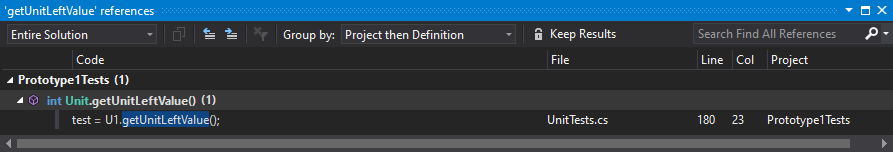
Assert.AreEqual(test, 1);

}

Result: Test pass (Evidence at end of section)

* Unit get unit left value method.

References of method in the program to identify valid use of methods:



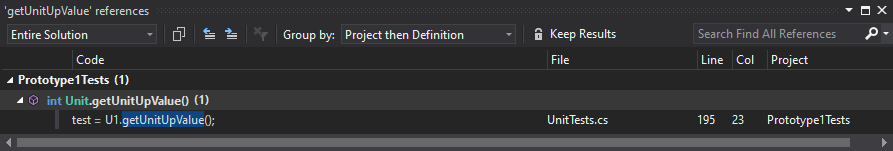
This method is only referenced in the UnitTesting file which proves it is not used anywhere else in the program. After investigation, I have found the reason for this is that the unitLeftValue attribute is only used from inside the unit class and is only accessed indirectly outside of the class using the unit location method.

The method will be removed from the class for security reasons.

Result: Test Fail due to no valid reference.

* Unit get unit up value method.

References of method in the program to identify valid use of methods:



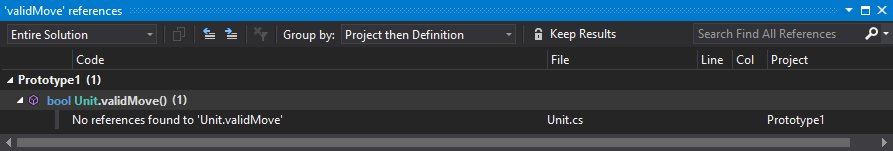
This method is only referenced in the UnitTesting file which shows it is not used anywhere else in the program. After investigation, I have found the reason for this is that the unitUpValue attribute is only used from inside the unit class and is only accessed indirectly outside of the class using the unit location method.

The method will be removed from the class for security reasons.

Result: Test Fail due to no valid reference.

* Unit valid move.

References of method in the program to identify valid use of methods:



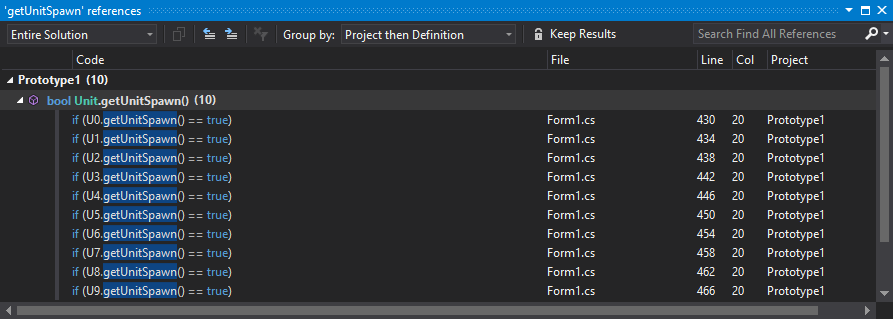
This method is only referenced in the UnitTesting file which proves it is not used anywhere else in the program. After investigation, I have found the reason for this is that the getValid method already completes the purpose the validMove method was created for. Therefore, validMove will remain unused and will be removed.

The method will be removed from the class for security reasons.

Result: Test Fail due to no valid reference.

* Unit get unit spawn State.

References of method in the program to identify valid use of methods:



It is shown that the getUnitSpawn method in the unit class are used throughout the program and play a part in key algorithms in the Form1 class movement method. It is not used anywhere where it is unneeded.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods (Including methods creating only for testing use). Attribute default changed by assigning new value using the testing set method, and a test variable is assigned a new value from changed attribute.

Test:

public void getUnitSpawnTest()

{

//Using set method only for testing

//Defult spawn state is false

Unit U1 = new Unit();

U1.setUnitSpawnValue(true);

bool test = false;

test = U1.getUnitSpawn();

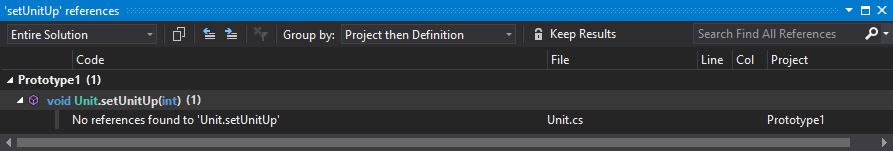
Assert.AreEqual(test, true);

}

Result: Test pass (Evidence at end of section)

* Unit set unit up.

References of method in the program to identify valid use of methods:



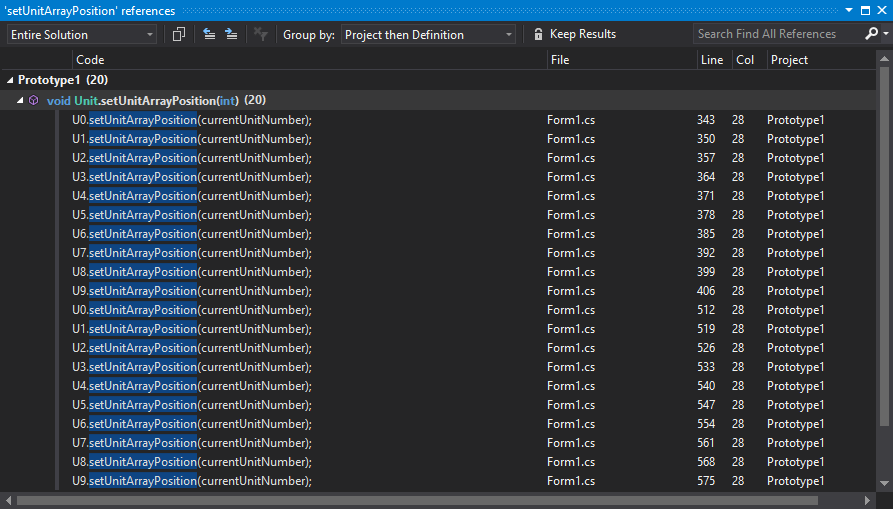
This method is only referenced in the UnitTesting file which proves it is not used anywhere else in the program. After investigation, I have found the reason for this is that the setUnitUp method already completes the purpose of the setUnitUpValue method was created for. Therefore, setUnitUp will remain unused and will be removed.

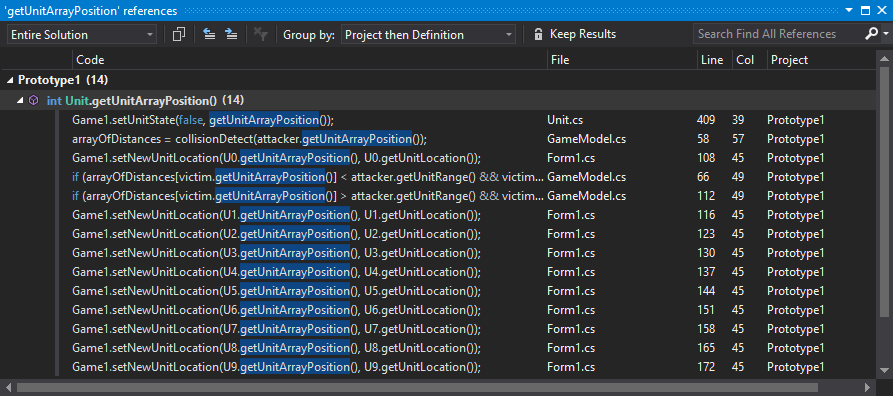
The method will be removed from the class for security reasons.

Result: Test Fail due to no valid reference.

* Unit set and get unit array position.

References of methods in the program to identify valid use of methods:





It is shown that the getUnitArrayPosition method and the setUnitArrayPosition method in the unit class are used throughout the program and play a part in key algorithms in the Form1 which change the unit’s array location. They are not used anywhere where they are unneeded.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods. Attribute default changed by assigning new value using set method, and a test variable is assigned a new value from changed attribute.

Test:

public void getAndSetUnitArrayPositionTest()

{

//Default array position is set to 0

Unit U1 = new Unit();

U1.setUnitArrayPosition(2);

int test = 0;

test = U1.getUnitArrayPosition();

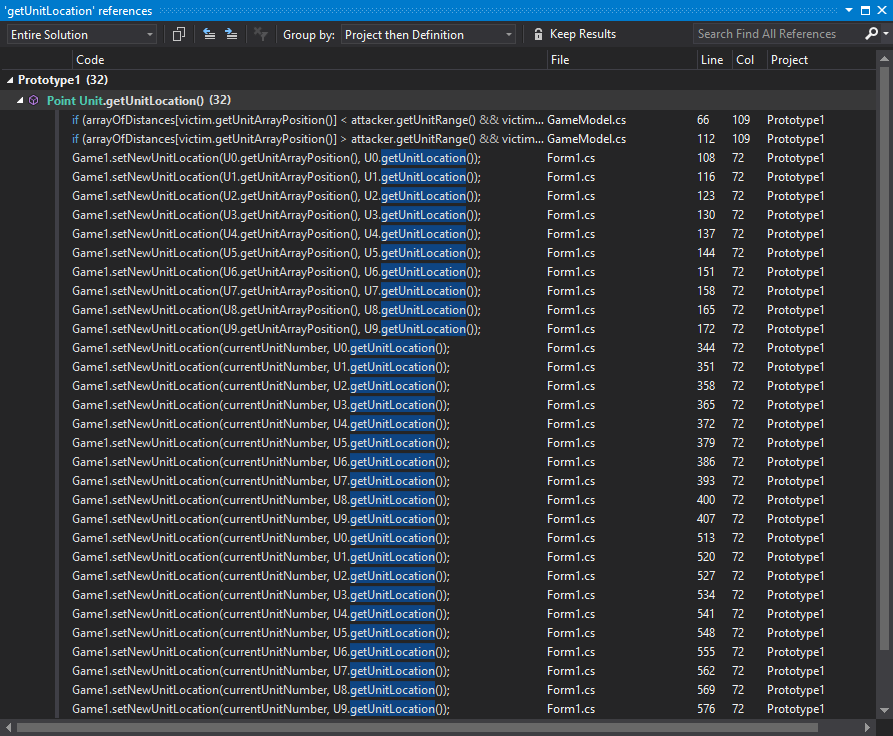
Assert.AreEqual(test, 2);

}

Result: Test pass (Evidence at end of section)

* Unit get unit location.

References of method in the program to identify valid use of methods:



It is shown that the getUnitLocation method in the unit class are used throughout the program and play a part in key algorithms in the Form1 class movement method. It is not used anywhere where it is unneeded.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods (Including methods creating only for testing use). Attribute default changed by assigning new value using the testing set method, and a test variable is assigned a new value from changed attribute.

Test:

public void getUnitLocationTest()

{

//Defult unit location is set to (0,0)

//Using testing methods setUnitUpValue() and setUnitLeftValue()

Unit U1 = new Unit();

Point test = new Point(0, 0);

Point valid = new Point(10, 10);

U1.setUnitUpValue(10);

U1.setUnitLeftValue(10);

test = U1.getUnitLocation();

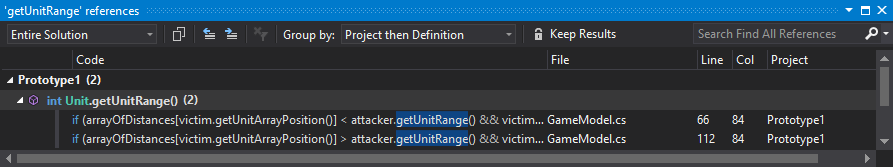
Assert.AreEqual(test, valid);

}

Result: Test pass (Evidence at end of section)

* Unit get unit range.

References of method in the program to identify valid use of methods:



It is shown that the getUnitRange method in the unit class are used in key algorithms in the GameModel class collisions method. It is not used anywhere where it is unneeded.

Expected result: Unit test to pass which prove the correct attribute is being set and get by the unit class methods (Including unit definition method). Attribute default changed by assigning new value using unit definition of the unit specific test class, and a test variable is assigned a new value from changed attribute.

Test:

public void getUnitRangeTest()

{

//Using test class and unit definition

//Defult range value of unit is 0

Unit U1 = new Unit();

int test = 0;

//Range value set to 4

U1.unitDefinition(0);

test = U1.getUnitRange();

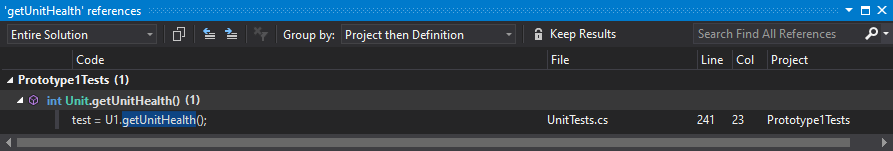
Assert.AreEqual(test, 4);

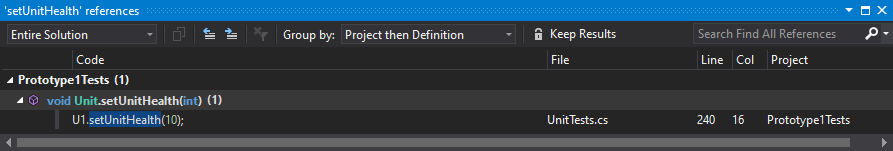
}

Result: Test pass (Evidence at end of section)

* Unit get and set unit health.

References of methods in the program to identify valid use of methods:





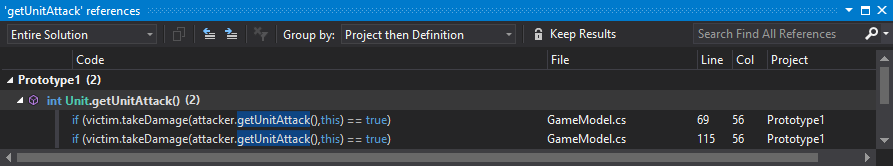
These methods are only referenced in the UnitTesting file which proves they are not used anywhere else in the program. After investigation, I have found the reason for this is that the health attribute of the unit class is only accessed within the class itself dismissing the need for get and set methods for the health attribute.

The method will be removed from the class for security reasons.

Result: Test Fail due to no valid reference.

* Unit get unit attack.

References of method in the program to identify valid use of methods:



The getUnitAlive method is used twice in the GameModel class. It is a very important as the GameModel needs to be able to access the unit attack value to control any unit interactions correctly.

Test:

public void getUnitAttackTest()

{

//Using test class and unit definition

//Defult attack value of unit is 0

Unit U1 = new Unit();

int test = 0;

//attack value set to 5

U1.unitDefinition(0);

test = U1.getUnitAttack();

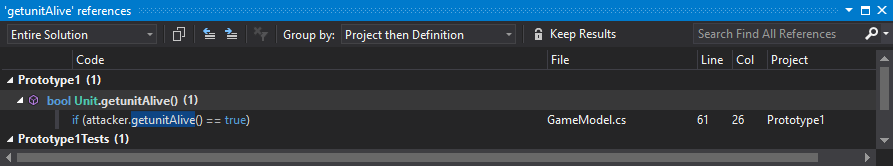
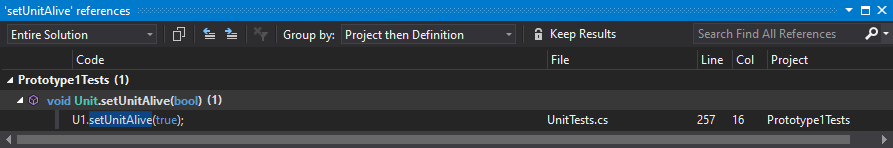
Assert.AreEqual(test, 5);

}

Result: Test pass (Evidence at end of section)

* Unit get and set unit alive.

References of methods in the program to identify valid use of methods:



getUnitAlive is used in the GameModel class and works in the method controlling unit interactions. setUnitAlive however is not referenced anywhere else in the program apart from the unitTest class and therefore is not relevant in the program. It is however needed for the testing of the getUnitAlive method and will be separated from the rest of the methods in a section dedicated for testing only.

Test:

public void getunitAliveTest()

{

//Defult value of unit alive is false

Unit U1 = new Unit();

bool test = false;

////

U1.setUnitAlive(true);

test = U1.getunitAlive();

Assert.AreEqual(test, true);

}

Result: Test pass (Evidence at end of section)

##### Result Evaluation

|  |  |
| --- | --- |
| Unit movement valid get and set methods. | Test pass |
| Unit get height method. | Test pass (with noted caution) |
| Unit get width method. | Test pass (with noted caution) |
| Unit get speed method | Test pass |
| Unit get unit left value method. | Test fail (No reference) |
| Unit get unit up value method. | Test fail (No reference) |
| Unit valid move. | Test fail (No reference) |
| Unit get unit spawn State. | Test pass |
| Unit set unit up. | Test Fail |
| Unit set and get unit array position | Test pass |
| Unit get unit location. | Test pass |
| Unit get unit range. | Test pass |
| Unit get and set unit health. | Test fail (No reference) |
| Unit get unit attack. | Test pass |
| Unit get and set unit alive. | Test pass (with noted caution) |

Methods which passed the testing are shown to be reliable and consistent enough to be used in further iterations of the game. Methods which have passed the tests with caution taken are methods which need to be re-referenced in the second iteration of testing purely to ensure that they are not pointlessly implemented into the program remaining unused. Method which have failed the testing are not referenced anywhere else in the whole program than where they are declared and will not become relevant even in further iterations due to their functions. (Further details are shown in the individual testing in the section above)

switch(Game1.getUnitNumber())

{

case 1:

timer1.Stop();

timer1.Interval = 20;

timer1.Start();

break;

case 2:

timer1.Stop();

timer1.Interval = 20;

timer1.Start();

break;

case 3:

timer1.Stop();

timer1.Interval = 18;

timer1.Start();

break;

case 4:

timer1.Stop();

timer1.Interval = 15;

timer1.Start();

break;

case 5:

timer1.Stop();

timer1.Interval = 10;

timer1.Start();

break;

case 6:

timer1.Stop();

timer1.Interval = 6;

timer1.Start();

break;

case 7:

timer1.Stop();

timer1.Interval = 2;

timer1.Start();

break;

case 8:

timer1.Stop();

timer1.Interval = 1;

timer1.Start();

break;

case 9:

timer1.Stop();

timer1.Interval = 2;

timer1.Start();

break;

}