Software Engineering Stage 6

(Year 12) – Major Software Engineering Systems Report

Liam Wu - RTS Game Development

Contents

1. Identifying and defining

1.1. Define and analyse problem requirements

Problem context

Students **analyse** the problem by **describing** each of its individual components and **explaining** how each of these components contribute to the problem needing resolution.

My client has experienced a lack of satisfaction in his life, due to limited new RTS releases. His disliking of current popular games, due to lack of strategy definition/focus, has led to him being unable to properly enjoy his free time and subsequently had an effect on his mental wellbeing. It has greatly affected his work with longer undefined hours of work that cross into rest periods and thus less efficient/effective productivity within set work times.

Needs and opportunities

Students **describe** the needs of the new system to be built based on the problem context and using the table given below.

Need	Description
1. Client satisfaction (Take up client down-time)	The final system should allow the client to properly enjoy his downtime and thus respect the boundaries of his work life balance.
2. Focus on Strategy	The system should have a strategy at the core of achieving success in the game. This means having unpredictability through unexplored and new strategies rather than RNG or random events.
3. Have core features of RTS	All of the RTS genre have features of unit development, resource management and unit control. This should also have these and expand on them.

Boundaries

Students **analyse** any limitations or boundaries in which this new system will need to operate. Boundaries can include but are not limited to: hardware, operating systems, security concerns etc.

The system will need to have hardware requirements such as running on a computer and despite aims in development the nature of RTS game especially later with higher unit numbers will require a

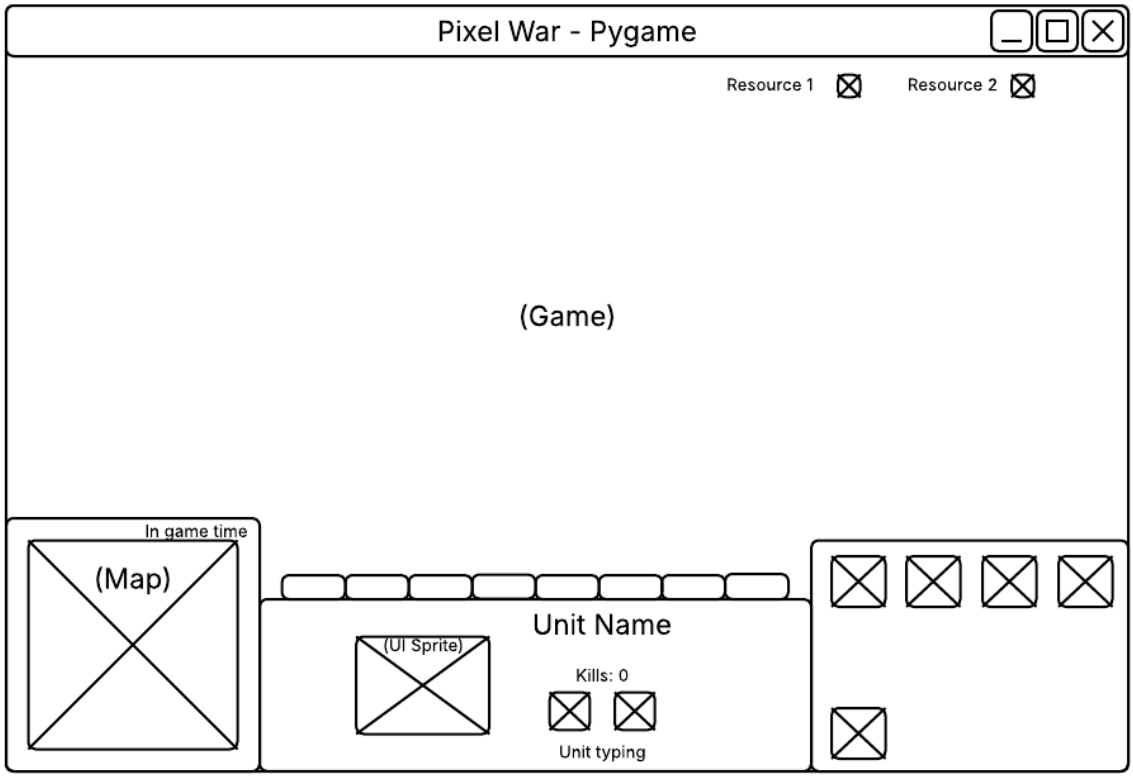
basic capacity to render all the assets at a relatively smooth frame rate. The system will require a separate API for downloading and registering accounts for the main game system.

1.2. Tools to develop ideas and generate solutions

Application of appropriate software development tools

Students **apply** appropriate tools such as brainstorms, mind-maps, storyboards and prototypes.

UI storyboard for the in-game client



1.3. Implementation method

Students explain the applicability of the implementation method for the current project. These methods are normally direct, phased, parallel, or pilot.

This project will use direct cut-over implementation. There is no need to have both systems working at the same time as patches should be applied as quickly as possible. There is no point in pilot implementations as the players should all equally get to experience newer versions of the game. This ensures that all game breaking bugs or exploits are removed as soon as possible from the project. This method however requires extensive testing as the whole system will fail if the new patch has a game breaking bug. Which is common for game development.

1.4. Financial feasibility

Students are to **conduct** a financial feasibility study, including producing an opening-day balance sheet, to assess whether their application is financially viable.

SWOT Analysis

Strengths: ● Use of agile and personal connection with the client allows for targeted and evolutioning response ● The product/system is free with no monetary systems	Weaknesses ● One man team leading to slower production of the system and changes from the client ● Weak maintenance ability post-deployment
Opportunities ● Well defined client ● Potential business growth	Threats ● The client could lose interest or be unable to fully interact with the final product due to changes in the lifestyle ● Shrinking demand for RTS genre

Study	Go or No Go?	Assessment and evidence
Market feasibility	Go	Even in spite of the shrinking market, there is a lack of other new releases meaning the RTS community will be encouraged to try out this system if it gets full release.

Cost of development	Go	An indie developer's cost of development is his time. Being motivated to start the project means that this cost is very low and worth going through with production
Cost of ownership	Go	Maintaining the potential online server and the monetary fees that come with it are minimal and for a relatively small application bug fixes should be not continue to cost a lot of time
Income potential	Go	The product would generate a dedicated fan base which would allow for the monetization of accessories and other non-core game features for people who want to show support for the project to grow
Future expansion opportunities	Go	If the system is successful with the client and a broader release is also well received there is opportunity to expand into a business.

Opening Day Balance Sheet

Pixel Studio Open Day Balance Sheet

Assets	***Amount (USD)***
Current Assets	

Cash (Founder's investment & savings)	*$10,000*
Prepaid software subscriptions (Unity Pro, Adobe, etc.)	*$1,000*
Total Current Assets	***$11,000***
Non-Current Assets	
Computer Equipment (laptops, monitors)	*$3,000*
Software Licenses (game engine tools)	*$1,500*
Office Equipment (desks, chairs)	*$500*
Total Non-Current Assets	***$5,000***
Total Assets	***$16,000***

2. Research and planning

2.1. Project management

Software development approach

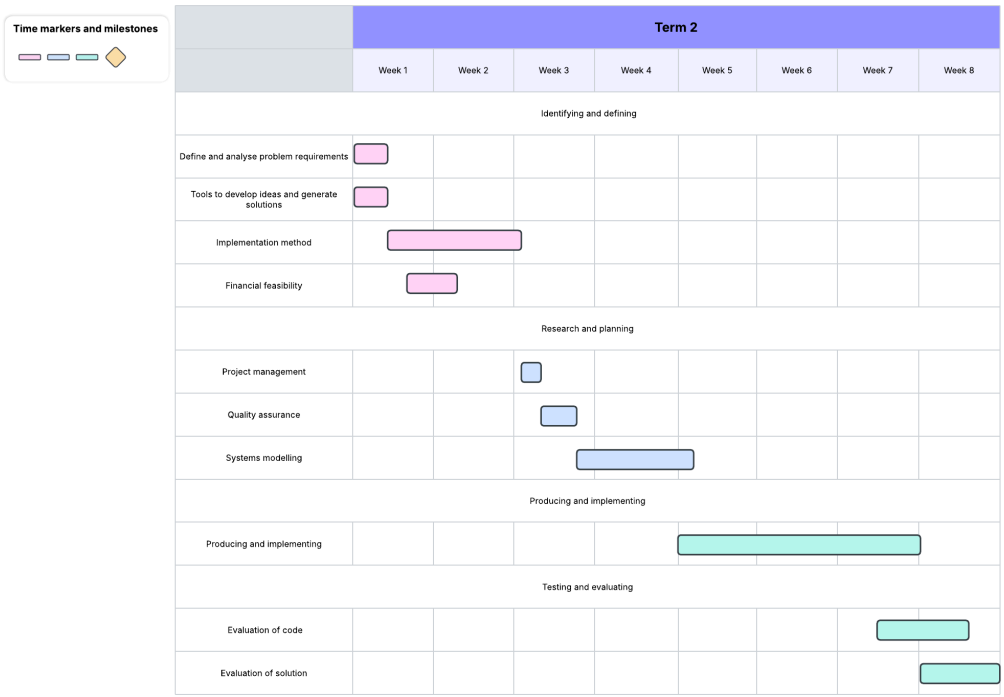
Students **explain** the software development approach most applicable for this current project. These are normally: Waterfall, Agile and WAgile.

Agile development is the approach for this project. This is because game development requires the feedback loop to previous stages in development to improve the final product. This project will also have frequent meetings with the client to have reviews at every stage of development which requires the flexibility of the agile process to work. The project will use the branches in GitHub as different sprites and have agile_artifacts for each sprint which contains the increment of each sprint and a review to reflect on the progress made in it. The product backlog will also be an updating list to show all small deliverables that need to be done in order to make the final product.

Scheduling and task allocation

Students **develop** a Gantt Chart that details the tasks required to be completed, person or people assigned to each task, timeline that does not exceed the project due date, resources required. In addition, students **identify** any collaborative tools used. For example Repl.it, GitHub and so on.

PixelWar RTS GANTT Chart



2.2. Quality assurance

Quality criteria

Students **explain** quality criteria based upon the needs from Section 1.1. These quality criteria should contain qualities, characteristics or components that need to be included or visible – based on Section 1.1. – by the end of the current project.

Quality criteria	Explanation
Independent application	The project should be packaged as a full application which can be opened on a PC like all other applications that are downloaded. E.g. through clicking the game shortcut or opening the application other means
UI (post app start)	An user interface that can allow the user to change their experience in the game. This will include changing keybinds, difficulty of opponents and also what factions they are going to play.
Complete PVE system	Upon interacting with the opening menu to start a game, the game engine should open to a complete user experience with challenging AI/hard coded bots and completely sprite-d with diverse factions with strategy as the core component. The game will also be overlaid with a game UI that tells information of: a mini-map, resource/unit collects, commands for the unit and an in-game timer.

Compliance and legislative requirements

Students **explain** compliance and legislative requirements their projects need to meet and how they plan to mitigate them where possible. For example, projects that deal with sensitive personal data being publicly available may fall under the Australian NSW Privacy and Personal Information Act (1998) and/or Federal Privacy Act (1988). Alternatively, international standards on information security management such as ISO/IEC 27001 may also be applicable.

Compliance or legislative issue	Methods for mitigation
Data Security (NSW Privacy and Personal Information Act 1988)	• All user inputs are going to be sanitised which will remove/pacify malicious data. This stops many sources of attacks such as SQL injection. • Post-sanitisation, all data will be encrypted and stored with salt and hash within salt and hashed databases • Strong authentication where clients are required to create an unique account with strict password requirements
User Anonymity (Federal Privacy Act 1988)	• A privacy policy that will dictate how the user's data will be used and methods for them to stay anonymous within the game system
Ownership (Copyright Act 1968)	• The project will not use copyrighted/patented material or media. All assets and features will be self-developed and therefore not violate any copyright laws.

2.3. Systems modelling

Students are to **develop** the given tables and diagrams. Students should consult the Software Engineering Course Specifications guide should they require further detail, exemplars or information. Each subsection below should be completed with Section 1.1. in mind.

Data dictionaries and data types

Students take the needs identified in Section 1.1. of this Systems Report. For each need, students **identify** the variables required, data types, format for display, and so on.

Not Irrelevant

Need

1.Client satisfaction/User Experience (Game experience)

Variable	Data type	Format for display	Size in bytes	Size for display	Description	Example	Validation

Need

2.

Variable	Data type	Format for display	Size in bytes	Size for display	Description	Example	Validation

Need

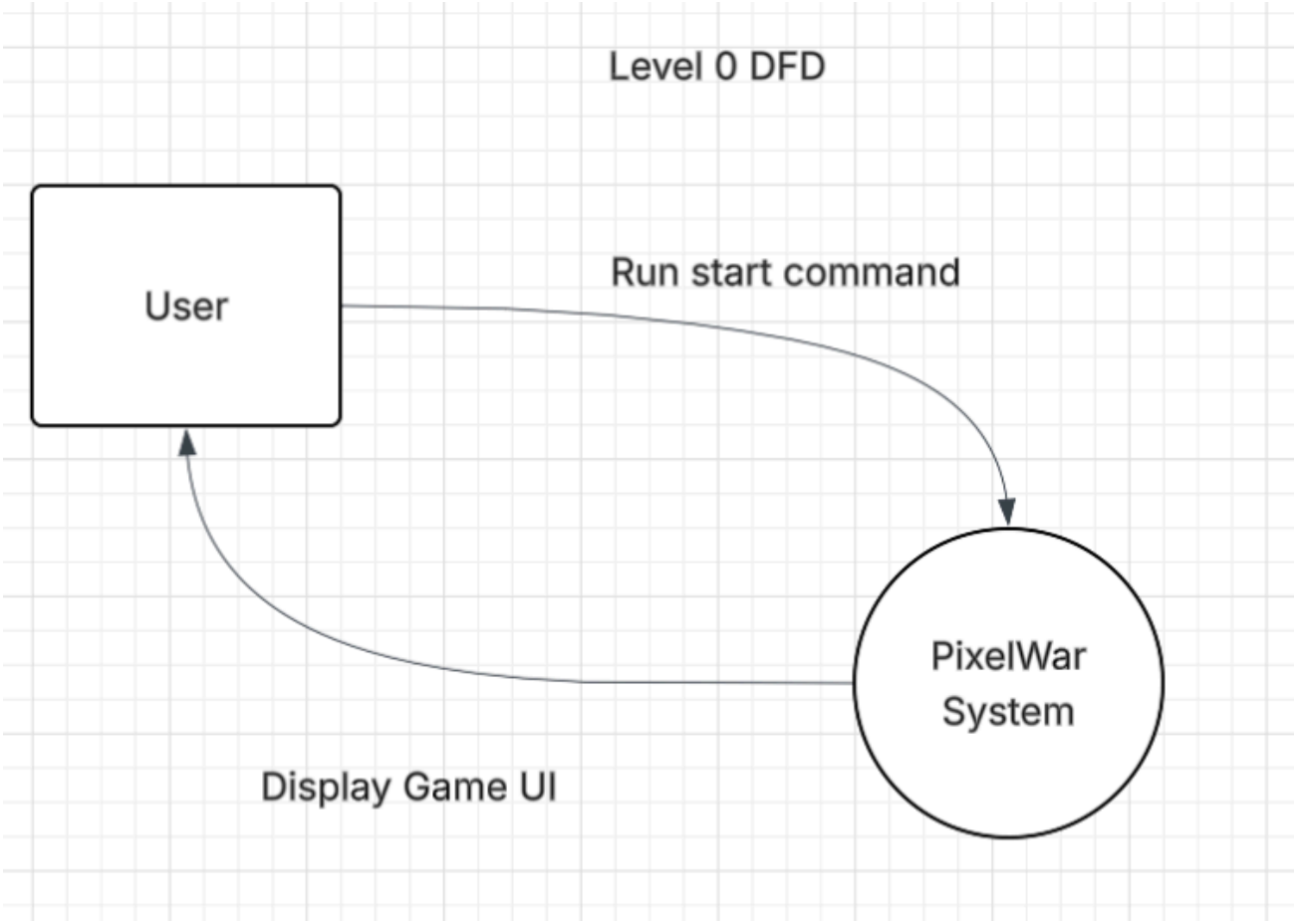
3.

Variable	Data type	Format for display	Size in bytes	Size for display	Description	Example	Validation

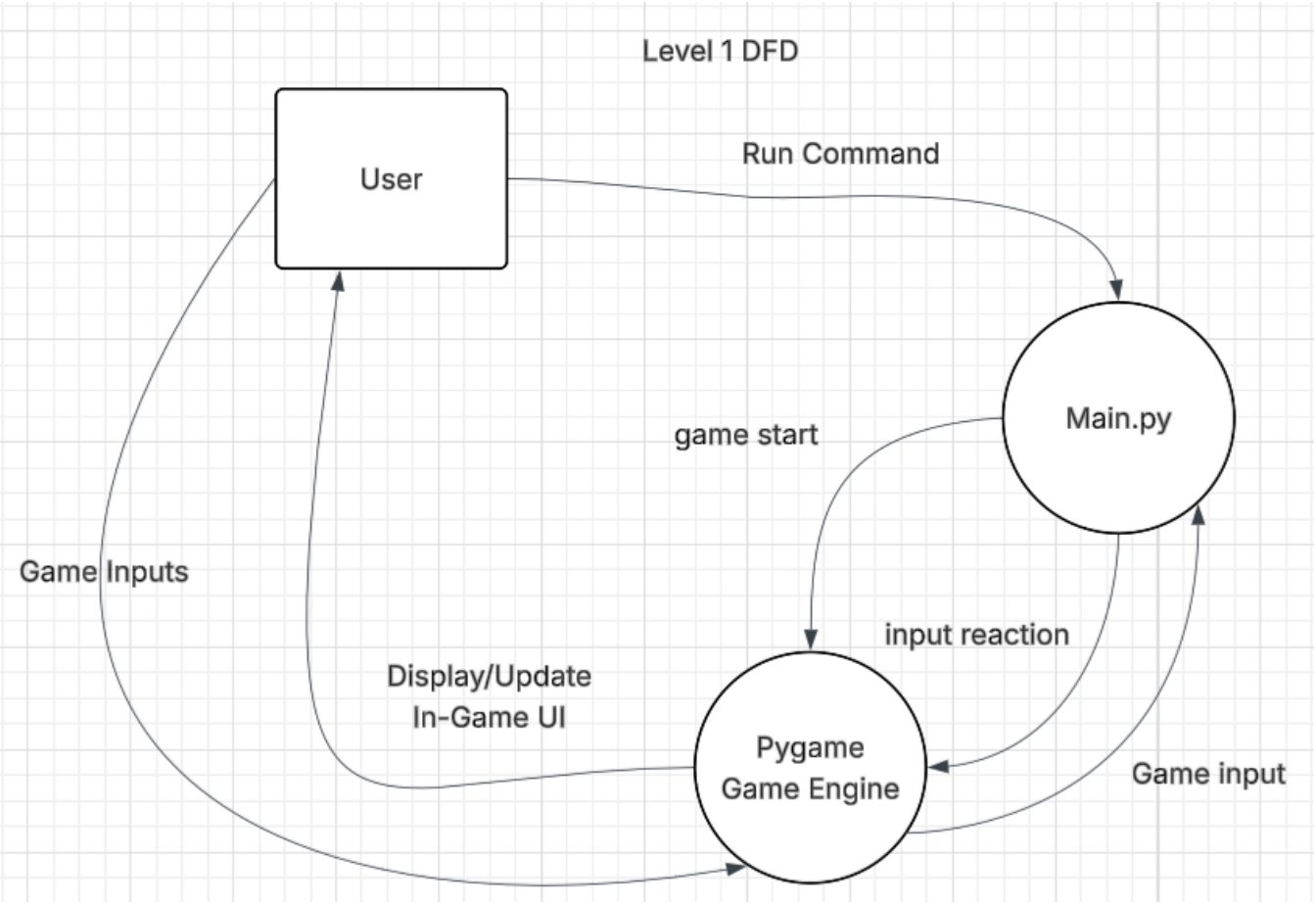
Data flow diagrams

Students **develop** data flow diagrams (DFDs) at Level 0 and Level 1. These diagrams should explicitly include the variables from the data dictionaries previously identified as well as the needs identified in Section 1.1.

Level 0

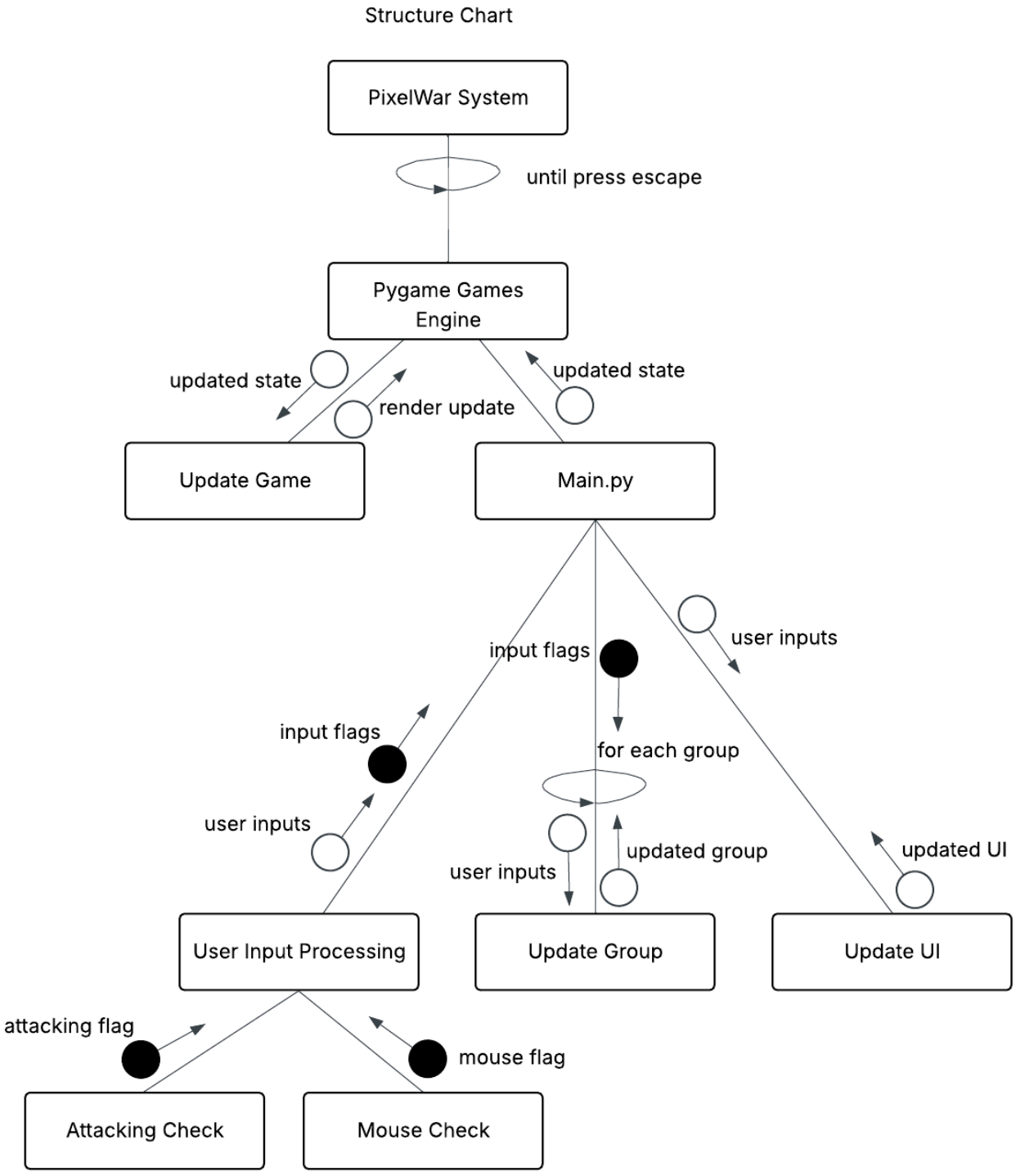


Level 1



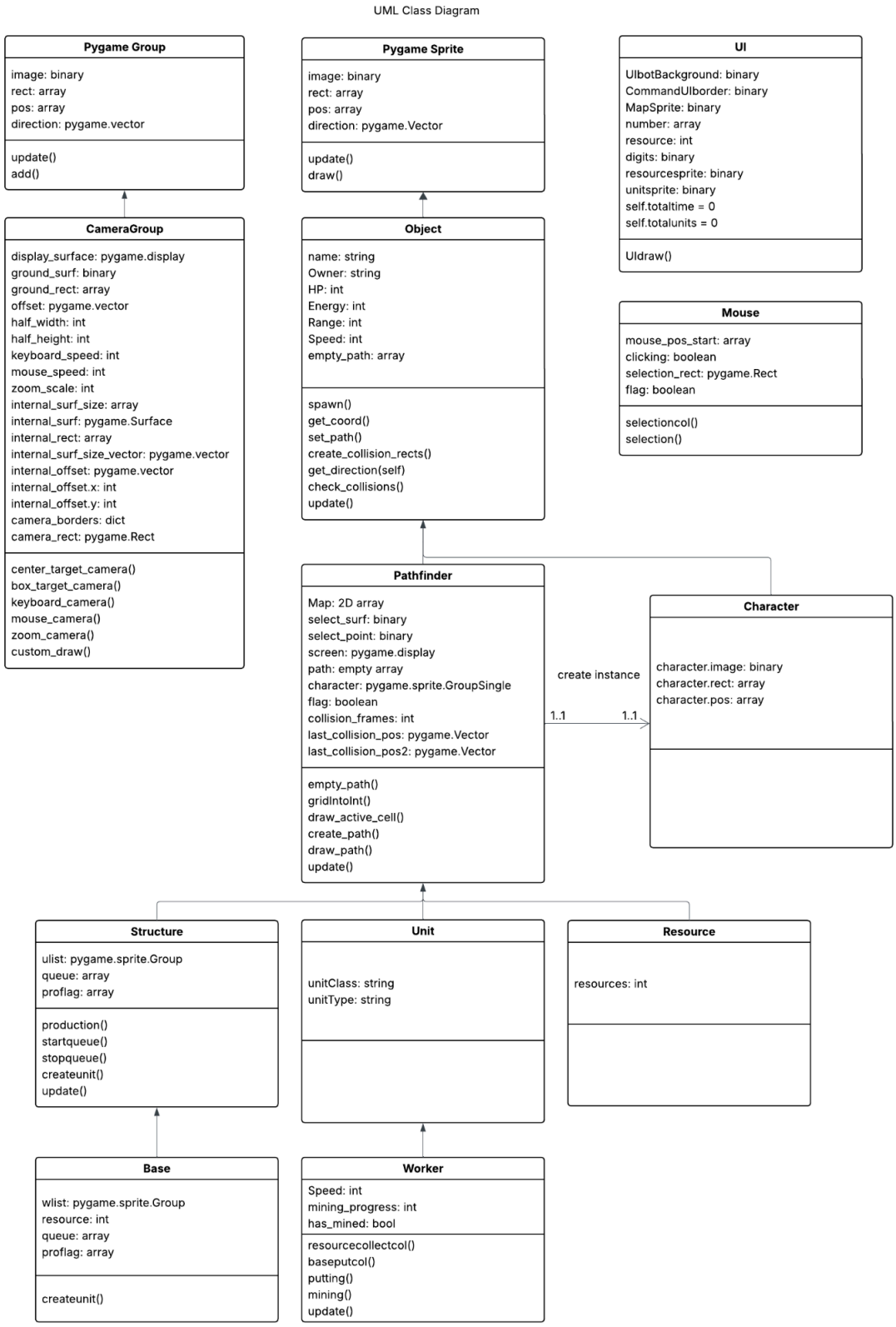
Structure charts

Students **develop** structure charts demonstrating how the procedures, modules or components of the final solution are interconnected.


Structure Chart
PixelWar System
until press escape
Pygame Games Engine
updated state
render update
updated state
Update Game
Main.py
input flags
user inputs
input flags
for each group
user inputs
user inputs
updated group
updated UI
User Input Processing
Update Group
Update UI
attacking flag
mouse flag
Attacking Check
Mouse Check

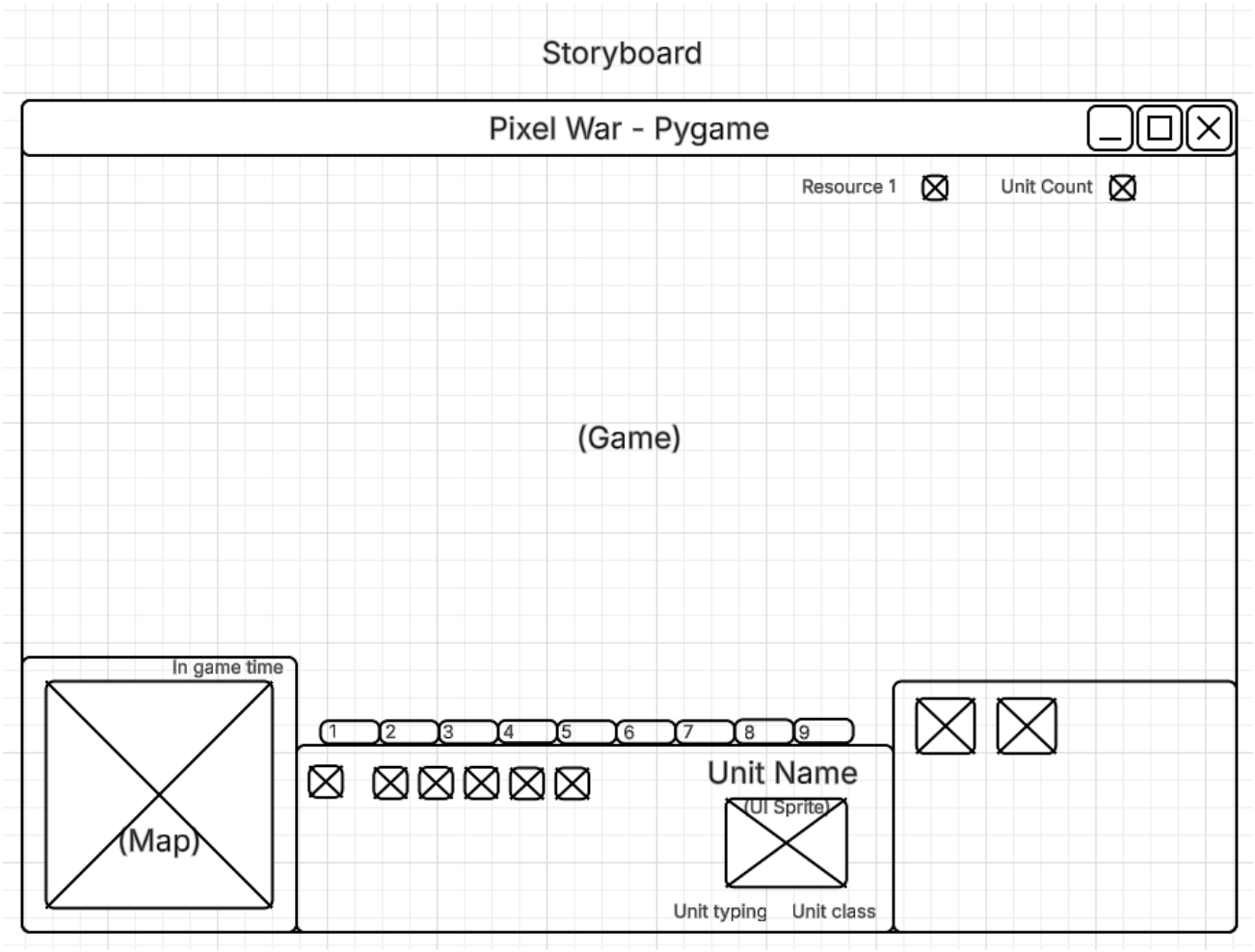
Class diagrams

Students **develop** class diagrams demonstrating how each class is related to the other.



Storyboards

Students **develop** storyboards, visually representing the software solutions they will build.



Decision trees

Students **develop** decision trees to visually outline the logic flow and chain of decisions or selections the final solution will need.

Irrelevant to program

Algorithm design

Students **develop** algorithms using methods such as pseudocode or flowcharts to solve the problem and meet the needs from Section 1.1. These algorithms should explicitly include the variables from the data dictionaries created in the previous section.

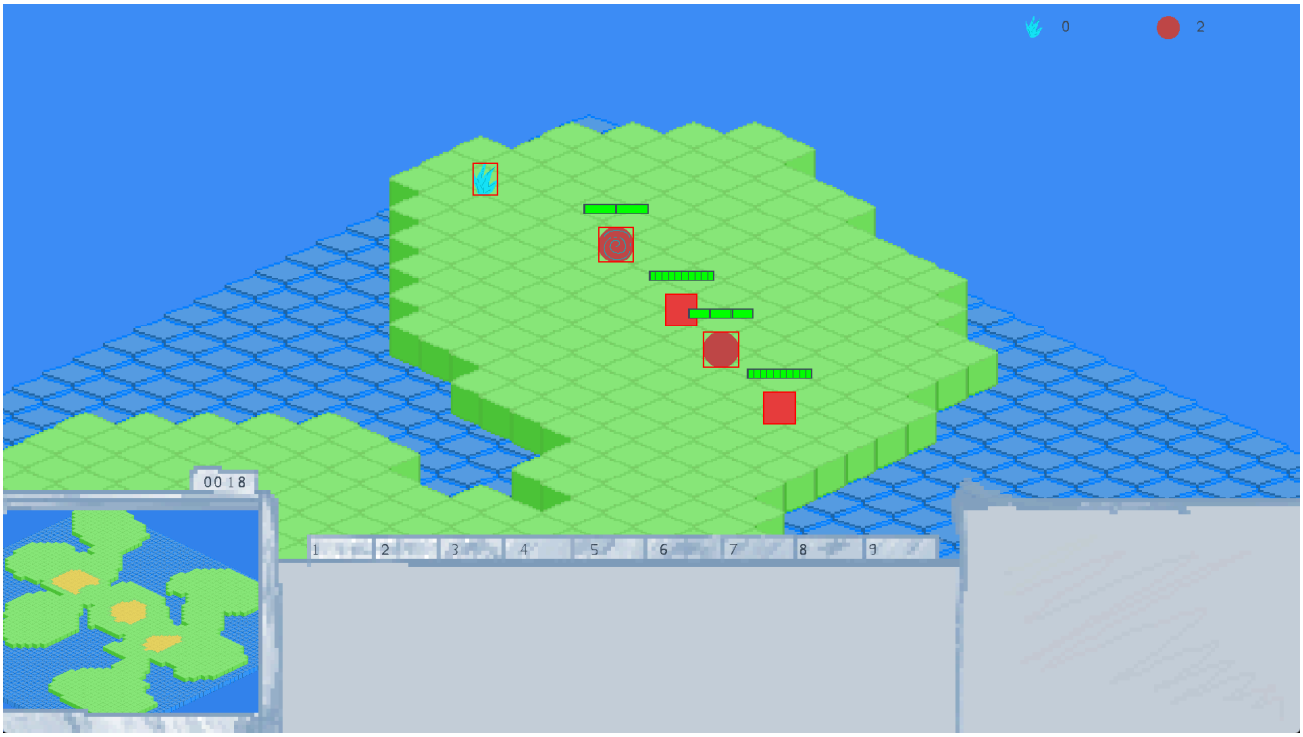
Irrelevant to program

3. Producing and implementing

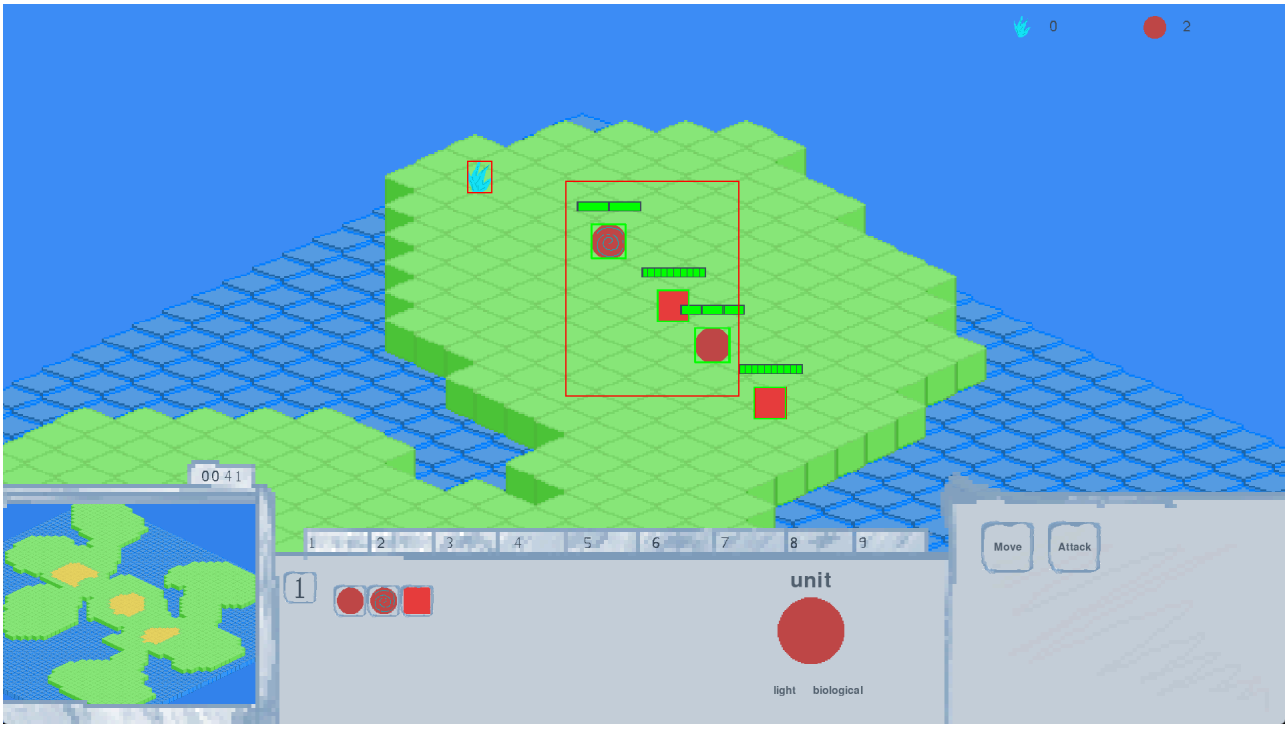
Solution to software problem

Students are to **include** screen shots of their final developed solution here. Each screenshot should include a caption that **explains** how it links to the:

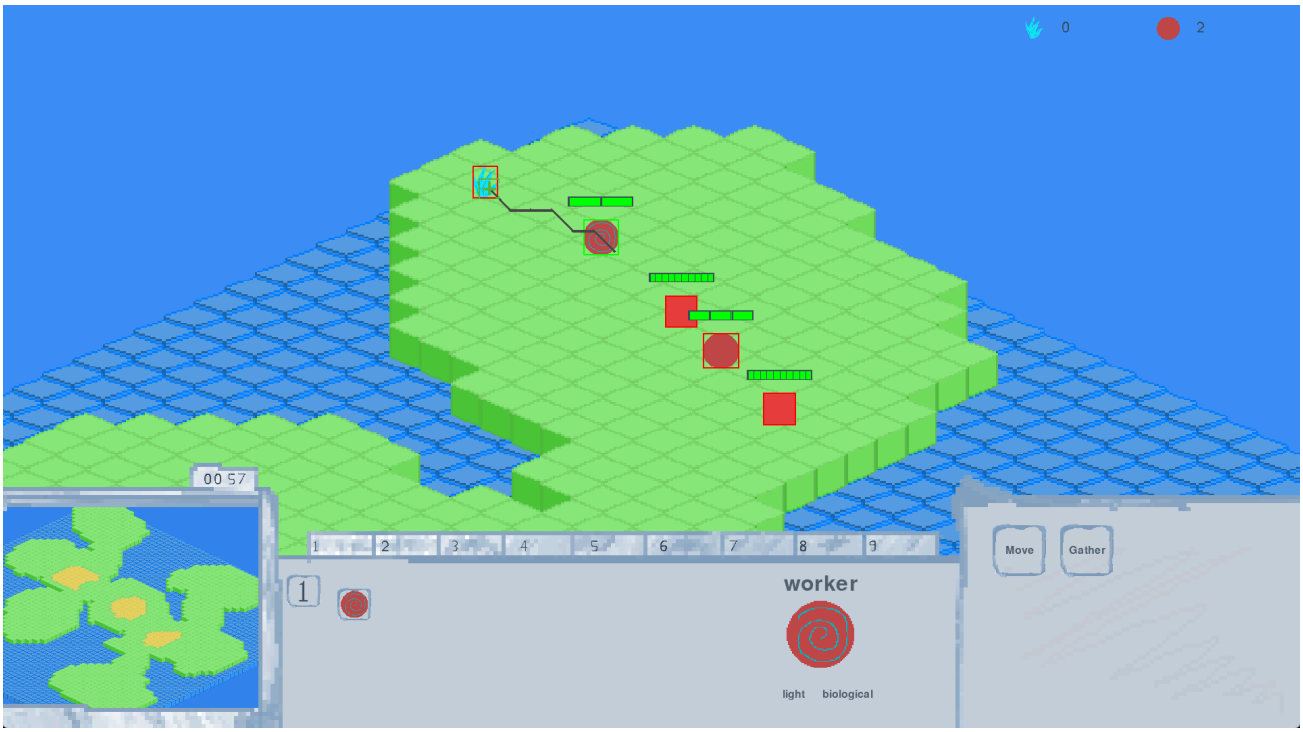
- Needs identified in Section 1.1.
- Components of Section 2.3. such as the storyboards, data dictionaries and so on.



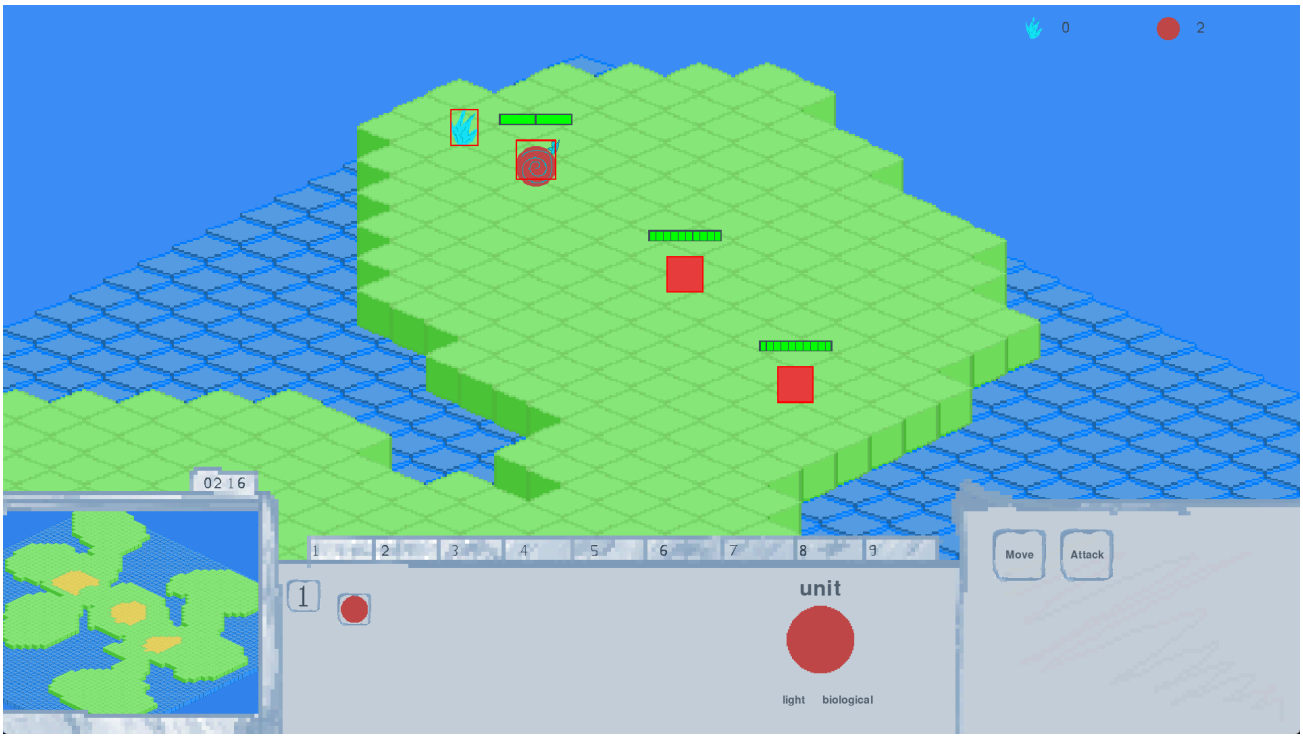
This is the base UI for the game as described in the storyboard with resource and unit count with the inactive areas in the bottom UI which will update with selection commands. This is before any user input into the game and the sprite is an instance of the UI class in the UML diagram. It shows all the classes (from left to right):resource, worker, base, unit, structure with class inheritance described in the UML class diagram.



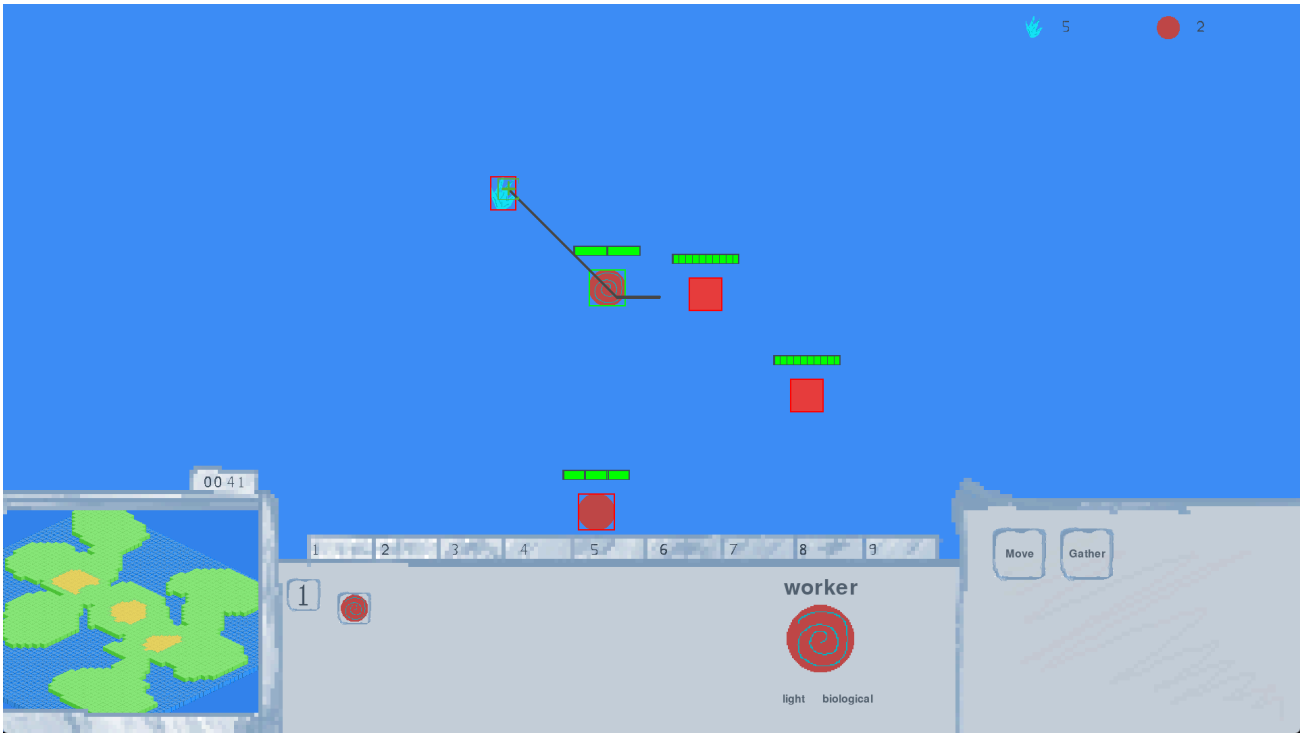
Selection box using left click hold and dragging it with the mouse. All hitboxes (shown by red boxes around the unit sprites) that collide with this boxes will be selected and played into the display list inside the UI class and rendered in topleft of the middle section with the number showing the page number. There are also command UI. This is the exact interface that the storyboard shows.



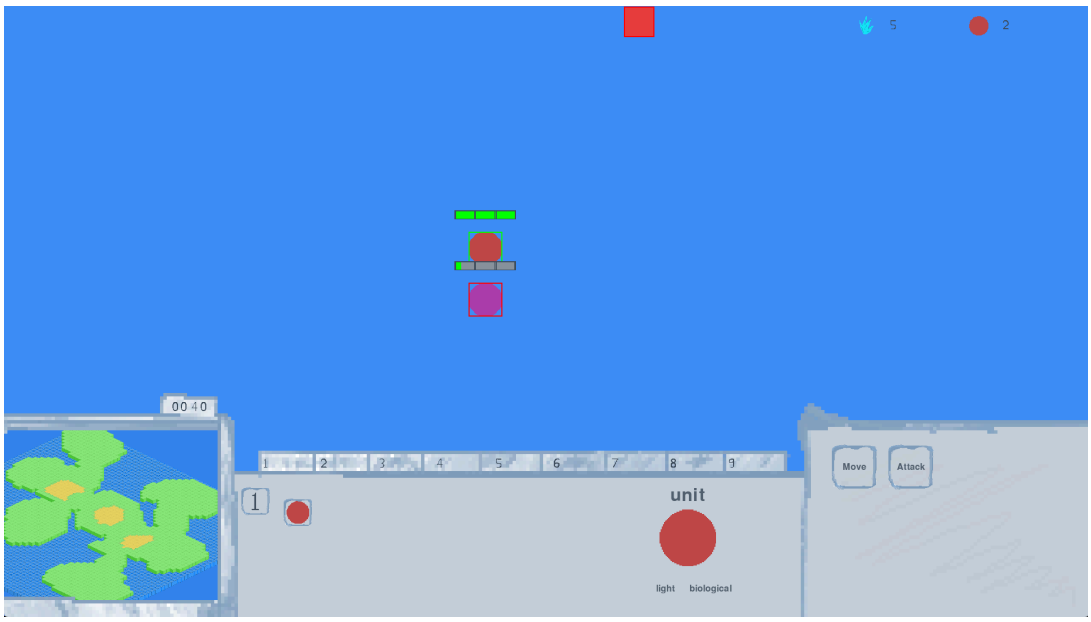
Pathing system rendering. Selected group rendering with the Cameragroup class



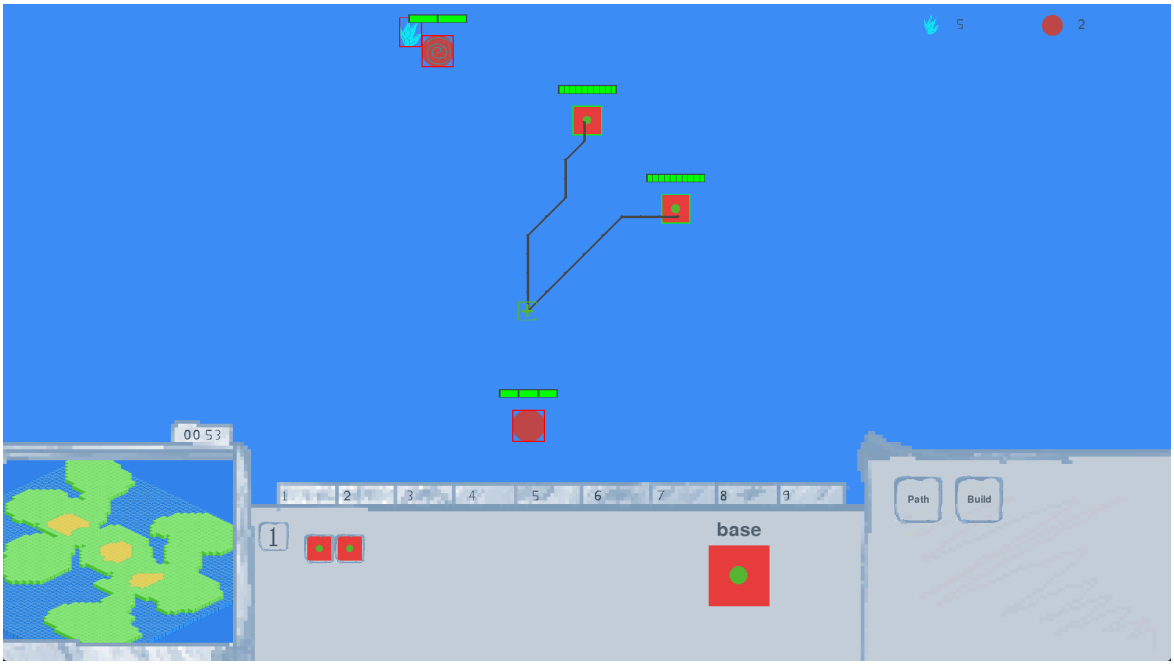
Resource collection process. The sprite update functions and rendering is also shown with the unit “holding” the resource and shows the new state of the unit. The autopathing from the resource to the base is also shown (the unit is moving back from the resource without user input)



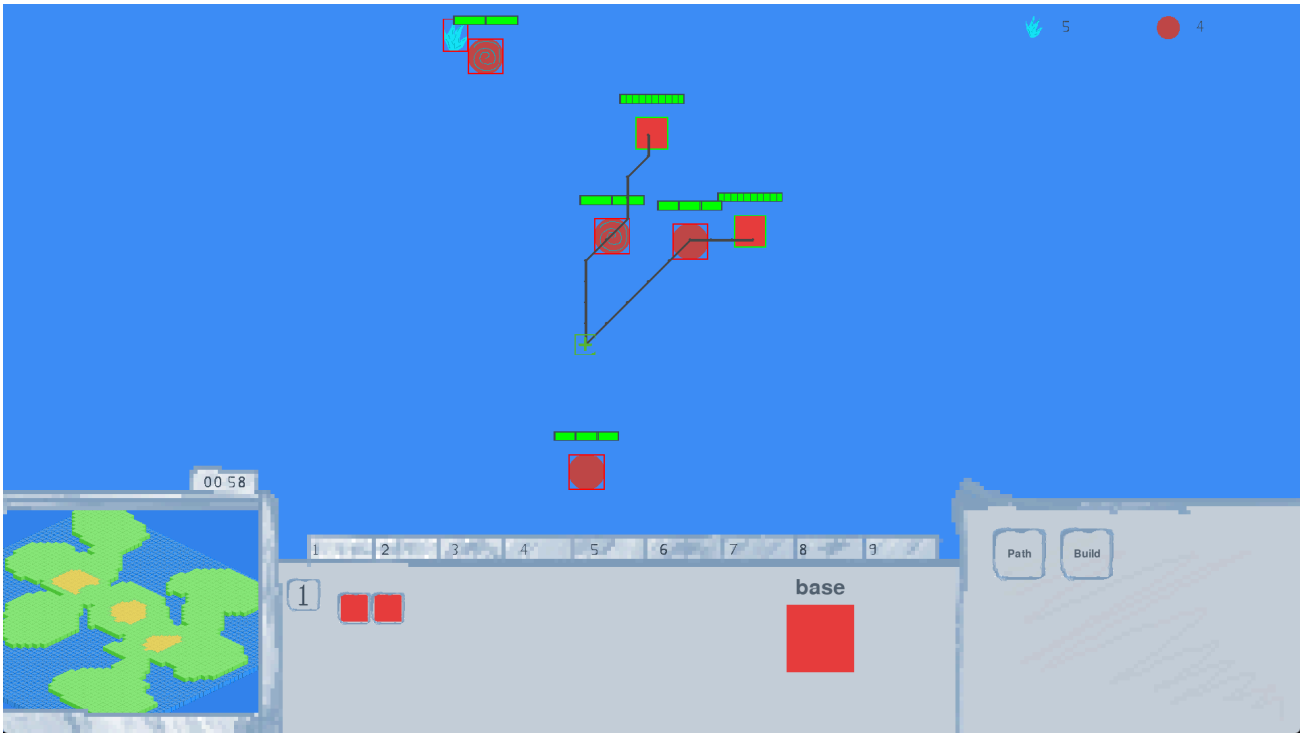
Autopathing process shown with a new path set after putting the resource in the base. (Map deloaded for higher performance). UI updating from the added resource.



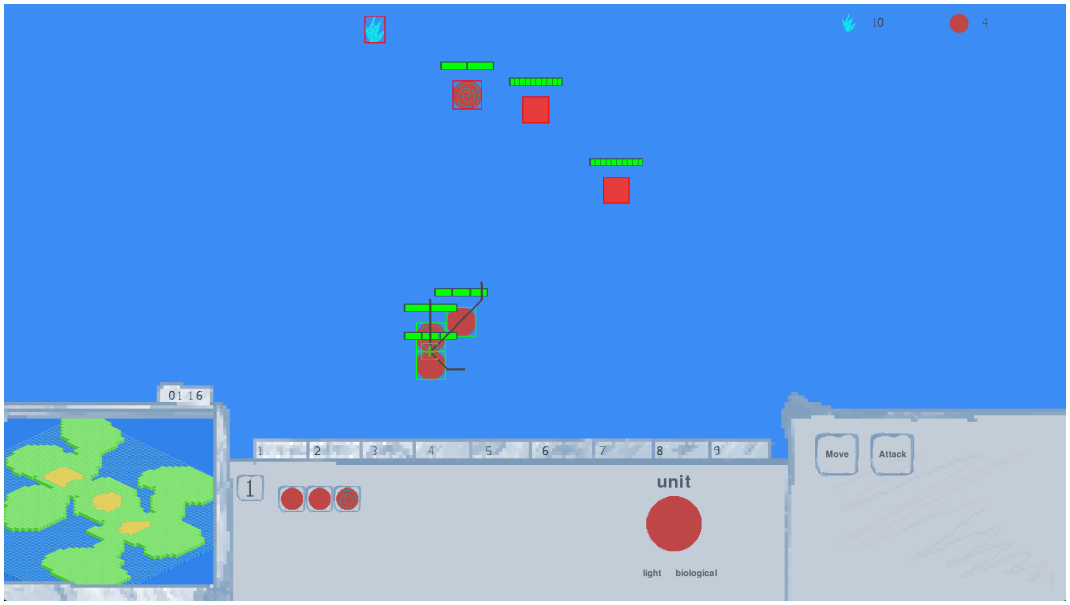
Attacking and HP UI updating to respective unit health. (taking user input from the processing described in structure chart) The location of the camera is updated by pushing the mouse to the edges of the screen (mouse camera movement) common in many top down games. Also showing of enemy system where the units can only attack enemies.



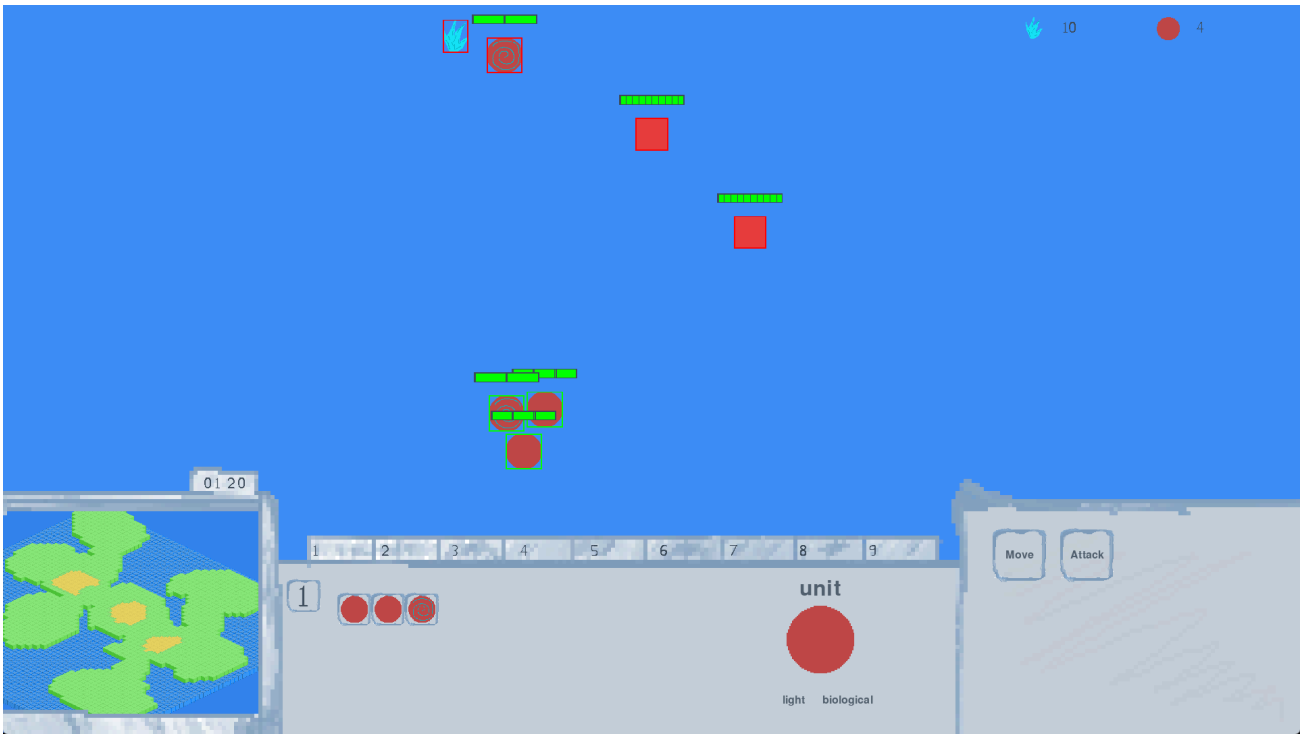
Production active state for structures. While structures are producing units their sprite is updated with the green circle to reflect this. This is based in intuitive UI design which are common in engaging interaction with the user. It also rewards information gathering for opponents where they will be rewarded for strategising around scouting (Needs 2) (if fog of war mechanic got implemented).



Path inheritance. When a unit is made they will inherit the path that the structure has similar to rallying mechanics in other RTS games matching “Needs 3”. The top right UI updated to unit count.



Group pathing (selection group and selected attribute rendering) and collision system



Constant-collision correction system. After a ten frames of consecutive recollision the units will stop “fighting” for the point they are pathing to. Similar mechanics exist in all pathfinding-based games (Need 3). And the set number of frames removes potential randomness from the system (Need 2)

Version control

Students **describe** what version control system or protocol was implemented.

I have used github and used branches to capture the different versions of the project throughout development. Smaller version changes such as unit changes within the system are saved through git commits which all have small titles and descriptions that represent the change to the system that has occurred with a line of codes added and deleted from the last commit. When a sprint is complete, branches merge into the main branch through pull requests. When accepting a pull request github automatically runs static and dynamic testing on the new version to ensure that it doesn’t clash with the main branch. Once merged a new branch is made to represent the next sprint which ensures the current version is continued to be developed.

4. Testing and evaluating

4.1. Evaluation of code

Methodology to test and evaluate code

Students **explain** the methodologies used to test and evaluate code. Methodologies include:

- Unit, subsystem and system testing

  Unit testing when the smallest component of a system is tested. In development I was able to achieve this by immediately testing that component post-implementation and checking if the requirements were set in the increment in the agile_artifacts.

  Subsystem testing is when combining multiple components. These tests come at the end of each sprint and tests all the new components and if they interact correctly against the increment criteria on the agile_artifacts. Problems that come from this testing will be addressed before merging and completing the branch/sprint. This would prevent possible oversights and downstream issues.

  System testing is done across sprints after subsystem testing by combining the subsystems of all previous versions to ensure subsystem integration continues to meet all the increments in agile_artifacts and performs as expected as a whole.

- Black, white and grey box testing

  Black box testing is dynamic testing where only the frontend is given to test. This was done at the end of the project alongside the system tests. It is to capture the final user experience within the project and try to make all UI elements intuitive and optimise performance of the product. The client would do a black box test at the end of each sprint to get feedback and redefine/solidify requirements.

  White box testing is a form of static testing where the tester is only exposed to the backend of the system. In the project, I regularly leave comments above functions and label testing lines to make it clear what each function does and constructs a clearer sense of the logic. Code review at the ends of sprints was the primary form of white box testing. Github also automatically does a static test at branch merges. This is done to analyse code logic and quality in each component of the project.

  Grey box testing is testing with access to both frontend and backend elements. Using debugging tools at unit tests and subsystem tests allowed for the visualisation of code logic while also showing the resulting frontend. This was critical in finding and accounting for bugs within the different versions of the game.

- Quality assurance.

Unit level debugging and consistent client meetings were all part of quality assurance throughout this project. By running debugging tools on the logic within individual functions the code quality could be optimised and help to meet the frontend requirements from the agile increments. Consistent meetings at the end of each sprint/version allowed the project to continue to progress in the desired direction while also getting feedback for the new features to allow for client satisfaction.

Code optimisation

Students **explain** the methodologies used to optimise code so that it runs faster and more efficiently. Methodologies include:

- Dead code elimination

  At the conclusion of development of the project. There was a system wide debug. Using the python debugging tool in Visual Studio Code and putting breakpoints at all class methods and functions to map out all of the logic of all the features in the game and, through removing breakpoints, code that is not executed within the running of the system but still compiled is removed. Testing functions are commented out to minimise the amount of code compiled.

- Code movement

  Using python debugging tools, co-pilot analysis and code reviews, code movement has been integrated throughout the project. Code movement is the idea of reducing the number of times code blocks are run without affecting the resulting program. By having general practices of moving definitions outside of loops when possible as allowed for better performance. Another form of code movement used is to have control variables for every “reaction” in the project. In the interactive media that the project is based in, running all actions every frame will severely affect performance by nesting large functions (reactions) inside in control structures like “if” statements it only executes the code when it is required from user input updating the flag.

- Strength reduction

  Strength reduction is the substituting of code blocks for simpler (less intensive) but equivalent functions. Through analysis from mentors and my personal understanding of pythonic coding, I have been able to optimise the code through catering the logic to be easier for the compiler. I also replaced, especially intensive functions containing large loops, with framing them in a way that best suits the final implemented version rather than a broader idea of future development. An example of this is the map rendering where it was changed to an image rather than using py.tmx to render every tile every frame which was very computer intensive. (this functions is still present in the code but commented out to not affect the program)

- Common sub-expression elimination

Sub-expression elimination is when repeated occurrences of the sub-expression are “substituted” by a temporary value that is passed through processes. In this project, this is most applicable to vector offsets of position and rendering. This process is throughout the rendering functions in CameraGroup draw functions and position updating functions in the update methods of all the classes. All objects are also instances of where this method is also applied like using pygame.Surface object to pass into pygame.__surfaceName__.blit() rather than passing in the whole subexpression.

- Compile time evaluation – constant folding and constant propagation

Compile time evaluation is having as many constants/variables defined within compilation and not runtime. This would improve the performance of the program while running. This technique is the basis of the OOP approach of the program such as loading sprites as attributes within classes and the definition of global flags before the game engine loop. Constant propagation optimised the pathfinding system by taking away the concept of cell size and replacing it with the constant of “32” as the project scope did not include different grid sizes.

- Refactoring

The OOP paradigm in the project is most of the refactoring in it with the abstraction of functions as methods in instances of classes. This is through the project where functions are separated into different files and functions away from the mainline where the code is executed. This allows for the runtime loop of pygame to be as readable as possible. Every subsystem of the project is separated into its own file which enables future development and was effective at the conclusion of implementation when system tests found bugs.

4.2. Evaluation of solution

Analysis of feedback

Students **analyse** feedback given to them on the new system they have just created. This feedback can be in the form of an interview, survey, focus group, observation or any other applicable method. Students should also include overall positive, negative or neutral sentiments towards the new system in their response.

Sprint/Version 1 Feedback Analysis:

The client meeting at the end of the sprint had very positive feedback for the pathfinding system developed. There was heavy emphasis on the movement of the sprite being smooth and the UI being intuitive. The critiques of this version of the system was its response to rapid input from the user causing the pathfinding to stop working. This was due to the system requiring the pathfinder to be in the center of the grid they are pathing to where constant input causes the pathfinder to keep returning to the nearest centre and thus not progressing. This is a constraint of the system I’m using

and can't be addressed without changing the library I'm using but the positive feedback of the intuitive UI and ease of user experience will be aspects I'll continue to develop.

Sprint/Version 2 Feedback Analysis:

This meet up with clients yielded more positive feedback with the system. The class system of workers and units was well received and the different sprites allowed them to differentiate between them. The production system also received praise with the sprite update when queuing units and producing units.

Sprint/Version Final Feedback Analysis (lack of client meetings):

Most of the feedback was very positively received in the version 6. The map background with the pathfinding valid tile system was the main positive along with the UI updates. The only critique was the performance of the game with the frame rate between terrible (3-6 fps). I took this feedback and did as much code optimisation as possible to improve performance.

Testing methods

Students **identify** the method or methods of testing used in this current project. For each they use, students are to **explain** how and why it was used.

Method	Applicability	Reasoning
Functional testing	The project uses the dynamic unit testing to test every function/ class method in the program.	The OOP paradigm is based around the idea of unit, subsystem and system testing. Functional testing is very effective in ensuring that cascading problems don't end up causing large scale issues within the system. Which is critical in a large scale project such as this.
User Acceptance testing	The client does tests on the specified aspect of the system using the UAT documentation provided previously.	Games are an interactive user experience and thus must be tested by users to fully capture the reception of the product. It is also a very easy way to also get feedback for the project from the testers.
Live data	The client playtesting the whole system, a black box test, where the user provides feedback on their user experience	Same reason as before the sole focus of a game is the player base and their reception.

Simulated data	To simulate enter state that the game could be in which allows for thorough subsystem testing and testing responses to said states.	This is to catch any issues before getting live data from playtesters and will likely cover areas that live data would miss in normal exposure to the game (catching unlikely situations).
Beta testing (same live data in this scenario)	The client playtesting the whole system, a black box test, where the user provides feedback on their user experience	Same reason as before the sole focus of a game is the player base and their reception.
Volume testing	Used by rapidly executing user inputs to stress test the system and account for failsafes within the system.	In a game where limit testing is commonplace in the community it will be surrounded by. Volume testing becomes invaluable in making sure the game continues to perform properly under high loads.

Security Assessment

Students are to **perform** an extensive security assessment of their final application and e**xplain** the countermeasures implemented.

Threat	Countermeasure
Game Crashing	• When the user clicks outside of the grid that is provided in-game, the pathfinding automatically correct to a place on the grid to prevent error crashing

Test data tables

Students **identify** variables which were used for either path and/or boundary testing. Students **develop** these test data tables based on their algorithms versus their real code. Students then **state** the reason for including said variables.

Variable	Maximum	Minimum	Default Value	Expected Output	Actual Output	Reason for Inclusion
zoom_scale	4	0	1	The map is scaled together with all the other sprites to give the effect of zooming.	The zoom scales all the sprites correctly.	The pygame zoom scale documentation has a point where the value is negative which causes an error
proflag	[1,1,1,1,1]	[0,0,0,0,0]	[0,0,0,0,0]	The proflag should reflect the number of units queued via user input	The units that come out of the production process should reflect the proflag values and the value should reset after this is done.	To check that unit queuing works and the shifting of the slots when a unit is produced works as intended for that range.
resource	100	0	0	The number of resource gathered by mining processes should be reflected in the UI in the top right	When a worker puts a resource into a base class the counter goes up by 5 each time and accounts for place values when it hits two to three digits	Tests that the UI is able to update to higher digit values.

unitcount	10	0	2	The number of units is reflected in the UI in the top right	Whenever a unit is produced the counter in the UI updates to match the new amount of units and accounts for digit changes correctly	Tests that the UI is able to update to higher digit values.
cameralist	<...sprite>, <...sprite>...	None	<...sprite>, <...sprite>	When a new unit instance is made the cameralist updates by now rendering that unit's sprite and updating location	When a new unit is added the cameralist gains the unit sprite and renders it accordingly	The primary rendering function which should update to new sprites being added to it. It is critical to the function of the game.

Boundary testing

Analysis of solution against quality success criteria

Students are to take each quality success criteria from Section 2.2 and place it here. For each quality criteria, **analyse** the components of the solution that met or did not meet each quality criteria. Give reasons why each success criteria were or were not met.

Quality criteria	Met?	Analysis
Independent application	Did not meet	The development scope was too big for the final project of being a published complete package to be realised for the final build of this project.
UI (post app start)	Did not meet	The development scope was again too large for any account/login features to be developed in the time frame

		given and for any faction selection or bot difficulty settings to be introduced.
Complete PVE system	Met	This was the primary focus of the project and thus was able to be meant with the final version of the project. The game itself is not complete in terms of mechanics and spriting but the system that it sits in is fully fleshed out with UI, camera movement and object interactions that a standard game should. This section was also set up from the start of development which allows for efficient and streamlined development of this criteria.