3. Architecture

0. Notes

1.3.1 Representation **[10] (3 pages)**
- Abstract and concrete representation of the structure of the software
- Statement of languages / libraries
- Tools used to create representation

1.3.2 Justification **[12] (2 pages)**
- Justify the architecture
- Relate the concrete architecture to requirements IDs
- Consistent naming of constructs

4.4 Architecture documentation

You are asked variously for abstract (conceptual) and concrete architectures.
- You will be marked on how well your architecture and reporting conforms to the appropriate level of abstraction: you will lose marks for making the architecture too detailed, or ignoring instructions.
- You will be assessed on the clarity and appropriateness with which you state the language(s) and tools that you use.
- You will be marked on how clearly your architecture justification follows the structure of your architecture, and accounts for unusual features or notations.

3. Architecture [22 marks]:

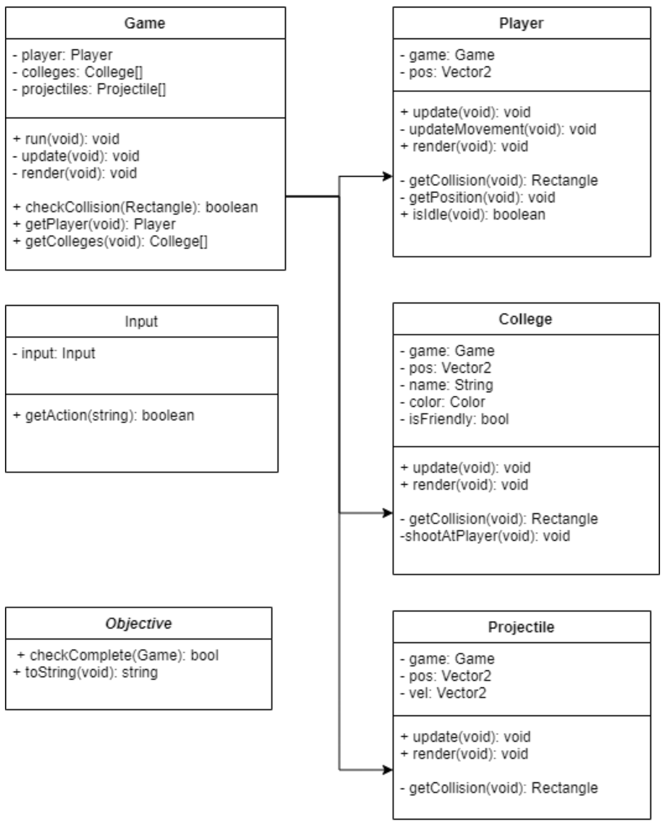
a) Give an abstract and a concrete representation of the architecture (structure) of the team's software, with a brief statement of the specific languages used to describe the architecture (for instance, relevant parts of UML), and, if appropriate, the tool(s) used to create the architecture representations (10 marks, ≤ 3 *pages*).
b) Give a systematic justification for the abstract and the concrete architecture, explaining how the concrete architecture builds from the abstract architecture. Relate the concrete architecture clearly to the requirements, using your requirements referencing for identification, and consistent naming of constructs to provide traceability. (12 marks, ≤ 2 *pages*)

By concrete we mean something in the world that can be observed directly (i.e. not from a model), like a software architecture implementation running in a certain hardware instance, or a business process going on in a company. Abstract means here something that is an abstraction of an ultimately concrete thing for prescription or description purposes

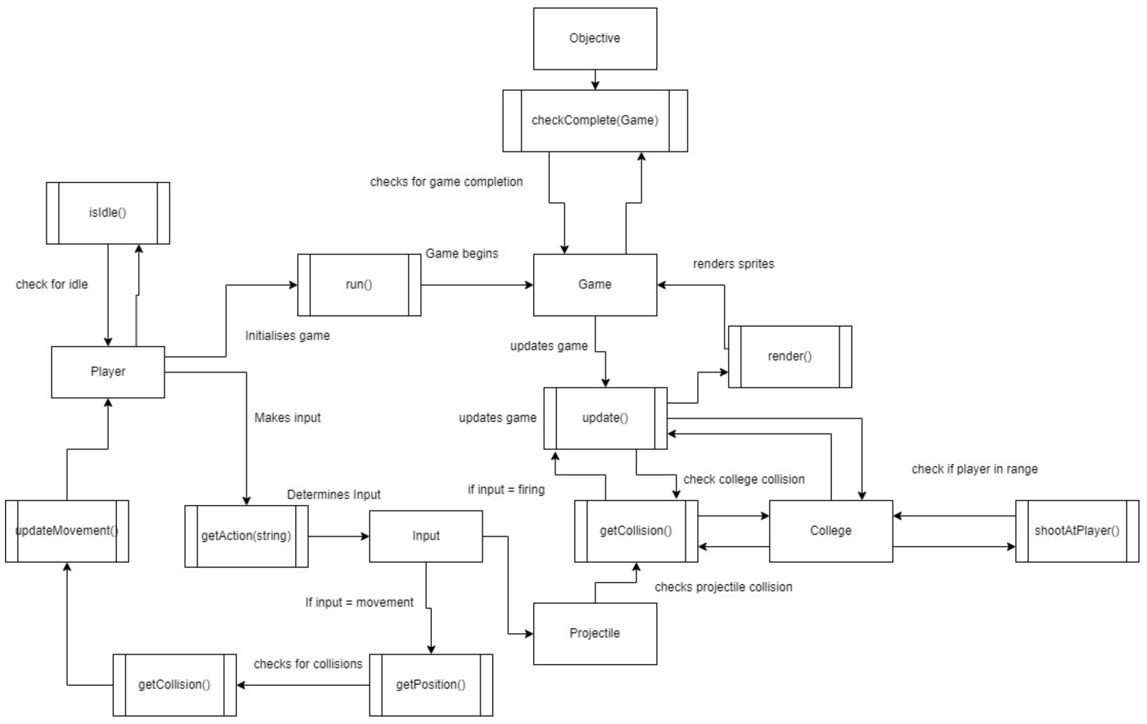
Possible justification

A. Representation

The initial abstract UML class diagram of our game:



The abstract data-flow diagram for our game:



Our initial abstract representation contains 6 main classes, these are the classes we have deemed as core components which we will need to meet the objectives. We have populated these classes with basic methods which we believe each class will need.

The 6 basic classes are:

- Game - The game class will be the main class for the game. It will keep references to a single Player class instance, and the multiple College instances. Within the Update and Render functions, it will call each instance's update and render functions. During construction, the Game class will load the tilemap, as well as extract the collision objects from it.
- Player - The player class will handle the players ship within the game. The player class will keep a reference to the Game, to allow it to access it's public functions. One of the main functions that it will access, is the checkCollision() function. This allows the player to check whether it is currently overlapping with the terrain, and react accordingly.
- College - The college class will handle all the colleges within the game. The class will keep reference to the game and will use its public functions. Like the player class it will make use of the render and update functions as well as have a shotAtPlayer() function to allow for combat.
- Objective - The objective class will keep track and check for completion of the game and individual objectives. This class will have functions to check the game state, and text which indicates the requirements.
- Projectile - The projectile class will create and manage all projectiles within the game. Projectile will also have its own Update() and Render() functions, which will make sure it continues on moving in the direction it is created with. It will also contain a reference to Game, which allows it to use the checkCollision() to be able to detect when it hits a Player or a Wall.
- Input - The input class will handle the keyboard and mouse inputs from the player and, through mapping, ensure the correct action is taken within the game. This will include moving the player, firing projectiles and resetting the game.

The data flow diagram above also gives a basic overview of how the classes interact with each other using the functions mentioned.

This is our concrete UML class diagram:

[ADD CONCRETE CLASS DIAGRAM]

[ADD CONCRETE DFD]

We created our abstract class diagram, and abstract Data-flow diagrams on diagrams.net (formerly draw.io) flowchart maker. The concrete class diagram was created using IntelliJ's built-in diagram maker. Both the concrete and abstract class diagrams use UML. UML was deemed appropriate for this project because of its tools which it provides for representing object oriented design.

B. Justification

The game takes place in a single screen, and as such a system for swapping between and handling multiple screen states for menu / game etc. will not be necessary; the game can be handled within a single Game() class. Variables are encapsulated by being made private, and providing public getter / setter functions when necessary. This means that they cannot be changed or accessed without the class's knowledge, allowing for maintenance of state.

One of the first major changes that was made when we began developing the game was the implementation of the checkCollidedHittable() function. This is a function that checks a Rectangle against everything that can be hit and damaged. This would also mean that there would need to be an IHittable interface, which Player and College implements and would check the difference between player friendly and enemy objects.

The 2nd change that was made was the implementation of the IHittable class. The player (and college) also will implement this IHittable class, and in turn implement the hit(float damage) function. This will deduct the damage from health, and also detect when it has run out of health.

The benefit of an interface over using a default implementation of the function is that the player and college can react differently - the Player will update the Game class when it is destroyed and the game state will change, meanwhile the College will destroy when it runs out of health. The visual update of destruction and health bars can be handled in other functions either in render() or update(). The College will need to update the Game class however, to allow for keeping track of the objectives such as the college becoming captured. This should also provide the player with points as well as gold.

Thirdly the UpdateAI() function for the colleges was expanded to include the primary logic for detecting the distance to the player as well as the direction to the player. The function will then decide whether to shoot at it or not. This uses the Game classes getPlayer() to acquire the reference, and then performs simple conditional statements and maths to figure out what to do. When deciding to shoot, this will in turn create a Projectile and call addProjectile() on the game class.

Next we adapted and expanded upon the Objective class. Given that the primary purpose of the game is to achieve a specific and different objective. We decided to implement Objective as an abstract class. Other Objective classes will inherit from this abstract class. This includes classes such as DestroyCollegeObjective, KillShipsObjective, AttainGoldObjective etc. The Game class will need to retain a reference to an Objective class.

We then added an Enemy class to add enemy ships to the game. This is to meet one of the requirements set in the design brief. In the current version of the game this class has limited functionality and only creates an enemy ship that can move and check for collisions with the map border. At present these ships do not have the ability to engage in combat and cannot be destroyed. The class has been built in a way that allows for its expansion in these areas.

Our concrete representation (as well as the initial abstract representation before it) is based on the requirements for the project. The architecture has been designed to meet all functional and non-functional requirements.

Firstly, the architecture has met requirements F001, F003, F004 and F005 by implementing an Input class which takes the keystrokes of the player and passes them to the Game and Player classes to act on accordingly. F015 is also met using the Input class as it simultaneously manages mouse input as well as keystrokes. The Input function also allows for keys to be remapped (NF007) and alternative inputs (such as a controller) to be used (NF008). This ensures the game acts in the correct way and is customisable to the players needs. The game also meets requirements F010 through the use of a Hittable class which continuously calculates hit detection and stops the player from moving outside of the playable area. F011 is also met via the College and Player classes. The college can check if the player is within a certain distance and will call the projectile class to fire at the player when appropriate. The Player and College classes also have functions (NAME) to update the Game state when their respective health reaches 0. For the Player class this update will trigger an end screen (F012, F014) and the college will become 'destroyed' (F013) this may also update the game's objective status. Colleges are also spawned randomly on the map (F016) by the Game class when the game is first started. The game architecture has been designed to be complete single player (NF004) and does not have the capability to access any kind of network (NF011, NF012, A004) this also stops any malicious attempt to alter the game via a network. The objective function controls how long the game is (NF002) as well as how difficult the game is (NF003) by selecting one of the pre-set objective paths. The game has also been designed to be colourful and bright which helps attract our target audience (NF001). This also allows for the game to be user friendly and easy to navigate. (NF005). The colleges, enemy ships, loot and map have distinct colours and shapes, there's also no overlap between red and green usage throughout the game, this allows colour blind people to play with less difficulty (NF006). The game also does not include any flashing images or lights to help reduce the risk of causing a seizure while playing. (NF009).

[COMBINE THIS PARAGRAPH WITH PREVIOUS TO SAVE SPACE AND HELP FLOW?]

In the Players handleInput() function, it needs access to the user's input to deal with movement. It will do this with use of a singleton Input class, which handles detecting user input as per **requirement NF007**, with preset bindings corresponding to specific actions, denoted by strings. This will also act as an InputProcessor for libgdx

The checkCollision() will loop over collision objects, which will be read from the tile map data and compare the passed rect against them, using an external intersection class.

References