CSC3003S Capstone Project: COVID Crossing

Chelsea van Coller
Department of Computer Science
VCLCHE001@myuct.ac.za

Lloyd Everett
Department of Computer Science
EVRLLO001@myuct.ac.za

Lynn Weyn
Department of Computer Science
WYNLYN004@myuct.ac.za

Abstract

We developed an engaging Android game to promote health awareness and socially responsible behaviours in light of the COVID-19 pandemic by following the principles of gamification and behavioural science. The primary game mechanic used is a scoring system that quantifies aspects of positive health behaviour and personal wellbeing as well as the tradeoffs between the two. Players can move through environments and interact with game objects in an RPG-style game where they are presented with narrator dialogue and player choices. Choices made influence player score. Scores are also affected by a randomised quiz as well as a Pac-Man inspired minigame set in a grocery store where the player avoids other NPCs while collecting randomised groceries in the store. A social aspect is introduced through an online leaderboard where friends can compete to obtain the highest score. The final product successfully carried out the key project requirements set out by the client and we were able to deliver a feature-complete Android application.

1. Introduction

We were asked to build an Android game centred around promoting good health practices and social awareness surrounding the coronavirus pandemic. Our client wanted us to use gamification and ideas within behavioural science to make the game engaging and fun while achieving these key educational goals. The client suggested a number of avenues to achieve gamification such as a scoring system for personal wellness and community responsibility and online leaderboards or other social features. Though we were given many ideas for game elements and mechanics, the design of the game was mostly open-ended and we were expected to exercise creativity in solving these problems while continuously obtaining client feedback regarding our ideas. The target demographic was flexible; we chose to focus on children between the ages of 9 and 15.

Establishing how we would achieve the educational goals while keeping the game engaging and fun was one of the central challenges of the project. We eventually chose to create a dialogue-focused RPG-style game with some key goals:

1. The player is able to make choices relating to daily activities and explore a set of environments within the game world.
    a. In the living room environment, the player can move their character, wash their hands, iron their mask and use the telephone.
    b. In the study environment, the player can take a quiz on COVID-19 on the computer.
    c. In the store, the player can move their character around the shop, collect randomised groceries, sanitise hands and avoid other shoppers.
2. The player can view their score, track how their actions affect their score and view their high score.
3. The player is able to compete with other players by means of an online leaderboard that compares their high score to those of other players.

Though we made small iterations on many game elements and features, there were no substantial changes to the overall scope of the project defined above. We delivered all of these objectives in our final build. Figure 1 is a screenshot of the final build of our game as running on an Android emulator.

Figure 1: *Living room scene as of the final build.*

Our overall development approach was to develop the game iteratively. There was a constant focus of keeping a functioning build of the game at all times and we progressively implemented high-priority features into the working build. We re-evaluated task assignments and project direction after implementing major features. Our approach to the software engineering process therefore quite closely matched the Agile methodology. We did however implement an evolutionary prototype after requirements analysis as we were reasonably confident in our overall game structure and we were wary of discarding work under relatively short time constraints. We implemented the game using the Unity engine with scripting in C# and made an effort to follow platform, engine and language conventions wherever possible.

2. Requirements captured

2.1 Initial requirement specification

We have already discussed the two key requirements of the project—educating players on responsible health behaviours and using gamification to make the game enjoyable—but there are many other requirements that were expressed by our client during initial discussions. The requirements we recognised are listed below as functional, non-functional and usability requirements.

Functional requirements

1. The game must have various activities or tasks that the user can perform that educate users of good health practices.
2. The game should implement a scoring feature using a personal and environmental score.
3. The game should incorporate a social element that uses that competition with other players as a means of gamification.

Non-functional requirements

1. The app size should be as small as possible, but ideally under 75mbs.
2. The game should be designed for Android smartphones and should be playable on lower end phones.
3. The game should work correctly even if there is no network connection (if with reduced functionality).
4. There should be full data transparency and efficiency.
5. The graphics should be of sound quality.

Usability requirements

1. It should be possible to discover game functionality without consulting a manual.
2. The game should be accessible to a younger age group (ages 9–15).
3. The game should encourage exploration without the fear of making irreversible actions.

2.2 Analysis and adaptations

We settled on a game design in the style of a dialogue-oriented RPG. This seemed especially suitable as many educational elements could be easily added to the game through dialogue interactions. Beyond that, the sense of direct interaction with the player character that the form of an RPG offers and the ability to move the player character allows for a more interactive experience. After positive feedback from the client we determined this would be the overall form of the game.

We then began to outline the game design in more detail. Our design has a player character exploring and moving around three environments: a living room, a study and a shop. The player is able to interact with objects in the environment by tapping on them. We began to sketch the game environments and produce use cases narratives according to this design. Figure 2 is a sketch we produced during the requirement analysis phase that depicts each scene with its corresponding game objects. Our use case narratives have been included in Appendix B; each is intended to play a part in fulfilling the functional requirements of the game.

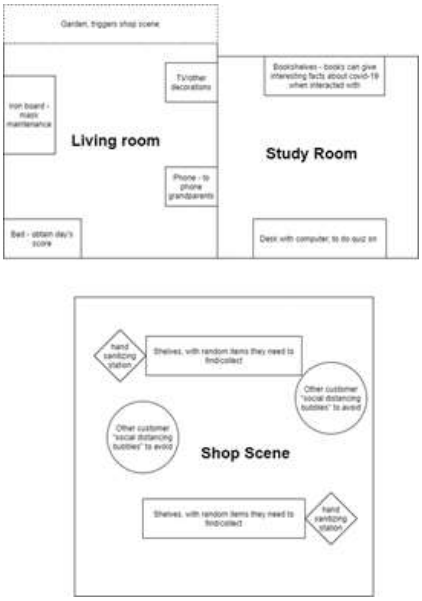


Figure 2: *Sketch of game environments produced during requirements analysis.*

2.3 Artefacts from further analysis

Once we completed the sketch and use case narratives, we began to set out system requirements in greater detail. The state diagram given in Figure 3 illustrates the intended state characteristics of the game at a high level. The analysis class diagram in Figure 4 depicts the overall domain objects as they were understood during requirements analysis and the relationships between them. These diagrams helped us model our specific requirements and gave us a conceptual model that we used during design and implementation.

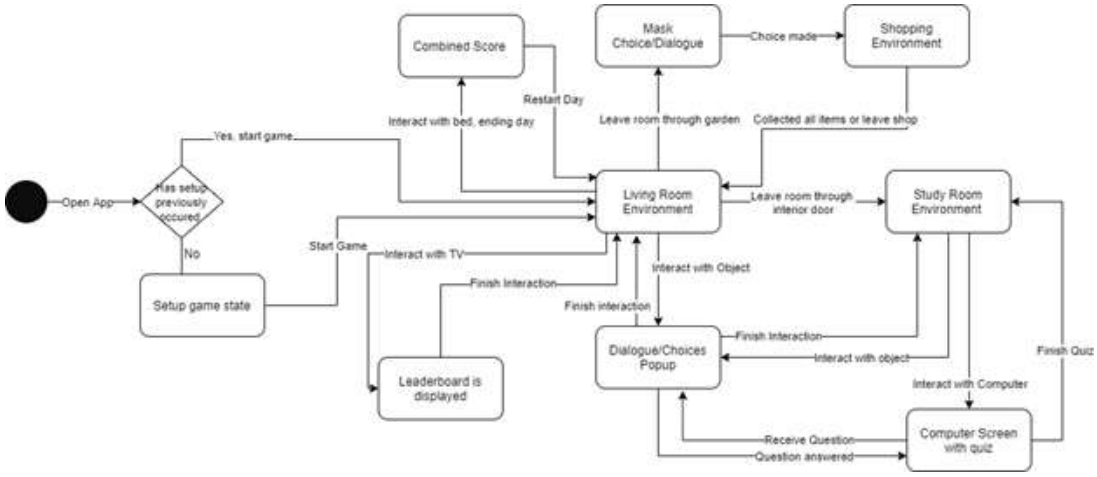


Figure 3: *State diagram depicting game states.*

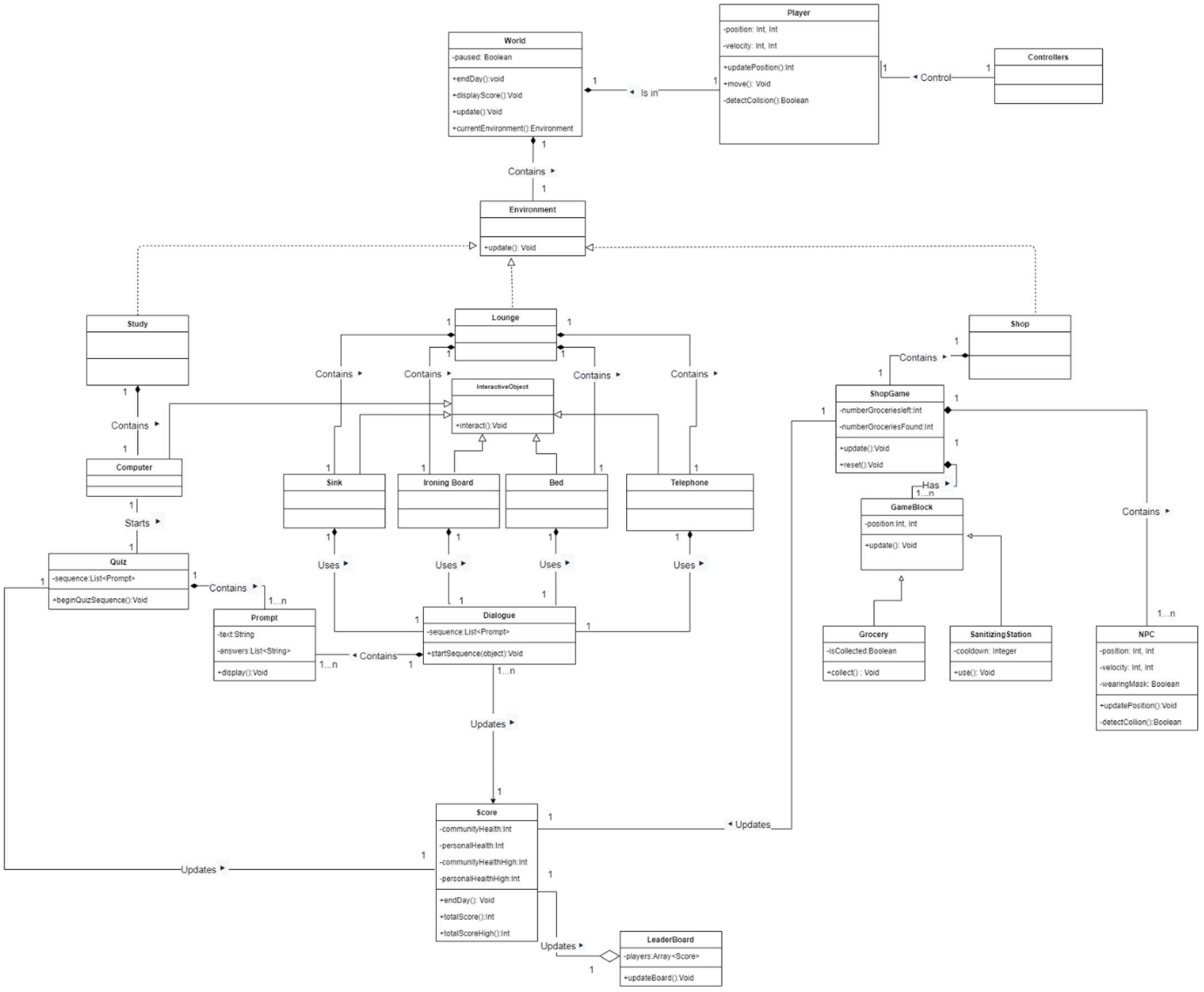


Figure 4: *Analysis class diagram*

3. Design overview

We began the design process by considering the intended behaviour of the high-level domain objects identified during the requirements analysis phase. The player character and elements such as the bed or the telephone are examples of these domain objects. In many cases the responsibilities that these elements would need to carry out are too broad to be expressed in a single class and furthermore their responsibilities need to be divided into reusable parts. As such, we devise a *Game Service Layer* that provides a set of services that game objects can use. Some of these services would still be too complex to implement without an additional layer of abstraction above the runtime libraries and therefore a small number of classes make use of a so-called *Utility Layer*. Finally, as one might expect, all of these layers are quite heavily dependent on the runtime libraries and Unity engine APIs, which we deem the *System Layer*. As such, our system seemed to be described by a layered systems architecture quite naturally once we considered how best to provide the high-level game functionality we identified during the requirements analysis phase. The layered architecture has the benefit of being simple to reason about and it seemed very appropriate in the absence of significant cross-cutting concerns.

Figure 5 summarises our architecture as a layered architecture diagram. Classes and services are not depicted exhaustively as there are very many of them especially in the higher layers. However, the diagram covers some of the most important class components and gives a brief description of the contents of each layer.

It may be helpful to think of this architecture in terms of examples. Consider how we implement the interaction with the ironing board. Dialogue is provided as a service from the layer underneath and uses the same implementation across all game objects, so all the ironing board interaction needs to do is pass on the dialogue tree to the dialogue service. The service takes care of displaying the dialogue and handling user interactions. The ironing board is notified of player responses that it is interested in through the use of callbacks. In this example, the Utility Layer does not play any role. We can also think about how the leaderboard interaction with the TV is implemented. The TV interaction class uses the dialogue service to ask whether the player wants to connect to the online leaderboard. If they choose 'yes', the class requests that the leaderboard service display the leaderboard. The leaderboard is too complex to implement as a single unit, so the leaderboard service relies on a static utility class that handles all of the network interactions with the leaderboard server and presents such functionality in terms of domain objects.

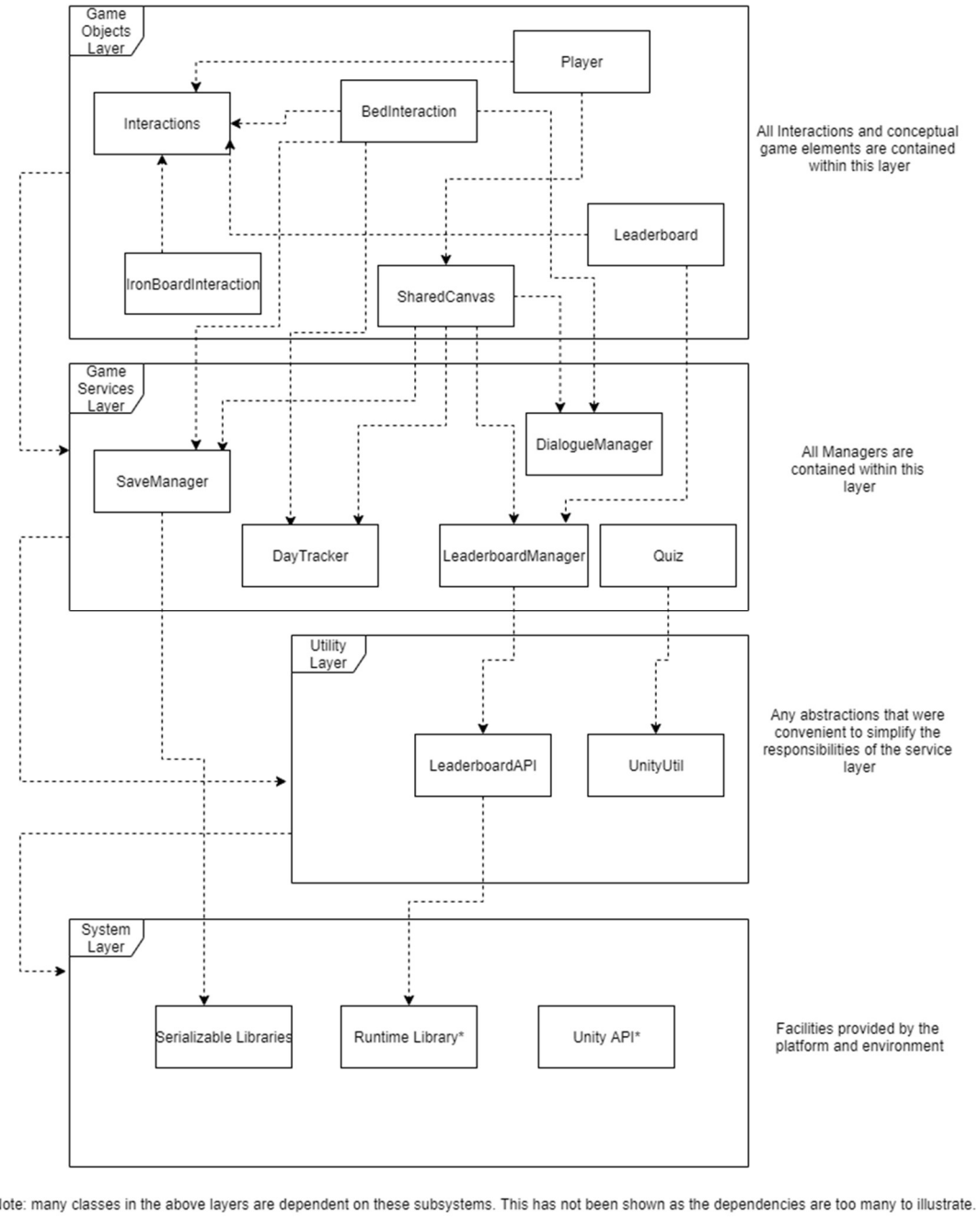


Figure 5: *Layered architecture diagram for the system.*

3.1 Model–view separation

Typically, the service layer is responsible for handling the visual display of UI elements. Likewise, game objects are responsible for their visual display. Unity seems to discourage patterns like MVC that attempt to separate the view layer from the model layer. This is perhaps because within

games there is typically a very close connection between the display of an object and its internal logic. For our application, keeping a strong architectural division between model logic and the visual layers certainly seemed to complicate the code rather than simplify it. It also seemed like we were going against the grain in terms of platform conventions and expectations. This seems to be in contrast to what we would expect from a business application where separation between the view and business logic is important and helps with maintainability and legibility. That said, we tried to remain aware of coupling and where it was harmful and implemented model and view separation wherever helpful (e.g. dialogue trees are always expressed completely independent of their visual representation).

3.2 Algorithms and data organisation

As the project is not data-oriented, our implementation made little use of algorithms in the traditional sense. We of course used basic algorithms throughout the codebase (detailed in section 4), but the Unity engine takes care of rendering and other aspects of the game that might be seen as algorithmically interesting. Likewise, none of the data organisation patterns were particularly notable; data was organised using ordinary structure composition and defined according to class responsibilities.

4. Implementation

Our game was implemented using the Unity game engine. Many classes are automatically inherited from Unity's `MonoBehaviour` class. As the name suggests, such classes each implement an individual 'behaviour' within the game. Subclasses override a set of methods which enable them to implement behaviours:

- `Start()` - called at the beginning of a scene
- `Update()` - called once per frame
- `FixedUpdate()` - similar to update but called at a fixed rate and not dependent to a frame rate
- `OnMouseDown()`, etc. - used to catch input and other events during gameplay

Many attributes had to be marked public, which breaks encapsulation, though this is necessary to utilize the Unity inspector and place references to `GameObject`s in the Unity scene.

Canvas and Managers

The `SharedCanvas` is a class that is shared between every scene in the game. It contains various managers that need to be accessed in every room and thus use the shared canvas to share information about various classes. It contains the `ScoreManager`, `DialogueManager`, `ButtonManager`, `SaveManager`, `SetupManager`, `LeaderboardManager` and `DayTracker`. Although, this is a fairly large amount of responsibility for one class it is necessary to track the various changes between scenes and this is the only `GameObject` that is available in each scene. The various managers will be discussed more in depth within the various classes they correspond to.

Movement

In order to move the player character around the various scenes, there are 4 responsible classes. `AbstractMovement` moves sprite across the screen, sets animation variables, and puts the mask in the correct direction. The most notable method is the `Update()` method, which gets the movement vector from its child class, if there is no dialogue or setup happening, and sets animation variables and mask positions accordingly. It is inherited by two classes, namely `PlayerMovement` and `NPCMovement`.

`PlayerMovement` manages movement specific to the player. Its `GetMovementVector()` method is responsible for taking input from the keyboard or from the buttons. The input can move the player up, down, left or right, however the player cannot move diagonally. The `ButtonManager` checks which buttons are being pressed, and can reset all buttons. It contains 4 arrows of type `ButtonBehaviour`. `ButtonBehaviour` implements the `IPointerUpHandler` and `IPointerDownHandler` interfaces to detect whether the user has pressed the button or stopped pressing the button.

The `NPCMovement` class inherits from the `AbstractMovement` class and is responsible for the movement of the NPC's. It controls their various directions and durations, which are chosen at random. It also does an extra check to determine whether the NPC is stuck and calls a new update to remove it from that position. The `getMoveSpeed()` method determines the speed and it is noted that the NPC's move slightly slower than the player in order to give the player a slight advantage.

Dialogue

Figure 6 gives an example of a dialogue tree that would be created when interacting with the sink.

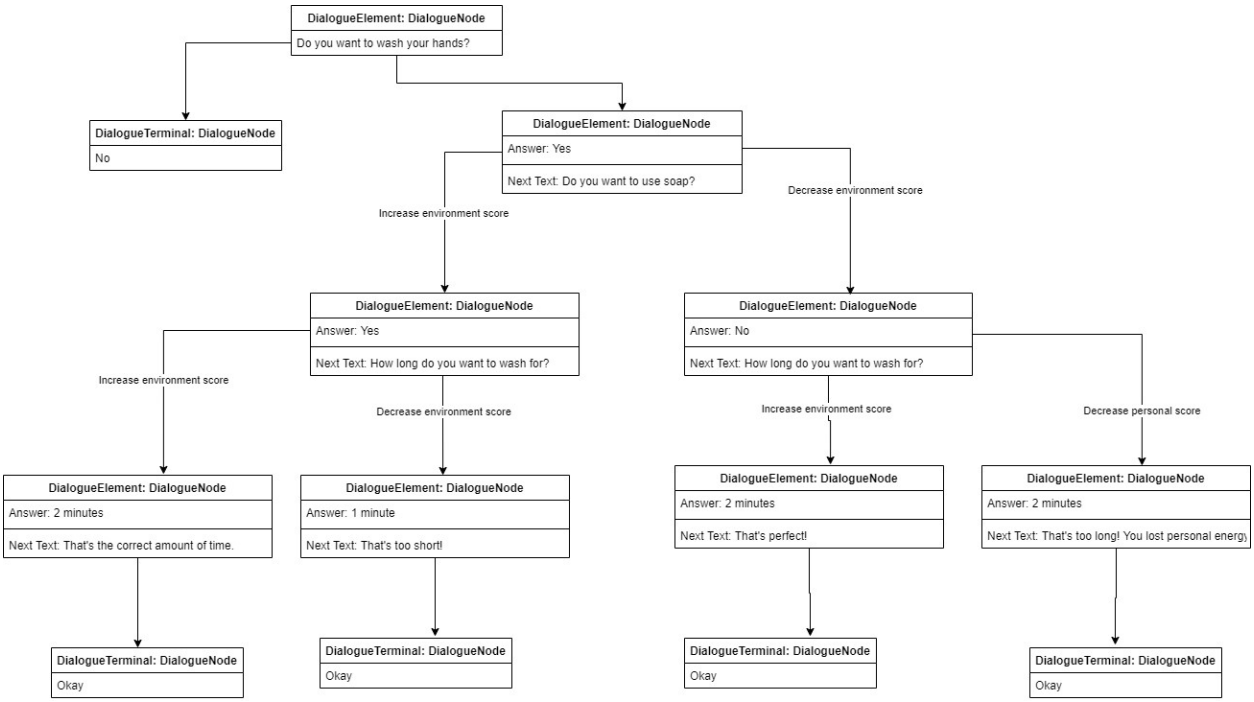


Figure 6: *Example dialogue tree for sink interaction.*

Both `DialogueElement` and `DialogueTerminal` are inherited from `DialogueNode` as this forms the base for the dialogue letting the node know what the title is and whether it has been reached. The `DialogueElement` is a class that contains the list of the children `DialogueNodes` that are to be called subsequently. The `DialogueTerminal` indicates to the `DialogueManager` that it is the last node in the tree and that specific dialogue path has been completed. Both the `DialogueElement` and `DialogueTerminal` have actions that can be called when reached, for example increasing the environmental score or changing the scene. The `DialogueManager` is responsible for managing the dialogue sequence, typing out the characters one by one when the dialogue appears, detecting whether a button has been clicked and how many buttons need to be displayed, as dialogue may have one to four answers. Notable programming techniques were a continued use of lambdas for changing the personal and environmental scores.

Interactions

We created an abstract parent class called *Interaction*, a subclass of `MonoBehaviour`. This is used to represent sprites in the game that can be interacted with for example the phone or ironing board. Subclasses can implement the `Interact()` method to specify what should happen during the interaction. The superclass ensures that the object will glow when the player is sufficiently

near to give visual feedback for an interaction and calls `Interact()` if the player character is near and the object is tapped by the player.

Scoring

Unity has a built in component called "Slider" which represents progress for a given range. We chose the range [0;100], with a starting score of 30. `HealthBar` was created in order to get and set the current value an individual slider should store and represent.

`ScoreManager` class manages and keeps track of two `HealthBar` objects - one for personal and one for community health. It calls the relevant methods to get and set `HealthBar` scores, can reset them and calculate a final score (average of the two scores).

Leaderboard

The `LeaderboardManager` class is responsible for managing the visual display of the leaderboard and initiating requests to the server and responding to results. The implementation of the requests are done through the API via the POST and PUT methods.

Quiz and Util

The quiz class is responsible for creating dialogue that displays a quiz. The quiz is in charge of reading in the text file under Resources named `QuizQuestions`. The quiz picks five random questions and displays them one after the other, allowing the player to choose the answer and gives feedback whether it was correct or incorrect. If the answer is incorrect, it also provides more feedback indicating why it was incorrect.

Groceries

The `RandomiseGroceries` class is responsible for randomizing groceries in the shop scene and placing them on various shelves and `GroceryInteraction` ensures they vanish when interacted with.

Masks

The `MaskShopActivation` adds a mask to the player in the shop scene if they choose to wear a mask during the `LeaveHomeDoorInteraction`.

NPC's and Player Immunisation

The `NPCBubbleBehaviour` class manages the bubble surrounding NPC. It randomises whether the NPC is wearing a mask and calculates a larger bubble if they are not and calculates the decrement of the player's score if they are not immune. If the player uses the hand sanitizing stations the `PlayerImmunityBehaviour` class activates the player immunity and the NPC's will not affect the player for a few seconds.

Days

The number of days is necessary to track as the player can only interact with an object once a day. The `DayTracker` class simply tracks the day the player is on and increments this when the player interacts with the bed. Most interactable objects have a method called `SetWasUsedToday()` which uses the `DayTracker` to check whether the object has been used that day.

Serialization

The `PlayerRecord` contains various information about the player, namely a unique GUID, nickname and high score. The `SaveManager` then ensures that this data is stored to secondary storage using JSON serialization.

Setup

`SetupManager` runs when starting the game, and checks if a username has previously been set. If setup still needs to be done, it starts up the username screen GUI, which asks for user input for a username and has a confirm button. On pressing the button, `NameTransfer's StoreName()` method is called, which sends the entered username to `SaveManager`'s nickname field and saves it. Should no username have been entered, it saves the name as "Anonymous". The confirm button also calls `SetupManager`'s `FinishUsername`() method, which deactivates the username screen and starts the tutorial. The tutorial is a series of annotated images, cycled through using a "Next" button. The cycling of images is managed by `TutorialManager`. Once the last image has been displayed, `SetupManager` will hide the tutorial, and setup is complete.

Game Scenes

Game scenes were created by placing open source art assets into a palette to draw Unity tilemaps for three scenes, and adding sprites to these scenes.

5. Testing

Our team carefully considered our approach to testing as we established our development workflow. A big concern we had was that the game was to have a great many classes and entities to represent all of the objects and services we needed in order to fulfil the requirements and we were wary that writing automated tests for all of these might be untenable. Our team was under significant time pressure and our development goals were substantial even after reducing the scope according to the most essential objectives of the project. Beyond that, many behaviours would be challenging to test automatically such as visual effects. As such, we are convinced that manual testing, though tedious and intrinsically more susceptible to being neglected, was the best decision for the project. However, if the game were to be developed further with a greater time budget, it would be strongly advisable to use an automated testing framework and develop a set of automated unit and integration tests.

Establishing a comprehensive testing plan was critical to ensuring the correctness and stability of the game and we adopted fairly strict requirements regarding code testing before merging code contributions into the main branch. Table 1 gives a broad overview of our testing methods. These methods seemed to offer the best balance between minimising development overhead and ensuring that the system functions according to expectations. We obtained good test coverage of all system functionality including boundary conditions without compromising key feature goals.

Table 1: *Summary testing plan.*

Process	Technique
1. **Class testing:** test methods and state behaviour of classes.	This took place in the form of white-box testing. We used test objects and classes during development to ensure the correctness of individual classes as we constructed them, but these were not formulated into formal automated tests for the reasons discussed above.
2. **Integration testing:** test the interaction of sets of classes.	Here, we used behavioural testing. Typically, this consisted of running through all of the functionality in the game and exploring all of the interesting edge conditions we had identified (these are listed in a table below). This was the test that we conducted most frequently as it was straightforward to carry out by running the game. More intensive testing occurred at significant development milestones and after the implementation of major features.
3. **Validation testing:** test whether customer requirements are satisfied.	We performed validation testing by conducting demonstrations of our system for the client at different

	development milestones. This can be seen as acceptance testing. We slowed the implementation of new features prior to these demonstrations and focused on thorough correctness testing (i.e. class testing and integration testing). These demonstrations were helpful to obtain feedback and determine if we needed to make small changes or reconsider parts of the system.
4. **System testing:** test the behaviour of the system as part of a larger environment.	Here, we tested that the game behaved correctly with respect to its external environment. This consisted of tests such as making sure that the game behaves correctly in terms of the Android lifecycle and ensuring the system works correctly when it connects to the leaderboard server. We also tested performance and made sure that the app binary satisfied size requirements.

Table 2 lists each of the test cases we used during integration testing. These test cases are intended to map to every nontrivial piece of functionality in the game. Table 3 identifies each of the system test cases.

Table 2: *Test cases used for integration testing.*

Functionality	**Expected behaviour**
Ensuring that clicking "wear mask" when exiting the living room, results in the player wearing a mask.	The player wears a mask in the shop scene.
Ensuring objects can only be interacted with once a day.	The player can only play the quiz once a day.
The player comes into contact with the NPC's bubble in the shop scene.	The player's personal and environmental score rapidly decreases.
Player presses arrow buttons on screen.	Character moves and is animated, always facing the correct direction. The player character collides with the various objects in the room.
Player interacts with objects, reads dialogue and makes choices.	Dialogue flows per script and personal and community health bar are appropriately updated according to choices
The player interacts with the TV to view the leaderboard.	The leaderboard is displayed with the player's high score highlighted in blue.
Player interacts with the bed in order to end the day.	Day starts anew with the leaderboard updated with player rankings and high scores. The scores are reset.

Player interacts with the hand sanitizer stations.	The player receives temporary immunity from the NPC's.
Interacting with groceries in the shop.	Increases personal score and makes the groceries disappear.
Server leaderboard API testing with Postman	The server correctly responds to API requests

Table 3: *Test cases used for system testing.*

Functionality	**Expected behaviour**
Build size is accessible.	The build APK is smaller than the 75mb requirement.
Network behaviour (once server was running).	The player is able to request the leaderboard to see where they rank.
UI elements and scenes fit on different display resolutions.	They appropriately adjust to Android phone screens and lower end models.
Player closes and reopens the application (with or without killing the app in-between).	All game state and scores should be preserved after the game is reopened.

These testing methods were often tedious and required discipline during development; however, we ultimately obtained comprehensive feature and code coverage and consider the system thoroughly tested and believe it will perform correctly. Beyond ensuring basic program correctness, the validation testing process indicates that the final game accomplishes the core requirements—that is, we have successfully delivered on the promise of an COVID-19-themed educational game that makes good use of gamification techniques.

5.2 Programming style

Generally, we try to throw exceptions when important preconditions in the code have been violated, especially at the start of methods. This was helpful to ensure that we could detect and diagnose issues at the point they occur as opposed having to find the origin of strange side effects or exceptions thrown later on. We believe this was an effective step towards achieving correctness though of course not all preconditions can be validated in a straightforward manner.

6. Conclusion

Validation testing with the client indicates that the main objective to develop a fully functional educational game about COVID-19 using gamification was successfully met. The game offers a unique, fun experience that educates users on the various health practices that should be

implemented during the pandemic. Our use of software engineering principles served us well and ensured we were able to deliver the final product within the allotted time.

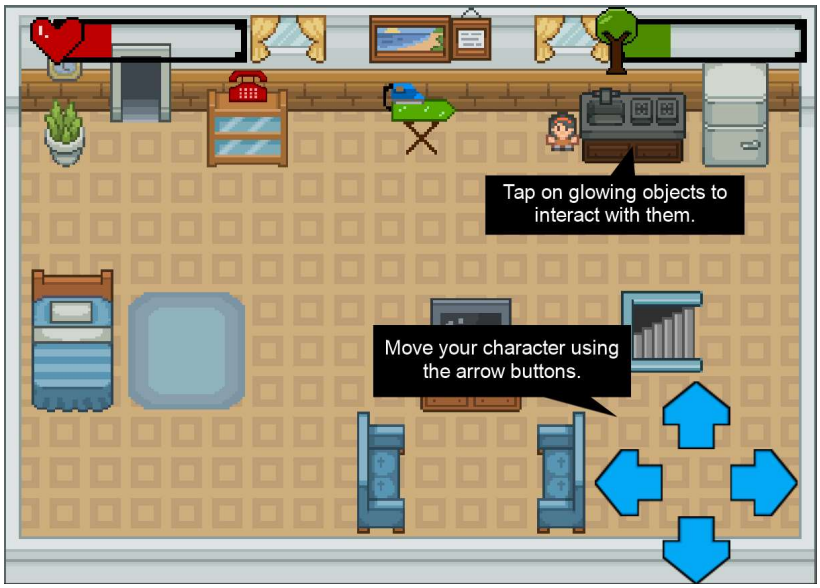
Going back to our original goals, we successfully met each of the key requirements:

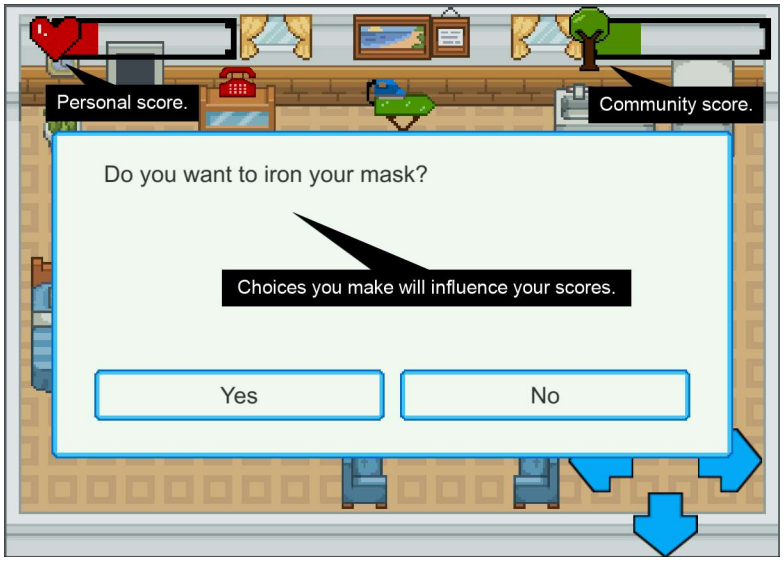
1. The player is able to make choices relating to daily activities and explore a set of environments within the game world.
2. The player can view their score, track how their actions affect their score and view their high score.
3. The player should be able to compete with other players by means of an online leaderboard that compares their high score to those of other players.

Our use of a layered architecture design made it easy to reason about how each layer would function and also allowed for consistency and flexibility throughout the software development process. All aspects of the game were thoroughly tested using a number of testing methods such that overall, we are confident in the correctness and suitability of the system

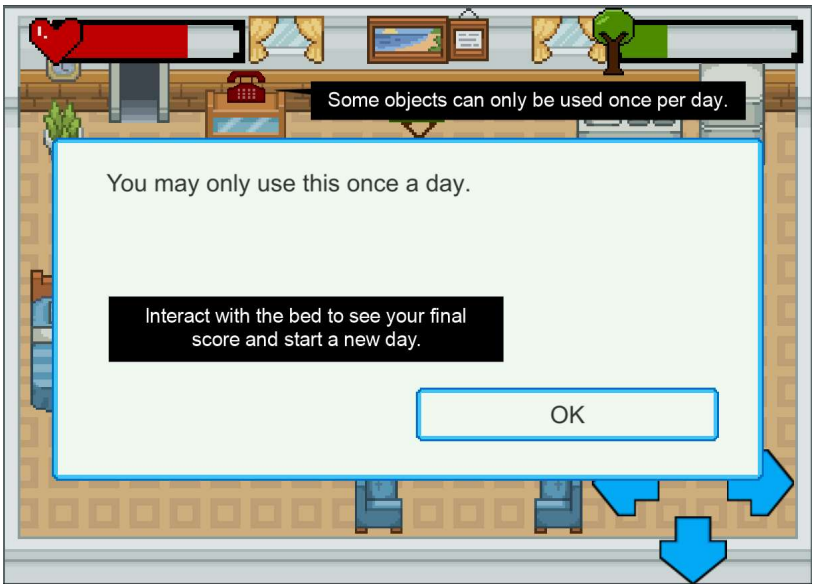
7. User manual

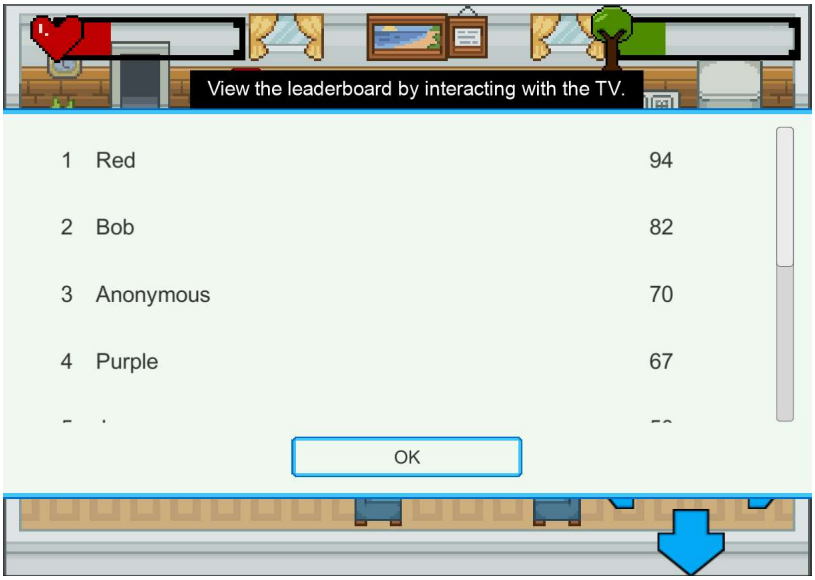
The game is intended to be playable without consulting any additional material. A tutorial is displayed to the player on first run as a series of annotated images explaining how game mechanics work. Additionally, we consider exploring and experimenting with objects throughout the game part of the fun. We have included the annotated tutorial images below.


Tap on glowing objects to interact with them.
Move your character using the arrow buttons.


Personal score.
Community score.
Do you want to iron your mask?
Choices you make will influence your scores.
Yes
No


Collect groceries in the shop.
Use sanitising stations to briefly get immunity from shoppers.
While maintaining social distancing.





Appendix A: Additional details

A.1 Build size

We managed to satisfy the build size requirement. The eventual APK was approximately 17 MB.

A.2 Play store upload

We went through all of the steps to ensure we had a valid build that could be uploaded to the play store. Unfortunately, Google is not currently allowing apps on the store related to COVID-19 without approval from the government or a public health entity (even just for internal testing).

A.3 Version control

Unity has built-in cloud enabled Collaborate features which we used extensively for source control due to the stability issues with the departmental GitLab server.

A.4 Leaderboard server implementation

The leaderboard server implementation has not been discussed in this report for the sake of brevity. It is an ASP.NET core application that uses the MVC model and EntityFrameworkCore to access a SQLite leaderboard database. The leaderboard controller class responds to client requests and mediates access to the leaderboard database, which is represented using a leaderboard context class. The code is generally very simple and has been documented.

A.4 Attribution

We are extremely grateful to the following artists who have kindly provided their work under a permissive and free license:

- The Tile Set - “Free tileset compilation 2014”, by PureAzuure, from https://www.deviantart.com/pureazuure/art/Free-tileset-compilation-2014-624467061
- The Character Art - “RPG Urban Pack”, by Kenney, from https://www.kenney.nl/assets/rpg-urban-pack

Appendix B: Use case narratives

Move around in environments within the game world.
Actor: Player

Four arrow controls are always displayed on the screen which allow the player to move in four directions. This allows them to move around within their current environment. The environment is displayed to the player top-down in the typical style of an RPG. When moving, the player sprite will animate to walk in the corresponding direction. Collisions with obstacles and walls will stop player motion.

Interact with features of the environment and observe changes in scores.
Actor: Player

When the player moves near an object in the environment that they are able to interact with, the object will glow, giving the player visual feedback for the availability of an interaction. If the user taps on the object, a dialogue will begin. Text is displayed to the player and they are asked to select one of several responses. Picking a response will either continue the dialogue (presenting more information and choices) or end the dialogue, depending on the response. Dialogues vary between different objects, but the essential structure is the same text-oriented experience.

An important component of these interactions is the way they affect player scores. As the game is played, a *personal comfort and wellbeing score* as well as a *community score* are presented to the player. Additionally, these two scores are combined for an *overall player score* at the end of the day. Typically, the interactions and dialogue sequences described above will either positively or negatively affect each of the two scores. The player is able to see the effect of their actions on their scores as visual feedback is provided as the score changes: changes in score appear directly on the screen for a short moment beneath the player scores.

Play the grocery store minigame.
Actor: Player

As the player enters the grocery store environment, they will see a maze-like organisation of aisles, with a random amount of groceries present in the store. The player can use the movement controls to obtain groceries along the aisles while avoiding other shoppers (similar to Pac-Man, for example). Entering close proximity with other shoppers will decrease the player's scores and collecting groceries will positively affect the personal wellbeing score. The player is also able to sanitize at hand sanitiser stations, giving them 'immunity' from other shoppers for a few seconds. Exiting the shop ends the minigame.

Take quizzes in the study environment.
Actor: Player

When the player interacts with the computer in the study environment, they will be asked whether they would like to begin a quiz. Questions will be presented to the player about good health practices and COVID-19. The question answers are multiple choice. Answering correctly increases community and individual scores, while answering incorrectly decreases these scores. Attempting to take the quiz more than once per day will result in an error message (i.e. when the player interacts with the computer).

Explore the home living room environment.
Actor: Player

The player can enter the living room and browse a number of objects with which the player can interact:

- If the player interacts with an ironing board, a dialogue will show asking if they want to iron their mask. If they say yes, the player's community score will increase but their individual points will decrease.
- If the player interacts with the kitchen sink, a dialogue will show asking if they want to wash their hands, with soap, and for how long. Handwashing for a long duration increases community points more but also slightly decreases individual points.
- If the player interacts with the telephone, they will be presented with the option to call friends and grandparents. This will allow them to have a basic conversation with whomever they choose to call. The player may be presented with the option to suggest meeting in person, but doing so fails and reduces community and personal points.
- The bed interaction presents a dialogue asking if the player wants to sleep. If so, the day ends and the overall score for the day is displayed.

Additionally, as the player leaves their home to go to the grocery store, they are asked questions on how to properly wear their mask. Depending on their answers, community and individual points are affected.

Keep a score on the leaderboard.
Actor: Player

When the game is opened for the first time on a device, the player will be asked for a nickname to be used on the leaderboard. No given username will set their name to "Anonymous". When in-game, interacting with the TV brings up a leaderboard where the player's high score and corresponding ranking will be displayed next to players with a similar ranking.