CS3217

Dash

Final Report

Name	Matric Number
Ang Jie Liang	A0149293W
Tan Yi Jing, Jolyn	A0143232W
Ang Yee Chin	A0161374H

Overview

Dash is a *progressive endless runner game* where the goal of the player is to travel as far as possible while staying within two borders - located at the top and bottom of the screen. The player-character automatically moves to the right of the screen through procedurally generated landscapes. Users can choose three modes of controlling the character: arrow, flappy and glide, each with their own unique gameplay and challenges. Obstacles, power ups and coins can be encountered and collected throughout the game which makes it more progressive, interesting and challenging. Throughout the game, as the player progresses, the stages will get harder with new obstacles and harder challenges. Users can also enter shadow play mode, and compete with his/her previous performance. Multiplayer mode is available, where users compete with each other real time. A global high score is included to display the top 10 scores achieved for the the three different game mode.

Gameplay

- The player character moves automatically towards the right of the screen through procedurally generated landscapes and obstacles
- The player can select three ways to control their player: Arrow, Flappy, Glide *(see Revised Specification: Controls)*
- Throughout the endless gameplay, the player character will advance to new phases after certain distances which then introduces a new extension to gameplay, this could include new kind of obstacles (moving obstacles), new power ups (dash forward, ghost mode, shrink) and tougher missions.
- The player loses on collision with obstacles or wall, or exceeds the screen boundary.

Additional Features

- Multiplayer mode: The player can choose to play individually or with friends using the multiplayer mode.
- Shadow Play: The player can choose to play with the previous gameplay of him/herself after the game is over.
- Missions: The player can complete and unlock missions (distance travelled, numbers of powerup collected, number of coins collected).
- High Score: The player can submit the score to global high score that share among different devices.

Similar Applications

While there are many existing 2D endless runner games, the existing games either consist of simple mechanics, or are stage-based, making them not truly endless as stages are pre-designed. For instance, “Flappy Bird” consists of only pipes, making the obstacle generation simple by adjusting the pipe gap placements. “Geometry Dash” is an action platformer with many variations of obstacles and game modes. However, stages are pre-designed and users have limited stages to play with. “Jetpack Joyride” is another endless runner game that does not guarantee that the players have a playable path, but instead provide “shields” through power ups which allow players to continue playing after colliding with obstacles. Most of the existing apps are single player based.

What makes our application special?

Many current endless 2D runner games are not endless, meaning that the users will be going through the same pre-designed levels and can be finished with the game easily once all levels are completed. We design our game to be truly endless to provide a seamless and enjoyable gaming experience, whereby the gameplay can potentially go on forever, and will be different every time.

Existing endless runner games which involve procedural generated maps consists of limited and simple types of obstacles, and only support one kind of player control. Our game includes a parameter-based procedural generator which generates floating and moving obstacles, and builds a continuous wall boundaries which are rare in runner games. Our game supports three unique ways of controlling the player-character, and guarantees a playable path for the different controls, making the game truly endless.

Our game also supports multiplayer and shadow play mode, allowing players to share their game experience and compete with other users, and better themselves.

Why our app?

We want to create a game which allow us to express our creativity and allow us to introduce new twists. We feel that we could improve the endless runner concept by designing our own procedural generator to make our game complex but still at the same time truly endless and playable, as opposed to pre-designed levels. We also include a multiplayer mode along with shadow play and extra missions to increase the social and fun aspect of the game. We also plan to include different ways to control the player character to introduce variations.

Revised Specification

Controls

Player can select different controls at the beginning of the game. The map and obstacle generation ensures that the game is playable and there will always be a possible path through the endless game for the different types of control. We finalised the available controls to the following three:

- Control 1: Arrow
  - Overview: Moves (vertically and horizontally) at constant velocity.
  - Control: Tap to change direction (up or down).
- Control 2: Flappy
  - Overview: Constant horizontal velocity but affected by gravity
  - Control: Tap to fly (Small force towards the opposite direction of gravity) Hold does tap action once.
- Control 3: Glide
  - Overview: Tap or hold
  - Control: Tap and hold to accelerate upwards.

Power Up

Throughout the game, the player can collect powerups. The power ups would aid the player in advancing further in the game. Final implemented powerups are:

- Dash: Move forward by certain distance.
- Ghost: Can pass through obstacles, but not walls.
- Shrink: Shrink down the size of the player.

Coin

We added coins to the game, allowing users to collect coins throughout the game, and complete coin-related missions. Locations of coins are guaranteed to be reachable by the player.

Multiplayer

Upon starting of the game at menu scene, the user is able to select between multiplayer or single player mode. When the multiplayer mode is chosen, the user can choose to host or join a room to play with friends. A seed will be generated for deterministic generation of map across different devices. Currently, there is no max number of players in a game, but it is best to maintain the number below 8 for better visual. The other players will be at a certain distance behind. It will be using Firebase to handle the network connectivity.

Shadow Play

When the game ends in single player mode, the user will be prompted to play again with his shadow to improve his/her skills.

Missions

During the game, players can complete three types of missions:

- Distance: Travelling a minimum distance
- Powerups: Consuming a certain number of powerups
- Coins: Collecting a certain number of coins

Since this is an endless runner game, the missions will be infinite by following a formula for the generation of next checkpoint (e.g. next checkpoint for the distance mission is met at the next 500m travelled). When a player has completed a mission, the player will see a popup text on the screen stating the mission that has just been completed. They will also be able to view the upcoming missions to complete in the game menu. To record the latest completed checkpoints for each mission, the game will keep a local storage of the last completed checkpoints.

High Score

To make the game more competitive and fun, we decided to add a high score feature for that works globally. The high score for different controls will be different.

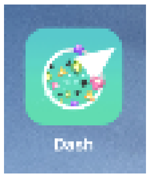
User Manual

Platform and Requirements

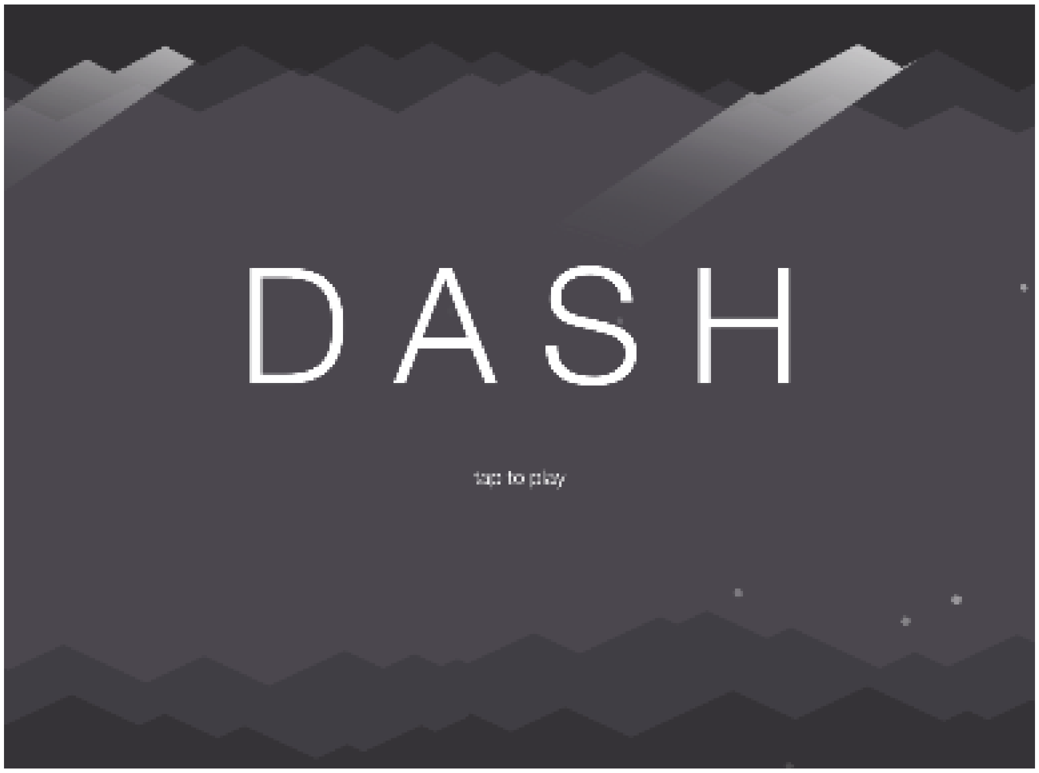
Dash is an iOS native app built for the iPad. An internet connection is required for access to multiplayer features.

Launching the App

From the iPad home screen, tap on the *Dash* icon below to launch the app.



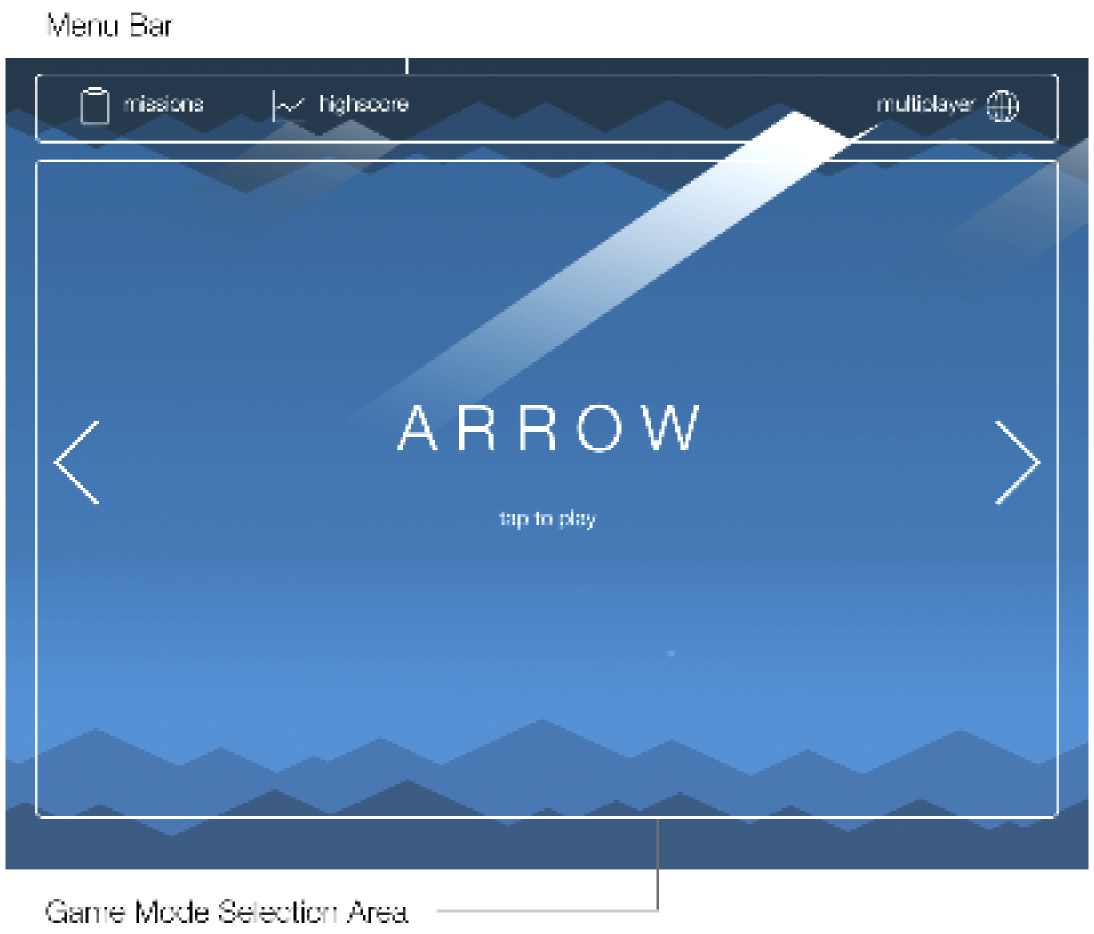
Upon launching *Dash*, you should see the application start screen:



Tap anywhere to proceed to the main menu.

Main Menu

The following image shows the main menu of the application.



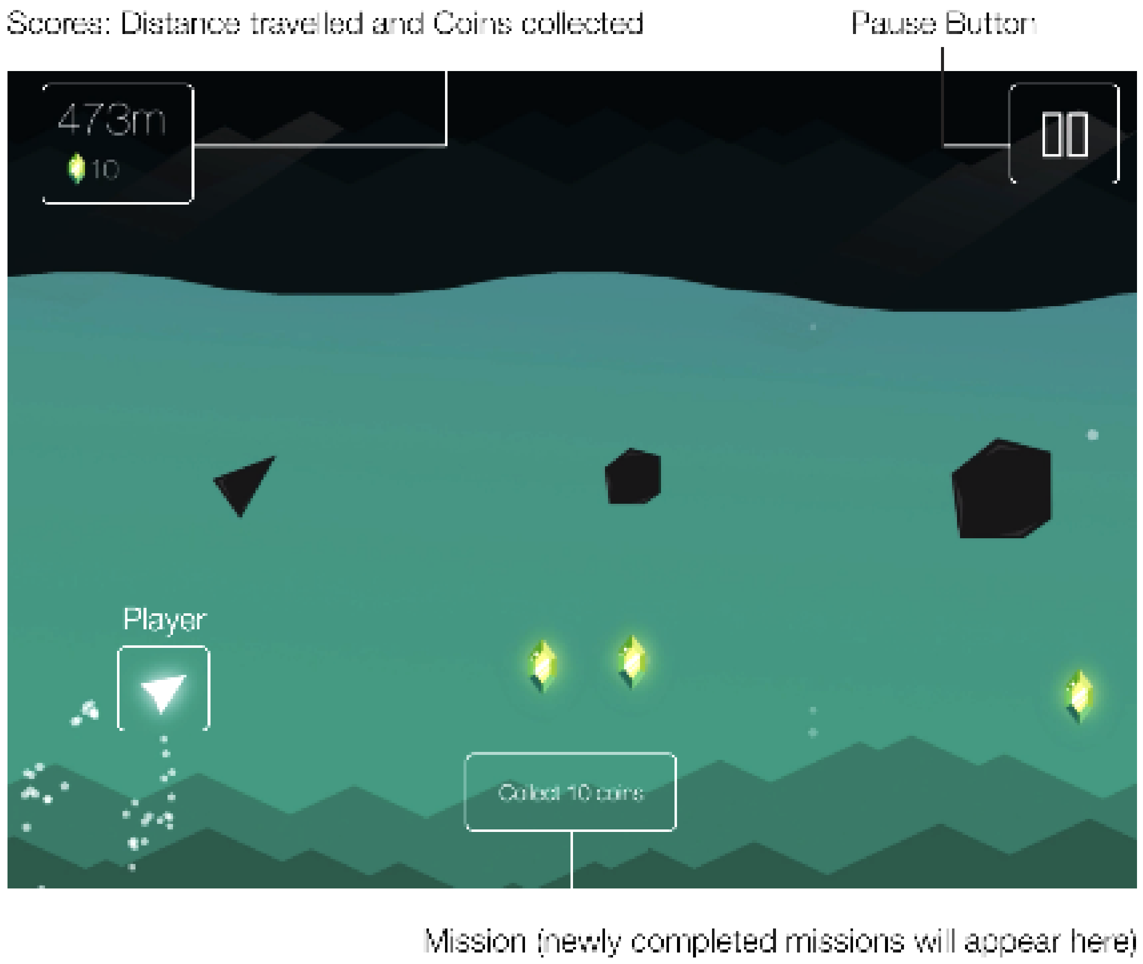
- Game Mode Selection Area
    - Press the left and right arrow keys to select a different game mode
    - Tap anywhere in the area to start the game with the displayed game mode
- Menu Bar
    - Missions: Shows the upcoming missions to be completed
    - Highscore: Shows the high score list for the displayed game mode
    - Multiplayer: Enters the multiplayer lobby

More details for the options in the Menu Bar can be found in the next section, Gameplay.

Gameplay

Interface

The following figure shows the important components in the game interface.


Scores: Distance travelled and Coins collected
Pause Button
473m
10
Player
Collect 10 coins
Mission (newly completed missions will appear here)

Game Rules

- The aim of the game is to travel as far as possible without colliding into obstacles.
- The difficulty of the game will increase the longer you play, with faster speed and more challenging obstacles.
- There are a few types of obstacles to look out for:
    - Walls: Located at the top and bottom of the screen.
    - Boulders: Appear in between the walls.
    - Missiles: Appear at longer distances. They will move in the opposite direction of the player.
- The game ends upon collision with any obstacle, wall or player is out of bound.

Game Modes and Controls

There are three playable game modes with different controls types. They can be selected from the main menu.

- Arrow: Tap to change direction
- Flappy: Tap to fly up once
- Glide: Tap and hold to accelerate upwards

Power Ups and Coins

The following are some collectable power ups which may be helpful to the player during the game.

Dash

Speeds up the player movement temporarily. During a dash, the player can pass through obstacles.

Ghost

Allows the player to pass through obstacles temporarily.

Shrink

Decreases the size of the player temporarily.

The player can also collect coins to complete missions (more on missions in the next section).

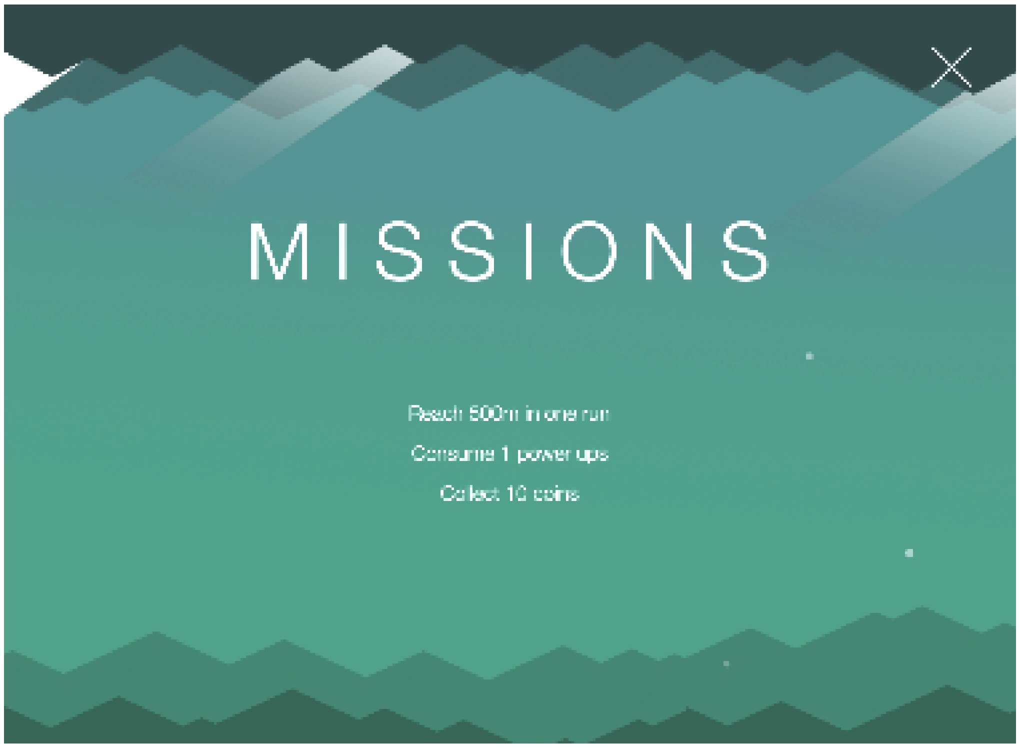
Coin

Missions

There are three types of missions which can be completed:

- Distance: Travelling a minimum distance
- Powerups: Consuming a certain number of powerups
- Coins: Collecting a certain number of coins

The upcoming missions can be checked by selecting the “Missions” option from the main menu. The Missions screen is as shown below:



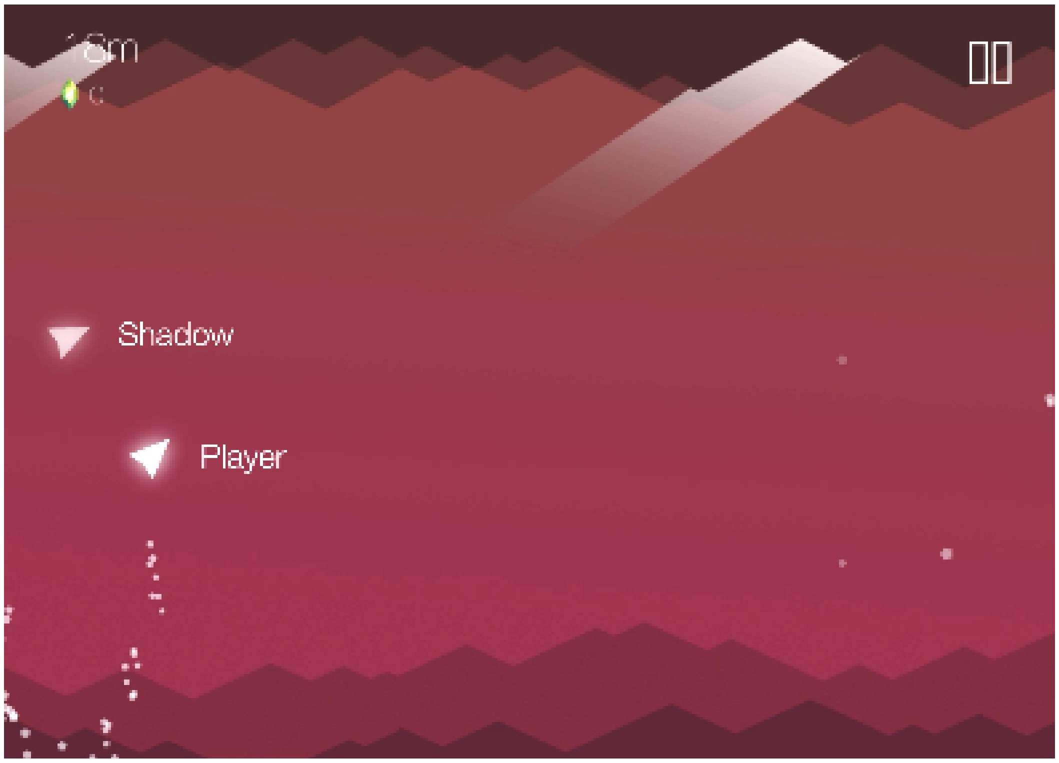
Reach 500m in one run

Consume 1 power ups

Collect 10 coins

Shadow Play

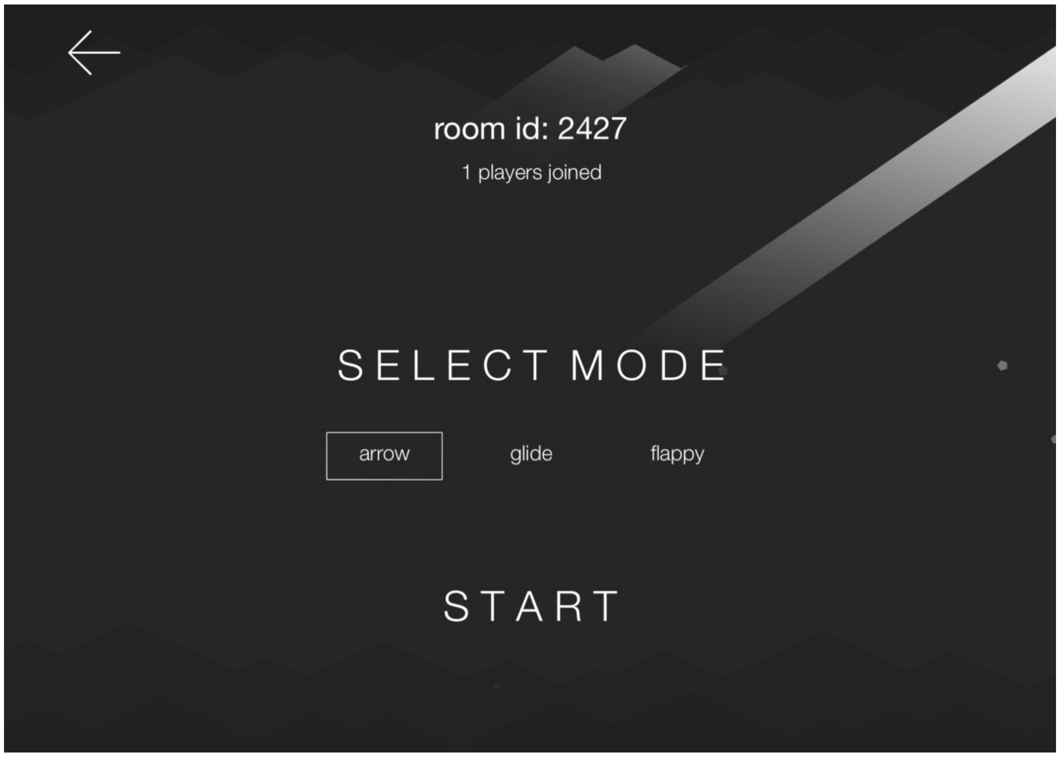
After a game has ended, the player can select the “Shadow Play” mode where the game will start again, displaying the player from the previous run. The map will be the same so the player can compete against the score from the previous run.



Multiplayer

From the main menu, the player can select the “Multiplayer” option to play with other players. The selection brings the player to the lobby screen with two options: “Host Game” and “Join Game”.

A multiplayer game requires a player to host the game. Selecting “Host Game” brings the player to the following screen:

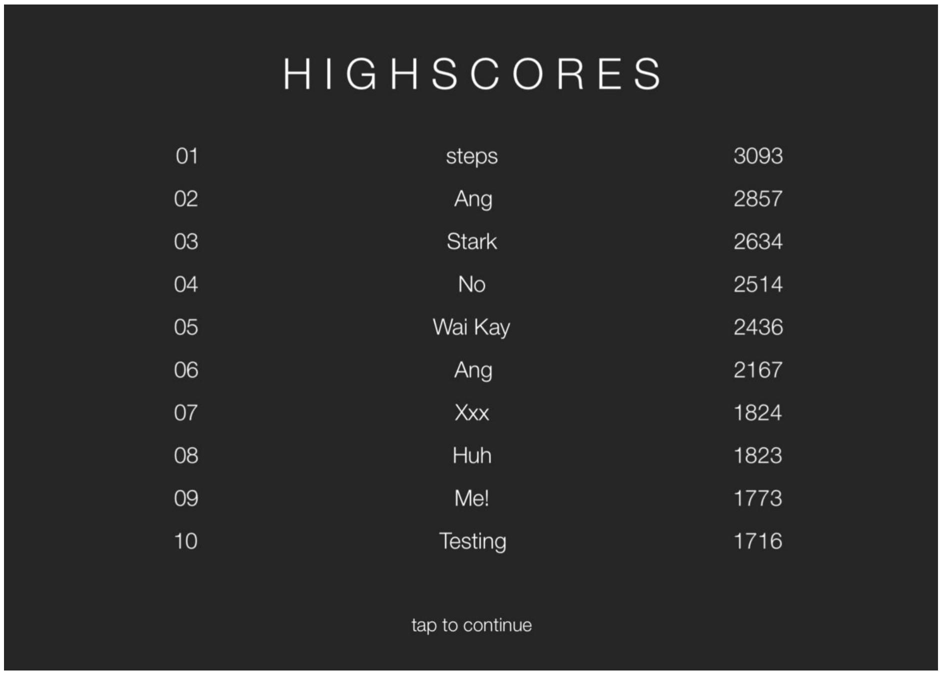


The host can select the game mode to play and start the game when all players are ready.

Other players can then join the game by entering the room ID as shown on the host player's screen. When a valid room ID is entered, the player will enter the room and is shown the same screen as the host's screen. The player is able to see the game mode selection made by the host. Note that only the host is able to start the game.

High Score

If the player has obtained a score in the top ten places, the player can enter their high score to join the global leaderboard. The high scores are kept separately for the different game modes. The high scores will be shown after every game, or can be accessed from the "High Score" option in the main menu:



HIGHSCORES

01	steps	3093
02	Ang	2857
03	Stark	2634
04	No	2514
05	Wai Kay	2436
06	Ang	2167
07	Xxx	1824
08	Huh	1823
09	Me!	1773
10	Testing	1716

tap to continue

Performances

With basic rectangular shapes to represent the obstacles, power ups and coins, we are able to achieve a stable 60fps at all times. After adding SKTextures to the SKSpirteNodes for visual and aesthetics reasons, the fps averages at 58 fps throughout the game.

Number of nodes in game is minimised and maintained from 20-30 nodes throughout the endless game. Memory usage is also optimised and maintained at around 165MB throughout the whole game, and CPU usage averaging at 40%. Memory usage reaches a maximum of 225MB when high score is displayed at the end of game.

Testing

Testing Strategy

Black-Box Tests

Refer to *Appendix*.

View and Controller are highly reliant of GUI and thus are conducted based on black-box testing. As views do not have much logic involved, if the Model and Logic are tested well, black-box testing should suffice. The testing can be conducted by navigating through all options in the main menu to ensure correctness of application flow.

Gameplay related black-box testing is categorised into two sections, one on the content generation, another on player controls.

As the Model and Logic are designed to be reusable for all three different kind of player control, content generation on all three stages are tested together on 10x the speed, to test that rendered nodes are valid. Player controls are tested separately, and ensures that user input handling is conducted correctly.

Once all the testings of the stages (e.g. obstacles, user input handling, player character, obstacle generation, collision detection) are completed, all stages are integrated together and an integrated testing is then conducted.

Networking related modules are also under black-box testing, as quantitative results are not necessary.

Unit Tests

Refer to *Appendix*.

Due to time constraints, we did not implement unit tests. Given sufficient time, we plan to write unit tests for Logic modules such as GameEngine, PowerUpGenerator, ObstacleGenerator and WallGenerator, and Model modules such as Player, Wall, Obstacle, etc.

The correctness of Logic modules can be ensured by feeding mock values and model objects into the modules and comparing the state of the models against the expected state in certain conditions. For example, the expected result for the generators is that the Obstacle and Wall should not overlap with each other. This also ensures methods used in runtime structures for the game such as Path and Point are valid.

Unit testing on Models ensures that the representation invariants are not violated, as well as correct implementation of computed properties and functions.

Stress Tests

Since we are building a endless runner game, it is important to ensure that the game is really endless and also smooth. One way we will be doing it is increase the speed by a certain factor and ensure that the content generation still works after disabling the player node. This allows the test of the generator to make sure it is not interrupted due to wrong triggering mechanism.

Performance Tests

SpriteKit comes with an option to display the node count and fps of the game scene, which we can use to measure the performance of the game during black-box testing. We aim to have as little node count as possible and a consistent fps of around 60 throughout the gameplay.

Regression Tests

We will run all unit tests and do black-box testing after every update to the game. We will do stress test and performance test after every milestone (see *Appendix*) when important parts of the game have been integrated together.

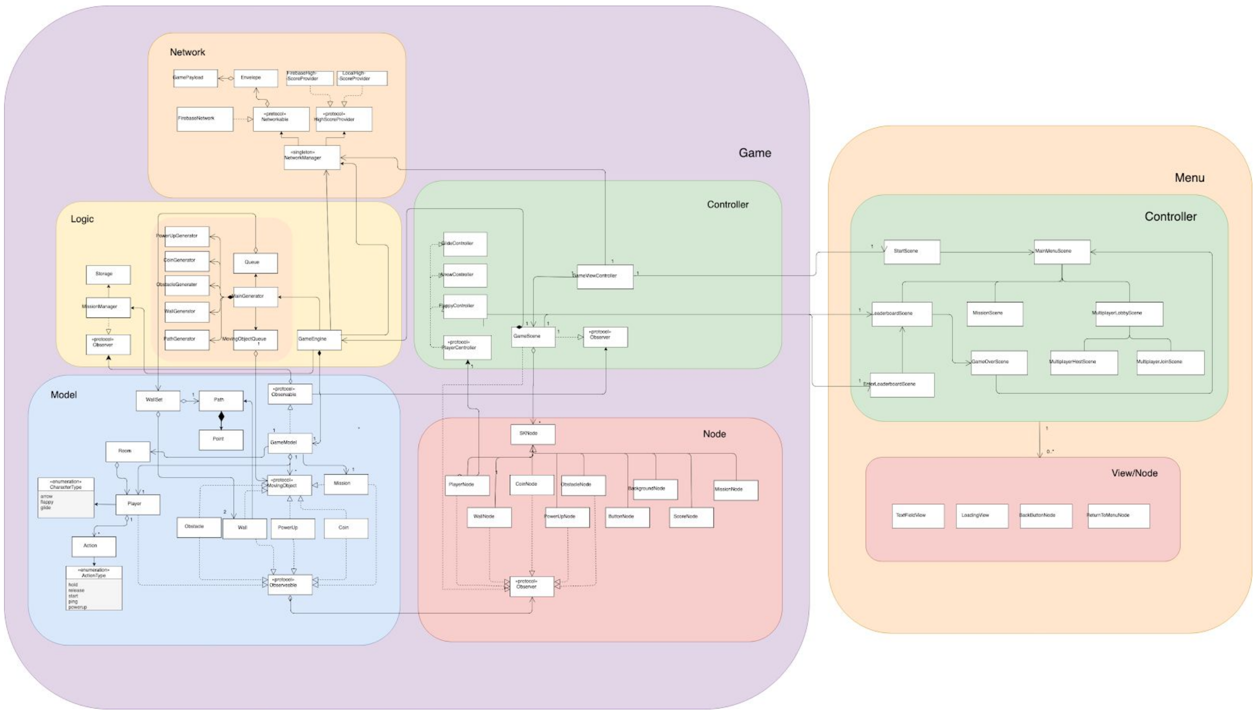
Test Results

From our current black box and performance testing, we have an average fps of 40 on the simulator and 60 on the iPad device. The performance on the simulator is an inaccurate representation of the actual performance. Thus, most testings are done on the iPad device. We achieved an average of 60 fps on iPad without SKTexture applied on SKSpriteNode, and an average of 58 fps with SKTexture. Number of nodes averages at 20 which is targeted and expected. Stress test indicates no reduction in generation performance when game is run 10x the speed.

Designs

Overview

Toplevel Organization



We organised our game program into four main sections: Model, Logic, Node and Controller.

Logic handles mainly the game logic of ***Dash***, including generation of walls, obstacles and power ups, changing the player-character or environment based on either user input or in-game events, as well as saving and loading game data.

Model contains the in-game objects of ***Dash,*** meaning the in-game logical representation, and thus contains the state of the different game objects. For example, for Player, what type of character is the player? How does the player react to user input? For obstacle, what kind of obstacle?

Node concerns with displaying the game objects on screen.

Controller handles user input, as well as linking Model/Logic with Node.

The organisation of code is discussed more in detail in the **Runtime Structure** and **Module Structure**.

Interesting Design Issues

Procedural Generation: Two-Stage Generation

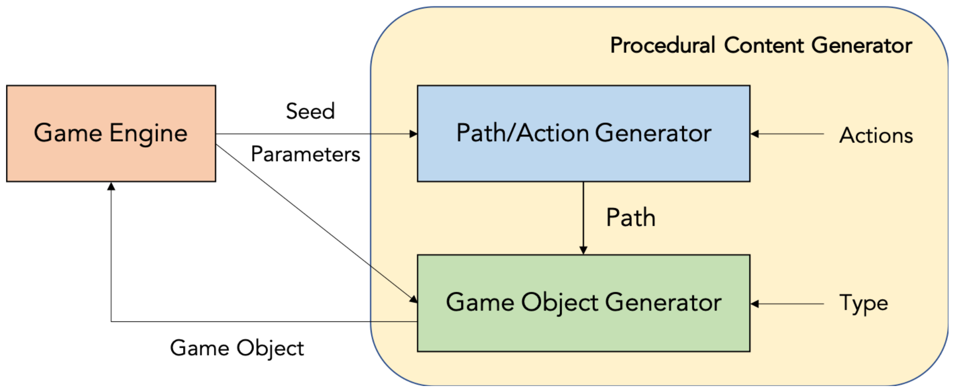
Our game supports procedural content generation (PCG) for the various game objects (walls, obstacles, coins and power ups), as our custom generator generates these in-game data through algorithmic means, instead of having the designer to create the individual levels manually, which is common in most games.

One common PCG implemented in games is chunk-based PCG, which essentially involves the storage of pre-designed modules/chunks, and the small chunks are pieced together via selective permutation to form the overall level. One advantage of chunk PCG is that since all the chunks are pre-designed, the game algorithm only needs to decide on how to combine and place the chunks, as opposed to generating the complex details and elements of the chunks. This however requires pre-designed modules, and would eventually lead to pattern repetitions which can be easily observed by the player. Thus, we chose not to adhere to the chunk-based PCG.

Instead, we designed a parameter driven algorithm for PCG, which generates content based on a number of custom parameters that allow us to tune the difficulty. One of the many parameters of our PCG is a random number generator seed, which allows us to recreate a level based on the seed, which is useful for multiplayer mode and replaying the levels. Besides adjusting the difficulty of the generated levels through parameters, we also aim to accommodate different types of player control by tweaking the parameters, and allow a more versatile level generator.

A common method of generating game level maps is through the interval system, in which at intervals throughout the game, the generator creates an object which makes sense in the game world. In the context of a 2D endless runner game, for example, at every 1.0 - 5.0 seconds, the generator introduces a new obstacle, which position and size allow the player character to technically be able to navigate through the course through position calculations. One downside of such implementation is that while the game is technically playable, the designer has limited control on how the final level will look like with the obstacles being pseudo-randomly generated. This method is seen in games such as Flappy Bird and Jetpack Joyride, in which game objects (e.g. pipes and laser obstacles) are discrete. In the case of our game, the top and bottom walls of the game are continuous throughout the game. Inspired by the state-of-the-art designs, we designed our own alternative method to generate our walls and obstacles.

Our parameter-based procedural level generation for Dash comprises of a two-phase approach:



The first phase involves generating the intended path for the player-character. The **Path Generator** constructs paths based on the player control methods and physics properties via parameters which also adjust the difficulty, along with a seed to randomise the path. Our initial approach is to imitate the player's actual actions (tap, hold, release) to generate the path. We however find the results to be too mechanical, requiring the player to adhere to our generate path strictly. Instead, we now construct paths with a mix of player controls and point of interests, thus allowing more flexibility to the player and also set the difficulty of the game to a suitable standard.

Based on the diagram above, the path generator passes the designated path to the **Game Object Generator (WallGenerator, ObstacleGenerator, PowerUpGenerator, CoinGenerator)** which adds game objects such as **Wall**, **Obstacle**, **PowerUp** and **Coin**. Similarly, a seed and parameters are passed into the generator to tweak the game difficulty such as size, leniency and frequency of game objects.

We also applied the chunk-based PCG concept into our parameter driven PCG, as we procedurally generate the paths and game objects chunk-by-chunk, and subsequently incrementing the difficulty of the chunks (a.k.a in-game stages). This reduces the present in game objects and saves memory and resource.

MainGenerator handles all the generation of in-game objects throughout the endless game, and contains sub-generators such as PathGenerator, WallGenerator, ObstacleGenerator and PowerUpGenerator. The **GameEngine** does not then have to be responsible for object generation, instead focus on updating the in-game objects' position and handles power up logic, and queries from the **MainGenerator** when game objects are to be added into the game.

Initially, game objects are created and added to the game real time when a distance is reached, this may cause delay if the object creation is not complete before the next tick. Now, the **MainGenerator** maintains a queue of **Wall** to be added to the game when a new chunk is reached, as well as a custom priority queue of **MovingObject** to contain the game objects (obstacles, coins, power ups) in order of generation time. These runtime structures will store and maintain incoming game objects to enhance efficiency. Then, for instance, the GameEngine compares its in-game time with the first element in the priority queue, and adds the object to the **GameModel** correspondingly. Everytime an object is removed from the queues, a new upcoming one is added asynchronously to the queue in a separate thread. The current game speed ensures that the queue is filled when dequeuing. However, multi thread access may be present and is currently not strictly guaranteed to be prevented. Currently, any latency when generating game objects are mainly due to rendering, and not the algorithm based on test results.

Rendering - Separating Model and View: Observer Pattern

The Observer pattern fits the game design and links the ***Model*** *with* ***Node****.* When the game objects in Model has any changes in its state or property (e.g. position), it notifies the observer nodes in Node. On one side, we have game objects which conform to the *Observable* protocol, and on the other side, we have node objects which conform to the *Observe* protocol. Game objects conform to the *Observable* protocol, and notifies its observers of any state changes.

```
protocol Observer: class {
    func onValueChanged(name: String, object: Any?)
}

protocol Observable: class {
    var observers: [ObjectIdentifier: Observation] { get set }

    func addObserver(_ observer: Observer)
    func removeObserver(_ observer: Observer)
    func notifyObservers(_ observers: [Observer])
}

struct Observation {
    weak var observer: Observer?
}
```

Another possible method of implementing the Observer pattern is using NotificationCenter. However, there are a few downsides of using NotificationCenter. One of them being that notifications are broadcasted app-wide without much containment. It would make relationships between the objects loose, thus harder to maintain clear separation between the different relationships and objects.

The **GameEngine** contains a **GameModel**, which consists of an array of **MovingObject**, which is respectively observed by its view representation **GameNode**. When the GameEngine updates the position of the **MovingObject**, the position of the rendered observer **GameNode** is also updated based on its model representation. The **GameScene** also observes the **GameModel**, and maintains a dictionary *[ObjectIdentifier: SKNode]*, to keep track of which **GameNode** is added and removed from the game. To summarize, the **GameEngine** updates and modify the logical representation **GameModel**, and its logical representation is then rendered by **GameScene** and **GameNode**.

An alternative to moving the game objects is by modifying the velocity within the SKPhysicsBody of the SKNode. Firstly, as the physics body is tightly coupled to rendering due to the design of SKNode, the resultant implementation will be heavily coupled. The current implementation allows GameEngine to set a global velocity for all moving objects, and add, remove and update obstacles easily with observation pattern. The SKPhysicsBody implementation is plausible, but not explored in this case.

Multiplayer and Shadow Play: Action Queue
There are a few methods on synchronizing the states between the users on remote devices. One of them is the consecutive sending of positions for each users. However, our game is fast paced and have smooth real time moving motions that are not easy to be recreated just based on the positions of the players.

Another approach is sending all performed user actions such as Touch Down and Touch Release. When these information are sent to the remote machines, the remote machines will simulate the user actions as if they are playing in real-time. However, there are a few problems such as frame skip issues and out-of-sync after a period of time due to carry forward error. (Discussed below at the “Other Discussions” section).

We implemented a hybrid of the two methods listed above. When an user actions are recorded, the positions of the user along with its physics properties (velocity) are sent together through the network to allow fixing the shortcomings of both methods. The user actions allow the smooth curves animations and at the same time the positions allow the adjustments of accumulated errors in a long run.

Currently, the user actions are sent instantly after the actions are performed. The actions will have a fixed delay of 0.2 seconds before it is performed on the remote devices to cater with any latency on the network. On the other hand, the position information are sent together with the action to correct the mismatch between the positions on different devices.

This multiplayer approach is working thanks to the same seed that is used to generate the maps by the Procedural Generation algorithm. The host of the room will generate a seed and send to every other clients so that the generated maps and obstacles will always be the same across all devices without the need of synchronize the map information.

Each player contains its own Action Queue both locally and remotely. The Action Queue contains all the Actions performed by the user. Each Action contains the timestamp and the action performed such as holding and release.

To synchronize the states and actions between users, we designed a Networkable protocol so any library and framework can conforms to it to allow the support of networking on the game. It can be implemented in any platform such as Peer-to-peer using Multipeer Connectivity, real-time database using Firebase or even custom websockets implementation using Socket.io.

When the game is played locally using Shadow play, the actions are recorded so that it can be replayed in the next round. The user can then play against the old movements of him/herself for practising purposes.

More detailed explanation on the difficulties we faced will be discussed at the next section under “Other Discussions”.

Handling Different Player Controls: Strategy Pattern

A challenge in handling different player controls is how to implement different behaviors for the same game without unnecessary code repetition. As the implementation for the control types supported may have very different structures (e.g. the arrow control requires direction of velocity to be changed directly, while the glide control requires a force to be applied to create upwards acceleration), it is not good maintainability to attempt to extract parameters by identifying common patterns between the different implementations. We also want to separate the physics body and rendering present in **SKNode**.

The simplest solution is using conditional if-else statements or switch cases to use the appropriate method given a player type, but this will lead to high coupling between the PlayerNode and the implementation of the controls, which should be decoupled for good separation of concerns and cohesion.

Another alternative is using inheritance, where we have different classes, each implementing a control type, to subclass from a general control class. However, the implementations for the various control types share little similarities, which makes having a parent class redundant without having useful methods to inherit from.

We decided to use the strategy pattern to support different ways of controlling the player. The pattern consists of an object using the strategy, the strategy protocol, and classes conforming to the strategy protocol (concrete strategies).

More of this is discussed in the section Module Structure: *Handling Different Player Controls: PlayerController, PlayerNode.*

Utilising protocols, the strategy pattern enables components to be more easily reusable and extensible as compared to using inheritance or if-else statements.

Libraries and Third Party Modules

We will be mainly using Apple's 2D game framework, SpriteKit, to help us with the game physics and animation effects.

We are using Firebase in the setting up of remote high scores. The responses from the server will be in the format of JSON.

We also use Firebase to establish the connectivity between peers on multiplayer mode. It used Event Handler pattern which makes the code cleaner and easier to understand. The comparison between FireBase and PeerKit, another library which we attempted to use, will be explained in the "Other Discussions" section.

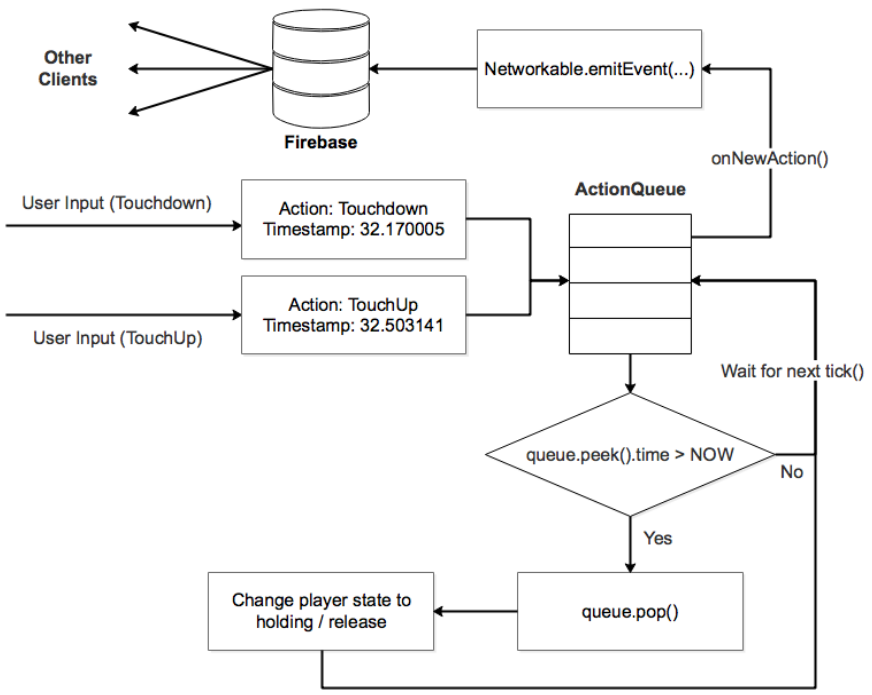
Other Discussions

Multiplayer

At the beginning, we thought that using peer-to-peer connectivity will be the best to reduce the latency because it used local approaches such as Bluetooth, Wifi and Wifi Direct. However, after trying to use the native library (MultipeerConnectivity) and third-party libraries such as PeerKit, the connection is not reliable and unknown errors occurred such as connection refused without much context or information.

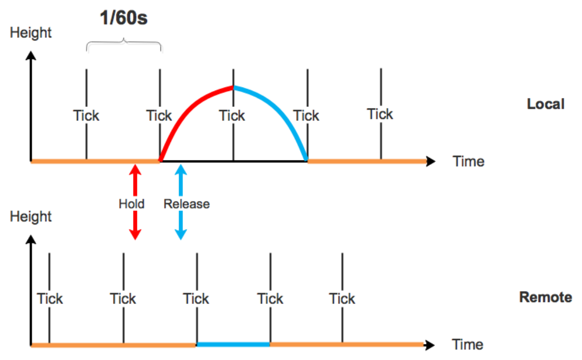
We choose to move to online platform through the Internet such as Firebase. Although the latency is slightly more than peer-to-peer, it is not easily noticeable. The disadvantage of it is the players have to be all connected to the Internet but at the bright side the players does not need to be physically nearby to each other now.

Using the action queues implementation as mentioned in the previous section, the flow of the actions between devices can be illustrated as follows:



It used the event handling design pattern to make the code decoupled and cleaner. Note that the networking component can be easily switchable to other backend system as long as it conforms to the Networkable protocol.

Besides that, we encounter with frameskip issue because different devices run the game at a different frame rate. Moreover, a slight different in the starting time of the game will cause the frames to be not aligned with each other. This is best illustrated with the diagram below.



Note that the actions will only be executed and performed at the next tick (1/60s in normal case), the slight out-of-sync between ticks on different devices will cause actions to overlap with each other. Therefore, we decided to limit at most one action in each frame. This might not be the best solution but calculating the subframes within frame required a lot more work and is prone to errors.

Scrolling Scene

Instead of moving the player and camera, we will move everything else in the scene (background, walls, obstacles) because the position of view is relative to the player. This allows us to implement parallax background easily as we can simply assign different scrolling speeds to different background layers. We use tiled background to generate the illusion of infinite background. The current design of the game logic, especially the obstacle and wall generator, works well with the “moving background” design, as we procedurally feed in obstacles or walls which in turn move towards the left, to create the illusion of the player moving forward. This also allows us to minimise the present nodes in the game scene.

Runtime Structure

MovingObject is a protocol in which logical representation of game objects which are handled by the GameEngine must conform. The protocol contains parameters that describes the game object, including object type, x and y position, width and height, and updates the position of the objects. It contains the required spatial and logical information by the GameEngine and Generator to process the objects to be translated and rendered to SKNode. **Wall**, **Obstacle, Coin** and **PowerUp** conforms to the **MovingObject** protocol. It also contains the parameter initialPos, which describes the position for the object to be generated, which is managed by **MainGenerator** and read by **GameEngine**. Having the game objects conforming to the protocol allow us to store and process the in game moving objects more easily and elegantly.

GameModel consists of all the present logic-representation of in-game objects: ***Player*** and an array of ***MovingObject***, as well as in-game data such as the current *stage*, the *speed* and *distance* of the game. When updating the game objects in the game loop, the game engine iterate through the game objects contained in ***GameModel***, as well as update the *stage*, *speed* and *distance* properties accordingly. It thus essentially contains all the necessary game data information for the ***GameEngine*** to read from and process into. It also conforms to the ***Observable*** protocol, as the game objects are observed by the **GameScene** and **MissionManager.** Storing the present in game data in **GameModel** allows us to render the objects easily.

MovingObjectQueue is a priority queue that contains **MovingObject**, and is found in **MainGenerator.** The **MainGenerator** generates upcoming **MovingObject** and **add** it into the internal priority queue, in which the top element is the object to be generated next. The **GameEngine** then *peeks* at the **MovingObjectQueue**, if the top element is to be generated into the game, the **GameEngine** polls from it and adds the **MovingObject** into the array in **GameModel**. When an element is polled from the **MovingObjectQueue**, **MainGenerator** automatically adds a new corresponding element into the priority queue. The usage of priority queue is due to the fact that objects may not be added in order. An alternative is to have separate queues for different type of objects, since objects for one type are generated in sequence. The **GameEngine** will then have to peek through all the different queues to add new objects. The separate queue implementation will also result in duplication of code for each different object type.

WallSet is a struct that contains a **Path**, and two **Wall** objects. It represents the generated **Path** designated for the player, as well as the top and bottom **Wall** based on the **Path**. It is used by **ObstacleGenerator** to generate **Obstacle**, as the generated objects are located in between the two walls, and also not intersecting the path. Encapsulating the three data into one struct allow us to pass data around easily for generation purposes.

Queue is a standard queue, adhering to its First-In-First-Out (FIFO) property. It is used in **MainGenerator** to store the upcoming **Path** and **Wall** (top and bottom) in a **WallSet**. As the path and walls are generated sequentially and according to time, a queue is used to store them instead of a priority queue. Similarly, **MainGenerator** stores the upcoming **WallSet** in a queue. Everytime when a stage ends, the **GameEngine** dequeues the **WallSet**, and adds the corresponding two **Wall** into the **GameModel**, the **WallSet** is also used by the **MainGenerator** to generate other MovingObjects. A possible alternative is to also store the **Wall** in the **MovingObjectQueue**. However, due to the two-phase design of our PCG, it would make more sense to generate the Path and Wall separately, and handle its generation chunk-by-chunk. The objects generated in MovingObjectQueue will only the ones which rely on the WallSet.

GameParameters contains all the tweaked parameters used by the game generators. The parameters includes obstacle generation frequency, distance between **Wall** and player, and distance between **Obstacle** and player for the three different player controls. These parameter values have been tweaked throughout the project to optimal values to achieve the best gameplay. It also contains a method *nextStage()*, which updates the parameters to increase the difficulty of the game at incrementing stages. It is used by the **MainGenerator**. It is responsible for handling increasing difficulty.

Point is a struct which represents a point in a x-y coordinate. It consists of two integer parameters *xVal* and *yVal*, as well as a function *gradient(with point: Point)* which returns the gradient between two Point structs. An alternative is to use CGPoint and extend to include more methods. We however decided to utilise our own struct to facilitate grid related calculations for procedural content generation, as we do not require to other properties that comes with CGPoint.

Path is a struct that describes a continuous trajectory connected point by one, and is represented by an array of *Point* objects. It is used to describe the designated path of a player, as well as the boundary of *Wall*. Our current implementation describes trajectory segment by segment connected by Points. It also contains method shift(by amount: Int) that translates the Path, as well as getPointAt(_:) and getAllPointsFrom(_:) which are used by generators to calculate more precise information regarding the path.

> Representation Invariant:
> 1) A path must be continuous to the right.
> In the array *points*, points[i].xVal < points[i+1].xVal, where i+1 is index contained in points.

Path is also used to describe a **Wall**, which contains parameter Path that describes the edges of a Wall.

To hold all the present SKNodes rendered, **GameScene** maintains a dictionary which maps the object identifier **MovingObject** in **GameModel** to the SKNode rendered in GameScene. The GameScene then refers to the dictionary to identify new objects added to the game model which would not be present in the dictionary, as well as old objects removed from the game model, which will be present in the dictionary but not in **GameModel**. This check is done when the array of **MovingObject** is modified in **GameModel**, which is observed by the **GameScene**.

Action is a model that contains the information about a relevant user action such as hold / release. It also contains the relative time of that action to allow synchronization between different devices and serialization for replaying purpose.

ActionQueue is fundamentally a queue that stores all the actions performed by a player. It allows the serializing, storing and transferring of all those actions so that different components can replay the actions easily. To ensure the actions are played sequentially and in order, the relative timestamp will be included to allow the replay in the same speed and interval as the original.

Envelope is a class that wrap the payload to be sent via network. It contains information such as the peer ID to identify the sender and receiver. It requires the payload to conforms with Codable for serialization and deserialization.

Networkable is a protocol that gives the basic features that is required for the game to connect with each other. Currently, it is designed with the following important methods. This allows different actions and events to be added easily as long as the relevant events are handled by the remote machines.

```
var peerID: String { get }
```

```
func createRoom() -> String
func joinRoom(_ roomId: String)
func onEvent(_ event: String,
    Run: ((_ peerID: String, _ object: AnyObject?) -> Void)?)
func emitEvent(_ event: String, object: AnyObject?)
```

HighScoreProvider is a protocol that gives the basic features that is required to store and retrieve high scores. It can be used to implement local and remote highscores with any kind of backend. In our app, **FirebaseHighScoreProvider** and **LocalHighScoreProvider** is implemented.

```
func setHighScore(_ record: HighScoreRecord,
                  category: HighScoreCategory, onDone: (() -> Void)?)
func getHighScore(category: HighScoreCategory,
                  onDone: (([HighScoreRecord]) -> Void)?)
```

NetworkManager is a singleton that manages which **HighScoreProvider** or **Networkable** to use. It also provides an action handler that allows the creation and deletion of handler easily.

MissionCheckpointList is a dictionary to keep track of the next mission checkpoints to be met. It has **MissionType** enum as the key and integer checkpoint value as the value. For example, `missionCheckpointList[MissionType.distance]` retrieves the distance the player has to reach in order to complete the next distance mission. This ensures the efficient checking of whether or not a checkpoint has been passed.

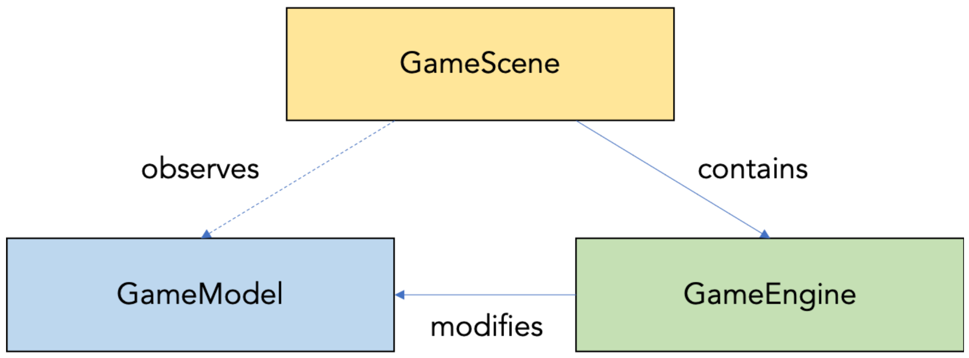
Module Structure

Class diagram can be found under the ***Designs: Overview*** section above.

MVC Pattern

We followed the main aspects of the MVC pattern by separating our classes into **Model**, **View** (Node) and **Controller**. However, we have an additional **Logic** module (can be categorised under Model) to decouple logic handling from the Model which mainly keeps track of the states of the objects.

Handling Game Logic: GameScene, GameEngine and GameModel

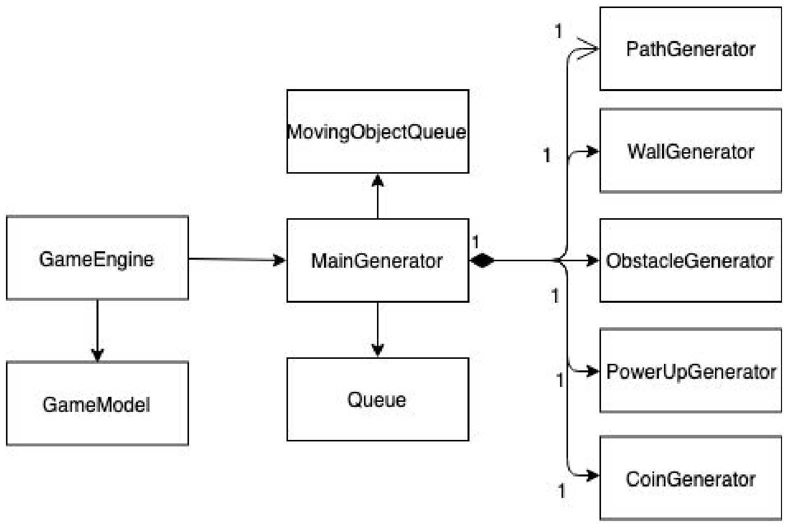


GameScene renders the game objects represented as SKNode. Most of the logic are however handled by *GameEngine*. *GameScene* observes *GameModel* via the Observation Pattern, and adds or removes new SKNode objects when *MovingObject* objects are added or removed within the *GameModel*. This is done by storing a dictionary [ObjectIdentifier: SKNode] which maps the GameObject ObjectIdentifier to the SKSpriteNode (WallNode, ObstacleNode, PowerUpNode), to keep track of the present nodes in the game to be rendered. *(explained in section Runtime Structure)*

To handle all game logic within the game, we organised the program to contain a Logic package, with main classes **GameEngine** and **MainGenerator** (as illustrated in the class diagram above). These classes would be in charge of the the various game events:

GameEngine is designed to manage the basic game logic of Dash instead of handling all within the *GameScene*. It has its own game timer, and updates the game state in the method *update()*. At every cycle of *update()*, it performs three main functions. First, it updates the position of all present game objects in *GameModel* as well as the in game information such as in game distance and speed. Second, it checks and handles power up status, and manages when to end a power up effect. Third, it checks for objects to be generated from the **MainGenerator** (to be explained next). It also handles power up trigger notified by the GameScene, as well as storing player actions for shadow play or multiplayer. Most of the logic handling is based on the in game time.

Handling Content Generation: MainGenerator, GameEngine



The responsibility of generating the in-game objects such as wall and objects are delegated to the **MainGenerator.** The MainGenerator contains two main runtime structure, a queue that stores incoming Path and Wall, and a priority queue that stores other incoming objects (Obstacle, Coin, PowerUp), as described in the section Runtime Structure. MainGenerator consists of multiple object generator as seen in the diagram above. These generators are in charge of generating a valid obstacle (type, size, position, etc) and adds the object to the relevant data structure. MainGenerator handles the logic of when to generate the objects.

The GameEngine will retrieve the relevant game objects from MainGenerator, and subsequently add the objects into the GameModel, with the following methods:

```
func getNext() -> WallSet
func checkAndGetObject(position: Int) -> MovingObject?
```

PathGenerator generates the designated path and passes the generated **Path** to the **WallGenerator, ObstacleGenerator** and **PowerUpGenerator**.

WallGenerator handles the generation of **Wall** for the game. It contains the algorithm to generate walls based on the current **Path**. It guarantees that there exists a path that the player can choose to not collide with any walls.

ObstacleGenerator handles the generation of **Obstacle** for the game. It contains the algorithm to generate obstacles based on the current **Path** and **Wall**. It guarantees that there exists a path between the walls that the player can choose to not collide with any obstacles, and that the obstacles do not intersect with the wall.

PowerUpGenerator handles the generation of **PowerUp** for the game. It generates the **PowerUp** object in line with the **Path** to ensure that the object is reachable by **Player.**

CoinGenerator handles the generation of **Coin** for the game. It generates the **Coin** object in line with the **Path** to ensure that the object is reachable by **Player.**

Handling Contact Detection: GameScene and GameEngine

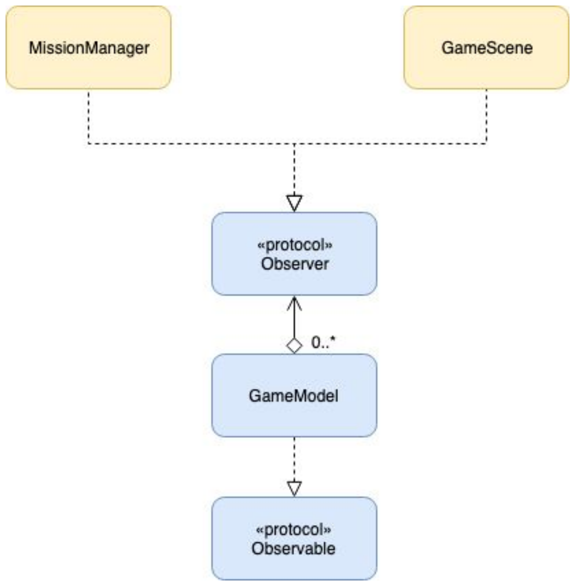
We use SKPhysicsContactDelegate that comes with SpriteKit to handle detect contact. Note that we are only concerned about contact between nodes and not collision, as events such as colliding with obstacle/wall and collecting power ups and coins only involve contact, not collision.

An enum **ColliderType** is created to set the contact bits of the different node types. By setting up the categoryBitMask and contactTestBitMask of the different Node objects, contact detection can be handled by **GameScene**. Thus, GameScene detects contact between nodes, and then sends the contact information to **GameEngine**, which handles the contact. This could include game over, and triggering of power up. Instead of detecting collision within the GameModel, we chose to implement contact detection within the GameScene with the in-built SKPhysicsContactDelegate for convenience purposes. It also makes sense as the contact is detected between the rendered nodes, and will look more real and valid to the player.

Handling Mission: GameModel and MissionManager

Another application of the observer pattern can be found in the implementation for Missions. A MissionManager will observe the GameModel for changes in the game state such as distance travelled and powerups collected. Note that since the

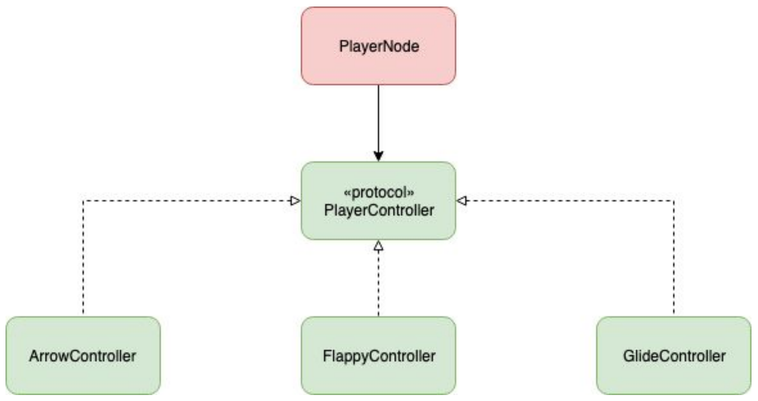
GameModel is also observed by the GameScene (for logic and view separation, as mentioned in the previous section), the GameModel will have two observers.
The GameModel will notify all of its observers for any important changes to the game state, where events will be identified with a name for observers to check if they should respond to the notification.



As can be seen from the above figure, using the observer pattern cleanly separates the observable (GameModel) from its observers (MissionManager, GameScene), and this pattern can be easily reused in various components in the project when a notification system is useful.

Handling Different Player Controls: PlayerController, PlayerNode

We use the strategy pattern to support different ways of controlling the player. The pattern consists of an object using the strategy, the strategy protocol, and classes conforming to the strategy protocol (concrete strategies). The figure below shows the relationship between relevant components in implementing the strategy pattern for handling different player controls.



PlayerNode: Uses the strategy, PlayerController, to handle different player behaviors depending on its player type (arrow, flappy, glide). At runtime, the player type is determined and the appropriate type of controller can be initialised and assigned to the PlayerNode.

PlayerController: The strategy protocol which defines the action to handle (i.e. moving the player)

ArrowController/FlappyController/GlideController: Implements the specific strategy for the action.

Updating and Saving Missions: MissionManager, Storage

The **MissionManager** detects and handles mission completion events. It observes the GameModel to keep track of the distance travelled and powerups and coins collected before validating whether or not a mission has been completed. Upon detecting a mission completion, the next mission checkpoint is generated and updated to the local storage through a static class, Storage. The Storage class is made static to allow for saving and loading data from anywhere in the project without the need for multiple instantiation of the class.

Handling Networking: Networkable, RoomViewController

The **Networkable** singleton is abstracted by **NetworkManager** and invoked by **GameEngine** and **RoomViewController**. It used event emitter pattern to allow cleaner code and different components can listen to different events easily. The events are all remotely from a backend server or database such as Firebase or Socket.io. In our case, **FirebaseNetwork** is implemented to handle with this.

At the same time, **GameEngine** will also observe any new **Action** that is being added to the current **Player** so that it can emit this **Action** to the backend server. This allows the **Action** to be transmitted to other remote devices easily by abstracting the underlying details.

Singleton Pattern

Network service are often designed as a Singleton because it is more efficient and different components will need to interact with the network. This can reduce the need of passing down the Network instance through different part of the program. Another way to do so is the publish-subscribe design pattern that allows components to subscribe to a topic such as “joinRoom” when another user joined the room, or “newAction” when another user execute a different action.

Reflection

Evaluation

Initially at the first two weeks, we were not be able to progress as planned mainly due to poor delegation of tasks. We splitted the procedural generation into three types based on control, and have each member handle one. We noticed that there were many repeated code or logic done by each member.

After rounds of discussion, we finalised on the generator design as a group (parameter-driven PCG) and delegate the game logic tasks to Jie Liang. Jie Liang handles all the object generation related code as well as the game logic. Next, we delegated the task of networking (multiplayer and shadow player) related tasks to Yee Chin. Jolyn handles all the player control related modules, as well as the missions controller and integration in the code. With the new task delegations, each member's work are almost independent with each other. For example, the map and obstacle generation is independent of player controls, as collision detection can be integrated later. We observe that our group progress is much more efficient after the task reformation, and are on track to complete the required tasks.

We have set aside Wednesdays for group meetings, as well as a whole-day codesprint to speed up development progress and facilitate better communication. Each member is expected to finish their respective tasks before every weekly meeting, and the individual parts are then integrated into the game.

By the last week of this semester (Week 13), we focused on the aesthetics part of the game such as sprites, animations and effects.

Overall, we felt we did well on the code structure and task delegation. We separated the different components such as game objects, player, networking and missions well. This made our development very smooth as each member is focused on different components. We were however not able to conduct much glass-box testing on our code, as we found it hard to test components such as Logic and Controller, which relies heavily on GUI. Testings is a field which we hope to master in the near future.

Even though the assets and aesthetics are not the focus of the project, we are also proud of the final design and User Interface of our final product, as we managed to maintain the minimalist design behind a complex code, and we did not spend much time on the assets. We received various compliments during STePs and were satisfied with the final product.

Lessons

The beauty of a game is to make it easy to play but hard to master. We applied various game-related techniques on game design to make the game complete and fun to play, such as random maps, power ups and multiplayer. We have successfully modularised and separate the game object and node object via observer pattern, and also dedicated a section to handle complex game logic, especially procedural generation of obstacles and wall. We applied additional design patterns such as strategy pattern for different player controls and singleton pattern for networking to reduce coupling and increase cohesion of our program. This would allow us to introduce more control types or multiplayer features easily for future development.

Due to lack of experience in game development, we had a hard time deciding on how certain things should be implemented, such as procedural content generation methods, as well as multiplayer mode implementation. This caused us to lag in terms of progress during the first few weeks. After group meetings and online research, we were able to finalize with the implementation, and the subsequent coding process was much faster and efficient.

A tradeoff we had to make was between polishing the game or including more features. Since this module is a software engineering course, we prioritise on making sure we have sufficiently complex features before setting aside time for polishing the game such as including more animations and assets. While we did not had time to implement some bells and whistles we had planned to do such as sound effects and background music, we felt that the game is presentable and enjoyable with the improvements on the visuals we did on the last week.

Appendix

Test cases

Test implementation of stages

- Test control 1 - Arrow
    - The generated wall should be edges with different length and rotation
    - Suppose the game has started
        - The character should be moving upward at constant speed
        - If the character is moving upwards
            - Tapping on screen should change the direction to moving downwards
            - Holding on screen should only change the direction once
        - If the character is moving downwards
            - Tapping on screen should change the direction to moving upwards
            - Holding on screen should only change the direction once
        - The distance at the top left hand corner should increase accordingly
        - The distance between walls should decrease gradually
- Test control 2 - Flappy
    - The generated wall should be edges with different length and rotation
    - Suppose the game has started
        - The character should be moving to the right at constant speed
        - Tapping on screen should make the character fly (A small force at the opposite direction of the gravity)
        - Holding on screen should just activate the tapping action once
        - The character should drop according to gravity if no input is given
        - The distance at the top left hand corner should increase accordingly
        - The distance between walls should decrease gradually
- Test control 3 - Glide
    - The generated wall should be edges with different length and rotation
    - Suppose the game has started
        - The character should be moving to the right at constant speed

        - Tapping on screen should make the character fly (a small force at the opposite direction of the gravity)
        - Holding on screen should increase the opposite force at the opposite direction of the gravity, thus accelerate upwards
        - The character should drop according to gravity if no input is given
        - The distance at the top left hand corner should increase accordingly
        - The distance between walls should decrease gradually

Test implementation of generators

- Test generators
    - Wall
        - It should generate Wall that is in between the boundary of the view
        - The distance between two walls should be larger than the height of the player-character
        - Wall should connect with the previous Wall directly.
        - With the same seed, generated Wall should be identical
    - Obstacle
        - It should generate Obstacle that is in between the 2 walls
        - The player-character should have a possible path through the game
        - Missile obstacle should move at twice the speed of a normal obstacle
        - With the same seed, generated Obstacle should be identical
    - Power Ups
        - It should generate Power Up that is in between the 2 walls
        - Power Ups should be possible to reach by Player
        - Type of Power Up generated should be randomised
        - With the same seed, position of generated Power Ups should be identical
    - Coins
        - It should generate Coin that is in between the 2 walls
        - Coin should be possible to reach by Player
        - With the same seed, generated Coins should be identical

Test implementation of networking / multiplayer

- Test multiplayer
    - If a player created a new room

        - A 4 digits room ID should be generated for the room
        - The room should have only 1 people (The player him/herself)
        - When another player joined the room, the room should have 1 more player
    - If a player joined a room using room ID
        - If the room exists
            - The player should join the room
            - The room should have 1 more people than previously
        - If the room doesn't exists
            - A "room does not exist" message should be shown
    - If the "Start" button is pressed from host
        - The game should start at almost the same time across different devices
        - The map generated on different devices should be the same
    - If a player tap on the screen in game
        - A tap action should be sent and reflected on other players' game in at least 0.2 seconds
    - If a player hold on the screen in game
        - A hold action should be sent and reflected on other players' game in at least 0.2 seconds
    - If a player leave the game halfway (disconnected)
        - The player will maintain its current position and velocity until it is out of bound
    - The pause button should be hidden in multiplayer mode

Test implementation of shadow play

- Test shadow play
    - Suppose the game has ended
        - If the "Shadow play" button is pressed
            - The game should restart with the exact same map
            - The shadow of the player from the last round should be shown with the exact actions with last round
            - The shadow of the player should have a 0.2 seconds delay

Test behavior of powerups

- Test powerups
    - Suppose it has collided with the character
        - Dash

            - Character should move forward by X metre
            - After X metre, game should return to normal speed, and character will blink and can pass through Obstacle and Wall for Y metre
            - Going out of bound should game over
        - Ghost
            - If character collide with obstacle
                - Character should pass through obstacle and not result in game over
            - If character collide with wall
                - The game should be over
        - Shrink
            - Character and its hitbox size should be decreased by 50%
            - Hitting with any obstacles or walls should game over

Test behavior of collision

- Test collision
    - Suppose the character has Ghost powerup
        - Collides with any obstacles should do nothing
        - Collides with walls should end the game
    - Suppose the character has Dash powerup
        - The character should move forwards for X meter
    - Suppose the character has Shrink powerup
        - The character should have smaller size (50%)
        - Collides with any obstacles or walls should end the game
    - Suppose the character has no powerups
        - Collides with any obstacles or walls should end the game

Test behavior of rendering

- Test rendering
    - Suppose the game control is Arrow (Control 1)
        - The rendered background should be blue
        - The rendered walls should be generated for the Arrow control
    - Suppose the game control is Flappy (Control 2)
        - The rendered background should be green
        - The rendered walls should be generated for the Flappy control
    - Suppose the game control is Glide (Control 3)
        - The rendered background should be red

            - The rendered walls should be generated for the Glide control
- Test parallax background
    - The moving speed of the front layer of the background should be faster than the moving speed of the back layer of the background

Test implementation of high scores

- Test high score
    - Suppose the game has ended
        - If the score is one of the top 10, the game should prompt for the name of the player
            - Pressing the “Submit my score” button should save the name into the high score list, sorted by score ascendingly
            - If the user is connected to the Internet, the score should be uploaded to the global ranking if it is within the top 10
        - If the score is not in top 10, the game should just end with the top 10 leaderboard shown to the user

Test implementation of menu interfaces

- Test main menu
    - If the right arrow button is pressed
        - The game mode for the next control should be displayed, in the order arrow > flappy > glide
    - If the left arrow button is pressed
        - The game mode for the previous control should be displayed, in the order arrow > glide > flappy
    - If “tap to play” is pressed, the game view should be loaded with a countdown to start the game in single player mode.
    - If “Multiplayer” is pressed, a new screen should be shown with 2 options: “Host Game” and “Join Game”
        - If “Host Game” is pressed, the host room view should be loaded with relevant information such as room ID
            - If a game mode under the label “Select Mode” is pressed, the selection box should move to the selected game mode
            - If “Start” is pressed, the game should start with the selected game mode
            - If the back button is pressed, the waiting room view should be dismissed, and the room should be deleted

        - If “Join Room” is pressed, the waiting room view should be loaded with the list of other devices
            - If the back button is pressed, the waiting room view should be dismissed
            - The previous selection of game mode in the main menu should be preserved
    - If “High Score” is pressed, the high score view should be loaded with a list of top 10 winners
        - If anywhere on the screen is tapped, the high score view should be dismissed
        - The previous selection of game mode in the main menu should be preserved
    - If “Mission” is pressed, the mission view should be loaded with a list of completed missions and the next 3 missions to be completed
        - If the cross button is pressed, the mission view should be dismissed
        - The previous selection of game mode in the main menu should be preserved
- Test pause
    - Suppose the game has started
        - If the pause button is pressed
            - The game should be paused
            - The pause button should now become a play button
    - Suppose the game has paused
        - If the play button is pressed
            - The game should continue
            - The score and speed of the game should remain the same
            - The play button should now become a pause button
    - Suppose the player is in multiplayer mode
        - The pause button should not be shown
- Test gameover
    - Suppose the game has ended
        - If the score is one of the top 10, the game should prompt for the name of the player
            - Pressing the “Submit my score” button should display the highscore list with the player name in it
                - If “Play again” button is pressed, window should close and the game is restarted

                - If “Shadow play” button is pressed, window should close and the game is restarted in shadow play mode
                - If “Return to menu” button is pressed, window should close and main menu window should display
        - If the score in not in one of the top 10, the game should display the highscore list without player name in it
            - If “Play again” button is pressed, window should close and the game is restarted
            - If “Shadow play” button is pressed, window should close and the game is restarted in shadow play mode
            - If “Return to menu” button is pressed, window should close and main menu window should display

Test implementation of missions

- Test initial missions
    - Suppose the game has not been played before
        - If “Missions” is pressed from main menu
            - Distance mission should display “Reach 500m in one run”
            - Powerup mission should display “Consume 1 power ups”
            - Coin mission should display “Collect 10 coins”
- Test clearing of missions
    - Suppose player has just completed a mission in-game
        - A popup text displaying the mission completed should appear from the bottom of the screen
- Test storage of missions
    - Suppose player has cleared some missions
        - If “Missions” is pressed from main menu after a game
            - The upcoming missions to be cleared should be displayed
        - If “Missions” is pressed from main menu after closing and starting up the app
            - The upcoming missions to be cleared should be displayed

Unit Tests

- Point
    - gradient(with point: Point)
        - Should return correct gradient between two Points
- Path

  - count
    - Should return correct number of points
  - lastPoint
    - Should return last element in points array
  - append()
    - count should increment by one
  - shift()
    - Should return translated path by input amount
- Obstacle
  - update()
    - xPos and yPos should update according to input velocity
- Wall
  - update()
    - xPos and yPos should update according to input velocity
- PowerUp
  - update()
    - xPos and yPos should update according to input velocity
- GameEngine
  - update()
    - Should increment *inGameTime*
    - Should update all *MovingObjects* in *GameModel*
- PathGenerator
  - generateModel()
    - Should return valid Path based on input parameters
- WallGenerator
  - generateTopWallModel()
    - Should return valid Wall based on input parameters
  - generateBottomWallModel()
    - Should return valid Wall based on input parameters
- ObstacleGenerator
  - generateNextObstacle
    - Should return valid Obstacle based on input parameters
- SeededGenerator
  - next()
    - Should return same value if the seed is the same

Roles and Responsibilities

Jie Liang	Procedural Content Generation Gameplay Logic (PowerUp, Coin, Collision Detection) Assets and Animations
Jolyn	Player Controls Missions Integration Assets
Yee Chin	Multiplayer Mode Shadow Play High Score

Detailed Schedule + Task List

Week	**Deliverables**	**In-charge**
~~9~~	Milestone 0.5: Basic prototype ● Able to move player ● Able to scroll background	-
	Preliminary design document (24 March)	Everyone
~~10~~	Milestone 1 Overview: Basic functional game ● Able to control player ● Able to generate walls and obstacles properly	
	➢ Prototype wall generation	Jolyn
	➢ Player Action queue for Multiplayer	Yee Chin
	➢ Generate wall and obstacles. ➢ Control player by tap and holding	Jie Liang
	24-hours codesprint (1)	Everyone

11	Milestone 2: Full working game ● Procedural Generation of Walls, Obstacles and Powerup ● Different player controls ● Multiplayer ● High Score (Shift to week 12)	
	➢ Handling different player controls	Jolyn
	➢ Basic Multiplayer connectivity using PeerKit	Yee Chin
	➢ Procedural Generation of Playable Walls, Obstacles and Powerup ➢ Increase generation difficulty through phases	Jie Liang
	Progress report II (7 April)	Everyone
12	Milestone 3: Production-ready game ● Multiplayer Menu and Selection ● Missions ● Power Ups, Coins Trigger ● High Score Sharing ● Shadow Play ● Assets and animations ● Menu/navigation	
	➢ Improve procedural generation algorithm (~Tues) ➢ Power Up and Coins (~Thurs) ➢ Integration (Sat) ➢ Animations (Sun)	Jie Liang
	➢ Multiplayer (~Wed) ➢ Shadow Play (~Thu) ➢ High Score (~Fri) ➢ Assets	Yee Chin
	➢ Improve player controls (~Wed) ➢ Missions (~Thurs) ➢ Menu/navigation (~Fri) ➢ Event system for animations ➢ Animations	Jolyn

	➢ Assets/sounds	
	Integration + Debugging + Buffer	Everyone
	24-hours codesprint (2) - Sat: If less than 80% done	Everyone
13	➢ Fix Bugs ➢ Overall Integration ➢ Enhance PCG algorithm ➢ Assets/Animation/Sound	Everyone
	STePs (17 April)	Everyone
	Final project report (21 April)	Everyone
	Final project presentation (22-27 April)	Everyone