System Manual

While the game is intended for use with the Fizzyo platform, full integration using the developer package has not been completed, as the latter was not finished at the time of writing. An adapter class $UserInput.cs$ (found in $Assets$) was however made to easily change user inputs (such as new device commands in case of firmare updates). The game should however function with the device at this point in time with the following included lines:

```
// Breath value (float)
FizzyoDevice.Instance().Pressure() || Input.GetAxisRaw("Horizontal");
// Button press (bool)
FizzyoDevice.Instance().ButtonDown() || Input.GetKeyDown("joystick button 0");
```

As mentioned in the conclusion, the game could be further worked upon by adding more content in terms of levels, and perhaps at a later stage shop contents.

Adding a level

All level scripts are a child of the $LevelContent.cs$ class, so that common functions (regarding coin collection, popups, and HUD counters) can be inherited and used easily. The child class only needs to call the initialisers for the HUD counters prefab and specify the respawn points for each hole in the course, see $Woods$ example. The child $LevelContent$ script needs to be assigned to an empty gameObject in Unity, and all the public counters (inherited from the parent) linked to the appropriate HUD gameObjects. To make a new level script therefore the developer just needs to create a child of $LevelContent$, or simply use the $Woods$ script as template.

Each course should follow a predetermined theme (such as Desert/Snow/Space) and have a total Par of 100, as this is the minimum length of the average airway clearance exercise session. Holes should be varied in length and difficulty, and give no more coins than half the value of the hole number as a rule of thumb. Each level could also feature a certain related challenge, for example Desert could be windy (constant slight sideways force), Snow could be slippery (momentum only decreases to a certain threshold, thus "slipping"), and Space could have low gravity. It is important for each course to be interesting, and follow the LEMONS framework. The ground should be set with the $Deathzone$ tag for respawn if out of bounds. As it stands, respawn locations are dependent on the hole number, so holes shouldn't be too long. Nevertheless, remember that the patient would rather continue playing the game than be cut short. Most importantly: DO NOT PUNISH THE PLAYER FOR INACTIVITY.

Because of how Unity handles lighting, it is important for there to be no static objects before a build, as the lightmaps will take an excruciatingly long time to generate. Level content has been generated using ProBuilder Advanced which is extremely useful for map creation. However, before building as a UWP app this import needs to be deleted as the UWP platform will not package the executables linked to the tools. This means PB gameObject may need to be converted to regular .obj files and re-imported before they can be used in a build (currently, this is a PB Advanced feature only). Vanilla Unity build tools can be used otherwise. Ensure only copyright free (or purchased) assets are used in the game.

It is important for HomePage to be first (Scene 0) as this will be the first scene rendered on launch. Subsequent levels need to also be in order as button listeners are assigned to existing buttons on launch. For example, in the $HomePage$ script, the $OnLevelSelect$ method's case 0 refers to $Tutorial$, and case 1 refers to $Woods$.

```
private void OnLevelSelect(int currentIndex) {
    switch (currentIndex) {
        case 0: SceneManager.LoadScene(1); break;
        case 1: SceneManager.LoadScene(2); break;
        default: SceneManager.LoadScene(0); break; }
    }
```

Special consideration must therefore be taken when assigning text to the buttons, and their order in the Unity editor.

Adding shop items

Test users always provide great ideas of content to add in shops. Both specific hats and whole other categories have been suggested (e.g. background music or accessories).

To add an extra category, the relevant PlayerController initialisers, HomePage script shop methods and Unity gameObjects can be used as templates and duplicated. In particular, the HomePage script features most variables or functions in sets of three, clearly labelled with the appropriate shop (Colour/Trail/Hat). It should therefore be relatively simple to copy this code to add a new category.

Additional similar changes need to be done in the $SaveState.cs$ file (used to save the game on item buy or level end) and $SaveManager.cs$ which saves the game state and doubles as a holder for the array of gameObjects to be sold, and additional purchasing functions which can be copied. The purchased and active items flags are saved using bitwise operations on integers in the $SaveState.cs$ file to save space.

New items can then be assigned to the array of gameObjects in $SaveManager$ through the Unity editor in the $SaveManager$ gameObject in the $HomePage$ scene. Images can be assigned to the individual scrollPanel buttons (add more buttons and lengthen appropriate arrays for more objects), and individual prices can be set in the $HomePage$ shopPrices arrays.