OULU UNIVERSITY OF
APPLIED SCIENCES

Armi Korhonen

SERIALISING AND SAVING DATA IN UNITY3D USING GOOGLE FIREBASE

SERIALISING AND SAVING DATA IN UNITY3D USING GOOGLE FIREBASE

Armi Korhonen
Bachelor's Thesis
Spring 2019
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author: Armi Korhonen
Title of the bachelor’s thesis: Serialising and Saving Data in Unity3D Using Google Firebase
Supervisors: Teemu Korpela, Lasse Haverinen
Term and year of completion: Spring 2019 Number of pages: 43

The purpose of this thesis was to get a better understanding on how to implement level sharing in a mobile game. An already existing prototype of a game made in Unity3D was used, as there was already a plan in place to publish the game on mobile devices. The plan was to look into if it was possible to implement level editing and sharing similar to Nintendo's Super Mario Maker. There were no technical nor architectural requirements, as there is no public info on how this kind of feature should be implemented.

A tutorial from LinkedIn Learning was used to learn how to integrate Google Firebase in Unity3D. This information was then adjusted to be used with the level data that was stored on the Google Firebase Realtime Database. A lot of research was done on how to structure the data, mainly using different technical forums. Through this research, it became clear that Google Firebase is not necessarily the best service to be used for level sharing the way it was desired due to the way the Realtime Database structures the data.

Even though the thesis did not create a practical solution to how the level sharing should be implemented, it did provide a good overview of how the issue should be approached in the future. Learning about JSON serialisation in Unity3D proved to be very valuable as it can be used in many different ways when storing and transferring data in games. A level editor was implemented, which can be repurposed to be used in other games as well. Learning about Google Firebase's usage was also very valuable. Firebase can be used for many different features in mobile games, even if it would not be used for level sharing specifically.

Keywords: Unity3D, Game Development, Google Firebase, Mobile Games

CONTENTS

ABSTRACT 3
CONTENTS 4
VOCABULARY 6
1 INTRODUCTION 8
2 SAVING DATA IN GAMES 10
2.1 Introduction 10
2.2 Commercial and contractual considerations 12
2.3 Saving data in Unity3D 13
2.3.1 DontDestroyOnLoad() 13
2.3.2 Player preferences 14
2.3.3 XML and JSON 15
2.3.4 Binary files 16
2.3.5 Other solutions 17
3 CREATING THE LEVEL EDITOR 18
3.1 Requirements 18
3.2 Level editor functionality 18
3.2.1 Level objects 22
3.3 Serialising level object data 24
3.3.1 Custom serialisation 24
3.3.2 Using JSON for serialisation 25
3.4 Using the interface method 25
3.5 Saving object data 26
3.6 Loading a game level 28
4 STORING LEVEL DATA IN THE DATABASE 29
4.1 Requirements 29
4.2 Using a BaaS 31
4.3 Choosing the service 32
4.4 Google Firebase products 34
4.4.1 Realtime Database 34
4.4.2 Authentication 35
4.5 Integrating Firebase with Unity3D 36

4.6 Structuring the data 37
4.7 Security considerations 38
5 CONCLUSION 40
REFERENCES 42

VOCABULARY

API	An application Programming Interface. A set of protocols and methods of communication between software components.
BaaS	Backend as a Service. A business model, which offers application developers a set of cloud computing services.
Firebase	A mobile and web application platform owned by Google.
free to play	A term used to describe especially mobile games, which are free to download and play, but might include other forms of monetisation.
game engine	A software-development environment for creating video games.
game jam	An event during which games are created alone or in groups. A game jam usually has a set theme and a time limit.
game level	A section of a game. Usually with a clear goal which needs to be reached before the player is allowed to advance to the next level.
game object	An object in Unity3D game engine, which functions as a container for different components. The components can be scripts, 2D sprites, 3D models, etc.
In-App Purchase	Buying of goods or services from inside a mobile application.
in-game currency	A virtual currency used to make purchases in a mobile game.
instantiate	In Unity3D it means to create an instance of a game object.

JSON	JavaScript Object Notation. A lightweight data interchange format. Also see XML.
NFC	Near-field Communication. A set of protocols that allow two devices to communicate within a few centimeters of each other.
Scene	A concept in Unity3D. A scene contains all the needed objects, scripts and cameras to display a menu or a game level.
SDK	Software Development Kit. A set of tools and libraries for creating software for a specific platform.
serialisation	Translating data structures of the state of an object into data to be stored.
Unity3D	A cross-platform game engine.
XML	Extensible Markup Language. A dedicated, declarative language used to describe data in a human-readable form.

1 INTRODUCTION

The purpose of the thesis was to study how to serialise game level data, how to store it and how to share it with other players. The idea came from a real world need of content creation for a game that would be published on mobile platforms.

The game “Interdimentional Nuisance” [sic] was developed in 2016 as part of the Castle Game Jam event. The game was developed using the Unity3D game engine. (1.)

As with most game jam games, the game is a prototype with a few levels, rather than a fully developed game ready to be published. In the game, the player aims and shoots projectiles at ghosts. The goal in the game is to hit, and subsequently destroy, all the ghosts in the level. An example level of the game is shown in the figure 1.

The game was demonstrated to the audience at the end of the game jam event. Based on the feedback, the game would be suitable for further development and release for mobile devices.

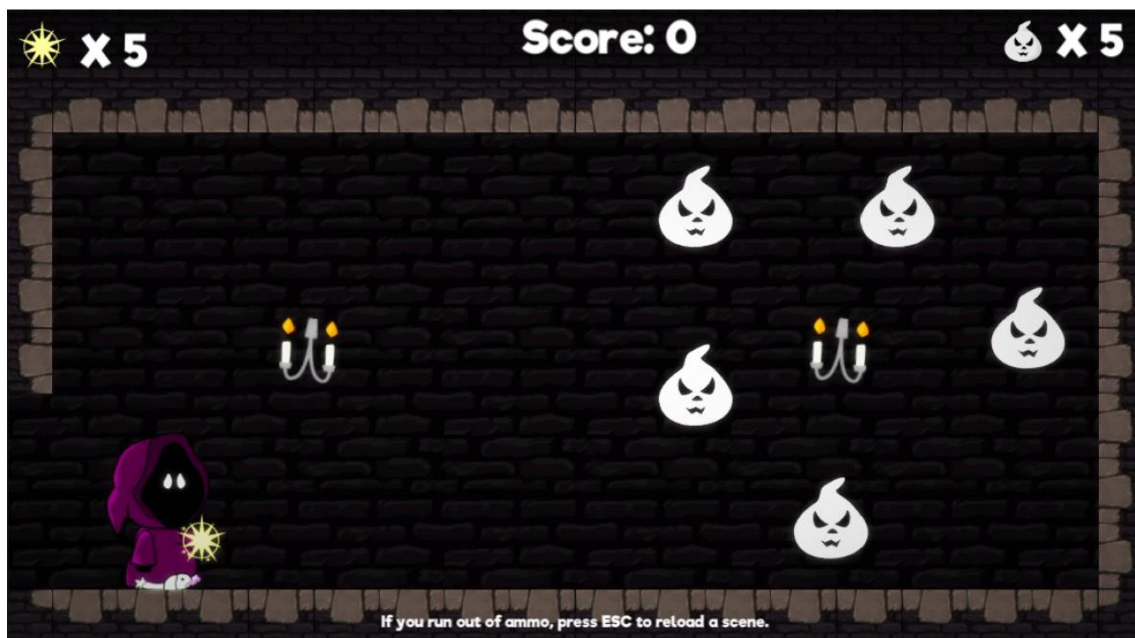

FIGURE 1. The first level in the game “Interdimentional Nuisance”

The largest workload for such a game would be designing and building content. Since this would be a huge undertaking for a sole developer, it would be very helpful if some of the content was created by the players themselves.

There are also other benefits for allowing players to create their own levels. People who create their own levels, would most probably share their creations with their friends, thus simultaneously creating word-of-mouth marketing for the game.

Nintendo’s Super Mario Maker was one of the main inspirations behind wanting to create a level editor and allowing players to browse and download levels made by others. There was an interest in finding out if it would be possible to create something similar on mobile, especially given the limitations of the game being developed by a sole developer and it being monetised only by advertisements. While there were no specific requirements for how the levels could be browsed, one of the main ideas was that users could watch a video advertisement in the game, and as a reward, they could download a number of random levels made by others. The levels should also be ones that the user has not played before.

There was no previous knowledge of serialising and saving data in games. There was also no previous knowledge of how cloud services are used for data distribution in games.

2 SAVING DATA IN GAMES

2.1 Introduction

The very first video games did not necessarily save any persistent game data. Games such as TicTacToe and Pong, were short and the game would simply end after the play session. Even after the games became longer, games did not necessarily store game state data on the computer. Instead, in games such as Metroid, the game would provide a password that could be later used to restore the progress. Some games used the inability to save the game state as a design feature; Treasure of Tarmin had four difficulty settings which were measured as the number of levels required to reach the final treasure. The gameplay at the easiest setting would be 5 minutes, while on the most difficult setting the game would last for 5 hours. (2.)

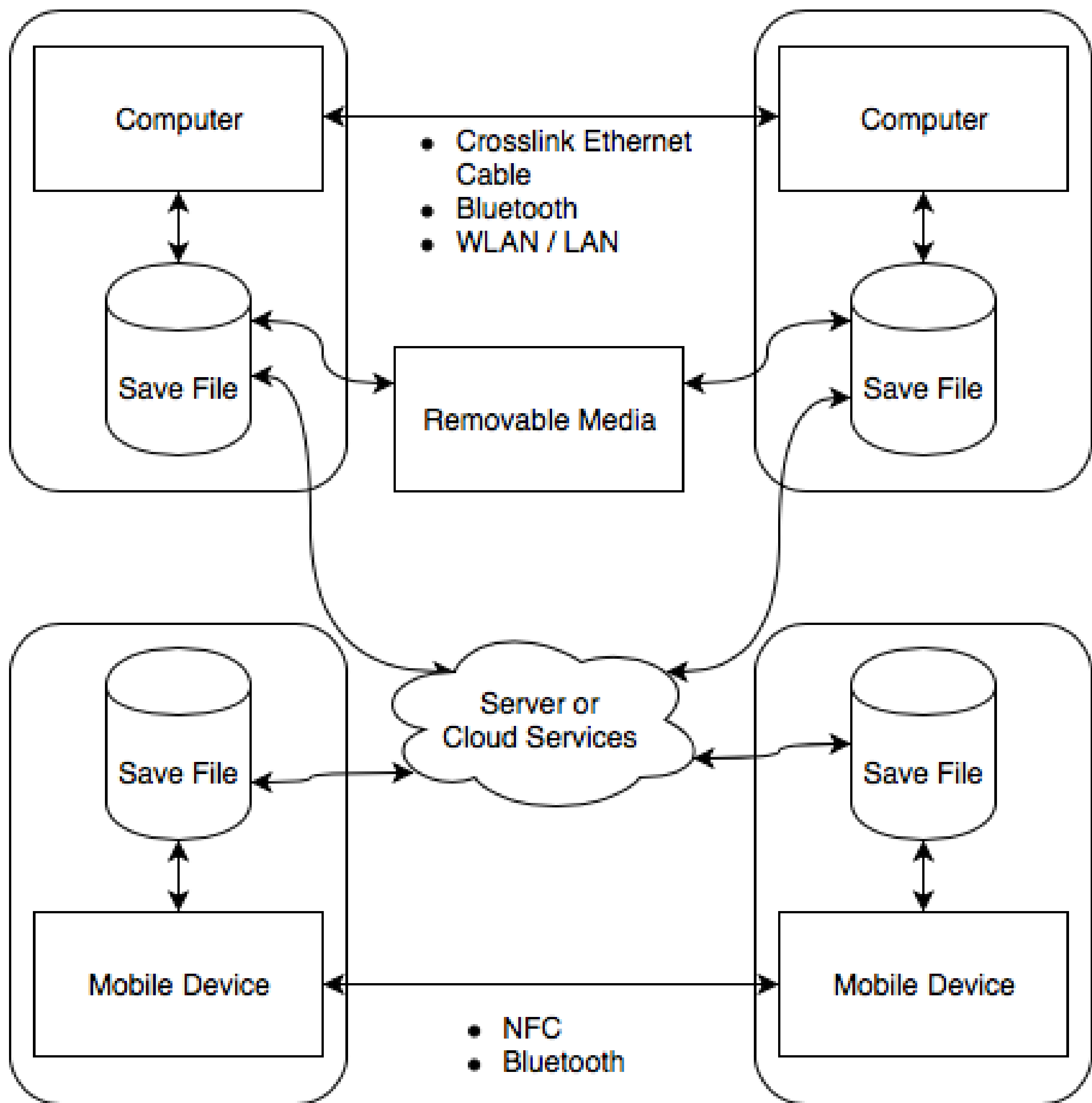

FIGURE 2. Different ways to distribute saved data between different devices

The save file of a game, which stores the information of the game state when saving, can be any kind of serialised data. When the game is launched again on the same or on another device, the game program should be able to read the data and restore the previous game state. Problems can arise if the save data is corrupted somehow.

The figure 2 shows several different ways for devices to share their saved data. Early on, computers could transfer data between them using some kind of removable media, such as floppy disks. Game consoles would have their own proprietary cartridges, which could sometimes be transferred between similar consoles.

Removable media is still used to transfer data between computers but with most computers being connected to the Internet, most such data would be transferred over the Internet using different protocols. This also makes it possible to share game data between different kinds of devices. Depending on the game, it could be possible to play it on a PC and later continue the same game on a mobile device. One of such cross-platform games is Blizzard's Hearthstone.

Mobile devices can also sometimes share data between them using NFC (Near-field Communication) or Bluetooth. While it can be convenient in some cases, it is even more limited compared to using a physical media to transfer data. The device users would need to be physically in the same space. This limitation can, however, also produce a positive effect. Games synchronise data during runtime between two mobile devices and two people can play the same game using their own devices, while still sharing the experience in the same physical space.

2.2 Commercial and contractual considerations

Mobile game distributors, such as the App Store, require that the developer provides a way for the user to restore any purchases they have made through the game. This means that the data for the purchases needs to reside on somewhere other than locally on the user's device. This ensures that the purchase is not device dependent and users will not lose what they have bought in case they switch devices. Other situations where restoring might be needed is if something happens to the user's device or if they uninstall, and later reinstall, the game on their device. (3.)

Storing data just locally can also be a commercial liability. Even mobile games can be hacked using different hacking tools. An example of a game that can be hacked is the popular Imangi Studios' Temple Run. By using a hacking tool, a user could access and change the locally stored game data. A hacker could add an unlimited amount of in-game currency into their game and use it to unlock features that would otherwise cost real money. (4.)

Games, such as Supercell's Clash of Clans, have made this impossible. The game will connect to their server every time it's launched and will ensure that the locally stored data is in synchronisation with the data stored on their servers. The only way to get more in-game currency is to make an actual purchase.

GDPR has also created new legislative issues that game developers need to be aware of when gathering data. Any personal data collected from users within the EU needs to conform with the GDPR legislation. (5.)

2.3 Saving data in Unity3D

The Unity3D game engine offers different ways to save persistent data. Additionally, there are different commercial third-party solutions.

2.3.1 DontDestroyOnLoad()

In Unity3D, the content of the game can be split into different scenes. As an example, one level in a game can be one scene. As the game progresses, certain data needs to be shared between these scenes. As an example, the current score might need to be saved during the runtime. This information can be saved into a particular game object which is loaded during each scene.

When a new scene is loaded, Unity3D destroys the currently loaded game objects and creates new game objects that are associated with the new scene being loaded. The DontDestroyOnLoad() -function tells Unity that the object should not

be destroyed when the new scene is loaded. This way the data stored in the object will be accessible in the new scene as well.

When using DontDestroyOnLoad(), it is important to remember that the scene where the game object was originally created might be loaded again. In this case, one might end up with two of the same game object. Therefore, it is important to use the Singleton-method to ensure that there will only ever be one instance which uses the DontDestroyOnLoad() method. (6.)

2.3.2 Player preferences

Player preferences, or PlayerPrefs in Unity3D, stores data locally in the file folder of the game. The purpose of PlayerPrefs is to store small data (e.g. preferred settings for resolution or the high score of the game) between gaming sessions.

By default, data stored in PlayerPrefs is not encrypted. This can become a problem if PlayerPrefs is used to store sensitive data, such as the amount of in-game currency. A user with sufficient abilities can “hack” into the file and change the information, thus bypassing the requirements to obtain this currency. (7.)

PlayerPrefs data can be encrypted using third-party assets or by creating a proprietary encryption solution. However, this does not ensure that the data would not be corrupted. (8.)

2.3.3 XML and JSON

Unity3D offers native support for JSON serialisation. XML is supported through the built-in XML parser of the C# language.

```
<root>
    <Player>
        <Name>Game Player</Name>
        <Level>18</Level>
    </Player>
    <Score>
        <HighScore>1000</HighScore>
        <Player>Game Player</Player>
        <LastScore>465</LastScore>
    </Score>
    <Preferences>
        <Resolution>1920x1080</Resolution>
        <FullScreen>true</FullScreen>
    </Preferences>
</root>
```

FIGURE 3. An example of the XML format

The figure 3 shows an example of the XML format. Due to the HTML like tags used, there is a bit of redundant data. This is not an issue with the less verbose JSON as shown in the figure 4.

```
{
    "root":{
        "Player":{
            "Name":"Game Player",
            "Level":"18"
        },
        "Score":{
            "HighScore":"1000",
            "LastScore":"465"
        },
        "Preferences":{
            "Resolution":"1920x1080",
            "FullScreen":"true",
        }
    }
}
```

FIGURE 4. An example of the JSON format

Both XML and JSON files are human readable. They can be designed to contain key-value pairs that are also understandable, but the data could also be obstructed by using names for keys which do not make sense to the user.

2.3.4 Binary files

The issue with the human-readable XML and JSON files is that when they are stored locally, a user can access them and change the values outside of the game. This can create issues with cheating or even game breaking situations when the values are not recognisable by the game. This can be prevented by storing the information in binary files.

In Unity3D, it is possible to store data in binary files by using the BinaryFormatter class of the C# language.

Although binary files are not human-readable, and thus harder to edit successfully, it should be pointed out that it is still possible for a skilled user to do so.

2.3.5 Other solutions

While Unity3D does not support them natively, INI files (initialisation files) are also a common way to store user data in games. INI files are human readable text files with key-value pairs, such as *Resolution=1920x1080*.

There are also 3rd party solutions made for Unity3D. These can be found at the Unity3D Asset Store. These assets can be convenient when there is a need to save the game state of large, complex worlds. (9.)

3 CREATING THE LEVEL EDITOR

3.1 Requirements

The main requirement for the level editor was that it could be used for creating, storing and restoring game levels. The levels themselves do not have to be actual playable levels in the prototype version.

A fully functioning level editor should enable the players to create their own versions of the game levels. Future development options would include testing the level in a separate play mode. Only levels that can be played through by the level creator can be saved and shared with others. This would prevent sharing levels which are impossible to complete. This kind of design is used, for example, in Nintendo’s Super Mario Maker.

3.2 Level editor functionality

The prototype of the level editor has a simplified UI for testing purposes. The most important function of the prototype is to be able to place objects and store information about the placed objects and their position. The level editor is also used for the purpose of loading previously saved level information.

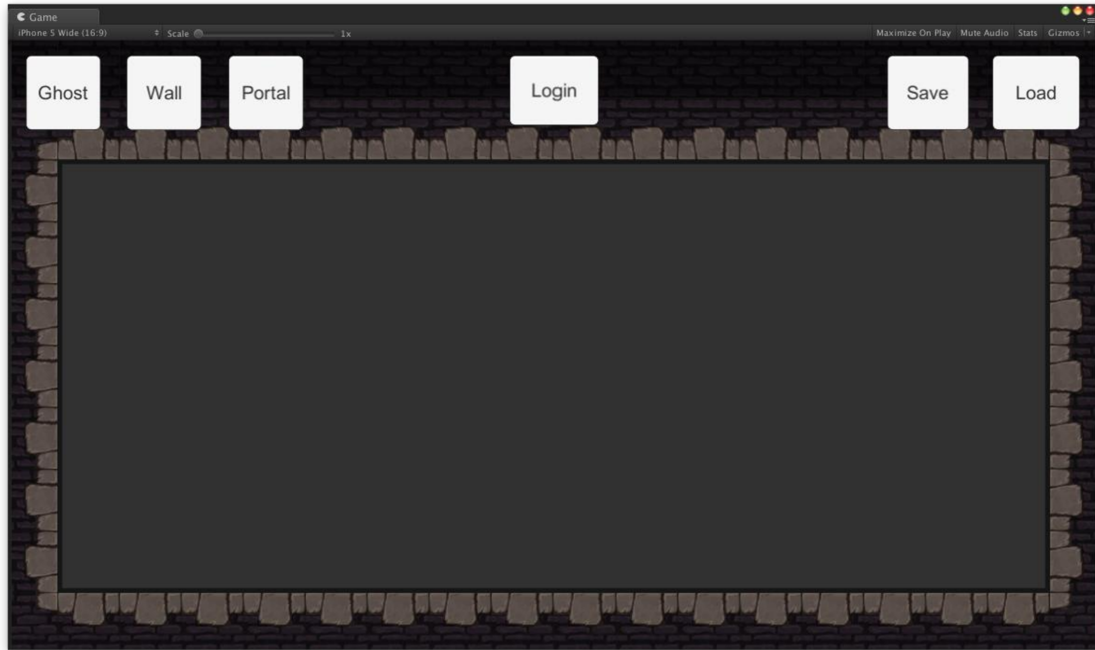

FIGURE 5. The level editor prototype

As shown in the figure 5, the buttons for different objects are located on the upper left corner. The area where the objects can be placed is the empty space with a grey background in the middle of the level editor screen. If an object is placed outside of this area, the object will be destroyed.

The login button in the UI was for testing login into the Firebase server using an e-mail address and password. In future development, this would be replaced with a different kind of authentication system.

The save button on the upper right corner is used when the user wants to save the level data. The load button is used for loading an existing level data.

Placing of objects in the game editor is described in the figure 6.

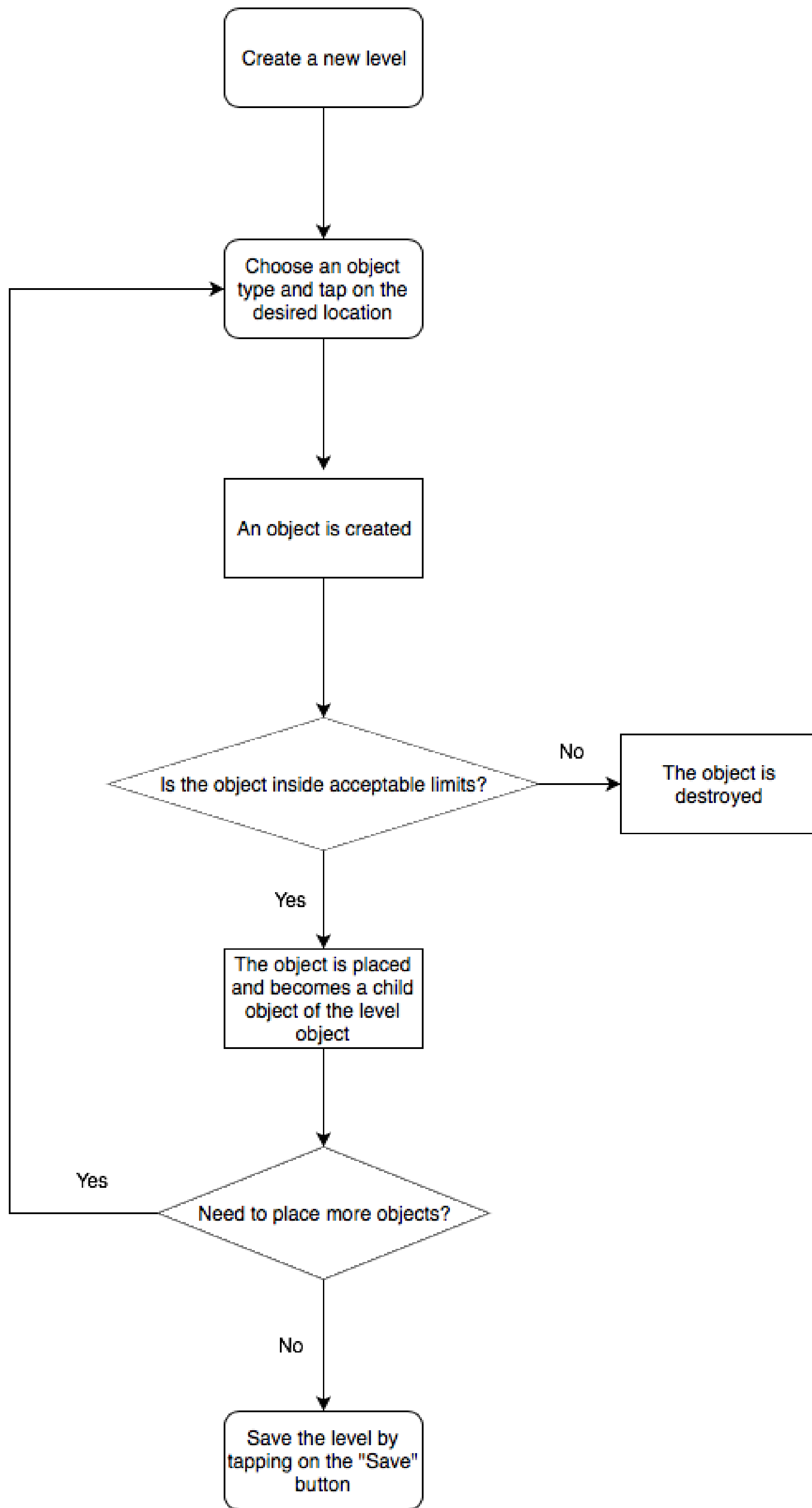

FIGURE 6. A flow chart of how the level editor creates new objects

When a user places a new object in the editor, an actual game object is created in the level editor scene and placed as a child object underneath a game object called *Level*. The hierarchy as shown in the Unity3D editor can be seen below in the figure 7.

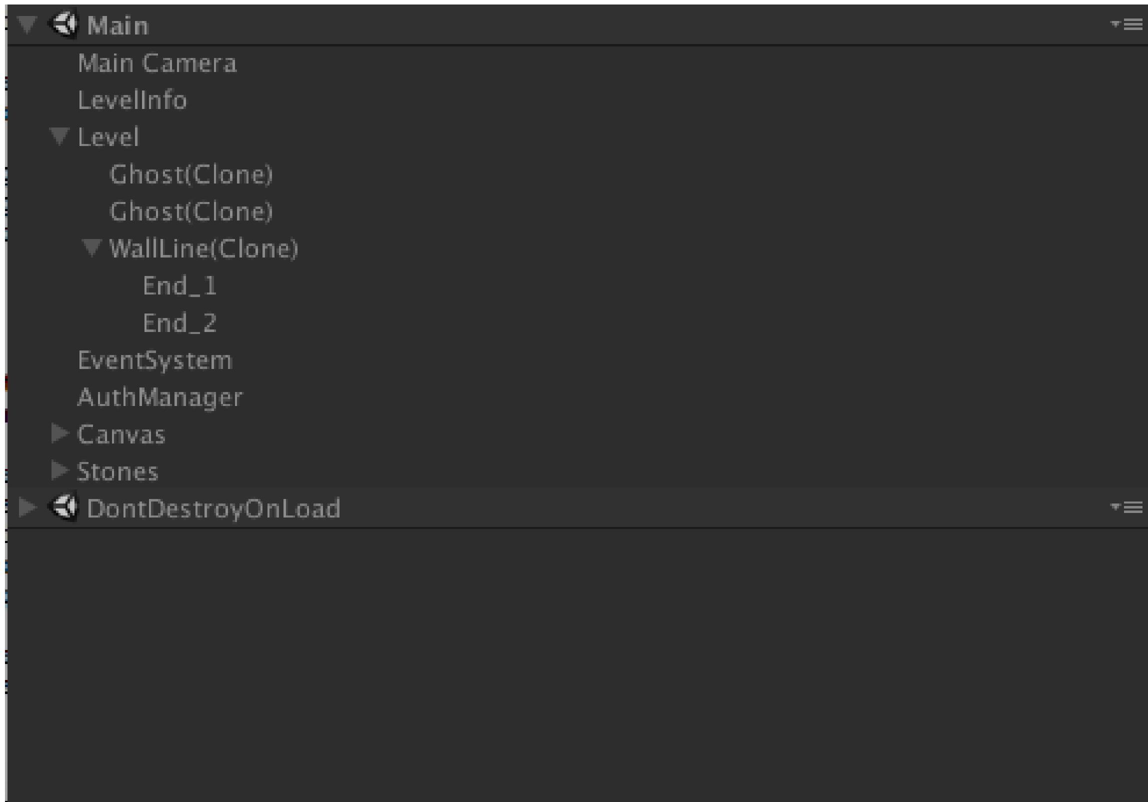

FIGURE 7. The hierarchy of the game objects in the level editor scene

3.2.1 Level objects

Since the objects in the existing game were fairly simple, static objects, some new ones were created for the sake of testing and to support future development.

The objects that can be placed in the level editor are as follows:

- Ghost - a static object
- A wall - an object which has a starting point and an ending point
- A portal - two separate objects which are conceptually linked together

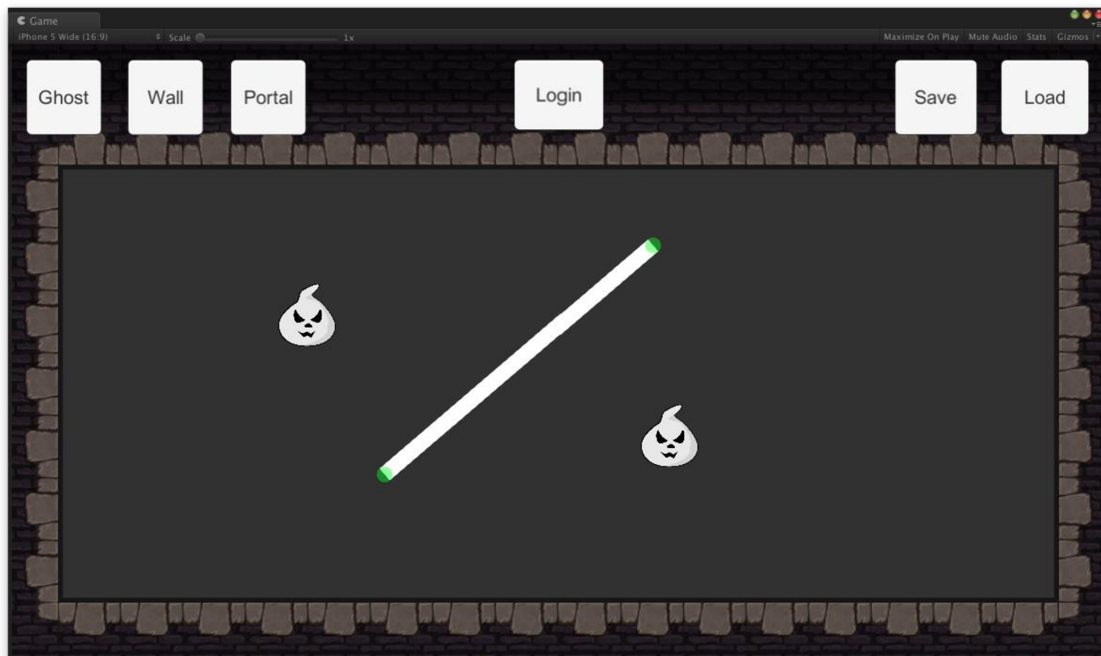

FIGURE 8. The level editor after two ghost objects and one wall object has been placed

Placing a static object is fairly straightforward: the user presses on the object button and then taps on where they want to place the object.

When placing a wall object, the user first presses on the corresponding button and then taps on where they want to place the wall object. After the wall is placed,

the user can then hold their finger on either end of the wall and drag the end point to where they want the wall to begin or end.

The portal object ended up not being part of the prototype. It would have been very similar to the wall object and the level editor development had already taken quite a lot of time.

3.3 Serialising level object data

Different objects within the level require different types of information to be stored. A singular, static object only needs information about what type of object it is and where it is positioned. The wall object requires information about where the object begins and where it ends, i.e. there have to be two different positions. The portal object is technically similar to the wall object as it also requires information of two different positions.

3.3.1 Custom serialisation

Since the information stored is fairly simple, a custom serialisation method was first developed. This was used for testing the functionality of the editor and saving data locally.

As shown in the figure 9, each object had an identifier such as “ghost”, followed by its x and y position. The information for each object was delimited by a comma. Each object was delimited by a semicolon. The whole string was stored in a text file. When the level was loaded, the string in the text file was parsed and the objects were created according to the information in the file.

```
ghost,-3.921875,0.3720486;
ghost,0.1465282,3.237326;
ghost,4.468577,0.07899284
```

FIGURE 9. An example of the custom serialisation

The custom serialisation method was not very elegant, but it worked, and it produced a small file size. The file size could have been reduced even further if the object identifier would have been just a number instead of a word. Another way to further reduce the file size would have been to design the level editor to have a grid. If the object can only be placed in a grid, the x and y coordinates

could have been described by using integers, instead of the larger floating-point values.

3.3.2 Using JSON for serialisation

Since the chosen cloud service provider, Google Firebase, uses JSON for storing data and Unity3D already offers JSON serialisation, it was decided that the serialisation should be done using JSON. While the custom-made string could have also been stored on the Firebase server, using JSON seemed like a more elegant solution compared to creating and parsing the string values. A further study could be made on what would be the actual performance differences between the custom-made solution and the native Unity3D JSON serialisation, when part of a more complex level editor.

An example of a level stored using the level editor prototype can be seen in the figure 10.


```
{
    "dataList": [
        "{\"objType\":0,\"objPos\":{\"x\":-0.21970559656620027,\"y\":0.24093663692474366,\"z\":2.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":0.04754361882805824,\"y\":-0.028926176950335504,\"z\":2.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":0.2656461000442505,\"y\":0.08075053989887238,\"z\":2.0}}",
        "{\"objType\":2,\"objPos\":{\"x\":-0.04537254944443703,\"y\":0.2841036021709442,\"z\":2.0},\"startPos\":{\"x\":
4.466002941131592,\"y\":0.048967257142066959,\"z\":-9.0},\"endPos\":
{\"x\":-5.744858741760254,\"y\":-14.92019271850586,\"z\":-7.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":-0.21970559656620027,\"y\":0.24093663692474366,\"z\":2.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":0.04754361882805824,\"y\":-0.028926176950335504,\"z\":2.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":0.2656461000442505,\"y\":0.08075053989887238,\"z\":2.0}}",
        "{\"objType\":2,\"objPos\":{\"x\":-0.04537254944443703,\"y\":0.2841036021709442,\"z\":2.0},\"startPos\":{\"x\":
4.466002941131592,\"y\":0.048967257142066959,\"z\":-9.0},\"endPos\":
{\"x\":-5.744858741760254,\"y\":-14.92019271850586,\"z\":-7.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":-0.21970559656620027,\"y\":0.24093663692474366,\"z\":2.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":0.04754361882805824,\"y\":-0.028926176950335504,\"z\":2.0}}",
        "{\"objType\":0,\"objPos\":{\"x\":0.2656461000442505,\"y\":0.08075053989887238,\"z\":2.0}}",
        "{\"objType\":2,\"objPos\":{\"x\":-0.04537254944443703,\"y\":0.2841036021709442,\"z\":2.0},\"startPos\":{\"x\":
4.466002941131592,\"y\":0.048967257142066959,\"z\":-9.0},\"endPos\":
{\"x\":-5.744858741760254,\"y\":-14.92019271850586,\"z\":-7.0}}"
    ]
}
```


FIGURE 10. An example of a level stored in the JSON format

3.4 Using the interface method

Even though the prototype only has very simple objects, the plan was to design something that could be easily used with any future additions. For this reason,

each object handles its own data serialisation, which is invoked through a common interface.

As seen in Figure 11, an interace class ISaveLevelData has two functions: GetObjectJSON(), which returns the object specific JSON as a string and RestoreObject(string objJSON), which restores all the needed information for the recreated object during the level loading phase.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public interface ISaveLevelData {
6
7     string GetObjectJSON ();
8     void RestoreObject(string objJSON);
9 }
```

FIGURE 11. A code sniplet of the interface used

3.5 Saving object data

When the user has placed all the desired objects in the level editor, they press on the “Save” button. Pressing the button executes a function called SaveData(), which is located in the Serializer.cs script.

```
29    public void SaveData(){
30        LevelData sdl = new LevelData();
31        sdl.dataList = new List<string> ();
32
33
34        //käydään läpi jokainen levelin lapsiobjekti ja lisätään listaan
35        foreach (Transform child in level.transform) {
36            objects.Add (child.gameObject);
37        }
38
39        //käydään läpi jokainen objekti listasta ja haetaan interfacen kautta JSON string
40        //mikä lisätään datalistaan
41        for (int i = 0; i < objects.Count; i++) {
42            ISaveLevelData isld = objects [i].GetComponent(typeof(ISaveLevelData)) as ISaveLevelData;
43            sdl.dataList.Add (isld.GetObjectJSON ());
44        }
45
46        //koko datalista tallennetaan JSONiksi
47        string json = JsonUtility.ToJson (sdl, true);
48        Debug.Log(json);
49    }
```

FIGURE 12. A code sniplet of the how the level data is saved

As seen in Figure 12, the SaveData() function creates a new LevelData object and a new list for storing strings. It then loops through each child object of the Level game object and adds them to a list. This list is then looped through. Each object has a common interface, which enables the script to get the data of the corresponding object. After the data of each object has been fetched and stored in a list, this list is then stored as a JSON file using Unity3D's JSON utility.

3.6 Loading a game level

When loading the saved game level, the JSON file will be read into a string variable. Using the Unity3D's JSON utility, the string variable is then turned into a new LevelData object with a new list of the saved game objects.

The list with all the objects of the game level is looped through and depending on the type of object, a corresponding new game object will be instantiated. This object will then be added as a child object of the level. It will then receive the information about its position through the ISaveLevelData interface.

The code for how the level data is stored is shown in the figure 13.

```
51    public void LoadData(){
52        string filename = Path.Combine(Application.dataPath, SAVE_FILE);
53        string jsonFromFile = File.ReadAllText (filename);
54        LevelData sdlout = JsonUtility.FromJson<LevelData> (jsonFromFile);
55
56        //per object restore
57        for (int i = 0; i < sdlout.dataList.Count; i++){
58            //ensiksi testataan objekti tyyppi
59            ObjectTypeClass otc = JsonUtility.FromJson<ObjectTypeClass> (sdlout.dataList [i]);
60            //tyypin perusteella instantioidaan uusi objekti ja lähetetään tälle JSON interfacen kautta
61            GameObject restoreObj = Instantiate (GetComponent<LevelInfo> ().objList [otc.objType]);
62            restoreObj.transform.parent = level.transform;
63            ISaveLevelData isld = restoreObj.GetComponent (typeof(ISaveLevelData)) as ISaveLevelData;
64            isld.RestoreObject (sdlout.dataList[i]);
65        }
66    }
```

FIGURE 13. A code sniplet of how the level data is restored

4 STORING LEVEL DATA IN THE DATABASE

4.1 Requirements

There was no detailed design on how the storing and sharing of the level data would happen between users. The main requirements were that the level data, in which ever form, could be saved directly from the game onto a server and that this level data could later be retrieved and the level recreated.

A good example of a game that would work like this would be RobTop Games' Geometry Dash. Both the paid mobile version and the PC version include a level editor and an option to search and play other people's levels. The free mobile version does not offer the editor nor the possibility of playing the user made content.

The figure 14 shows what the level editor looks like to the player in Geometry Dash. Users have the possibility to create new levels by clicking on the "Create" button or to browse for levels created by other users by clicking on the "Search" button.

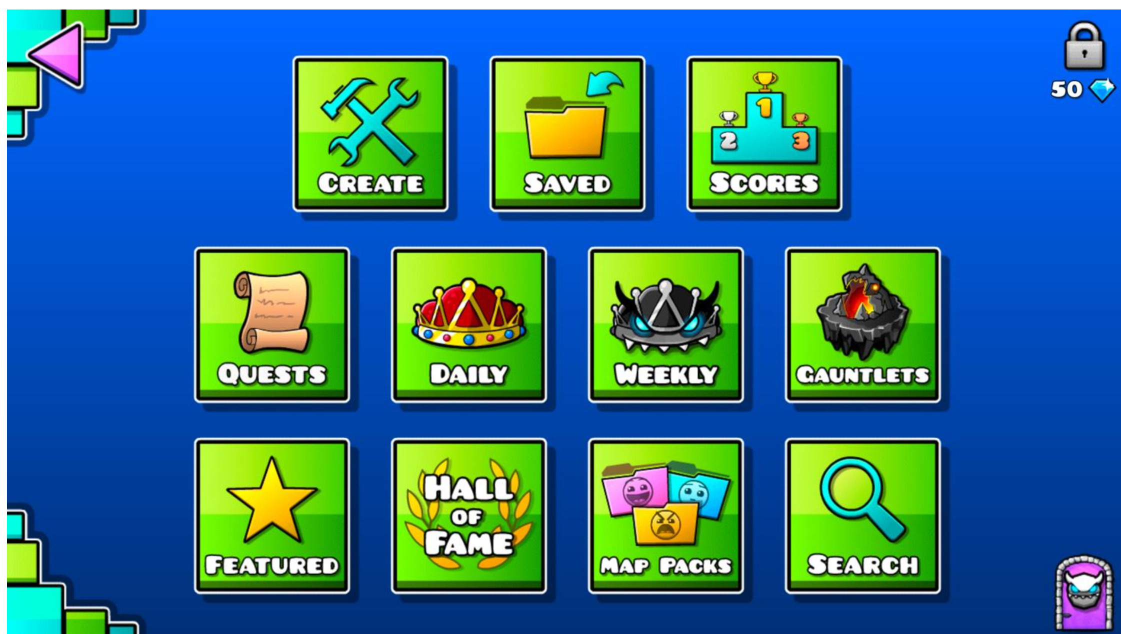

FIGURE 14. The main menu of the level editor in Geometry Dash (RobTop Games 2019, retrieved on 18.4.2019)

The figure 15 shows the different ways that users can search and browse for levels made by other users. When looking for the most downloaded or most liked levels, the game displays them as a paginated list, as shown in the figure 16.

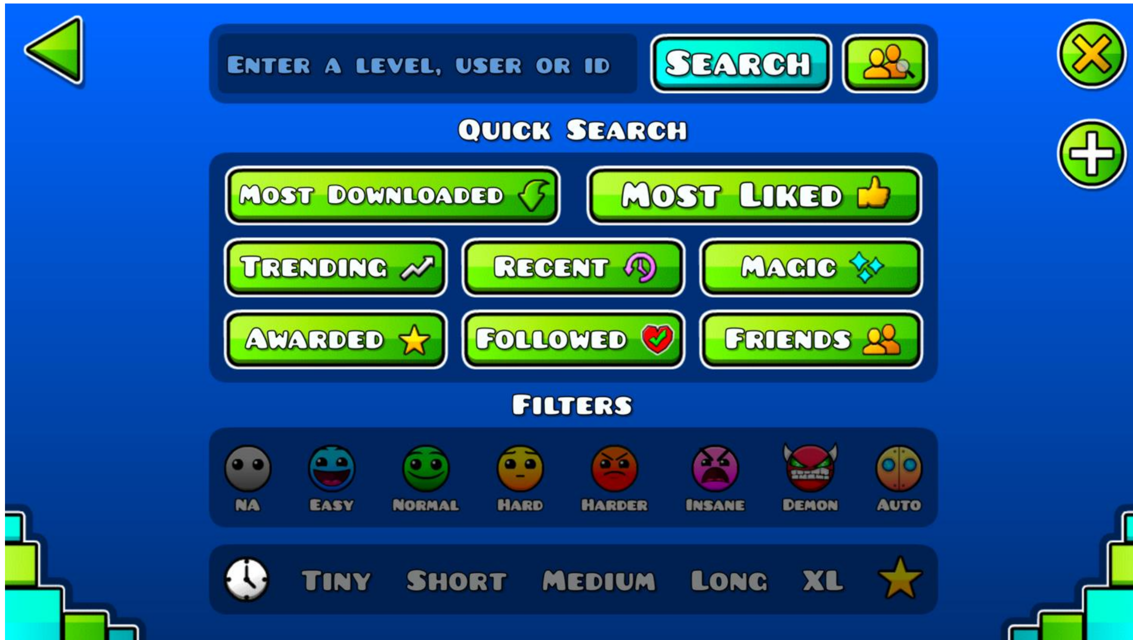

FIGURE 15. The different options for browsing levels on Geometry Dash (RobTop Games 2019, retrieved on 18.4.2019)

FIGURE 16. The third page of the most downloaded levels list (RobTop Games 2019, retrieved on 18.4.2019)

Since there was very little previous knowledge of how cloud computing works, it was important that the database maintenance would be fairly straightforward and would not require a lot of focus. Building a fully functioning backend with commercial viability would have been a large project of its own. For now, the focus was on inspecting how Unity3D communicates with a suitable backend and how the data should be structured efficiently.

4.2 Using a BaaS

BaaS, an acronym for Backend-as-a-Service, is a cloud-based service model. It allows software and application developers to connect to backend systems through an API. A backend service aimed at mobile developers can also be called MBaaS, or Mobile Backend-as-a-Service.

BaaS often includes services like cloud storage, push notifications, user and file management, social networking integration and user management.

4.3 Choosing the service

Different service providers were considered to be used for storing the level data on a server. The main criteria for the service were as follows:

- being able to store and retrieve data in a game made with Unity3D
- free tier for testing and for a game that does not have many players yet
- ease of use and maintenance; something that one person can handle
- safe to use

The service providers were evaluated. The results of the evaluation can be seen in the table 1.

TABLE 1. Comparisons of the evaluated cloud service providers

Service provider	**Positives**	**Negatives**
Amazon AWS	huge amount of different services possibilities for expansion	significant learning curve difficult to determine costs
Google Cloud	easier to understand than Amazon AWS	only usable for registered companies within the EU
Google Firebase	designed to be used with games and has an SDK for Unity3D easy to use	currently only for mobile and web player games client driven, not server driven

Microsoft Azure		big learning curve
Heroku	cheap	not a lot of ready-made solutions

In the end, Google Firebase was chosen mainly because of the ease of use. All other services would have required a lot of time to be spent on learning how to use and create cloud services. For just one person that would have been too big of an overtaking, especially considering that the service has to have proper security measures when taken to production.

Google Firebase offers solutions that are aimed at game developers and has built-in authentication. At the time of writing this thesis, Google Firebase was designed to only work with mobile and web-player games made in Unity3D. This was acceptable, as the game was designed to be released as a mobile game.

4.4 Google Firebase products

Products and services offered by Google Firebase are mainly aimed at mobile application developers. Firebase offers integration when developing apps with Swift, Objective-C, Java, JavaScript, C++ or in Unity3D. In Unity3D, Firebase is only supported when developing games for iOS, Android or web. This means that if the game is later released on PC, Mac or on a game console, Firebase cannot be used as a backend system.

Google Firebase offers pricing tiers for different needs. The free "Spark Plan" was used for the prototype developed in this thesis.

Out of the different services offered, Realtime Database and Authentication was used with the prototype created during this thesis.

4.4.1 Realtime Database

The Firebase Realtime Database is a NoSQL database hosted in Google Firebase’s cloud. Data can be synchronised between the cloud database and the client so, that even if the client goes offline, they can still access the data.

4.4.2 Authentication

Google Firebase offers different types of authentication for identifying end-users as shown in the figure 17. For the sake of simplifying the process, the prototype created during this thesis uses email and password pairs for authentication as shown in the figure 18.

A possible end-product would most likely use Game Center on iOS and Google on Android for authentication.

The purpose of the authentication is to allow end-users to create and manage their own level designs. Authentication can also be used for other purposes, such as storing information on any in-app purchases.

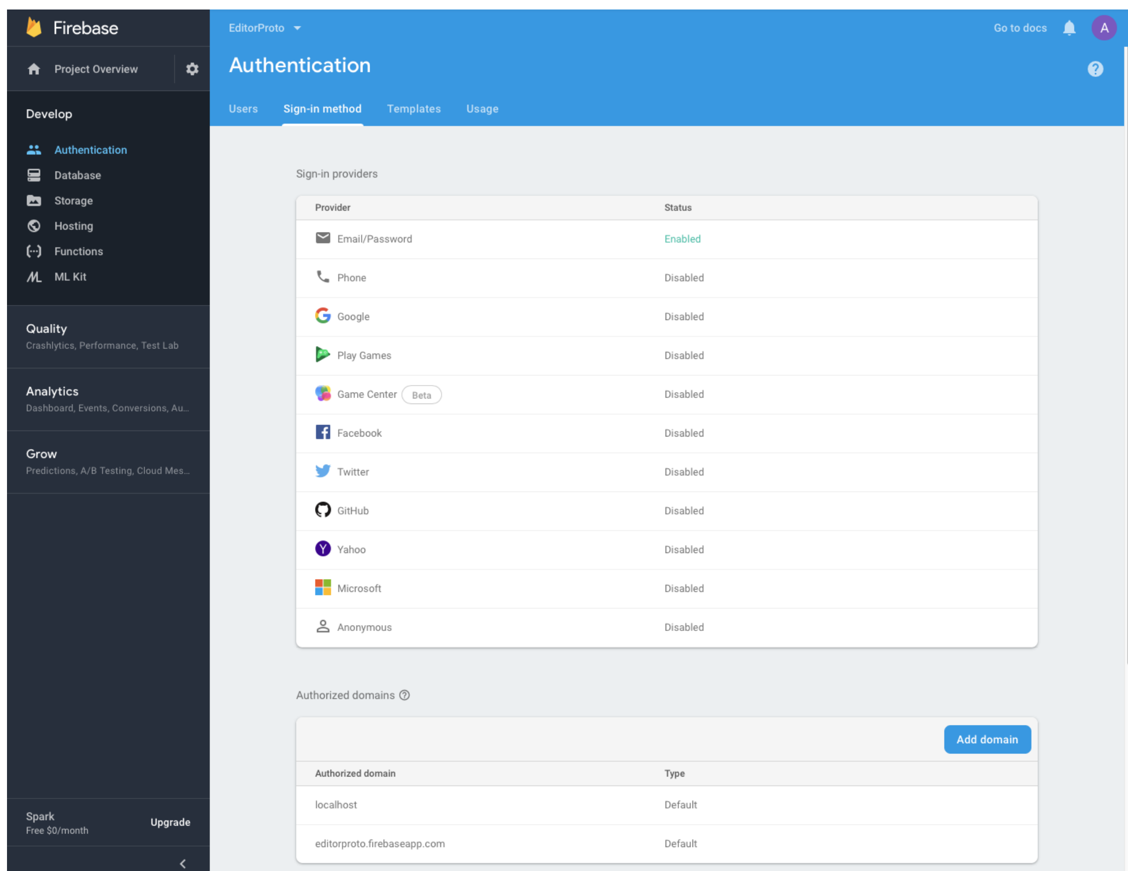

FIGURE 17. Authentication options in the Google Firebase dashboard (Google 2019, retrieved on 5.4.2019)

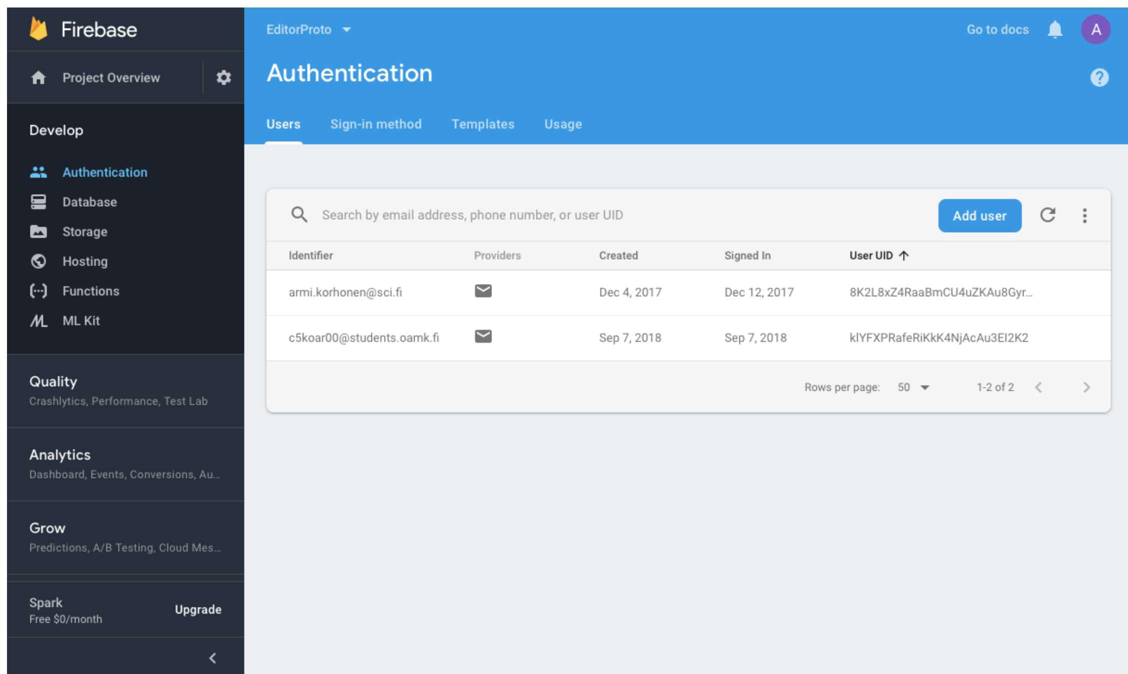

FIGURE 18. Two registered e-mail addresses shown on the Google Firebase dashboard (Google 2019, retrieved on 5.4.2019)

4.5 Integrating Firebase with Unity3D

Most of the Firebase services can be integrated with native iOS or Android applications. The game used in this thesis was developed with the Unity3D game engine, thus the services that can be used are more limited. The services that can be used with Unity3D are Realtime Database, Cloud Storage, Authentication and Cloud Functions. (10.)

When integrating the Firebase SDK, a tutorial from Lynda.com (now known as LinkedIn Learning) was used as a guide through the process. The tutorial stored different kind of data but it was helpful in understanding how to use the authentication and storing the data in the Realtime Database. (11.)

4.6 Structuring the data

Designing the data structure is one of the most challenging parts of creating and using a database. This became evident as the lack of clearly defined requirements made it very difficult to plan how the data structure should be designed. A tentative research was still made to see what kinds of possibilities using the Firebase Realtime Database would offer.

When designing a relational database, data redundancy should be avoided when possible. This is not necessarily the case with the Firebase Realtime Database. The database is a JSON tree, which should be kept as flat as possible. This means that even though the database JSON tree can have up to 32 levels of nested data, every time a node is retrieved, all its child nodes are also retrieved.

The security of the Realtime Database heavily relies on rules, which are combined with authentication. If a person has access to one node, they will have access to all the child nodes as well. To avoid too much nested data, some data needs to be duplicated. This enhances the security of the database, as well as the performance. (12.)

Because of these requirements for the data structure, it was decided that the user data would be in its own node and the levels in a separate node. If each user's own levels would have only resided within the user nodes, other users would not have been able to access that data without also having access to other user data. This would have posed a potential security risk.

Having all the level data nested within a dedicated node produced further problems. The original idea was to let users get, for example, 10 random new levels created by other users. The Firebase SDK does not offer a way to access a random node within the database. It also does not offer a neat way to know how many nodes there are, thus creating a random number on the client side would not have worked either.

Another idea was to just get the first 10 levels from a list of levels and later the next 10 and so on. This could be done by using the startAt() and endAt() queries. This would be similar to how Geometry Dash lists levels and paginates the lists. The issue with this is that people would again download a lot of data without necessarily using it. In a premium game, such as Geometry Dash, the usage has already been paid in advance. When designing a free to play game with only advertising monetisation, all cloud data usage should be minimal or offered as a reward for watching advertisements.

One way to get the levels would be to create a list with all the level names, without the level data. This list would then be downloaded to all devices. Each device would then locally keep track of which levels they have played and retrieve new level data based on the list. The issue with this approach is that each device would download a large amount of data, regardless of if they would end up playing the game or not. Since the game would be a free to play game, there would be no guarantee that the advertising revenue would cover the costs of the Realtime Database service.

4.7 Security considerations

Due to the way Google Firebase has been designed to work, certain security measures should be taken into consideration before moving into production. The reference to the database and its API key are stored on the client. A knowledgeable hacker can access that information and use it for accessing and tampering with the database, as demonstrated by Elliot Alderson, who hacked the “Donald Daters App”. Because of this, the way to secure the database is to use the Firebase Realtime Database Rules. (13.)

The Firebase Realtime Database Rules determine who can read and write data in the database, what indexes exist and how the data is structured. By default, the rules do not allow anyone to access the database. When creating a new database, it is also possible to start in "test mode", which enables all access to the database, as shown in the figure 19. Using the test mode can make it easier

to focus on developing the data structure and the client-side functionality. However, before moving into production, the rules should be set to only allow authorised clients to access their own data.

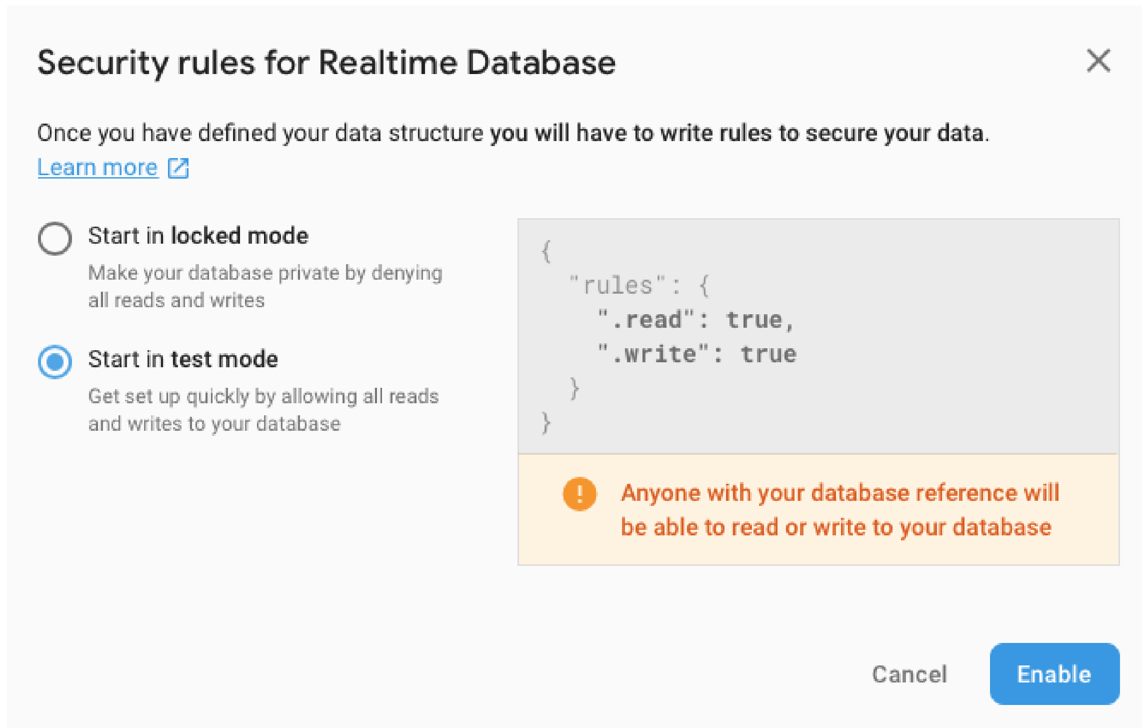

FIGURE 19. Security rules for Realtime Database (Google 2019, retrieved on 7.4.2019)

5 CONCLUSION

In general, the thesis gave a good insight into what would be required to develop a game with a level sharing mechanic. It did not, however, give a clear answer to how it should be done. Just designing the data structure would have been a large undertaking, worthy of a thesis of its own.

Using the Unity3D's native JSON serialisation seemed like a very clear and easy way to store data in games. The stored data size could be made smaller by using a custom serialisation method. However, a further research should be made to determine whether the custom serialisation would be more efficient than the native JSON serialisation when used in a simple mobile game.

While storing the data to and retrieving it from a Google Firebase Realtime Database is fairly simple, sharing the levels as originally planned did not seem as straightforward. Unlike a relational database, a JSON database does not work well with complex search and filtering algorithms. In general, a much more elaborate planning is needed when designing the data structuring for a mobile game backend.

One of the main concerns of using cloud services for sharing the user created content was the service costs. In the case of a premium mobile game, where the user pays for downloading the game, it is already known how much revenue each user will create. The money is also paid before the service will be used. This is not the case with the so called freemium and free to play games. Freemium games are free to download and play, but they offer in-app purchases as a way to monetise the game. The risk with these games is that a large portion of the users might never spend money on the game, but would still use the cloud computing, which costs money.

As the original plan was to publish "Interdimentional Nuisance" as a free to play game with only advertisements as its monetisation, this would produce a conundrum on how to make it financially feasible to use cloud computing while allowing the users to play for free. Since the users would "pay" by watching video

advertisements, the cloud computing usage should be designed so that most usage would be only allowed after watching a video advertisement.

Whether it would be feasible to include level sharing using cloud services in a free to play game would require a very clear plan on what actually would be offered and how.

REFERENCES

1. Korhonen, A. 2016. Interdimentional Nuisance. Retrieved on 06.04.2019 https://northernerd.itch.io/interdimentional-nuisance

2. Williams, A. 2017. History of Digital Games. Focal Press. Retrieved on 16.04.2019 from http://proquest.safaribooksonline.com.ezp.oamk.fi:2048/book/programming/game-programming/9781317503804/altering-time-in-home-console-games/sec115_html?uicode=ouluuas

3. Apple Inc. 2019. App Store Review Guidelines. In-App Purchase. Retrieved on 05.05.2019 https://developer.apple.com/app-store/review/guidelines/#in-app-purchase

4. Ilindra, A. 2019. Temple Run Cheat – Get Unlimited Coins and Gems. Retrieved on 05.05.2019 https://www.geekdashboard.com/temple-run-cheat/

5. General Data Protection Regulation. Regulation (EU) 2016/679.

6. Jakupec, Z. 2015. Saving Data Between Scenes in Unity. Retrieved on 05.05.2019 https://www.sitepoint.com/saving-data-between-scenes-in-unity/

7. Zucconi, A. 2015. A practical tutorial to hack (and protect) Unity games. Retrieved on 05.05.2019 https://www.alanzucconi.com/2015/09/02/a-practical-tutorial-to-hack-and-protect-unity-games/

8. Fatih, R. 2014. SecurePlayerPrefs. Retrieved on 05.05.2019 https://github.com/rawandnf/SecurePlayerPrefs

9. Unity Technologies. 2019. Unity Asset Store. Retrieved on 05.05.2019 https://www.assetstore.unity3d.com

10. Google Ireland Limited. 2019. Firebase Products. Retrieved on 16.04.2019 https://firebase.google.com/products/

11. Ferrone, H. 2017. Unity: Working with Google Firebase. Retrieved on 05.04.2019 https://www.linkedin.com/learning/unity-working-with-google-firebase/firebase-101

12. Google Ireland Limited. 2019. Understand Firebase Realtime Database Rules. Retrieved on 07.04.2019 https://firebase.google.com/docs/database/security/

13. Alderson, E. 2018. How I “found” the database of the Donald Daters App. Retrieved on 05.05.2019 https://medium.com/@fs0c131y/how-i-found-the-database-of-the-donald-daters-app-af88b06e39a