Score: 95/100

National Park Builder

Coding Project Final Report

Prepared by

Viktor Kirillov
Mykyta Parovyi
Volodymyr Vakhniuk
Aleksandra Dmitrieva

for use in CS 440 at the
University of Illinois Chicago

November 2020

Table of Contents

List of Figures

List of Tables

I Project Description

1 Project Overview

National Parks Builder is a gaming application that allows users to build and manage a national park. You are responsible for building it from scratch by selecting each building and placing it whenever you want on your imaginary island. Earn revenue from guests coming to your park, keep the excitement points up so that more guests come, complete daily tasks and get rewards for that, visit other islands and look how nice the other parks are as well as yours.

2 Project Domain

Domain of the project is as follows: involve and show people national parks in a way that they may have a desire to come to a real national park. As we wanted to make the whole game process look like managing the national park, some people from business spheres might want to play this game as well.

3 Relationship to Other Documents

This document is a final coding project report based on the National Parks Builder [1] project development report.

4 Naming Conventions and Definitions

4a Definitions of Key Terms

Guest: represents a virtual guest (within a game) that is visiting the island and generates in-game revenue.

Excitement (or Excitement score): an integer value used to represent the popularity of the island, which directly reflects how many guests are coming to the user's island.

DB (or db): refers to a database.

4b UML and Other Notation Used in This Document

This document generally follows the Version 2.0 OMG UML standard, as described by Fowler in [2]. Any exceptions are noted where used.

4c Data Dictionary for Any Included Models

Models in our project are "**Buildings**" and "**Player**".

Buildings Model provides us a suitable way of getting such data as: *all building objects* and *building images*.

Player Model is much complicated model which has:
Information of a single Player - all data about a current Player.
Guesting player information - all data about a guesting Player.
Profile info of a specific Player - all info about a specific Player.
Leaderboard Players - Getting the first 10 players already sorted by excitement score.
All tasks info of a current Player - Getting all the tasks associated to a Player
Building process of a single building - Set a Building to a Player that had built it.
Guest Revenue event - Adding revenue amount to a Player Balance.
Visit Revenue Event - Adding the revenue based on the last time the player entered the game.
Setting a ticket price - Changing the ticket price that guests pays for entering the park.
Setting Camera spawn and Resetting Player's Island - Setting the camera spawn of a current Player on their island and ability to reset their island completely (delete all the buildings and change the generated terrain to a new one)

Player's park excitement value = $building_1$.excitement + $building_2$.excitement + … + $building_n$.excitement, where $building_i$ is an object of a building, placed in player's park.

Guest visit chance = 100 - (ticket price * 100 / 30)

II Project Deliverables

1 First Release

- C++ development environment is completed.
- Both GUI and Wrappers for its classes were developed.
- Big switch from C++ to Javascript environment transition done.
- Basic UI layout and elements were developed.
- Game and Home basic Scenes functionality were implemented.
- Basic Controllers were developed and connected with UI.
- Rendering setup for WebGL was done.
- MVC architecture was brought to the project.
- Database was configured and connected to the models.
- UI loader component was developed.

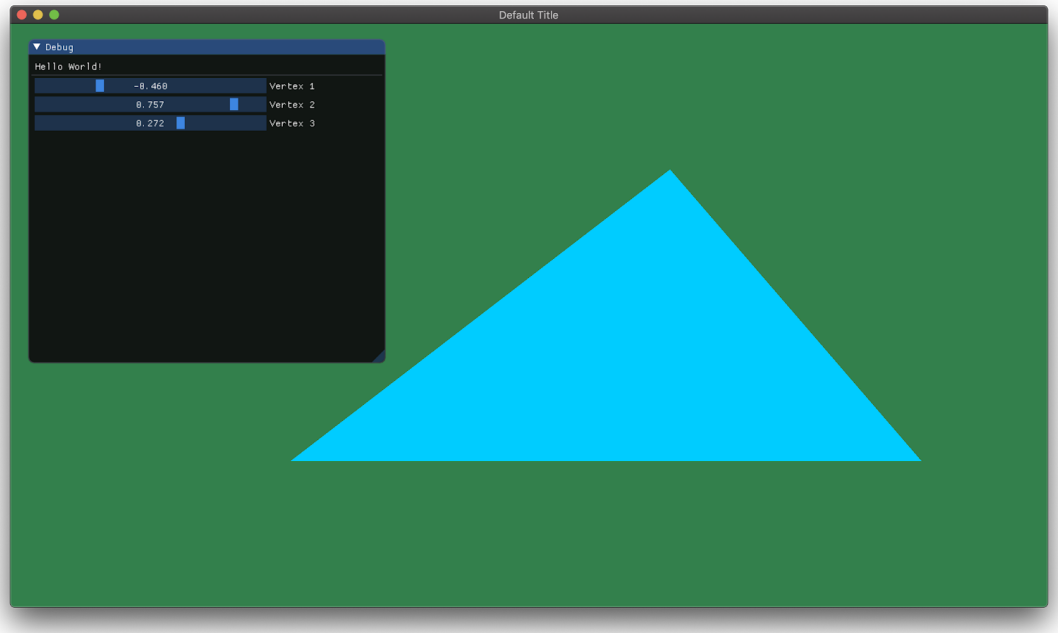

Figure 1 - User View of the App in C++ environment setup.

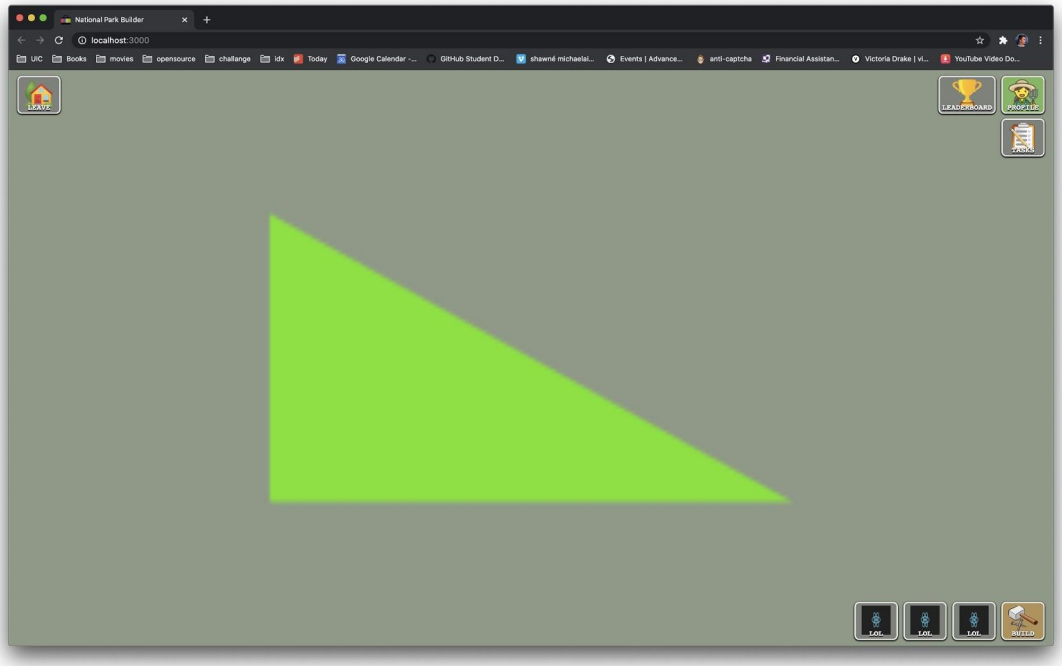

Figure 2 - User View of the App after transitioning to the Javascript environment.

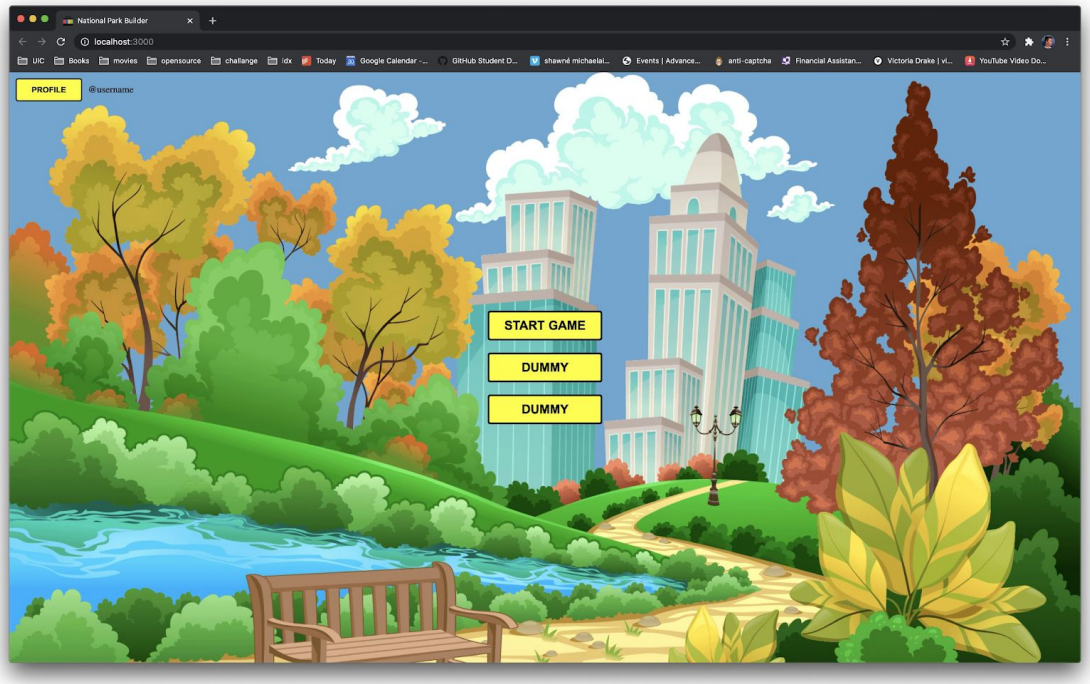

Figure 3 - User View of the App during 1st Release.

2 Second Release

- WebGL is set up.
- Real map is being rendered.
- Island generator (Perlin Noise) implemented.
- Camera is working correctly.
- Appropriate models and textures found and configured.
- User - Map interactions implemented.
- Guest system implemented.
- Ticket system implemented.
- Leaderboard implemented.
- Guest attendance chance dependency on excitement level implemented.
- Reset Island and Set spawn camera implemented

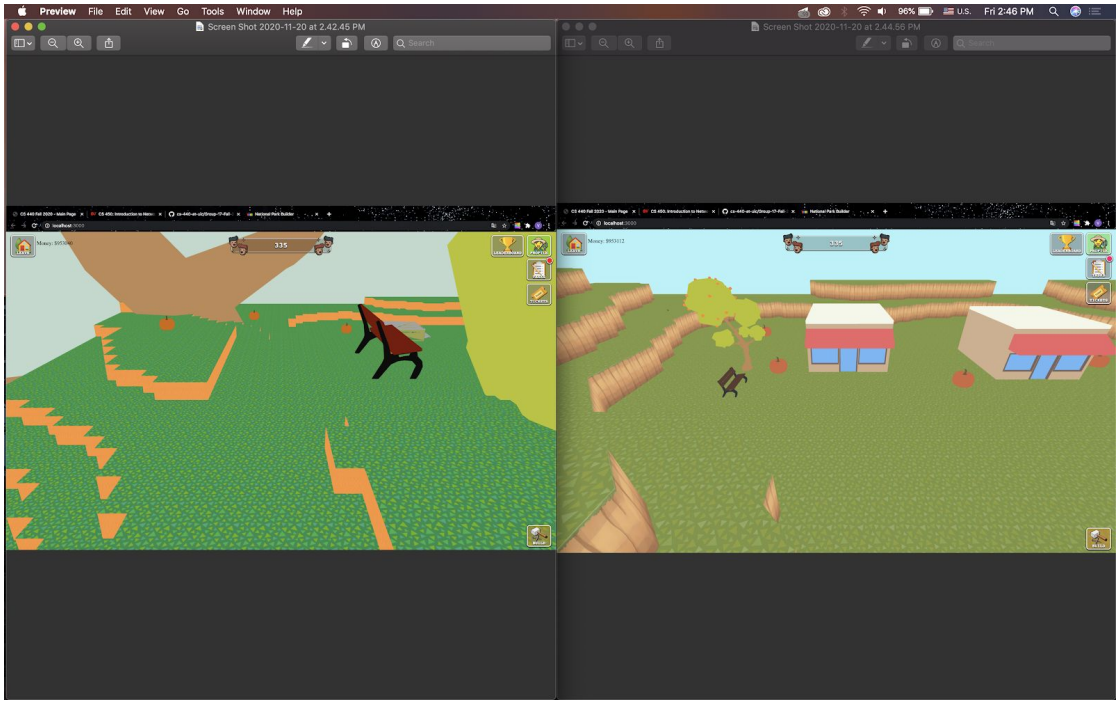

Figure 4 - Differences between the user view of 1st Release and 2nd Release.

Figure 5 - Final user view of the Application.

3 Comparison with Original Project Design Document

Things that were done in a different way:

- The park is fully virtual (not a real park in the US), thus it does not represent any real park and any similarities between the park in our game and real park is unintended. Moreover, we did not include the feature of selecting the park's location from the list of real locations from the US. **[ref. 1.1]**
- The employees were not implemented along with their morale level. Although, we developed the following substitution: excitement points. Players can gain it by building different buildings, and the more they build, the more excitement points they earn. Depending on the excitement score, the number of virtual visitors to the park is going to vary. The more excitement score you have, the more virtual guests you are going to attract. **[ref. 1.1]**
- The fire, donations, cleaning area and (item) microtransactions features are not included in the game. **[ref. 1.4b.]**
- The targeted platform was Mobile, although our game is available on web browsers. **[ref. 1.6b.]**

III Testing

1 Items to be Tested

I. Database Interactions
   A. Stable DB connection.
   B. Checking Ticket revenue from guest "coming" action.
   C. Getting people ordered by excitement points amount.
   D. Setting Last Visit entry of a person after each player game session.
   E. Getting all the buildings objects from the DB.

II. UI & Controllers functionality
   A. Building button functionality (first phase)
   B. Accept/Decline Building buttons functionality (second phase)
   C. Tickets (setting new ticket price [drag])

2 Test Specifications

ID 1.B - Checking Ticket revenue from guests "coming" action.

Description: Test the real value of a ticket price in a database of a user to add to their balance.

Items covered by this test: User ticket price, user balance

Requirements addressed by this test: Adding correct Ticket price to a User balance and checking if that's the balance that is supposed to be.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A

Test Procedures: Run the db along with the game, log into it and wait for the virtual guest to come. (guest coming action should be fired)

Input Specification: Player's ticket price and balance entries in the db.

Output Specifications: Test should produce the correct sum from adding the ticket player price and balance.

Pass/Fail Criteria: For the test to pass the resulting sum should be correct as specified in the ***Output Specifications*** above.

ID 1.E - Getting all the buildings objects from the DB.

Description: Getting all the buildings objects from DB

Items covered by this test: Building data objects

Requirements addressed by this test: Returning all the buildings data objects.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A

Test Procedures: Run db along with the server and wait for any client (game) incoming request for getting all the available buildings data.

Input Specification: All buildings data objects entries in db

Output Specifications: The test should produce the right amount of data objects and retrieve all their properties as it is in the database.

Pass/Fail Criteria: For the test to pass the resulting buildings data objects should be as it is specified in the ***Output Specifications*** above.

ID 1.D - Setting Last Visit entry of a person after each player game session.

Description: Setting the Last visit entry of a user

Items covered by this test: User's last visit entry value

Requirements addressed by this test: Setting the last visit entry as it is specified in the request and checking it to be valid.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A

Test Procedures: Run db along with the server and wait for any client (game) to log in and start playing.

Input Specification: Specific User's last visit entry

Output Specifications: Test should produce the updated version of a last visit entry as specified in the request (value should be the last seen timestamp represented as seconds from the 1970 year).

Pass/Fail Criteria: For the test to pass the resulting last visit entry should be as it is specified in the ***Output Specifications*** above.

ID 1.C - Getting users ordered by excitement points.

Description: Getting first 10 users ordered by excitement points in decreasing manner

Items covered by this test: Users entries and their excitement scores

Requirements addressed by this test: Getting first ten users from db and correctly sorting them by their excitement score, returning the resulting data.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A

Test Procedures: Run db along with the server and wait for any client (game) to request for the leaderboard users.

Input Specification: All users from the DB

Output Specifications: Test should produce the list of users that are correctly ordered by excitement score in decreasing manner (starting from the highest).

Pass/Fail Criteria: For the test to pass the resulting output should be as it is specified in the ***Output Specifications*** above.

ID 2.A - Building button functionality.

Description: Clicking the Building Button and Rendering all the building options for the user.

Items covered by this test: Building data and options being displayed to a user that is rendered based on that data.

Requirements addressed by this test: Getting all the building data objects and correctly rendering all the available options to a user.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A and 1.F

Test Procedures: Run db along with the server and wait for any client (game) to click the build button.

Input Specification: Options being rendered to a user and buildings objects data incoming from the server.

Output Specifications: Test should produce the correctly displayed options that are generated based from the received data from the server.

Pass/Fail Criteria: For the test to pass the resulting output should be as it is specified in the ***Output Specifications*** above. Tester should manually check if there are enough options being displayed and data in them is valid and corresponds to the data received from the server.

ID 2.B - Accept/Decline Building buttons functionality.

Description: Testing the placement accept/decline functionality in order to make sure that the data is being either stored or just discarded in the db of a particular user.

Items covered by this test: User's entry of buildings they have and their excitement score.

Requirements addressed by this test: Adding a new building if an "accept placement" event has occurred, else current app state should be discarded and switched to its previous state. If a user accepted the building placement, their excitement score should be increased accordingly with the building excitement score value.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A

Test Procedures: Run db along with the server and wait for any client (game) to place and build any building.

Input Specification: All users from the DB

Output Specifications: Test should result in rendering exactly that building that user built and reflect the updated excitement score change by displaying the latest updated score to the user.

Pass/Fail Criteria: For the test to pass the result should be as it is specified in the ***Output Specifications*** above.

ID 2.C - Setting the ticket price.

Description: Testing whether setting ticket price event reflects changes in the db and correctly updates the ticket price entry of a specific user.

Items covered by this test: User's ticket price entry in db and displayed price in the ticket window.

Requirements addressed by this test: Client should get the updated value of a price for a ticket after changing it in the ticket window.

Environmental needs: Firebase DB running along with the app.

Intercase Dependencies: 1.A

Test Procedures: Run db along with the server and wait for any client (game) to change their ticket price.

Input Specification: Updated numeric value from the client.

Output Specifications: Test should result in correctly updated user ticket price value to be displayed and reflected in the user's entry in the db.

Pass/Fail Criteria: For the test to pass the resulting output should be as it is specified in the ***Output Specifications*** above.

3 Test Results

ID 1.B - Checking Ticket revenue from guest "coming" action.

Date(s) of Execution: November 23rd 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results: Ticket Price $17 + Current player balance $10357 = $10374

Actual Results: $10374

Test Status: Pass.

ID 1.C - Getting people ordered by excitement points amount.

Date(s) of Execution: November 24rd 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results:

- admitr3@uic.edu - 1559
- liooren4625@gmail.com - 275
- nickparov@gmail.com - 201
- vvakhn2@uic.edu - 68

- bahehsaldeen@gmail.com - 52
- gueejla@gmail.com - 41
- vladimir18045@gmail.com - 7
- tkvapi2@uic.edu - 2
- matvei.parovoy@gmail.com - 2
- admitr.sv@gmail.com - 2

Actual Results: Same as the expected

Test Status: Pass.

ID 1.D - Setting Last Visit entry of a person after each player game session.

Date(s) of Execution: November 22nd 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results: 1606446701

Actual Results: Same as the expected

Test Status: Pass.

ID 1.E - Getting all the buildings objects from the DB.

Date(s) of Execution: November 23rd 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results: Complete array of building data objects as it is in DB.

Actual Results: Same as the expected.

Test Status: Pass.

ID 2.A - Building button functionality.

Date(s) of Execution: November 22nd 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results: Figure 6

Figure 6 - Test expected user view: rendered building options.

Actual Results: Same as the expected.

Test Status: Pass.

ID 2C - Setting the ticket price.

Date(s) of Execution: November 22nd 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results: Figure 7

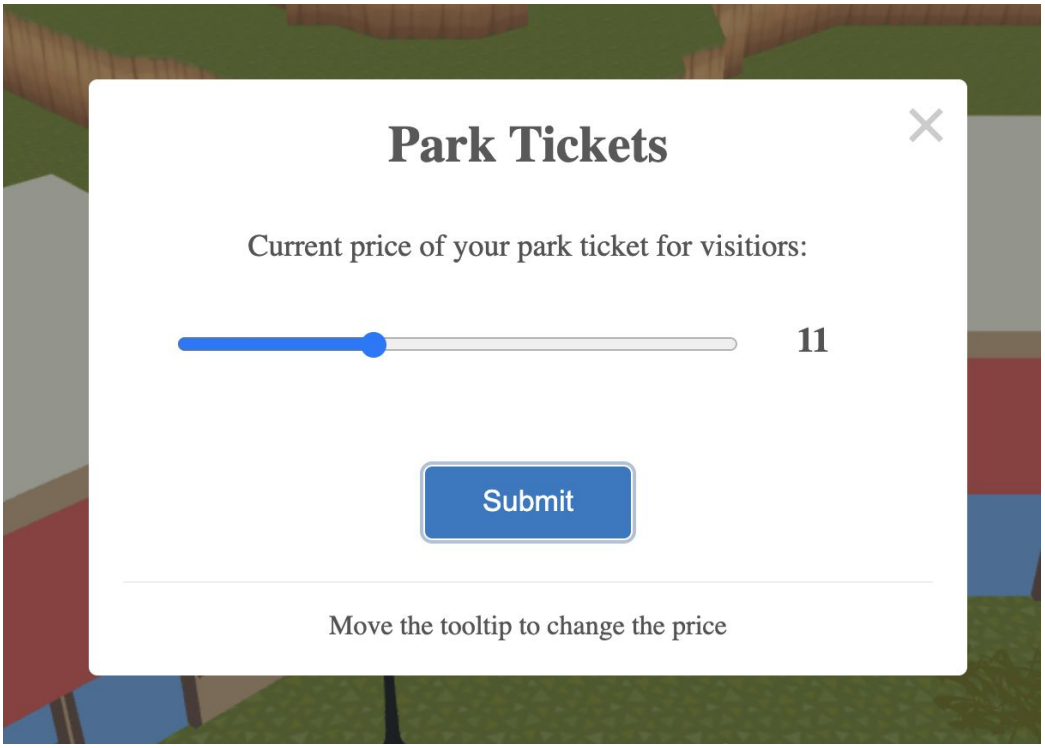

Figure 7 - Test expected user view: Park ticket updated price.

Actual Results: Same as the expected.

Test Status: Pass.

2.B - Accept/Decline Building buttons functionality.

Date(s) of Execution: November 25th 2020.

Staff conducting tests: Mykyta Parovyi and Alexandra Dmitrieva

Expected Results: Building should be placed and saved in db or if the reject button was clicked, the current state should be discarded and switched back to the previous state.

Actual Results: Same as the expected.

Test Status: Pass.

4 Regression Testing

None of the tests should be run in the future. Only if the updates on the specific items are made, only those of them should be run again that check the specific update to be working correctly.

IV Inspection

1 Items to be Inspected

```
_taskProcesser(uid, action, amount) {
    return new Promise((resolve, reject) => {
        this._getTasks(uid).then(currentTasks => {
            let promises = []
            for (let i=1; i<=numTasks; i++)
                if (currentTasks[`task${i}`].action == action) {
                    promises.push( admin.database().ref(`users/${uid}/tasks/task${i}/progress`).set(currentTasks[`task${i}`].progress + amount) )

                    // If user completes the task for the first time
                    if (
                        currentTasks[`task${i}`].progress < currentTasks[`task${i}`].count &&
                        currentTasks[`task${i}`].progress + amount >= currentTasks[`task${i}`].count
                    ) {
                        promises.push(
                            new Promise((resolve, reject) => {
                                admin.database().ref(`users/${uid}/excitement`).once('value').then(snapshot => {
                                    snapshot.ref.set(snapshot.val() + currentTasks[`task${i}`].excitementReward).then(() => resolve())
                                })
                            })
                        )
                    }
                }

            Promise.all(promises).then(() => {
                resolve()
            })
        })
    })
}
```

Figure 8 - Viktor's Piece of Code

```
__getInterpolatedNoise(x, z)
{
    let floorX = Math.floor(x);
    let floorZ = Math.floor(z);

    let s = this.__getNoise(floorX, floorZ);
    let t = this.__getNoise(floorX + 1, floorZ);
    let u = this.__getNoise(floorX, floorZ + 1);
    let v = this.__getNoise(floorX + 1, floorZ + 1);

    let res1 = this.__cosineInterpolate(s, t, x - floorX);
    let res2 = this.__cosineInterpolate(u, v, x - floorX);
    let res3 = this.__cosineInterpolate(res1, res2, z - floorZ);

    return res3;
}
```

Figure 9 - Volodymyr's Piece of Code

```
function checkLastVisit() {
    return new Promise((resolve, reject) => {
        const oldMoney = Global.gameSceneState.state.money;
        // add revenue accordingly
        Player.addVisitRevenue(Global.homeSceneState.state.user.idToken).then(res => {
            Global.gameSceneState.setState({
                ...Global.gameSceneState.state,
                money: res.newMoney
            })

            if(res.newMoney - oldMoney > 0 && !Global.isPopupOpened && !Global.isVisiting && !Loader.isShown())
                Toast.fire({
                    icon: 'success',
                    title: `While you were offline, you earned: ${res.newMoney - oldMoney}$`
                })

            resolve(res);
        });
    })

}
```

Figure 10 - Mykyta's Piece of Code

```
    // On building button press (displays a list of available buildings)
    onBuildBtn() {
        if (this.buildingsShowing) {
            // Remove building buttons from the scene
            Global.gameSceneState.setState({ ...Global.gameSceneState.state, buildingsToBuild: [] })
            this.buildingsShowing = false
        }
        else {
            // buildingsData: array of buildings data from the database
            Buildings.getAllBuildings().then(buildingsData => {

                // Prepare the data to render
                let buttonsData = Object.keys(buildingsData).map(buildingId => {
                    const building = {...buildingsData[buildingId], id: buildingId}
                    return {
                        text: building.name,
                        price: building.price,
                        image: Buildings.getBuildingImage(buildingId),
                        onClick: this.onExactBuildingBtn.bind(this, building)
                    }
                })

                // Sort by price
                buttonsData = buttonsData.sort((a, b) => b.price - a.price)

                // Render the data
                Global.gameSceneState.setState({ ...Global.gameSceneState.state, buildingsToBuild: buttonsData })
                this.buildingsShowing = true
            })
        }
    }
```

Figure 11 - Aleksandra's Piece of Code

2 Inspection Procedures

Code Review Checklist

A. Is code formatted in a way that it is overall understandable for all the team members?
B. Are the naming conventions fit to the language used?
C. Does the code have meaningful comments?
D. Does the code follow any architectural approach such as MVC?.

E. Does the code not disrupt the existing architectural structure?
F. Are there any hard coded values?
G. Are hard coded values represented or named in a way that indicates them being as constants?
H. Are there any existing relevant libraries/classes that are used instead of writing code from scratch?
I. Can the purpose of the given code be understood just by looking at it?
J. Are the code blocks concise and formatted the right way along with the proper indentations?
K. Is there any code duplication present in the provided piece of code?
L. Are there any unreadable or improperly named variables, methods of the class or values present in the provided piece of code?
M. Does the provided code have the proper error handlers?
N. Is the code secure enough to be safe from SQL Injections, XS and etc. ?
O. Does the code pose any threats to overall performance of the application?

The Checklist above was assembled by our team during the code review meetings that were held at least once per 10 days. All the meetings were held electronically.

3 Inspection Results

Table 1 - Inspection Results table

What was inspected?	Who did the inspection?	Date	Result of the inspection
Figure 8 - Viktor's Piece of Code	Mykyta, Volodymyr, Alexandra	10/16/2020	Code meets all the checklist requirements apart from the following revealed issues: Proper Exception handler should be added (reject block) along with more comments present in before each reference to the db in order to clarify the idea of what it is specifically requesting from it.
Figure 9 - Volodymyr's Piece of Code	Mykyta, Viktor, Alexandra	10/26/2020	Variable naming should be improved (res1, res2, res3) or comment describing the purpose of those variables before their declarations should be present. More comments should be present in order to clarify the purpose of the code. However, everything else in the code meets the checklist requirements.

Figure 10 - Mykyta's Piece of Code	Volodymyr, Viktor, Alexandra	11/06/2020	Proper Exception handler should be added (reject block), more comments should be provided along with an additional brackets "{}" in the if blocks. Apart from these issues, code meets all the checklist requirements.
Figure 11 - Aleksandra's Piece of Code	Volodymyr, Viktor, Mykyta	11/16/2020	Code is fully self-explanatory and easily readable, thus it meets all the checklist requirements.

V Recommendations and Conclusions

Judging by the results of the inspections above, the following changes should be made to the overall code of each team member:

- Improve variable naming, specifically rename such variables as res1, res2, res3 into something meaningful.
- Add such an amount of comments that clearly shows the purpose of the code.
- All the exception handlers should be added (reject blocks), in order to avoid possible errors in the future.
- Code should be rewritten in a way that describes itself. This change should be made in any possible piece of code as much as needed.

VI Project Issues

1 Open Issues

- Keeping the customers for longer at each entry to the game, which is a real issue due to the fact that for now we have the following triggers for the player to come back: revenue from being offline, guests coming to the park, daily tasks along with their rewards. We do not know if we need more of the same type triggers or just leave as it is but add something unique feature that would keep them in the game much effectively.
- Choice between leaning more towards game experience or National Park Simulation (straying from original idea) as we do not know exactly in which direction to lean. If we choose game experience direction, then we would expect longer play time and more popularity, although if we choose National Park Simulation (Builder) we might attract people into visiting National Parks much effectively.
- Availability to play on mobile platforms (IOS/Android). We do not know whether it is going to be comfortable to play the game on mobile with our current UI.

2 Waiting Room

A. Developing Better notification system (more interactions)
B. Adding Shadows and Textures
C. Paying for an entry when visiting other players' island that they set for the other players
D. Developing better economy and consistent player progression in game over time
E. Bringing in game items that give a slight boost that players can get from doing tasks and sell on in-game auction to the other players with some fixed price of in-game currency.

3 Ideas for Solutions

1. To develop idea **"A"** the existing notification system can be improved by adding tooltips and hint under each button upon the first entry to the game, adding task completion animation with a separate window which shows something like: "Congratulations on completing the daily task" followed by the reward the user earns, adding dialogues above the virtual guests based on the excitement points of the park.
2. To develop the idea **"C"** a separate button for each player can be added to the UI along with the needed controller function that will set the ticket price for other **REAL** players and store it in the database. Next step that would be made is adding an additional check for the player's ticket price of the park that the player is visiting to decrement that price from the current visiting player's balance. Also, the popup can be added to ensure that the player is ready to pay the other player's ticket price.
3. To develop the idea **"E"** different in-game items should be added that will inflict a significant impact on money income for the park for some particular amount of time. Those items can be given to a player after either completing a daily task for a certain amount of times or playing in the game for a certain amount of time as well. Auction should also be added to keep players for more and let them sell these different boosting items for their fixed price (in in-game currency).

4 Project Retrospective

Methods we used for completing this project didn't exceed the regular type of dividing the work methods. Alternatively, a large amount of attention was paid to the original document that we referred to during development.

VII Glossary

Firebase: a platform developed by Google for creating mobile and web applications, used as a Database in this project.

MVC: software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements.

VIII References / Bibliography

[1] "National Parks Builder" by Maaz Ahmed, Bryan Gutierrez, Ammar Idrees, Zaid Patel

[2] M. Fowler, UML Distilled, Third Edition, Boston: Pearson Education, 2004.

IX Index