Abstract

Mr.Maze is a multiplayer maze game focused on building an easy game with simple controls and rules, but with the interesting theoretical and practical part under the hood. Project implements different approaches on generating mazes with various difficulties. In this work we will go through every important part of our project, describe what does what and focus on some details. You will know about our backend part. We will show how we let users to interact with mazes. And, finally, we will describe our approach on UI/UX. Introduction will give a brief overview on paper, chapter about backend will cover processes on generating mazes and multiplayer, frontend chapter will focus on interaction with users, and design chapter will show our UI/UX vision.

Аңдатпа

Mr.Maze қарапайым басқару элементтері мен ережелері бар, бірақ негізінде қызықты теориялық және практикалық бөлігі бар жеңіл ойын құруға бағытталған мультиплеер лабиринт ойыны. Жоба әртүрлі қиындықтармен лабиринттерді құрудың әртүрлі тәсілдерін жүзеге асырады. Бұл жұмыста біз жобамыздың әрбір маңызды бөлігін қарастырамыз, не үшін қажет екенін сипаттаймыз және кейбір бөлшектерге назар аударамыз. Сіз біздің сервер туралы білесіз. Біз пайдаланушыларға лабиринттермен өзара әрекеттесуге қалай мүмкіндік беретінімізді көрсетеміз. Соңында біз UI/UX-ке деген көзқарасымызды сипаттаймыз. Кіріспе қағазға қысқаша шолу жасайды, артқы жағында лабиринттер мен көп ойыншы ойындарын құру процестері туралы айтылады, интерфейс пайдаланушылармен өзара әрекеттесуге арналған, ал дизайн UI/UX туралы біздің көзқарасымызды көрсетеді.

Аннотация

Mr.Maze - это многопользовательская игра-лабиринт, ориентированная на создание легкой игры с простым управлением и правилами, но с интересной теоретической и практической частью под капотом. Проект реализует различные подходы к созданию лабиринтов с различной сложностью. В этой работе мы рассмотрим каждую важную часть нашего проекта, опишем, что для чего нужно, и сосредоточимся на некоторых деталях. Вы узнаете о нашей серверной части. Мы покажем, как мы позволяем пользователям взаимодействовать с лабиринтами. И, наконец, мы опишем наш подход к UI/UX. Введение даст краткий обзор о работе, глава о бэкэнде расскажет о процессах создания лабиринтов и многопользовательской игры, глава о интерфейсе будет посвящена взаимодействию с пользователями, а глава о дизайне покажет наше видение пользовательского интерфейса.

Contents

Chapter 1

Introduction

1.1 Description

Mr.Maze is an online maze game. It creates mazes using procedural generation, so it will be challenging enough for players. Also users will be able to invite friends and play against them on generated mazes.

1.2 Motivation

We were inspired by Wordle game. It is an easy game with simple rules, so everybody can start playing it. Our case is similar. Everybody is familiar with mazes. Most likely You have played with mazes in childhood, it means You already know the rules. Our mission is only to give You a platform and content.

1.3 Thesis Outline

In this work we will go through every important part of our project, describe what does what and focus on some details. In Chapter 2 You will know about our backend part. In Chapter 3 we will show how we let users to interact with mazes. And, finally, in Chapter 4 we will describe our approach on UI/UX

Chapter 2

Backend

2.1 Microservices

Our backend designed in microservice architecture. In this section we will describe advantages and disadvantages of that approach

2.1.1 Pros

Load handling

Each microservice has own resources. It means we can control which of them is going to have most computational power and distribute resources depending on the service's needs

Developing

Developing a new microservice is always easy, because all other services, except the one which is being developed, can be developed independently. Team's work is parallel. So the development stage is shorter

Database

Microservices has own database, so access is not limited for whole system. In monolith access to database is limited to number of database replicas and connection pool and whole system is limited by that. In microservices we have the same limitations, but only for microservice.

2.1.2 Cons

Integration

As we said, mircoservices are good with databases, but in case we need to move data across services, it becomes a problem. Microservices' internal communication becomes a problem for system's network. Also, requests' idempotency becomes a problem, so we have to use message brockers to guarantee a recieve by another service. If we some data on different microservices are related, we will need to guarantee data's consistency, for us it means, we will load network with another bunch of requests between services

Bug tracking

If some service uses a lot of other services, it becomes hard to track bugs and handle them through all services. In such cases it is good to have some logger, which will collect logs from services and show us.

Refactoring

When we refactor microservices and change data representation or communication protocols, we should be ready for other services to be unable to communicate with current one

2.2 Authorization

2.2.1 JWT

In project we are using JWT (JSON Web Token [2]). Token is divided on three parts by dot.

- Header
- Payload
- Signature

Structure: **xxx.yyy.zzz**

Header

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA (For example see Figure 2.1). Then, this JSON is Base64Url encoded to form the first part of the JWT.

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data (For example see Figure 2.2). There are three types of claims: registered, public, and private claims. The payload is then Base64Url encoded to form the second part of the JSON Web Token.

Registered claims These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.

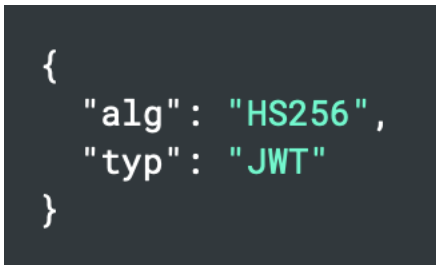
Public claims These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

Private claims These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

Signature

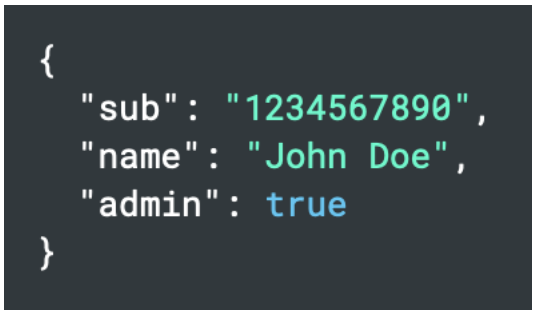
To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.



```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Figure 2.1: Header



```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Figure 2.2: Payload

Summary

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, so we can easily transfer information like user id, and permissions through this token, and be sure data verified, because it is signed.

2.2.2 Implementation

In our project we developed an authorization service which will sign tokens and refresh them. Also we have verification middleware on each service, so we can be sure token is valid and not expired.

When user credentials passed to server it hashes it and compares to hash we have in database for this user. We keep no plain text passwords and hash functions can't be reversed, so they can not be leaked. In case everything okay it returns token string.

We are using jwt-go package to sign token.

2.3 Database

For our project we chose Postgres [3] database because we are familiar with it, also it is open source, that's why we can use it for free.

SQL databases provide fast data access and data modification. We are not going to delete often or change table structure, so there is no need to use collection based databases like Mongo.

2.4 Maze generation

In this chapter we will explain some structures and approaches to make it clear how we produce valid maze with almost linear time complexity (We will talk about this "almost" a bit later)

2.4.1 Disjoint Set Union

First what we need is disjoint set union (DSU). It is a structure that stores indexes and the set it belongs to. Initially all indexes belong to set with only that index, so it is a representative of this set. DSU can say which set does exact index belong to and can unite two sets.

Data structure

In this code we see DSU structure with three fields.

```
type DSU struct {
   Size   int
   parent []int
   rank   []int
}

func NewDSU(size int) *DSU {
   ret := &DSU{
      Size:   size,
      parent: make([]int, size),
      rank:   make([]int, size),
   }

   for i := 0; i < size; i++ {
      ret.MakeSet(i)
   }

   return ret
}
func (d *DSU) MakeSet(v int) {
   d.parent[v] = v
   d.rank[v] = 0
}
```

- Size - number of indexes
- parent - an index which is also is in that index's set
- rank - approximated height of tree, in case it is a perfect binary tree, used for optimization. (It is a thing that makes time complexity almost constant, because we will compress tree by reassigning a parent node every time we get a query to find set)

NewDSU function is a constructor, here we are declaring data according to size and setting them an initial value, which is 0 rank and set is same as number of node.

MakeSet method sets node's values to default

Methods

Here we will go through data structure's two methods

```
func (d *DSU) FindSet(v int) int {
   if v == d.parent[v] {
      return v
   }

   d.parent[v] = d.FindSet(d.parent[v])
```

```
    return d.parent[v]
}

func (d *DSU) UnionSets(a, b int) {
    a = d.FindSet(a)
    b = d.FindSet(b)

    if a != b {
        if d.rank[a] < d.rank[b] {
            a, b = b, a
        }
        d.parent[b] = a
        if d.rank[a] == d.rank[b] {
            d.rank[a]++
        }
    }
}
```

- FindSet - returns a set of node. It goes up through parent array at exiting it reassigns parent value, so each time we make a request we make it method faster
- UnionSet - unites two set. First it find representatives of two set. Then it checks their ranks and assigns a set with bigger rank to the one with the smaller.

Time complexity

Time complexity analysis shows that rank heuristic optimization and path compressing gives us $O(\alpha(n))$ per query. $\alpha(n)$ is an Inverse Ackermann function and it is not exceeding 4 for $n <= 10^{600}$, that why we can consider it as constant. For any additional information it is reasonable to check out Princeton lecture on DSU [5]

2.4.2 Eller's algorithm

This algorithm [1] for maze generating works row by row and it is crazy, because on output we will have perfect maze. Each point can be reached with exactly one unique path. Algorithm:

1. Initialize the cells of the first row to each exist in their own set.
2. Now, randomly join adjacent cells, but only if they are not in the same set. When joining adjacent cells, merge the cells of both sets into a single set, indicating that all cells in both sets are now connected (there is a path that connects any two cells in the set).

3. For each set, randomly create vertical connections downward to the next row. Each remaining set must have at least one vertical connection. The cells in the next row thus connected must share the set of the cell above them.

4. Flesh out the next row by putting any remaining cells into their own sets.

5. Repeat until the last row is reached.

6. For the last row, join all adjacent cells that do not share a set, and omit the vertical connections, and you're done!

Step 1

```go
row = make([]string, width*2+1)
row[0] = "|"
row[width*2] = "|"
row[width*2-1] = "_"

dsu := domain.NewDSU(width)
```

Here we initialize a row and DSU. (width is twice bigger just for usability. We will go through odd indexes, so it makes no sense to algorithm itself)

Step 2

```go
rand.Seed(time.Now().UnixNano())
for i := 0; i < width-1; i++ {
   row[2*i+1] = " "
   if dsu.FindSet(i) == dsu.FindSet(i+1) {
      row[2*i+2] = "|"
   } else {
      res := rand.Intn(10)
      if res > 3 {
         row[2*i+2] = " "
         dsu.UnionSets(i, i+1)
      } else {
         row[2*i+2] = "|"
      }
   }
}
```

Here we go through cells and decide if we are adding a wall between them or not.

Step 3

```go
sz := map[int]int{}
for i := 0; i < width; i++ {
   sz[dsu.FindSet(i)]++
}
open := map[int]int{}
for k, v := range sz {
   open[k] = rand.Intn(v)
```

```
    }
    for i := 0; i < width; i++ {
       cur := dsu.FindSet(i)
       if open[cur] == sz[cur] {
          row[2*i+1] = "_"
       } else {
          res := rand.Intn(10)
          if res > 3 {
             row[2*i+1] = "_"
          } else {
             row[2*i+1] = " "
          }
       }

       sz[cur]++
    }

    maze.Rows = append(maze.Rows, strings.Join(row, ""))
```

Here we count number of cells in each set and generating a number between 0 and number of cells, so we choose an index at which wall downwards will not appear, after that we go through cells and adding a wall with 0.6 chance

Step 4

```
    newDsu := domain.NewDSU(width)

    sets := map[int]int{}

    for i := 0; i < width; i++ {
       if row[2*i+1] != "_" {
          val, ok := sets[dsu.FindSet(i)]
          if !ok {
             sets[dsu.FindSet(i)] = i
          } else {
             newDsu.UnionSets(val, i)
          }
       }
    }

    dsu = newDsu
```

Reseting dsu to initals and keeping nodes together if they are in the same set

Step 5

```
    for k := 0; k < length; k++
```

Looping it

Step 6

```
 for i := 0; i < width-1; i++ {
        row[2*i+1] = "_"
        if dsu.FindSet(i) != dsu.FindSet(i+1) {
            dsu.UnionSets(i, i+1)
            row[2*i+2] = " "
        }
    }
    maze.Rows = append(maze.Rows, strings.Join(row, ""))
```

For the last row, if two cells are in different sets, we are connecting them

2.4.3 Summing up

We go through each row multiple times and using maps. It is not affecting our time complexity in terms of big O notation and Golang's map implementation works with constant time, so we just skipping it and consider we are doing it once and have $O(length)$. At each row we are using DSU for each cell, so it gives us $O(width * \alpha(width))$ per row, as we already noticed inverse Ackermann's function can be written out, because it is constant for all reasonable numbers. So we have $O(length * width))$, which is pretty good at this point.

2.5 Multiplayer

At this point we going to have only local multiplayer, but our project stands to be scalable and expandable, so in the future, with the relevant resources it won't be hard to add multiplayer games through the internet.

2.5.1 Minds on implementation

For sure, move tracking should be fast enough on both sides, on writing and reading. Such database as Redis [4] could help us. It is a NoSQL key-value database and most important thing, it has amazing performance, so it is a good choice for us. Also, we should be able, to send information to the client as soon as possible, so it will be reasonable to use websockets, to keep connection with user client and send information on player's position update.

Chapter 3

Frontend

3.1 Stack of technologies

3.1.1 Typescript

JavaScript is a dynamic typing language, which means that type of the variable can be changed and will be defined by assigned value. Is it always suitable? Surely, no. Sometimes we need to know which exactly type is a variable, cause our application could crash, if we don't receive proper field or field format. Typescript can help us to prevent this type of issues, cause it's a static typing language. So we can't assign value with type B to the variable once we've assigned value with type A to it. Also Typescript has such features like the Generic types, Declaration files etc.

3.1.2 React

React is a JavaScript library. It's a one of the most popular JavaScript frameworks(libraries) for building user interfaces. It's free and open-source, which means that everyone can contribute to it. Developing in React based on UI components. React code is easy to understanding, because of JSX(JavaScript Syntax Extension). JSX allows you to assign html elements to the variables and write JavaScript code inside of html code. React has a Virtual DOM, which provides fast changes in DOM tree with reconciliation. Also notable features of React: declarative, lifecycle methods, hooks, state and props.

3.1.3 Redux

Redux is a state management tool for React based on flux architecture.

Redux flow:
1. UI (User Interface)
2. Actions

3. Reducer
4. Store
5. State

3.1.4 Material-UI

Material UI is a tool which provides scalable UI components for React.

3.1.5 Axios

Axios is a tool for easily making a request for server.

3.1.6 react-router-dom

react-router-dom is a library for configure pages routing in react app.

3.1.7 SCSS

SCSS is a tool which help to work with styles easier.

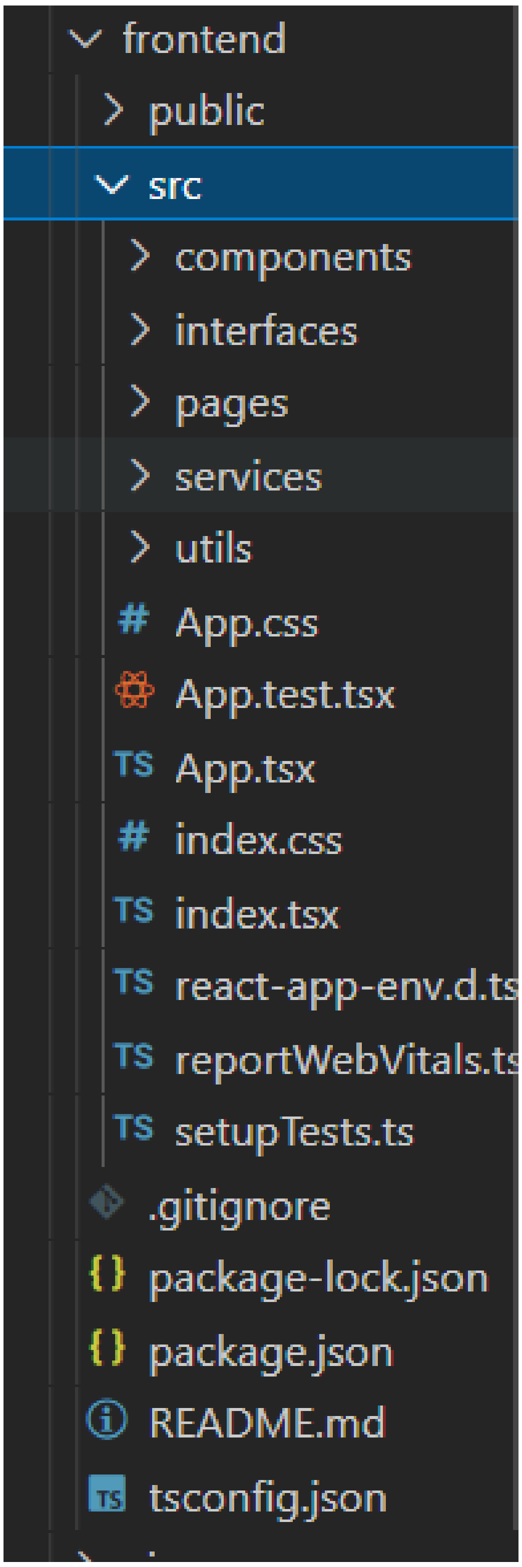
3.2 Maze game implementation

3.2.1 Throttling

To throttle a function means to ensure that the function is called at most once in a specified time period (for instance, once every 10 seconds). This means throttling will prevent a function from running if it has run “recently”. Throttling also ensures a function is run regularly at a fixed rate.

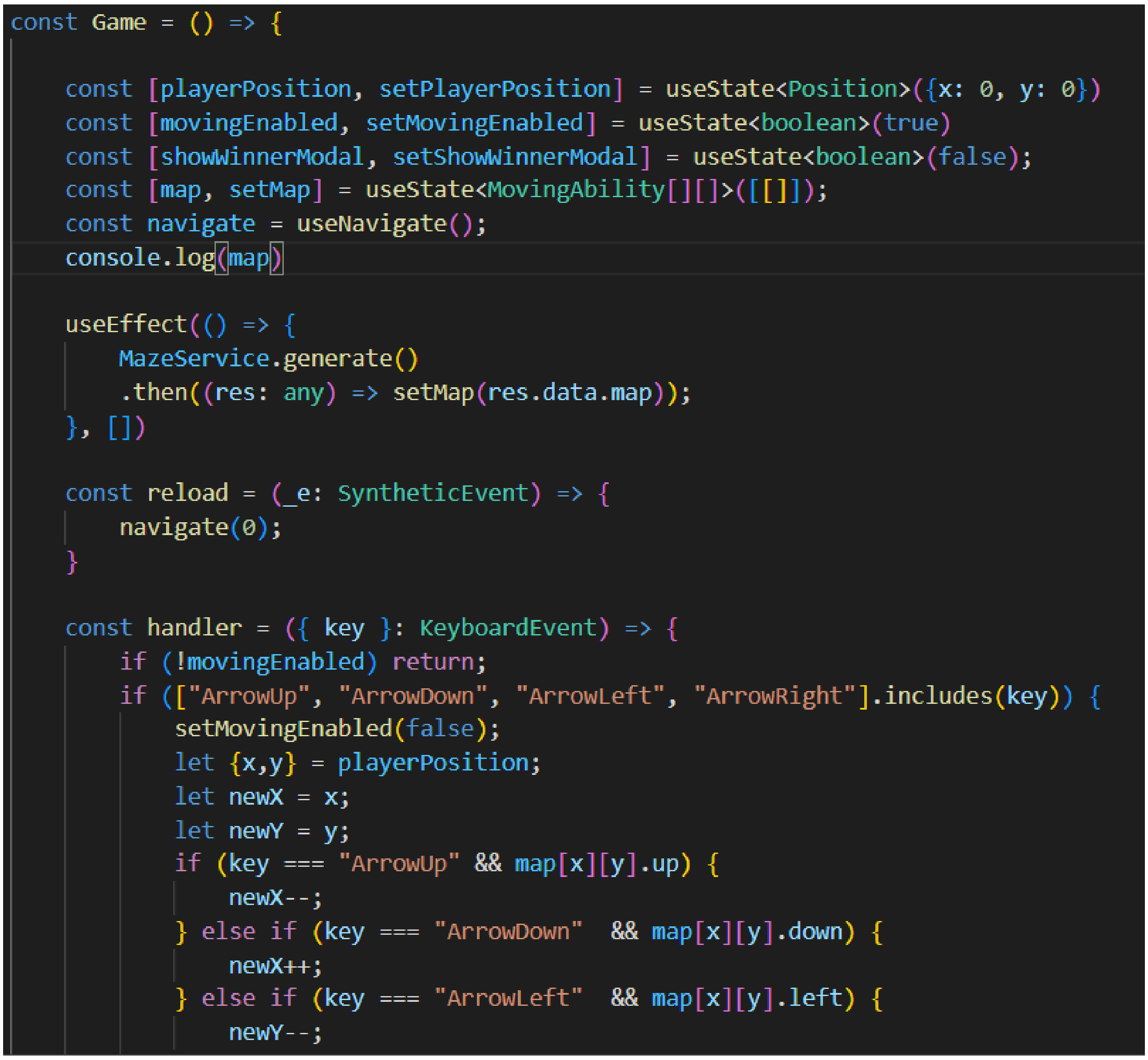
Simple example of throttle function:

```
export function throttle(callback, interval) {
    let enableCall = true;

    return function(...args) {
      if (!enableCall) return;

      enableCall = false;
      console.log(args)
      callback.apply(this, args);
      setTimeout(() => enableCall = true, interval);
    }
}
```



```
frontend
  public
  src
    components
    interfaces
    pages
    services
    utils
    App.css
    App.test.tsx
    App.tsx
    index.css
    index.tsx
    react-app-env.d.ts
    reportWebVitals.ts
    setupTests.ts
  .gitignore
  package-lock.json
  package.json
  README.md
  tsconfig.json
```

Figure 3.1: The project structure



```
const Game = () => {

    const [playerPosition, setPlayerPosition] = useState<Position>({x: 0, y: 0})
    const [movingEnabled, setMovingEnabled] = useState<boolean>(true)
    const [showWinnerModal, setShowWinnerModal] = useState<boolean>(false);
    const [map, setMap] = useState<MovingAbility[][]>([[]]);
    const navigate = useNavigate();
    console.log(map)

    useEffect(() => {
        MazeService.generate()
        .then((res: any) => setMap(res.data.map));
    }, [])

    const reload = (_e: SyntheticEvent) => {
        navigate(0);
    }

    const handler = ({ key }: KeyboardEvent) => {
        if (!movingEnabled) return;
        if (["ArrowUp", "ArrowDown", "ArrowLeft", "ArrowRight"].includes(key)) {
            setMovingEnabled(false);
            let {x,y} = playerPosition;
            let newX = x;
            let newY = y;
            if (key === "ArrowUp" && map[x][y].up) {
                newX--;
            } else if (key === "ArrowDown"  && map[x][y].down) {
                newX++;
            } else if (key === "ArrowLeft"  && map[x][y].left) {
                newY--;
```

Figure 3.2: GamePage component

Chapter 4

Design

4.1 Tools

4.1.1 Figma

Figma is the main tool of the UX/UI designer. On this service we have developed the design of our website. In figma we can:

- Develop the interface and prototype of the site
- Save files to the cloud
- Integrate with different design applications
- Several people can enter and edit at the same time

4.1.2 Adobe Illustrator

This application was needed to create the logo for our game. This application makes it possible to:

- Easy to create vector design
- Save it in any convenient format
- Save as best quality

4.2 Graphic Design

This is the main part of the design. At this stage, we already decide on colours, fonts and patterns. Graphic design helps to build the right communication with users. The visual concept begins to be developed at this stage and the most important part of graphic design is the logo.

4.2.1 Logo

As you can see, the logo consists of three figures. Each figure represents a player, and by this we want to make it clear that this is a multiplayer game. The logic behind the figures is that they are a connection between two points and represent the player's movement from one point to the other, which is the main function of our game.

4.3 UX/UI design

In design, it is very important that the structure is created according to logic and all the rules of design. No matter how beautiful the site would be, if the structure is created incorrectly, the site will be non-working.

4.3.1 Main page

Header

- Our logo is located on the upper left side.
- "About us", "Contact", "Log in" will be located on the upper right side

Onboarding

- Also on the main page there is a brief information about the game, this is done so that when a person first visits the site, he understands where he got to.

Starting game

- In the middle there will be 2 main buttons "Start game", "Join the game".
- "Start game". The emphasis is on the "Start game" button by coloring the button with a bright color. The emphasis was on this button in order to encourage more users to register in the game, since when you click on the "Start game" button, if a person is not registered, a registration window will pop up for him. And if a person is registered, then when you click on this button, he can immediately start playin.
- Join the game". This button is designed to allow players to join the creator's game. Clicking on this button will pop up a window with an input where players could enter the id of the game they want to join.

Footer

- Footer will contain links to social networks in the form of icons

Chapter 5

Conclusion

5.1 Idea

At the start we decided to reformat a well known classic game. Mazes are good time-killer, rules are easy and difficulty is amusing. To add some competitiveness, we added a possibility to play with a friend.

5.2 Implementation

5.2.1 Design

For now, we brought an amazing design, which makes our application look friendly and interesting. UX is also user-friendly, right after user loads main page he can see "PLAY" button and that's all what he needs from this page, also it contains some information about project.

5.2.2 Backend

For the backend we did a big work on making mazes look natural and interesting enough to keep user's focus. Implementation based on algorithms and data structures. In the relevant chapter we added all details on optimization and some math calculations.

5.2.3 Frontend

Frontend is a quintessence of our work, because it is where our design meets technical part. Using some techniques and approaches to make playing easy and comfortable, we built a user-friendly game for wide auditory.

5.3 Problems

As any team, we faced some problems and failed to implement some of thing for this stage. Firstly, it is lack of resources. We can't get high-performance hardware and software for free, so it is the place we met problems. Any online communication with a high tempo should be done with a high stability and accurate approach. So, for now, we decided to make some things easier and leave place for future expansion.

References

[1] Jamis Buck. *Maze Generation: Eller's Algorithm.* URL: `https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm`.

[2] *JWT.* URL: `https://jwt.io`.

[3] *PostgreSQL.* URL: `https://www.postgresql.org`.

[4] *Redis.* URL: `https://redis.io`.

[5] Kevin Wayne. *Union-Find.* URL: `https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/UnionFind.pdf`.