Metropolist

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

Shizheng Yang

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2019

Metropolist

By Shizheng Yang

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. Kasi Periyasamy Examination Committee Chairperson	Date
Prof. Steven Senger Examination Committee Member	Date
Prof. Kenny Hunt Examination Committee Member	Date

Abstract

Yang, Shizheng, "Metropolist," Master of Software Engineering, May 2019, (Kenny Hunt, Ph.D.).

This manuscript describes the development of a web-based map generator, which allows the user to create randomly generated fantasy maps of cities and additionally allows the user to annotate and edit city elements at a fine granularity.

Acknowledgements

I would like to express my sincere appreciation to my project advisor Dr. Kenny Hunt for his invaluable guidance and untiring support. I would also like to express my thanks to the Department of Computer Science at the University of Wisconsin–La Crosse for providing the learning materials and computing environment for my project.

Table of Contents

Abstract i
Acknowledgments ii
List of Tables v
List of Figures vi
Glossary vii
1. Introduction 1
    1.1. Background 1
    1.2. Similar Systems 1
        1.2.1. Minecraft 1
        1.2.2. Medieval Fantasy City Generator(MFCG) . . 2
        1.2.3. Azgaar's Fantasy Map Generator(FMG) . . . 2
    1.3. Project Goal 4
2. Requirements 5
    2.1. Overview 5
    2.2. Functional Requirements 5
    2.3. Non-Functional Requirements 7
    2.4. Selection of Software Development Life Cycle Model . . 8
3. Design 10
    3.1. Architecture Design 10
        3.1.1. Web Browser or Client 10
        3.1.2. Web Application Server 10
        3.1.3. Database Server 11
    3.2. Database Design 11
    3.3. REST API Design 12
    3.4. Map Generator Design 13
        3.4.1. Canvas and SVG 13
        3.4.2. Graphics Libraries 13
        3.4.3. Height Map 14
4. Implementation 17
    4.1. Frontend 17
    4.2. Backend 20
    4.3. Map Generator 21
        4.3.1. Polygons 21
        4.3.2. Elevation 21
        4.3.3. Contour Lines 22
        4.3.4. Soft Brush 24
        4.3.5. Layers 24
        4.3.6. City Wall 25
        4.3.7. Types 25
        4.3.8. Buildings 27
        4.3.9. Streets 28
5. References 30

6. Appendices 31

List of Tables

1 REST API Design Table 14

List of Figures

1	A screenshot of a planetary terrain region of "No Man's Sky"	1
2	A screenshot of "Minecraft"	2
3	A screenshot of the Medieval Fantasy City Generator	3
4	A screenshot of the Azgaar's Fantasy Map Generator	3
5	A screenshot of the current project	4
6	Use Case Diagram	6
7	Agile Development Life Cycle Model	8
8	Entity Relationship Diagram	12
9	Signup page	18
10	Login page	18
11	Community page	19
12	User dashboard page	19
13	Profile page	20
14	Voronoi polygons	21
15	Voronoi polygons after 5 times of relaxation	22
16	Continent and Water	23
17	A map with contour lines	24
18	A map with walls	26
19	A city map with blocks	27
20	A city map with buildings	28
21	A city map with streets	29

Glossary

LaTeX

LaTeX is a document markup language and document preparation systems for the TeX typesetting program.

1. Introduction

1.1. Background

Generating content for computer games, CGI effects, feature films, print media, or Role-playing games is a significant bottleneck in terms of effort and resources. A typical game contains many thousands of audio files, images, textures, and 3D models. Procedurally generated content provides a cost-effective alternative to the manual creation of models, textures, images, and sound assets and can expand playability beyond what is otherwise possible. The video game "No Man's Sky," for example, was released in 2016 and relied on procedural asset generation to create over 18 quintillion planets each of which has a unique ecosystem composed of flora and fauna. Such a scale is, of course, beyond the reach of any manually created system. Figure 1 is a screenshot of a planetary terrain region of "No Man's Sky."

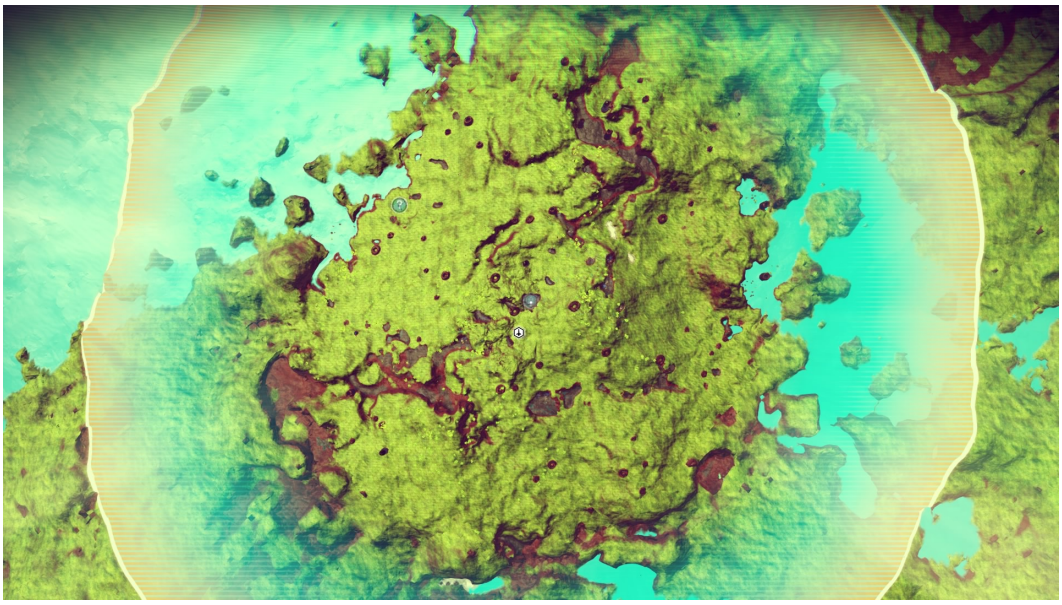

Figure 1. A screenshot of a planetary terrain region of "No Man's Sky"

1.2. Similar Systems

1.2.1. Minecraft

"Minecraft" is a computer game that can produce massive worlds composed of fine details, like elaborate cliff faces and waterfalls. Moreover, it relies on procedural generation, which automatically creates environments and objects that are at once random but guided by rules that maintain a consistent logic. Mountains are always rocky and sprinkled with snow, for example, while the low lands are typically full of grass and trees. Figure 2 is a screenshot of "Minecraft."

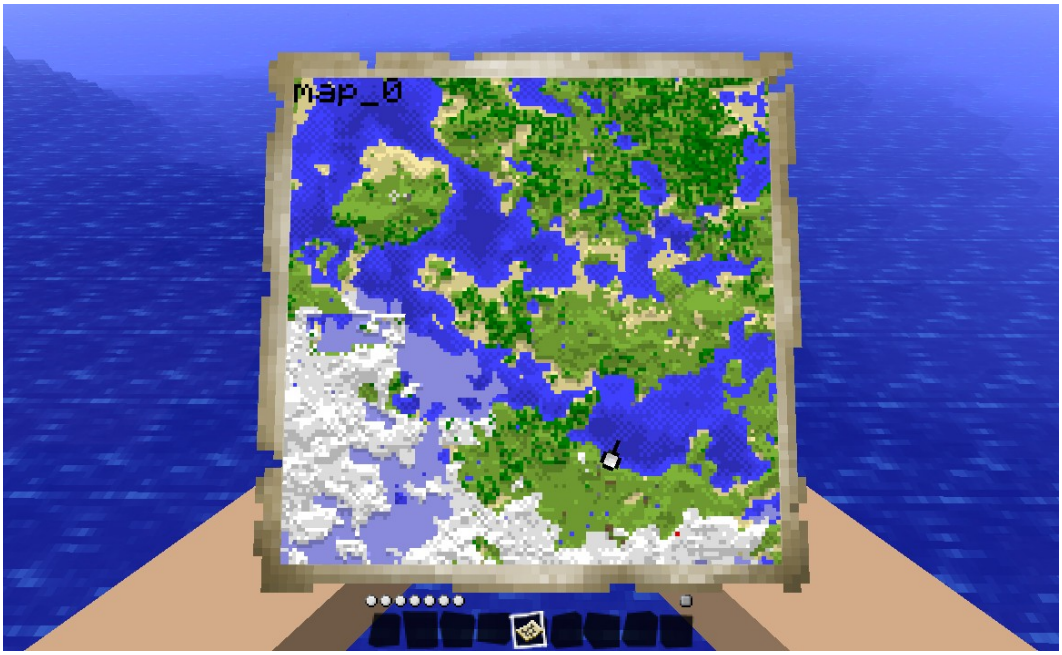

Figure 2. A screenshot of "Minecraft"

1.2.2. Medieval Fantasy City Generator(MFCG)

The "Medieval Fantasy City Generator(MFCG)" is a web application. This application generates a random medieval city layout of a requested size: small, medium or large, which is made up of different types of regions, and the generation method is stochastic. Furthermore, elements are provided for the user to add to the city, such as farm fields, citadel, plaza, temple, river, coast and so on. Because of the premise of medieval fantasy, the map always includes the walls and castle, but the user can decide whether to display them. It allows the user to edit the map to modify unsatisfying layers using a warping tool. The author also mentioned that the goal of the application is to produce a nice looking map, not an accurate model of a city. Finally, the user can save the map as an image in the "png" or "svg" format by using the export feature. Figure 3 is a screenshot of MFCG.

1.2.3. Azgaar's Fantasy Map Generator(FMG)

The "Azgaar's Fantasy Map Generator(FMG)" is another similar system, which is a larger scale world map. The size of the map made by FMG is not just an island, but a random fantasy map represents a pseudo-medieval world. Just like the real world, the map generated by FMG has constraints that the continent is always surrounded by the ocean and will never touch the border of the map. Although it is a randomly generated map, it is still based on real-world rules. While its most prominent feature is that the user can choose the type of map he likes, which provides the following 5 map types: political, cultural, height, biomes and pure landmass. It also supports user-defined map

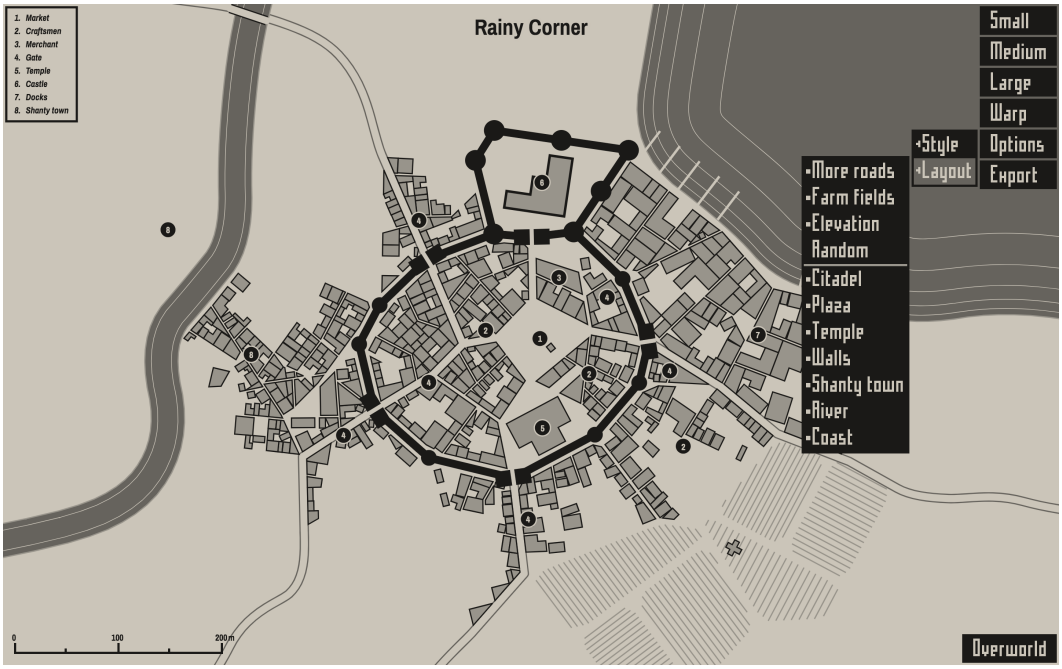

Figure 3. A screenshot of the Medieval Fantasy City Generator

type, and the user can add additional layers on existing map layers: rivers, temperature, and population. Also, it allows the user to annotate and edit the map using various such editors: layout, style, template, scale, countries, or cultural. It also supports exporting the map in the PNG or SVG format, but unlike the previous application, if the user wants to come back to edit the map in the future, he can save it in the "map" format. Figure 4 is a screenshot of FMG.

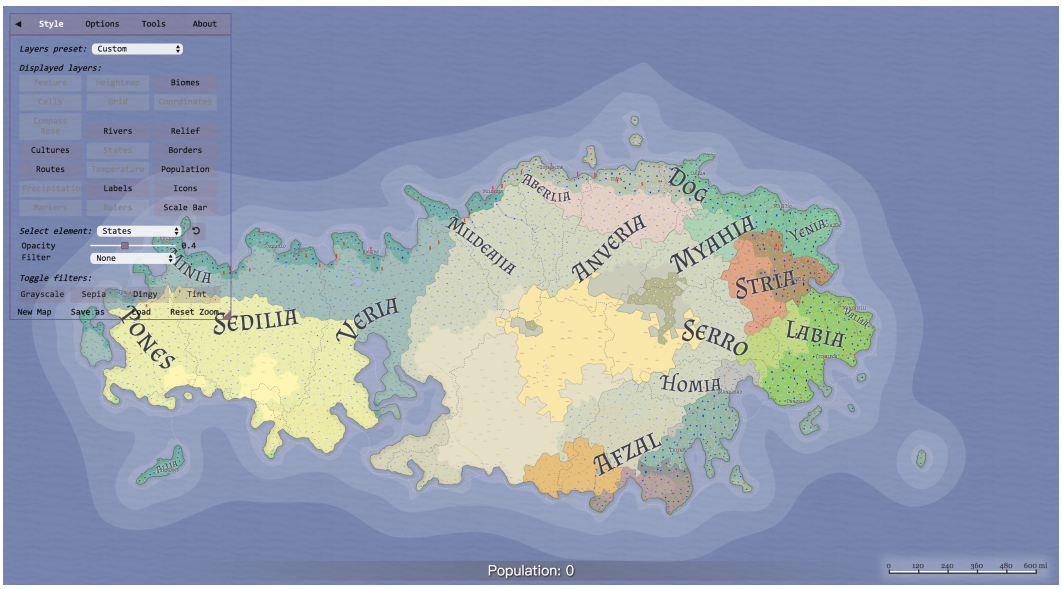

Figure 4. A screenshot of the Azgaar's Fantasy Map Generator

1.3. Project Goal

The goal of this project is to automatically generate city maps for use in role-playing games (RPG) or worldbuilding narratives. While the ultimate goal of this project is to procedurally replicate maps of a quality similar to the best cartographic hand-created maps by expert artists, we have obtained a modest approximation to the desired level of quality. Our system will have essential features, such as allowing the user to annotate or edit, allowing the user to export the map as an image in the PNG or SVG format, and allowing the user to save it to a database and retrieve it. Figure 5 is a screenshot of the final project.

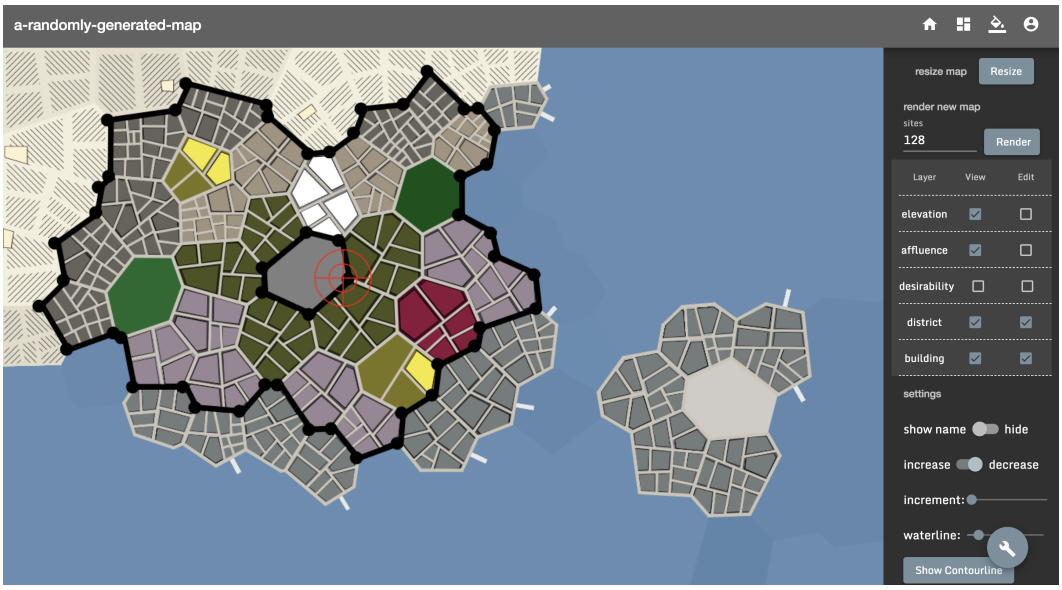

Figure 5. A screenshot of the current project

2. Requirements

2.1. Overview

Initially, this project was conceived by Dr. Kenny Hunt, who served as the advisor, and he gave almost all of the requirements in an introductory meeting. Both the advisor, who is also the project sponsor, and the developer, who is also the author, are the stakeholders of the project.

After that, the advisor met with the developer every week to gather informal functional requirements of this project, and he would also refine any previous requirements that were given. These requirements played a crucial role in the development of the map generator. On the one hand, the scope of development was clarified, and on the other hand, the necessary algorithms were compiled to maintain the continuation of development.

2.2. Functional Requirements

There are two roles for this system: "admins" and "users". As a result, the following functional requirements were established for the project:

1. As an admin or a user, I need to be able to:
   (a) Log in with a unique email and password
   (b) Log out with the ended session
   (c) Edit personal profile by changing email, first name, last name, and password
2. As an admin, I want to be able to:
   (a) View all users
   (b) Enable or disable users
   (c) Search for users using any combination of email, first name, and last name
3. As a user, I want to be able to:
   (a) Sign up with a unique email, first name, last name, long password, and confirmed password
   (b) View all of the maps that I have created
   (c) View all maps shared with me
   (d) Search for maps using any combination of the map name, created date, edited date, the owner's email, the owner's first name, and the owner's last name
   (e) Download viewable maps in PNG or SVG format

(f) Make maps public or private

(g) Create maps with a name

(h) Delete maps

(i) Edit maps

Figure 6 is the Use Case Diagram that describes the functionalities to be implemented by this web application.

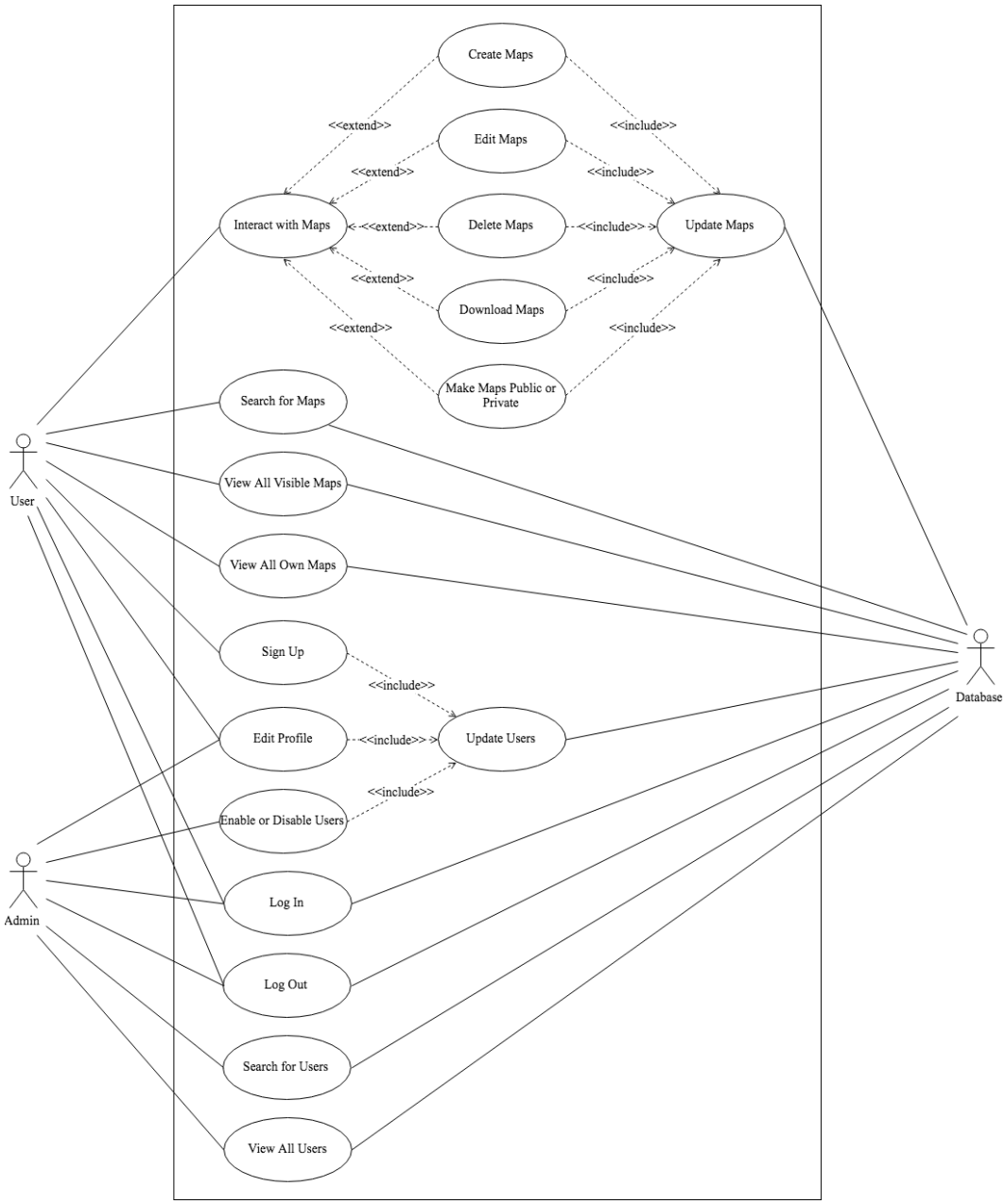

Figure 6. Use Case Diagram

2.3. Non-Functional Requirements

There are numerous non-functional requirements for this system: response time, availability, usability, security (authentication, authorization, integrity, privacy, etc.), and so on. For this application, we focused on the following requirements:

1. Security:
   (a) Input Validation:
      i. Add validations at both client and server sides
      ii. All methods should always validate all parameters in the server-side
   (b) Password Encryption:
      i. Shall always be encrypted
      ii. Not even the system administrators shall see clear text passwords
      iii. Applying a hashing algorithm to passwords
   (c) Prohibiting Cross Site Scripting (XSS):
      i. Never insert data anywhere in a "script" element
      ii. Never insert data in an HTML comment
      iii. Never insert data anywhere in CSS
      iv. Never insert data in an attribute name
      v. Never insert data in an attribute value
      vi. Never insert data in a tag name
   (d) Secure Session Cookie:
      i. Set the "Secure" cookie attribute and instruct web browsers to send the cookie only over encrypted, e.g., HTTPS, links.
      ii. Set the "HttpOnly" attribute true and prohibiting the JavaScript code from reading the cookie via the DOM "document.cookie" JavaScript object.
      iii. Always change the session id after login
2. Performance:
   (a) High performance of rendering map:
      i. Compare and select the most appropriate and the fastest way to render maps, e.g., canvas, SVG

2.4. Selection of Software Development Life Cycle Model

We analyzed and summarized the following possible risks:

1. Lack of experience developing in a map generation
2. The certainty whether a third-party library was available as the map rendering engine
3. The potential misunderstanding between the advisor and the developer with respect to the requirements

To reduce or avoid these risks, two life cycle models were considered: waterfall and agile. Because of our development model is flexible and the frequent communication between the sponsor and the developer, the waterfall model, is beyond our consideration. Unlike the waterfall model, we only need initial planning to start this project, and every time the new requirements are almost based on the previous one. Finally, the agile model was chosen, which is shown in Figure 7.

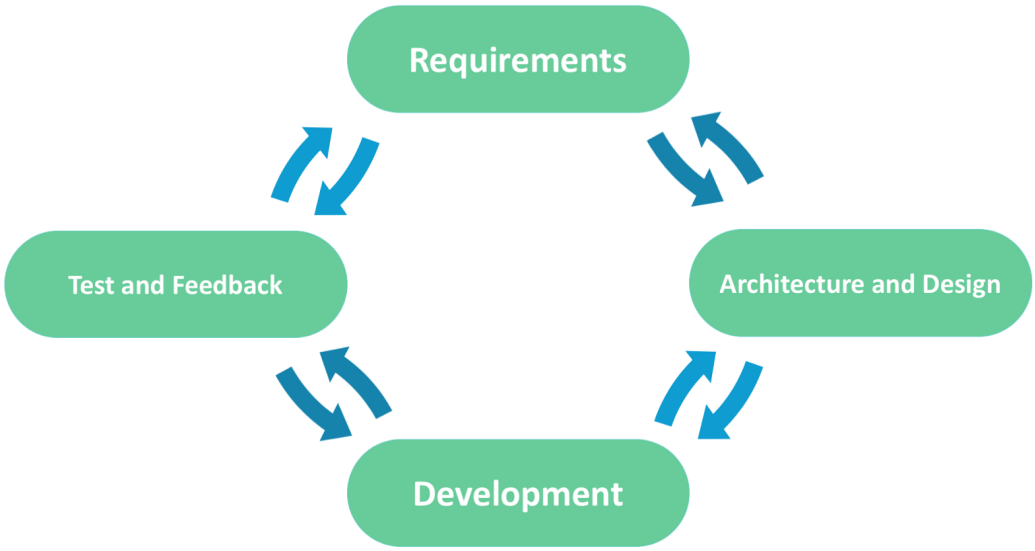

Figure 7. Agile Development Life Cycle Model

An agile methodology is a practice that helps continuous iterations of development and testing in the software development process. In this model, development and testing activities are concurrent. Furthermore, the agile model is a combination of iterative and incremental process models, which breaks the product into small incremental builds. These builds are provided in iterations. Each iteration typically lasts from about one to three weeks.

At the end of each iteration, a working product demo is displayed to stakeholders. There are several known advantages and disadvantages of using the agile model in this project:

1. Advantages:
    (a) Easy to manage
    (b) Little or no planning required
    (c) Gives flexibility to developers
    (d) Resource requirements are minimum
    (e) Suitable for fixed or changing requirements
2. Disadvantages:
    (a) More risks of sustainability, maintainability, and extensibility
    (b) An overall plan, an agile leader (the advisor) is a must without which it does not work
    (c) There is a very high individual dependency since there is minimum documentation generated

3. Design

3.1. Architecture Design

Because we selected the web application model from the beginning, the classic client-server web architecture was quickly adopted. The architecture is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requests called clients. In theory, any device connected to the Internet can try to access and obtain the resources or services of the server while the server is running normally. Based on the client-server structure, we have detailed its component structures below:

3.1.1. Web Browser or Client

The web browser or client is the interface rendition of a web app functionality, with which the user interacts. This content delivered to the client is developed using HTML, JavaScript, and CSS and does not require operating system related adaptations. In essence, the web browser or client manages how end users interact with the application.

There are three, well-known Web Application Architecture types available in the modern landscape: Single Page Applications (SPA), Microservices, and Serverless Architectures. Because the SPA is super-simple to deploy if compared to more traditional server-side rendered applications: it is just one index.html file, with a CSS bundle and a Javascript bundle. Furthermore, our frontend and backend are developed separately. We chose the SPA structure.

The SPA model interacts with the user by providing updated content within the current page rather than loading entirely new pages from the server with each action from the user. It helps prevent interruptions in the user experience, transforming the behavior of the application such that it resembles a traditional desktop application.

3.1.2. Web Application Server

The web application server manages business logic and data persistence and can be built using PHP, Python, Java, Ruby, .NET, Node.js, among other languages. It is comprised of at least a centralized hub or control center to support multi-layer applications. The essential purpose of a web server architecture is to complete requests made by clients for a website. The clients are typically browsers and mobile apps that make requests using secure HTTPs protocol, either for page resources or a REST API.

Since the focus of this project is on the client side (frontend), which is the map generator (written in JavaScript), we chose the Node.js. Moreover, the Node.js is written using JavaScript and is the same technology as frontend components. This makes it easier for the developer to program backend services and frontend user interfaces. It also provides consistency, code sharing

and reusability, simple knowledge-transfer, and a large number of free tools. These benefits bring flexibility and efficiency when building this project.

3.1.3. Database Server

The database server provides and stores relevant data for the application. Additionally, it may also supply the business logic and other information that is managed by the web application server. There are multiple popular database systems available: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, MongoDB, MariaDB, DB2, and SAP HANA.

Because the project involves a small number of entities, the relationships among entities are not complicated, and the choice of the Software Development Life Cycle (SDLC) model, we chose the MongoDB, which is dynamic, flexible and easy to get started. It also provides high performance, high availability, and high scalability. More importantly, our main purpose is to save maps and implement basic CRUD (create, retrieve, update, delete) operations of the database. As MongoDB is a schema-less database (written in C++), we can serialize the map data to JSON, send it to MongoDB and then save it.

3.2. Database Design

NoSQL stands for "not only SQL," which is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats. MongoDB is a type of NoSQL database. A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects, because the values of fields may include other documents, arrays, and arrays of documents.

The system supports only two document types: the user and the map. A user can have many maps, but a map can only belong to one user. Thus, the relationship between the user and the map should be one-to-many.

MongoDB provides two ways of data modeling: Embedded Data Modeling and Normalized Data Modeling. Using Embedded Data Modeling, we may embed related data in a single structure or document, which means we should store maps in the user schema. However, the data of the map is relatively large, in general, it may exceed the maximum BSON document size set by the MongoDB, and the map cannot exist as a separate entity. Though embedding provides better performance for reading operations, as well as the ability to request and retrieve related data in a single database operation, it is still beyond our consideration.

Normalized Data Modeling provides One-to-One and One-to-Many Relationships. We chose the Normalized One-to-Many Structure, which uses references between documents, and provides more flexibility than embedding. Figure 8 is the Entity Relationship Diagram that describes the relationship between the user and the map.

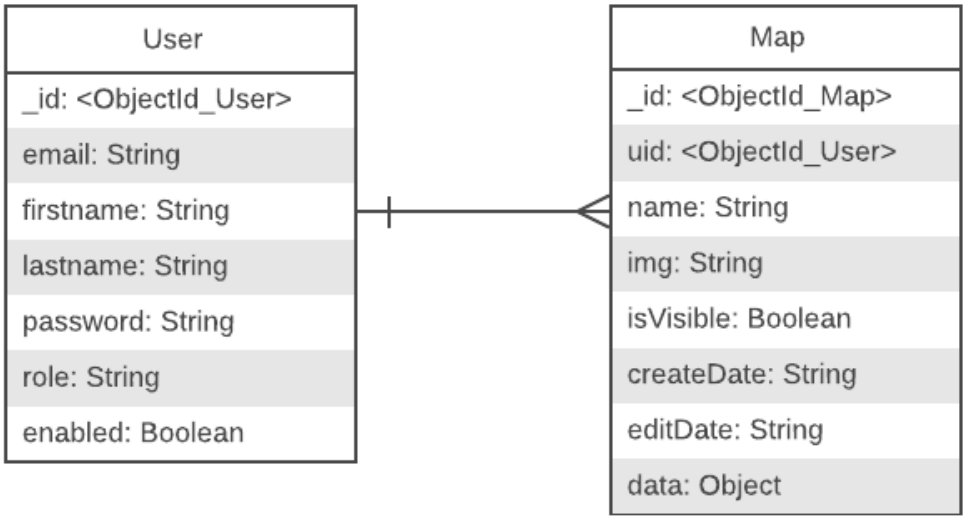

Figure 8. Entity Relationship Diagram

3.3. REST API Design

REST is the acronym for Representational State Transfer. It is an architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation [1].

Like any other architectural styles, REST has its own 6 guiding constraints, which must be satisfied if an interface conforms to the RESTful. These principles are listed below:

1. Client–server. By separating the user interface concerns from the data storage concerns, it improves the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

2. Stateless. Each request from a client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

3. Cacheable. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

4. Uniform interface. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified, and the visibility of interactions is improved.

5. Layered system. The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.

6. Code on demand (optional). REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

Based on these principles, we have designed the APIs shown in Table 1.

3.4. Map Generator Design

3.4.1. Canvas and SVG

Since the project is a web application, the map generator should run on the web page. We need to choose a 2D HTML Graphics Tool to render it, such as Canvas or SVG.

SVG is a language for describing 2D graphics in XML, which means every element is available within the SVG DOM, you can attach JavaScript event handlers for an element. In SVG, each drawn shape is remembered as an object. If attributes of an SVG object are changed, the browser can automatically re-render the shape. Canvas draws 2D graphics, on the fly (with a JavaScript) by rendering pixel by pixel. In canvas, once the graphics are drawn, it is forgotten by the browser. If its position should be changed, the entire scene needs to be redrawn, including any objects that might have been covered by the graphics. We chose Canvas not only because there are always many elements in a map, but also there are many third-party JavaScript graphics libraries that can provide support.

3.4.2. Graphics Libraries

The more important thing is that we have not had any experience in developing a map generator before. However, we learned the necessary knowledge to maintain the continuation of development from similar websites, which mentioned in the section Background. In addition, developing this generator requires much professional geometric mathematics. If we build our own wheels, the time may not be enough. There are many open source third-party graphics libraries on the web, such as D3.js, Paper.js, Fabric.js, Three.js, and so on, which provide many ready-made and mature geometric construction methods. D3.js was chosen because of the following reasons:

1. It is open source: so we can freely use the library, code and build our own maps.

2. It has good and neat documentation of all the available functions and methods.

HTTP Verb	URI	Description
GET	/metro/api/v1/users	Get a list of all users
GET	/metro/api/v1/users/{uid}	Get the user with {uid}
GET	/metro/api/v1/users/{uid}/ maps	Get a list of all maps of the user with {uid}
PATCH	/metro/api/v1/users/{uid}/ password	Verify the password of the user with {uid}
PUT	/metro/api/v1/users/{uid}/ password	Update the password of the user with {uid}
PUT	/metro/api/v1/users/{uid}/ email	Update the email of the user with {uid}
PUT	/metro/api/v1/users/{uid}/ name	Update the name of the user with {uid}
PUT	/metro/api/v1/users/{uid}/ enabled	Update the enabled of the user with {uid}
GET	/metro/api/v1/maps/{mid}	Get the map with {mid}
GET	/metro/api/v1/maps/ ?page={page}&limit={limit}	Get a list of {limit} maps on page {page}
POST	/metro/api/v1/maps	Add a new map to the database
PUT	/metro/api/v1/maps/{mid}	Update the map with {mid}
DELETE	/metro/api/v1/maps/{mid}	Delete the map with {mid}

Table 1. REST API Design Table

3. There is a gallery with unique charts which can be referred to while developing our project.

3.4.3. Height Map

Before we get started to develop this map generator, we need to understand what is a map and decide what methods we are going to use to generate it.

Simplifying the map is a two-dimensional representation of the surface of the world. The main thing that allows us to say whether an image is a map or not is the obvious partition between land and water parts. To have an ability to divide a canvas into land and water we need to know what pieces are lower than others, it means we need to generate a height map.

There are three main ways to generate a heightmap: use noise functions, use graph structure, or use both. Most procedural map generators use noise functions (midpoint displacement, fractal, diamond-square, Perlin noise, etc.) to generate a height map, but the result may look a bit like artificial. So we decided to use graph structure to model the things directed by gameplay constraints (elevation, affluence, desirability, district, buildings). We want this project to have the following main features:

1. Once opening the map editor page, it will show an empty map with random map seeds by default, then the user can start making and editing maps by accessing the menu on the right.

2. Entering the number of map seeds to create a new empty map in the menu or using the default seeds.

3. Viewing different types of layers for the map: elevation, affluence, desirability, district, and building, which can superimpose on each other.

4. Selecting different types of layers for the map: elevation, affluence, desirability, district, and building, which can superimpose on each other.

5. Displaying street names or not.

6. Manipulating the “increase/decrease toggle” to increase or decrease the value of elevation, affluence, and district.

7. Manipulating the “increment sliding” to set the size of the increment.

8. Manipulating the “waterline sliding” to make map cells as water or city.

9. If the waterline is changed, the continent will be changed accordingly.

10. Displaying contour lines or not.

11. After selecting one or more than one layers under “edit” mode, the user can change the size of the ”soft brush” by using the mouse wheel, which determines the area where the map will be edited.

12. Editing the map by clicking and dragging the “soft brush.”

13. The districts and buildings are procedurally generated.

14. Allowing to change the type of the current district by right-clicking and selecting a new type from the context menu.

15. The map provides 13 different types of districts: rich, medium, poor, empty, plaza, park, farm, water, harbor, university, religious, castle, and military.

16. Zooming in and zooming out the map by pressing on the alt key and using the mouse wheel.

17. Using a specialized button to resize the map on the right top.

18. Saving the map or downloading it by clicking the button on the right bottom.

All these are based on the Voronoi diagram. Thankfully, generating Voronoi diagrams in JavaScript is easy to be implemented if we use d3.voronoi of Fortune's algorithm by Mike Bostock.

4. Implementation

4.1. Frontend

There are many mature SPA frameworks today, such as Vue.js, ReactJS, Angular.js, Angular, and so on. Since the author is familiar with Angular, the frontend part of the project is implemented using Angular, which comes with almost everything we need, from powerful templates to fast rendering, data management, HTTP services, form handling, and so much more. Moreover, Angular provides a UI component library called Angular Material, which is inspired by Google Material Design. Angular Material components help in constructing attractive, consistent, and functional web pages and web applications while adhering to modern web design principles like browser portability, device independence, and graceful degradation. It helps in creating faster, beautiful, and responsive websites.

In our design, the user must register to log in before he can use the application, so the home page is the login page. Luckily, Angular provides multiple routing guards, and the CanActivate interface is a good way for us to implement our own authentication routing. In addition, as we mentioned in Section 2.3. Non-Functional Requirements, every form must be validated by the client side before being submitted to the server side. The Implementation of GUI is listed below:

1. The "Sign Up" form requires the user to enter a unique and valid email, first name, last name, password, and confirmed password. The user can click the icon on the right of the password input to make his password visible or invisible. Also, the "clear" button on the left bottom corner is used to clear all inputs of the "Sign Up" form. The user can also click the "Log in instead" button to be redirected to the "Log In" page if he already has an account. Figure 9 is the interface of the "Sign Up" page.

2. The "Log In" form requires the user to enter the email and the password. Also, the user can access the "Sign Up" page via the "Create account" button on the bottom. Figure 10 is the interface of the "Log In" page.

3. The "Community" page shows all the maps that can be viewed, which are marked by their own owners as "isVisible." Besides, the user can filter maps using map name, owner's email, owner's first name, or owner's last name via the "Filter" on the left top corner. Figure 11 is the interface of the "Community" page.

4. The "User Dashboard" page displays all the maps belonging to the current user. The user can filter them using any information of maps via the "Filter" on the left top corner. The user can click on the header to perform the sorting of the corresponding map properties. Also, the user is allowed to make the map be visible or invisible by manipulating the

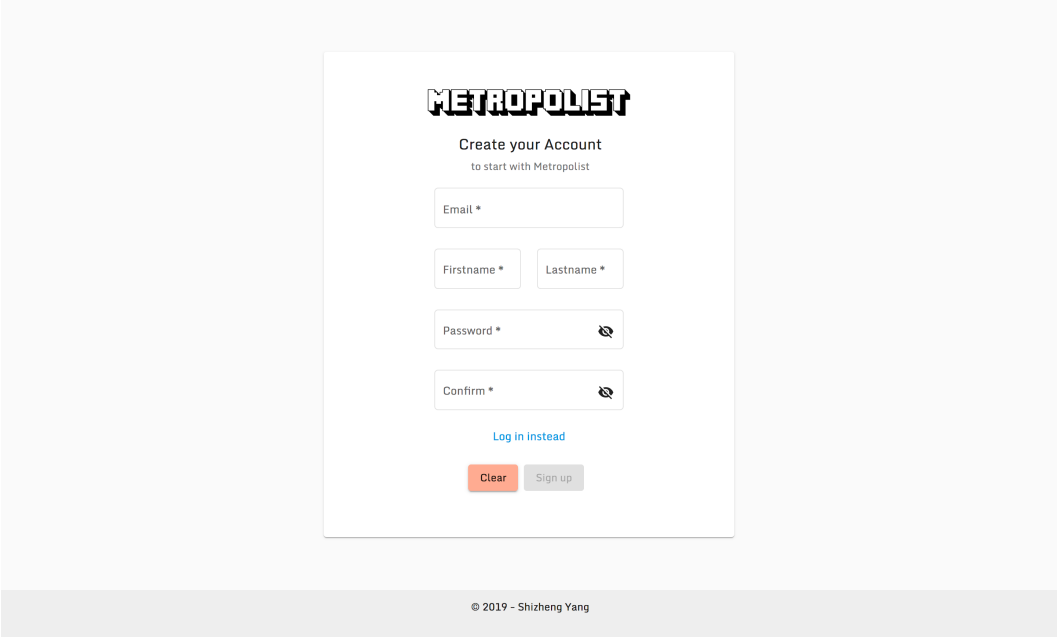

Figure 9. Signup page

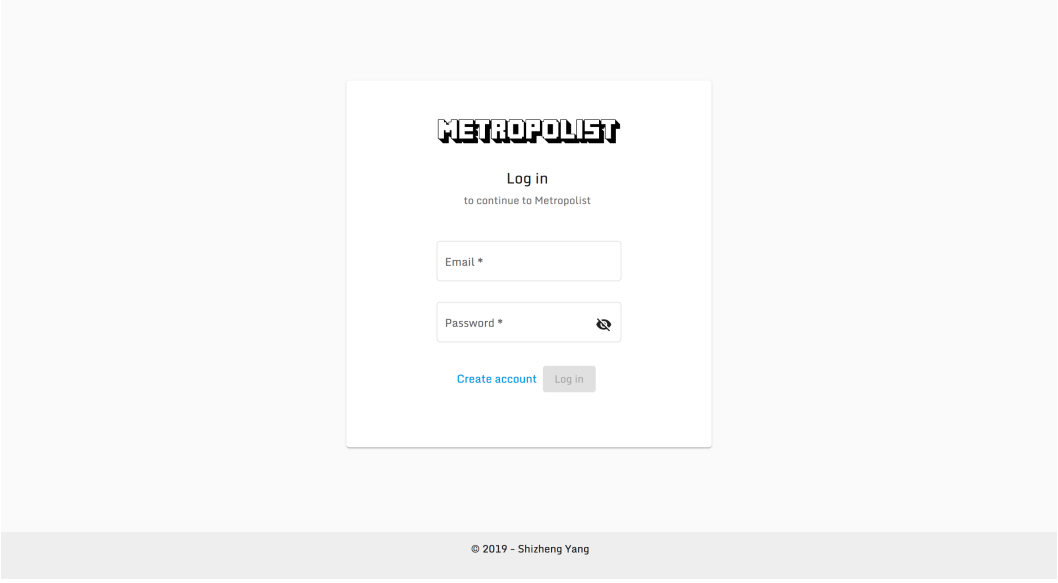

Figure 10. Login page

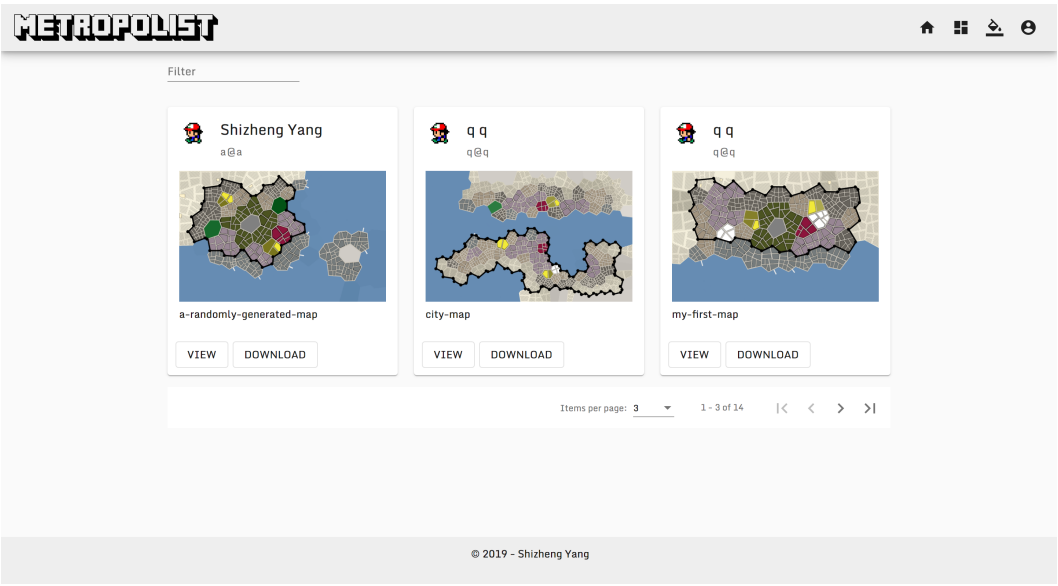

Figure 11. Community page

slide of "isVisible", and edit, delete or download it via the three buttons in the "Operation" column. In addition, The button for creating a new map is in the lower right corner of the page. The user can enter a map name in the pop-up window after clicking it. Figure 12 is the interface of the User dashboard page.

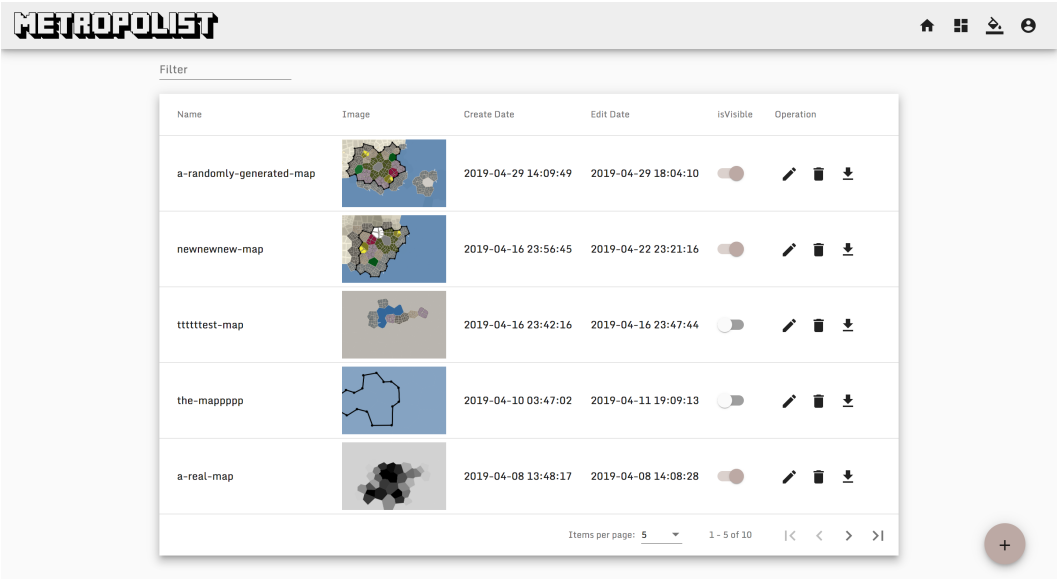

Figure 12. User dashboard page

5. The "Profile" page allows the user to change his email, password, first name, or last name. By the way, before changing the password, the user must first enter the old password. After the verification is passed, the

new password can be entered. Figure 13 is the interface of the Profile page.

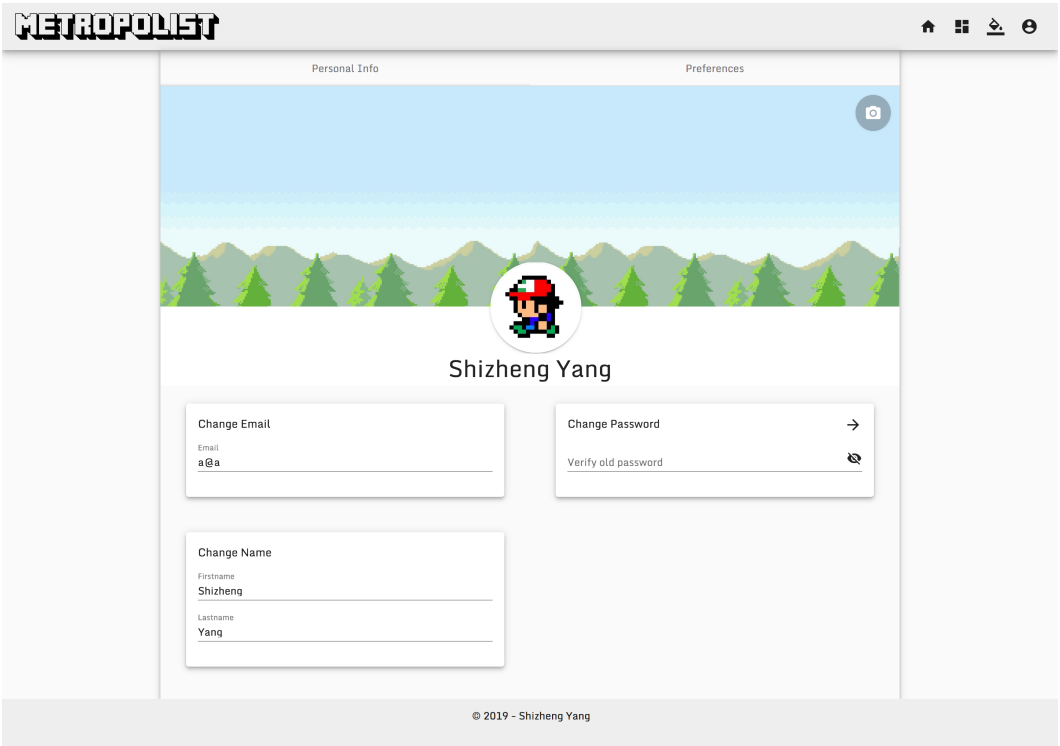

Figure 13. Profile page

4.2. Backend

The backend consists of the server, database, and APIs. The server running on the author's computer, the database is powered by MongoDB, and the APIs are RESTful. On this basis, we chose Express.js because it is a lightweight Node.js framework, which allows us to define routes of our application based on HTTP methods and URLs so that we can easily apply the REST API design to them very well. Also, it includes various middleware modules that we can use to perform additional tasks on the request and response.

To connect MongoDB to our Express application, we had to use an ORM to convert information from the database to a JavaScript application without SQL statements. ORM is short for Object Related Mapping, a technique that can be used to convert data among incompatible types. More specifically, ORMs mimic the actual database so we can operate within a programming language (e.g. JavaScript) without using a database query language (e.g. SQL) to interact with the database. For this application, we used Mongoose as the ORM, which is an object document modeling (ODM) layer that sits on top of the Node.js's MongoDB driver.

4.3. Map Generator

4.3.1. Polygons

The first step is to render some polygons (cells). As we mentioned before in the design phase, we used the d3.voronoi library to implement Steven J. Fortune's algorithm for computing the Voronoi diagram or Delaunay triangulation of a set of two-dimensional points. Figure 14 is an example of 100 dots (red) and corresponding 100 polygons:

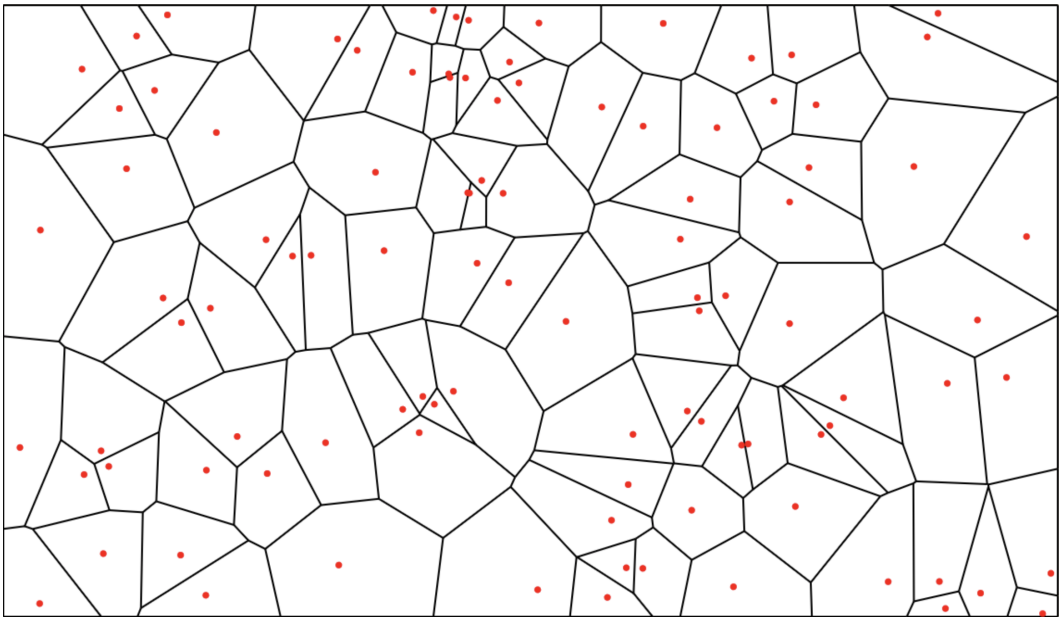

Figure 14. Voronoi polygons

As we can see these random points' tessellation is a little bit irregular, we wanted something closer to semi-randomness or quasi-randomness, not completely random points. So we approximated that by using a variant of Lloyd relaxation, also known as Voronoi iteration or relaxation, which is a fairly simple tweak to the random point locations to make them more evenly distributed. Lloyd relaxation replaces each point by the centroid of the polygon. The more iterations, the more regular the polygons get. Figure 15 is the result after running approximate Lloyd relaxation 5 times:

4.3.2. Elevation

As we mentioned in the map generator design phase, the map should be a height map, so we introduced the concept of "elevation". We called these dots that generate polygons as sites, and each site corresponds to a polygon. Then we derived the sites object from the d3.voronoi object, and assigned the "elevation" property to it, whose value from 0 to 1. Before making the height map, we defined the waterline, which is also from 0 to 1. If the value of elevation is greater than the waterline, the polygon represented by the site

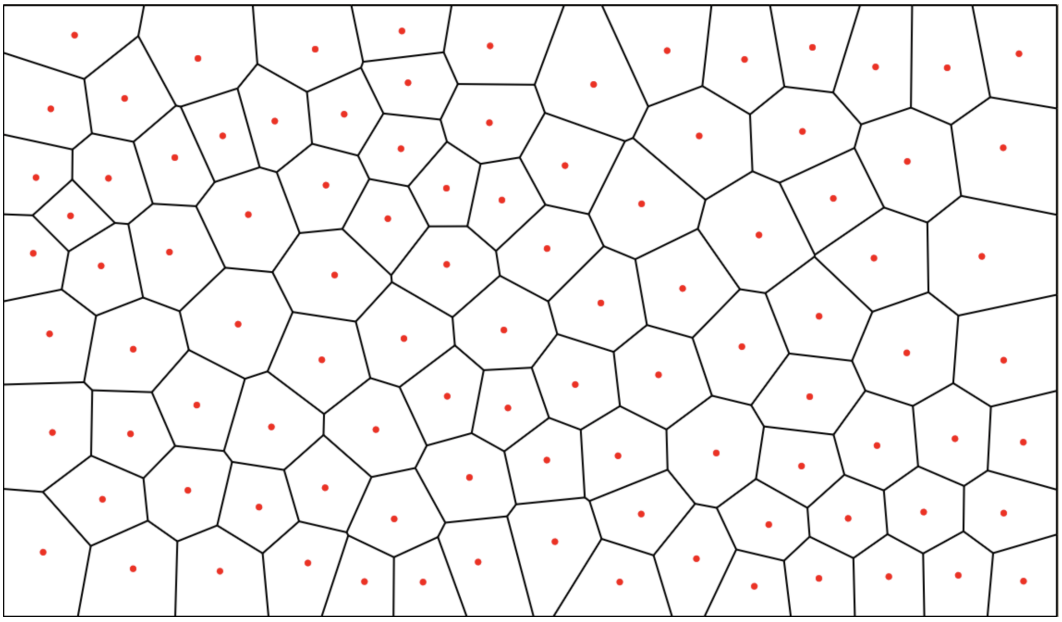

Figure 15. Voronoi polygons after 5 times of relaxation

is the continent. Otherwise, it is water. Also, the darker the polygon is, the higher the elevation it represents. As the waterline rises, the continents with lower altitudes would be flooded and become water. So in the beginning, we set all the polygons to be continents and have an initial elevation that is 0.35. If the user wants to create water, then he should manually edit them. Figure 16 is an example that divides the world into the continent and water:

4.3.3. Contour Lines

We calculated the contour lines of the map, which are determined by the elevation of each polygon. The calculation is based on the Delaunay triangle, which is related to the Voronoi diagram. Every triangle in the Delaunay triangulation corresponds to a polygon corner in the Voronoi diagram. Every polygon in the Voronoi diagram corresponds to a corner of a Delaunay triangle. Every edge in the Delaunay graph corresponds to an edge in the Voronoi graph. We got these triangles directly from d3.voronoi.triangles().

These triangles link to every site, and each site has an "elevation" attribute. We set an elevation value between 0 and 1, and then looped every edge of every Delaunay triangle. If this value is between the elevation values of the two sites to which it is connected, we marked the point represented by this value and then connected all such points. Algorithm 1 describes the idea.

We expected the user can edit the terrain by contour lines by the tools we provided. Figure 17 gives an example that draws four contour lines based on the value of waterline (blue) and several elevation values: 0.25 (red), 0.5 (green), and 0.75 (yellow).

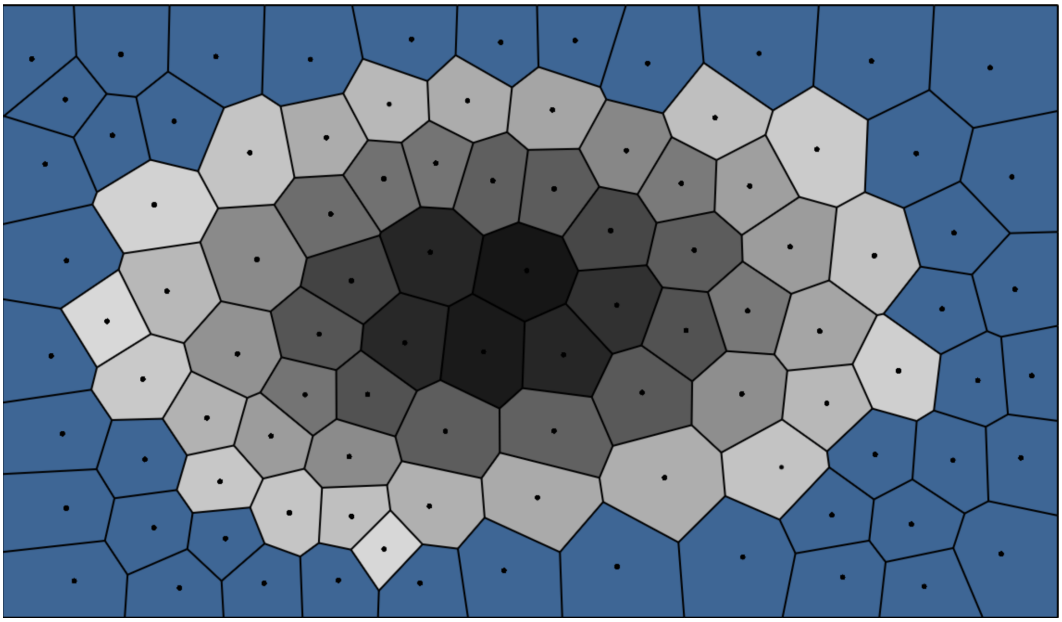

Figure 16. Continent and Water

Algorithm 1 Draw contour lines for elevation X

Require: $X \geq 0 \wedge X \leq 1$
  for every triangle T **do**
    for every edge $E1$, $E2$ in T **do**
      if $X \in$ [E1, E2] **then**
        Draw a line from $E1$ to $E2$
      end if
    end for
  end for

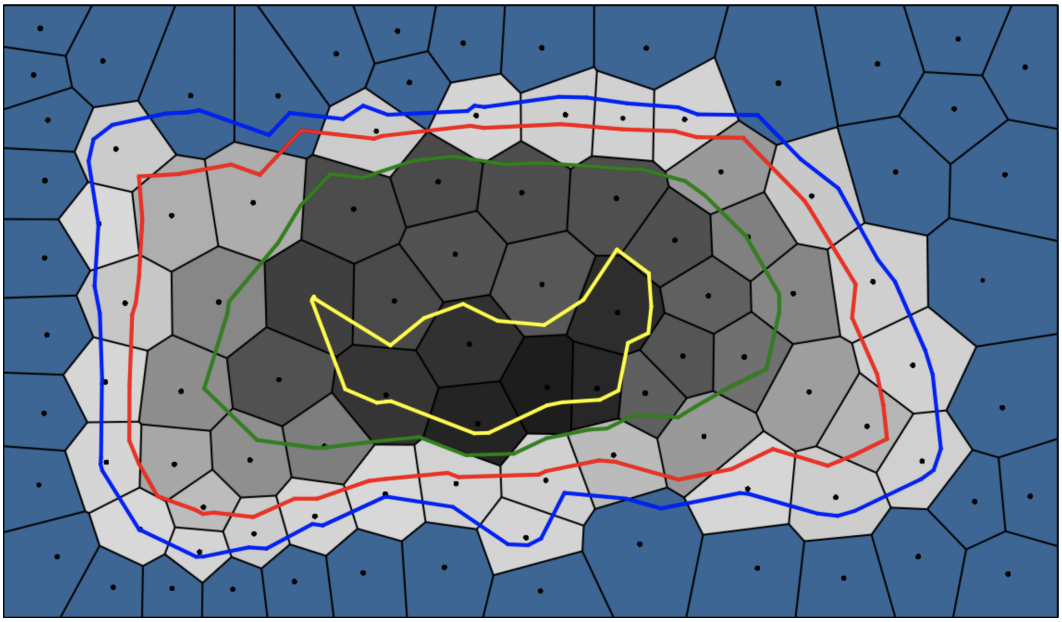

Figure 17. A map with contour lines

4.3.4. Soft Brush

As mentioned in the previous section, the user needs to edit the map manually. So we provided some methods of operations, the most important of which is the soft brush. The soft brush is a circle that follows the mouse movement, and its role is to change the specific attribute values of all sites in the range.

In terms of elevation, the user can click and drag the soft brush to change the elevation of the selected sites accordingly. Generally speaking, the longer the distance is dragged, The more values are changed. The user first selects one of the two modes of "increase" or "decrease". Correspondingly, the attribute value of the site is increased or decreased under the operation of the soft brush. And the size of the soft brush can be enlarged or reduced by the mouse wheel.

4.3.5. Layers

In our design, 5 different layers were applied to the map. Each layer comes with two modes, "view" and "edit". Once the user has checked the edit option of a layer, its view option would also be checked; once the view option of a layer is unchecked, its edit option would also be unchecked. We always keep at least one layer to ensure the normal display of the map if the user unchecked all of them.

Just like introducing the concept of elevation and assigning it to sites, we then introduced the concepts listed below (including elevation) and assigned them to sites:

1. The elevation layer represents the altitude of the area, whose value is from 0 to 1. The elevation of any place in a polygon is the same.

2. The affluence layer represents the wealth of the district (polygon), whose value is from 0 to 1. The lower the value, the poorer the area, vice versa. It is usually used to divide residential areas.

3. The desirability layer represents the attraction of the region to people, from 0 to infinity. People generally prefer to live in places with higher desirability.

4. The district layer allows the user to assign types to districts. Correspondingly, different types of districts have different background colors. In addition, the user can use the soft brush to render the city wall under this layer.

5. The building layer displays buildings of various districts. These buildings are procedurally generated, so the user can change them by changing the type of the district.

The user can view them by checking different layers, and layers can be superimposed on each other. For each polygon, we took the average of the sum of the currently selected layers of color values as its background color.

4.3.6. City Wall

As a fantasy map of the medieval city, the city wall is always an essential part. Our map generator allows the user to create walls manually.

The basic idea is to assign the "wall" attribute to each site and then give it an initial value with 0, which means there are no walls in this district. Another value is 1, representing there is the city wall in this district. The user increases or decreases the value of the wall by using the soft brush to render or erase the city wall. We used the d3.topojson library to merge all the districts with walls, which returned multiple merged polygons. Interior borders shared by adjacent polygons in merged polygons are removed, as we can see in Figure 18.

4.3.7. Types

Based on the continent and water, we subdivided the districts (including water) into 13 types, and we assumed that the value of desirability determines most of them. The equation and the assignment conditions are listed below:

$$desirability = (elevation + affluence)/2$$

1. rich: $0.8 \leq desirability \geq 1$

2. medium: $0.6 < desirability > 0.8$

3. poor: $waterline < desirability \geq 0.6$

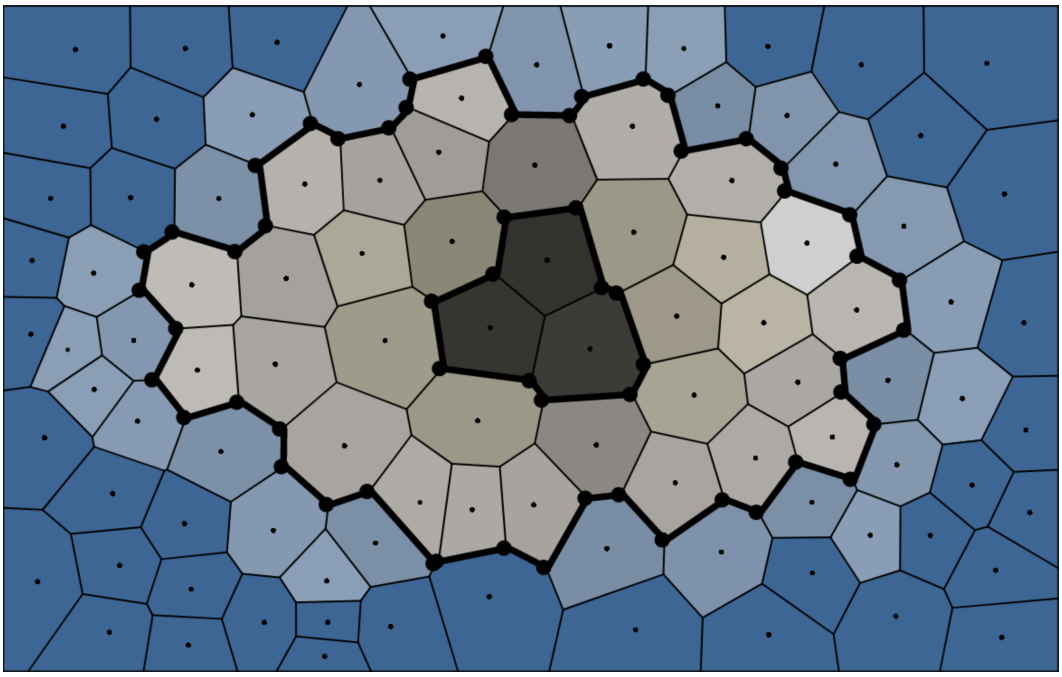

Figure 18. A map with walls

4. university: $\{rich, rich, rich, rich\} \subset \{neighbours\} \wedge \textbf{count}\{university\} < 2.5/100 * \textbf{count}\{polygons\}$
5. park: $\{poor, medium, rich\} \subset \{neighbours\}$
6. plaza: $\{poor, medium, rich\} \subset \{neighbours\}$
7. water: $elevation \leq waterline$
8. harbor: $\{poor, water\} \subset \{neighbours\}$
9. farm: $waterline < desirability \wedge elevation <= 0.45$
10. empty: $desirability = 0 \wedge (waterline < desirability \leq 0.45)$
11. castle: $\textbf{max}\, desirability$
12. military: The military always surrounds the castle
13. religious: $Math.random() > 0.99 \wedge \textbf{count}\{religious\} < 2.5/100 * \textbf{count}\{polygons\}$

In our design, sites closest to the boundary can only be assigned as empty, farm, or water. Different types of districts are represented by different background colors.

4.3.8. Buildings

One of the most important parts is the generation of buildings, which is procedural. The main idea of creating buildings is the segmentation of polygons. Because the time to build the wheel was too long, we used a third-party library called poly-split-js written by Kladess. It implemented the polygon-splitting algorithm written by Sumit Khetarpal based on Dr. Rossignac's research. The algorithm splits polygons into any number of equal areas while ensuring the minimum length of line based cuts. It also works for both convex and concave polygons, as long as they do not have non-manifold vertices, and can be traversed with a single loop. In a nutshell, this algorithm is designed to divide a polygon into two by their area ratio. It receives two parameters: the vertex set of the polygon to be split and the area ratio of the polygons to be input. It returns the vertices sets Polygon1 and Polygon2 of the two split polygons, and a secant line called Cutline.

Before generating the buildings, we first introduced the concept of a block. A district may have many blocks, and a block may contain many buildings. We split all polygons except type water, then we got the preliminary version of the city map. Figure 19 is an example of implementation.

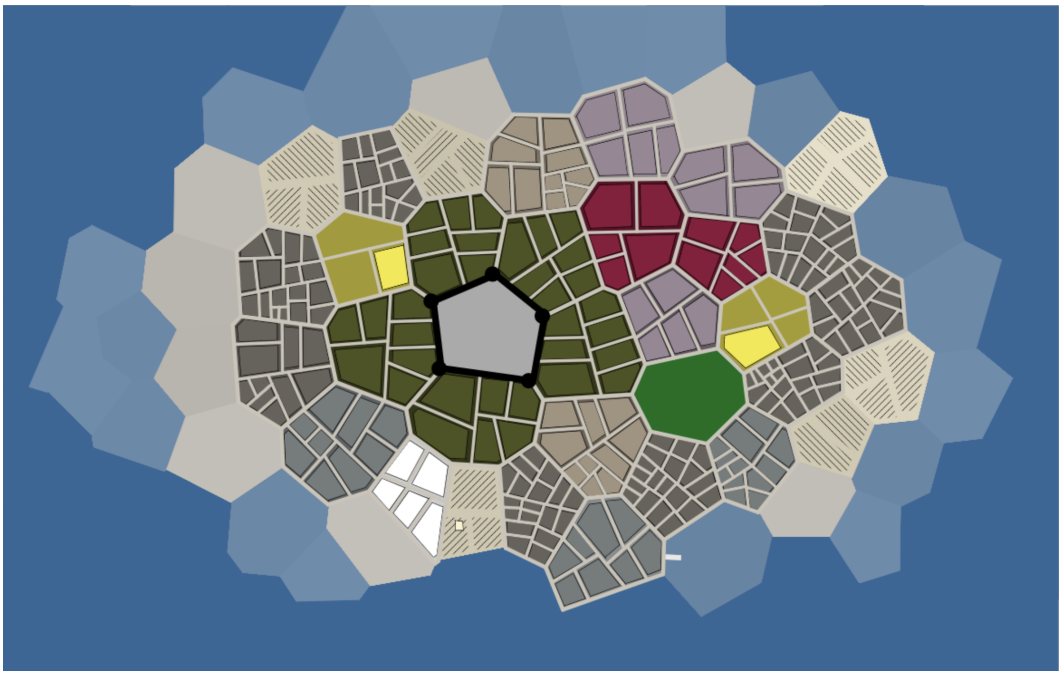

Figure 19. A city map with blocks

Then we split these blocks as before, and the small polygons returned this time are treated as buildings, which are located in blocks. Since the map would be filled with polygons in an instant, we designed such a mechanism, and the precondition is the map can be zoomed in or out normally. When the map is zoomed in to a certain extent, these buildings would be displayed. On the one hand, the map looks much better, and on the other hand, the rendering

work of the browser has been lightened, and the performance of the generator is improved. Figure 20 is the implementation.

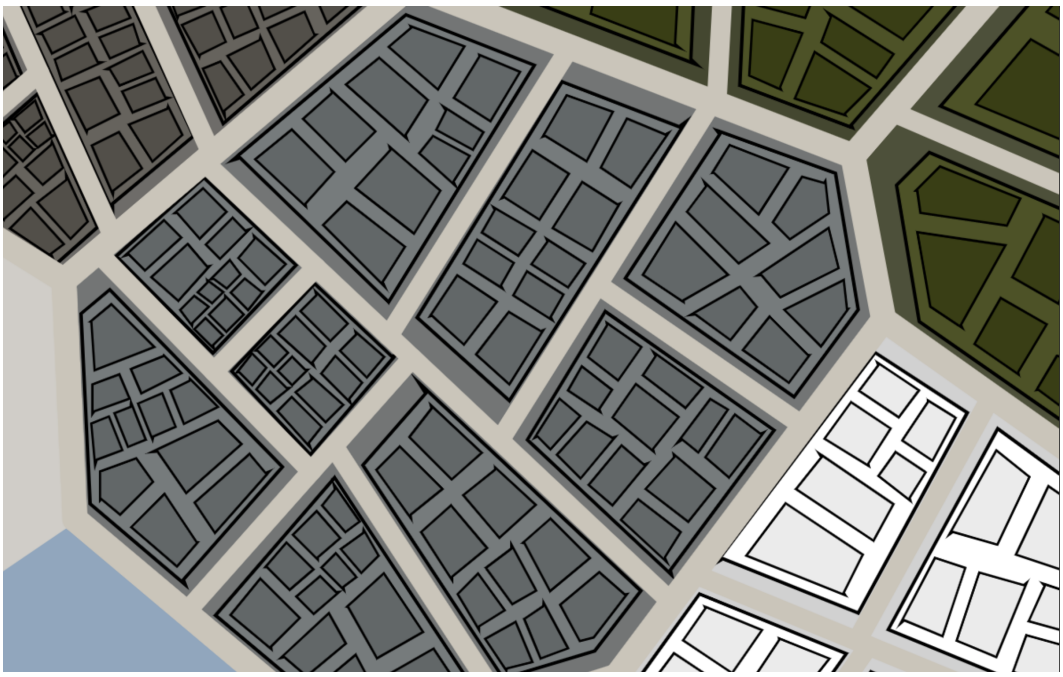

Figure 20. A city map with buildings

4.3.9. Streets

A feature of this map is the randomly generated street names. As we mentioned before, the polygon-splitting algorithm returns two polygons and a cut line. This cut line is the key to our implementation of the streets. Because there are no streets between the buildings, we only considered the streets in the blocks and the streets between districts.

First, each edge of each district is a street, and each edge of each block is also a street. Each cut line participating in the split block is also a street.

Considered that each polygon and each of its neighbors share a common edge, we avoided repeating these edges by hashing each rendered edge. Because one edge consists of two vertices, so each edge needs to be hashed twice. Then we put these hashes into a global variable. When an edge is ready to render, we took the two hash values of this edge and compared them with all the hashed edges in the global variable. If there is the same, skip rendering it, otherwise, render and add these two hashes to the global hash collection. After rendering each edge, we cleared the hash collection in the global variable for the next rendering.

Essentially, each street has its own street name. First, we got a collection of 400 random street names, and then randomly placed these street names at the midpoint of each street. Because each street is only traversed once, there

would be no situation where two street names appear on one street. Then we adjusted the angle between the street name and the x-axis so that it can always be displayed well on the street. Figure 21 shows the streets with street names.

As the number of polygons increases, the street would increase, and street names would inevitably repeat. We hope to solve this problem in future work.

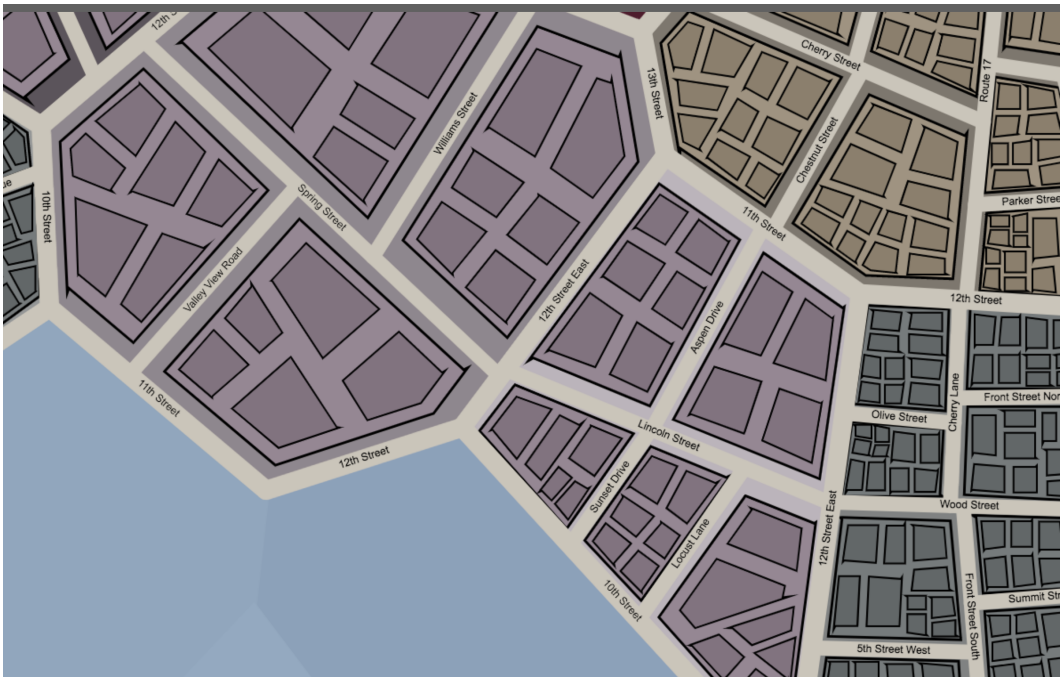

Figure 21. A city map with streets

5. References

[1] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures, 2000.

6. Appendices