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

List of Tables

List of Figures

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."

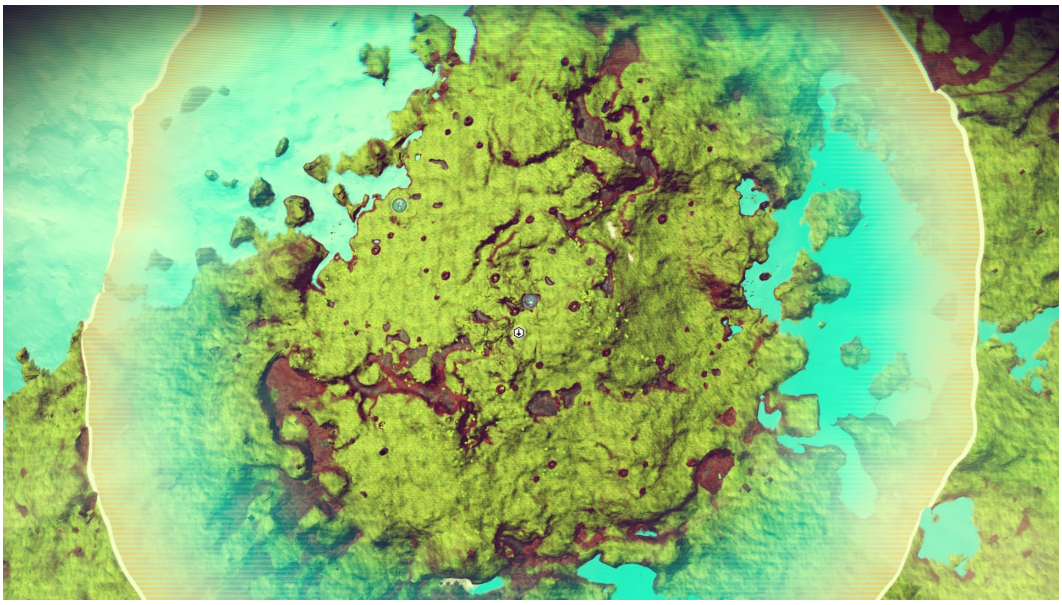


Figure 1. A screenshot of a planetary terrain region of "No Man's Sky"

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."

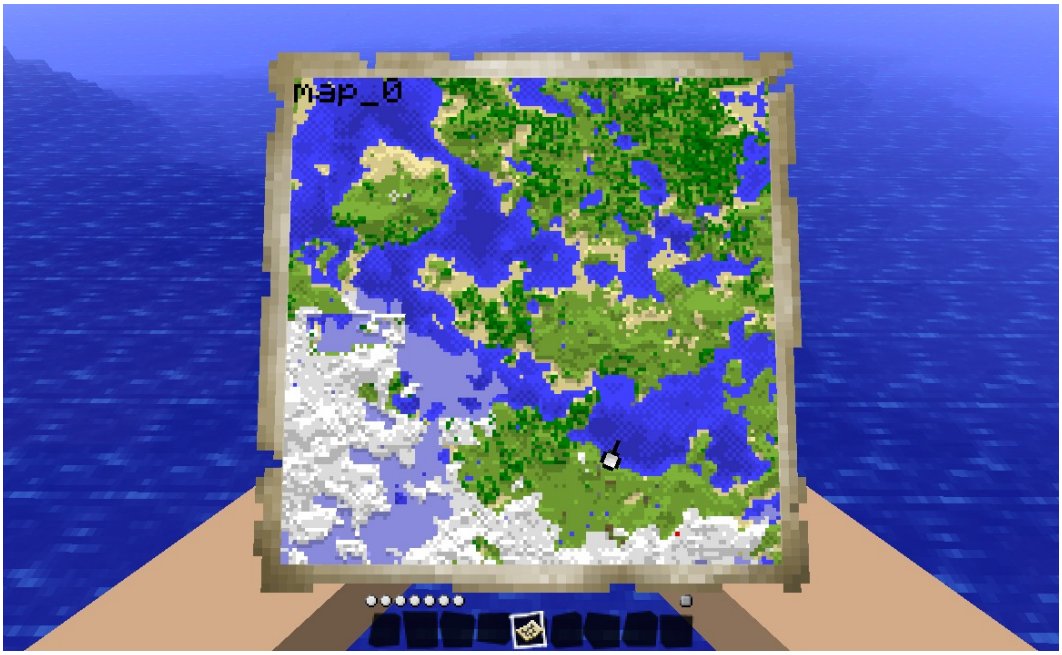


Figure 2. A screenshot of "Minecraft"

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

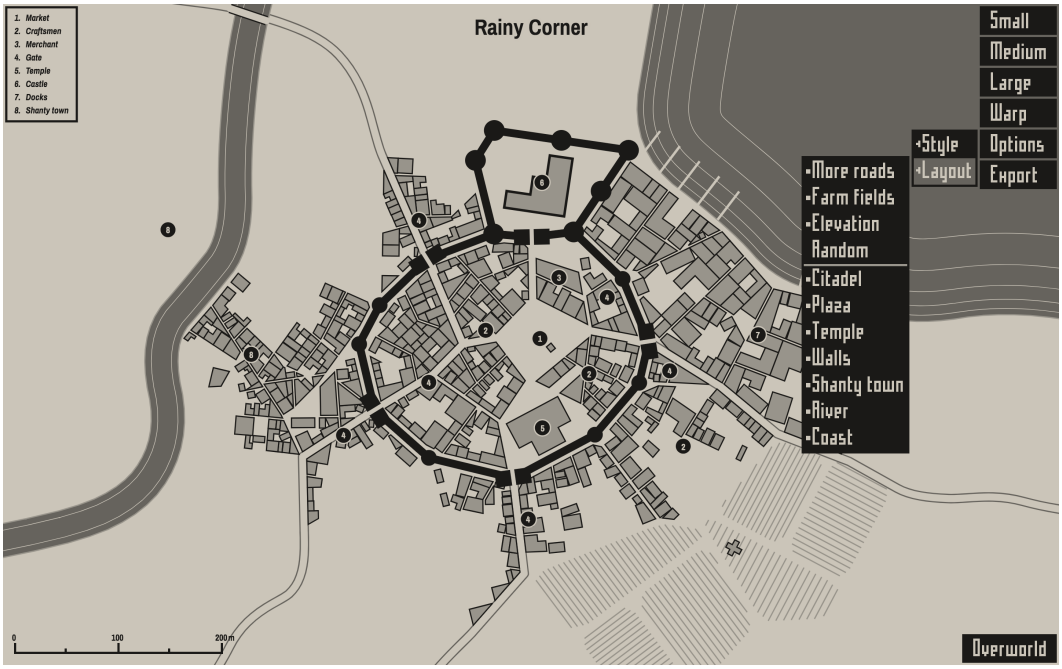


Figure 3. A screenshot of the Medieval Fantasy City Generator

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.

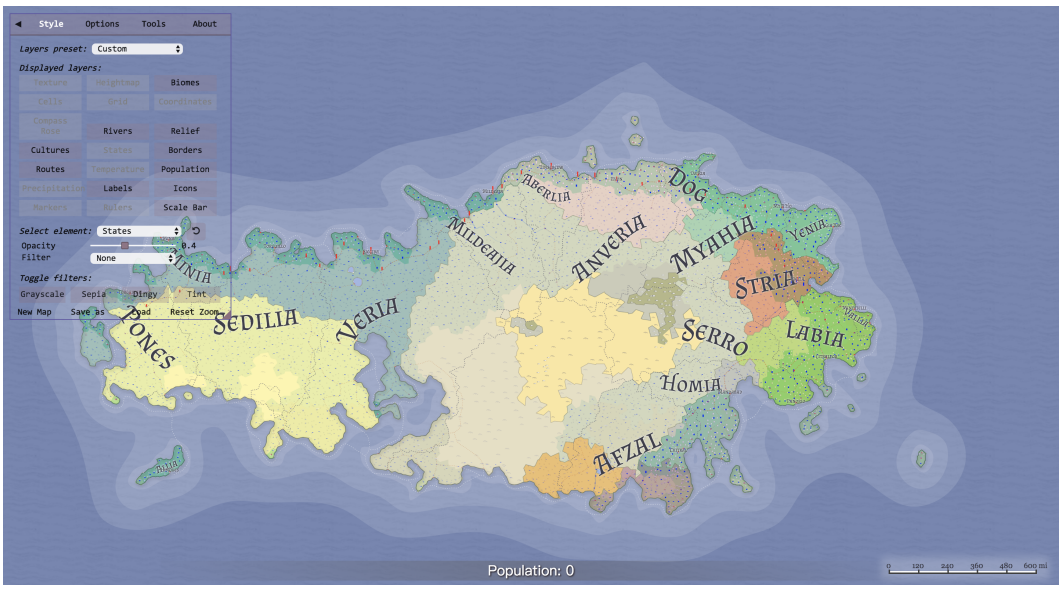


Figure 4. A screenshot of the Azgaar's Fantasy Map Generator

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.

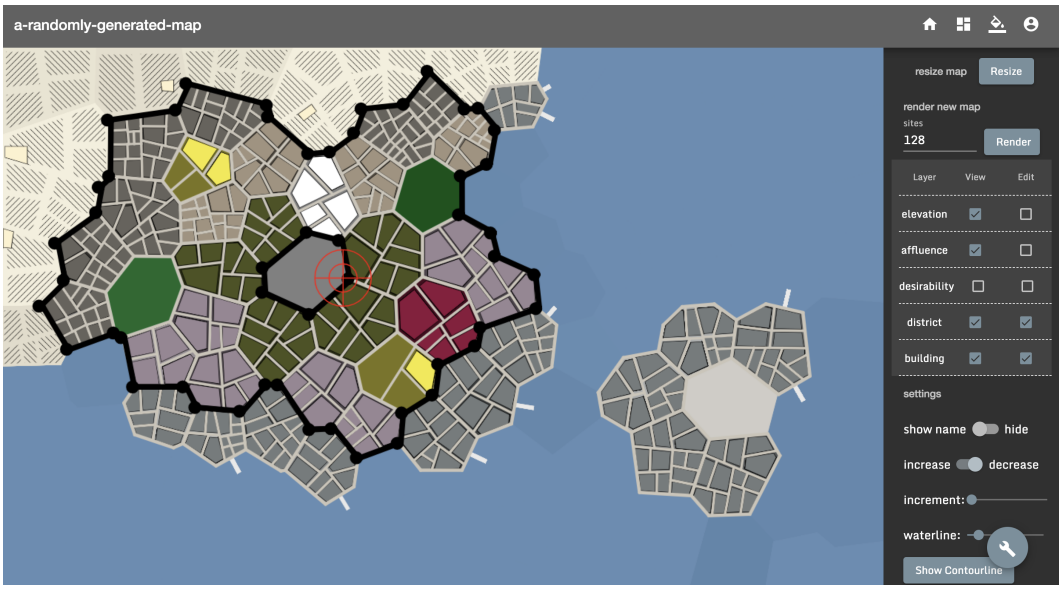


Figure 5. A screenshot of the current project

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 [2] in the first meeting. Both the advisor, who is also the project sponsor, and the developer, who is also the author, are the stakeholders of the project.

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 able to:
   (a) Log in with a unique email and password
   (b) Log out
   (c) Edit my personal profile by changing my email, first name, last name, or password
2. As an admin, I need 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 need to be able to:
   (a) Sign up with a unique email, first name, last name, and password
   (b) View all of the maps that I have created
   (c) View all of the maps shared with me
   (d) Search for maps using any combination of the map's 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.

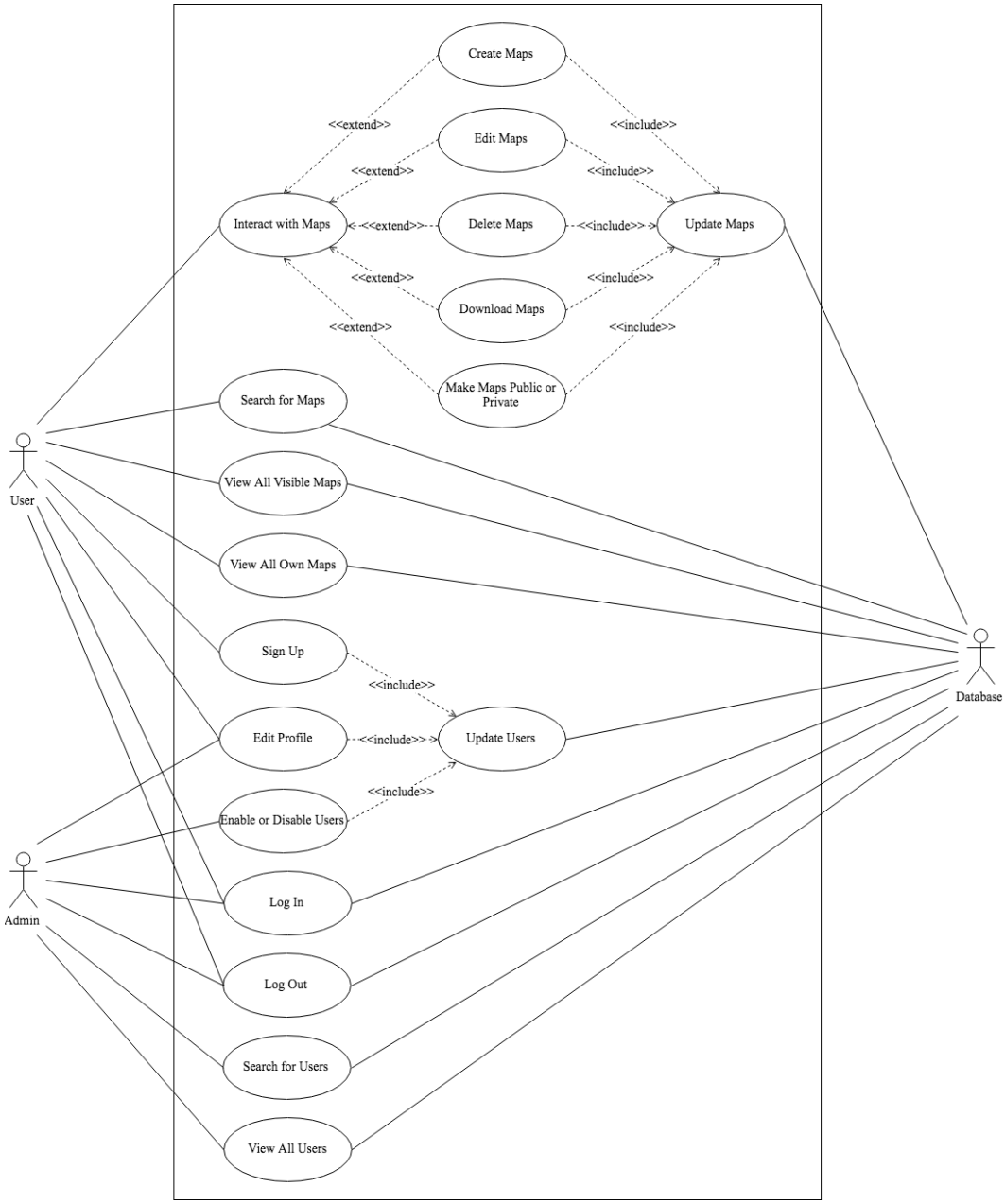


Figure 6. Use Case Diagram

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 [5]. For this application, we focused on the following requirements:

1. Security:
    (a) Input Validation:
        i. All user supplied data is validated at both client and server
    (b) Password Encryption:
        i. Password must be encrypted
        ii. Not even the system administrators shall see clear text 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. Only use secure cookies
        ii. Only use "HttpOnly" cookies
        iii. Always change the session id after login
2. Performance:
    (a) High performance of rendering map:
        i. The map rendering engine must be fast enough to support a viable user experience

2.4. Selection of Software Development Life Cycle Model

We analyzed and summarized the following possible risks:

1. Lack of experience developing semi-automatic map generation algorithms
2. Uncertainty regarding third-party libraries that would provide assistance
3. 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. If we use the Waterfall model for software development, then we have to be clear with all the development requirements beforehand as there is no scope of changing the requirements once the project development starts. But for Agile methodology, on the other hand, is quite flexible, and allows for changes to be made in the project development requirements even after the initial planning has been completed. So finally, the Agile model was chosen, which is shown in Figure 7.

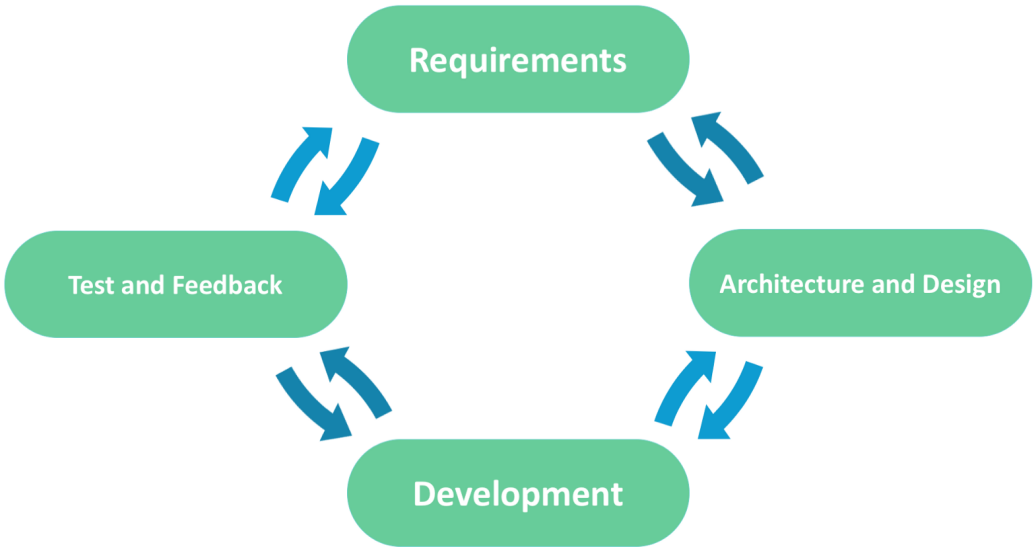


Figure 7. Agile Development Life Cycle Model [6]

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 SPA provides more responsive user experience and clean separation between data and views. 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 Node.js. Moreover, Node.js is written using JavaScript and is the same technology as the frontend components. This makes it easier for the developer to program backend services and frontend user interfaces simultaneously. 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 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 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.

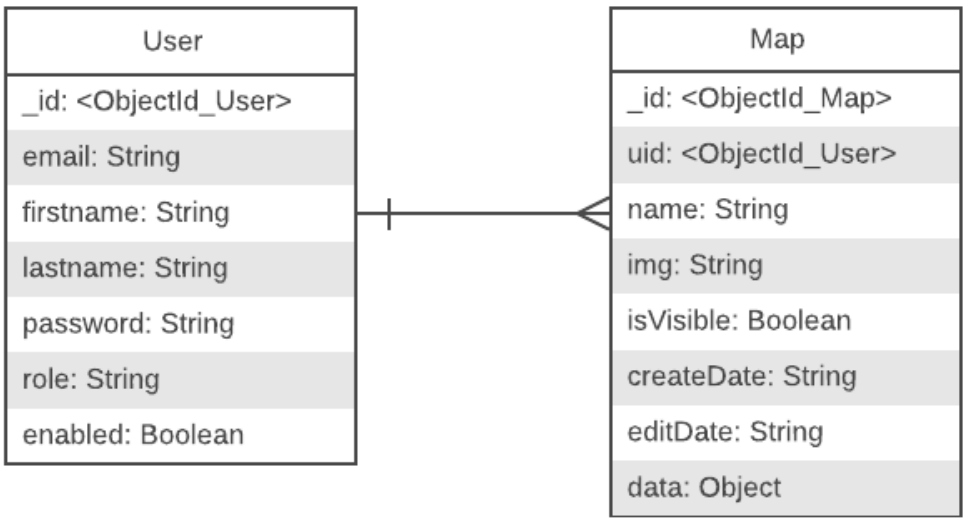


Figure 8. Entity Relationship Diagram

3.3. REST API Design

An API is an application programming interface. It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allows the client to talk to it.

REST determines how the API looks like, which 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 should 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. e.g., clients may call the API to get a UI widget rendering code, which is allowed.

We found ourselves did not implement some of them, such as the Cacheable, Layered system, and Code on demand. But we were still making a RESTful API, although not "truly RESTful [7]." We hope to achieve them all in our future work. However, our APIs as shown in Table 1, also implemented the rest of these principles:

1. All endpoints follow a canonical form using a sequent of nested collection/item types. This practice conforms with the provision of a Uniform interface.

2. GET requests do not alter the state, which also conforms with the provision of a Uniform interface.

3. The client application and server application are able to evolve separately without any dependency on each other. And the client only knows resource URIs, which conforms with the provision of a Client–server.

4. The server will not store anything about the latest HTTP request client made. It will treat each and every request as new. No session, no history. This practice conforms with the provision of a Stateless.

3.4. Map Generator Design

3.4.1. Canvas and SVG

Since the project is a web application, the map generator must be able to run in the web page. We need to choose a 2D HTML graphics tool to render the map. Two technologies are available: SVG and Canvas.

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 redraw the shape. Canvas draws 2D graphics, on the fly (with JavaScript) by

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

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.

Because the map generator allows the user to annotate and edit the map, every time the user edits the map, the map will be redrawn, which contains

thousands of objects, so we chose Canvas. By the way, SVG renders slowly if there are plenty of complex objects (anything that uses the DOM a lot will be slow).

3.4.2. Graphics Libraries

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

3.4.3. Map

Before we started to develop the map generator, we needed to understand what is a map and then decide what data structures and methods we are going to use to write. A map is a two-dimensional, abstract representation of the surface of the world. There are three main ways to generate a map: noise functions, random partitioning, or a hybrid of the two. Most procedural map generators use noise functions (midpoint displacement, fractal, diamond-square, Perlin noise, etc.) to generate a randomized map, but the results often look evenly artificial. So we decided to use random partitioning to model the mapping constraints. And we defined the following layers of the map:

1. elevation: the elevation layer represents the altitude of the area. The elevation of all locations in a polygon is considered the same.
2. affluence: the affluence layer represents the wealth of the district (polygon). The lower the value, the poorer the area, vice versa. It is usually used to classify residential areas.
3. desirability: the desirability layer represents the attraction of the region to people, from 0 to infinity, which determined by elevation and affluence. People generally prefer to live in places with higher desirability.
4. district: the district layer allows the user to assign types to districts. Correspondingly, different types of districts have different background colors. As a fantasy map of the medieval city, the city wall is always an essential element. Our map generator allows the user to create walls manually, so the user can use the warp tool to render the city wall under this layer.

5. building: 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.

Based on these layers, we have expanded the districts. A city is always composed of different regions, so we introduced the concept of type:

1. rich: represents the rich area, the value of desirability is generally high
2. medium: represents the middle-class area, the value of desirability is generally medium
3. poor: represents slums, the value of desirability is generally low.
4. university: represents universities and always appearing in the middle-class areas and the rich areas.
5. park: represents parks and generally appearing at the junction of rich areas, slums, and middle-class areas.
6. plaza: represents plazas and generally appearing at the junction of rich areas, slums, and middle-class areas.
7. water: represents rivers, lakes, sea, or ocean.
8. harbor: represents harbors, and there is usually a deck near the sea.
9. farm: represents farms and usually there is a farmer's house on a large farm.
10. empty: represents empty areas.
11. castle: represents a castle. A city map has only one castle, and the castle is always surrounded by large militaries.
12. military: represents the military. The military always surrounds the castle
13. religious: represents the area with churches or temples.

After introducing these types, 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 district placements and shapes, then the user can start editing the map.
2. Create a map by entering the resolution of the map in the menu.

3. View the different layers of the map, these layers can be superimposed on each other.
4. Edit the different layers of the map.
5. Toggle the display of street names.
6. Set the waterline or elevation level
7. Display contour lines or not.
8. After selecting one or more than one layer under “edit” mode, the user can change the size of the ”warp tool” by using the mouse wheel, which determines the area where the map will be edited.
9. Edit the map by clicking and dragging the “warp tool.”
10. The districts and buildings are procedurally generated.
11. Allow to change the type of the current district by right-clicking and selecting a new type from the context menu.
12. Zoom in and zoom out the map by pressing on the alt key and using the mouse wheel.
13. Use a specialized button to resize the map on the menu.
14. Save the map.
15. Download the map.

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 developer 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.

4.1.1. Overview of Frontend Folder Structure

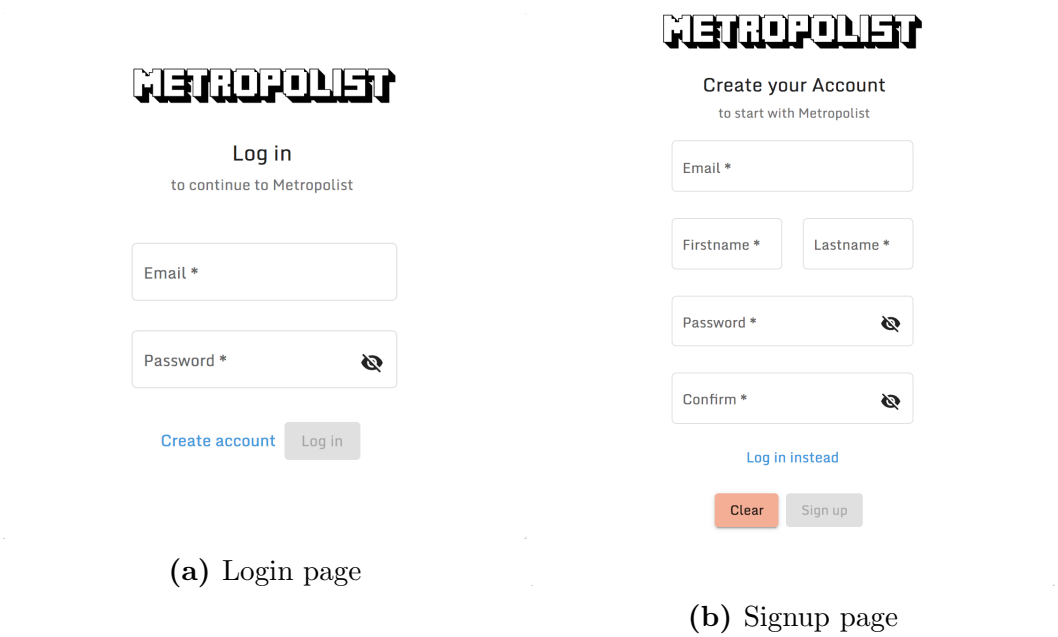
The Angular uses the concept of Angular Modules to group the related features. This gives us a helpful starting point to organize the folder structure. Each Module should get its folder named after the Module name. The Angular does not make any distinction between the Modules. However, based on how we made use of modules, we classified our modules into such categories: guard, model, pipe, service, and component. Figure 9 shows the final structure of our frontend application.

4.1.2. GUI Implementation

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 10b is the interface of the “Sign Up” page.

Figure 9. Frontend Folder Structure



(a) Login page

(b) Signup page

Figure 10. Login and Signup pages

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 10a 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 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.

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

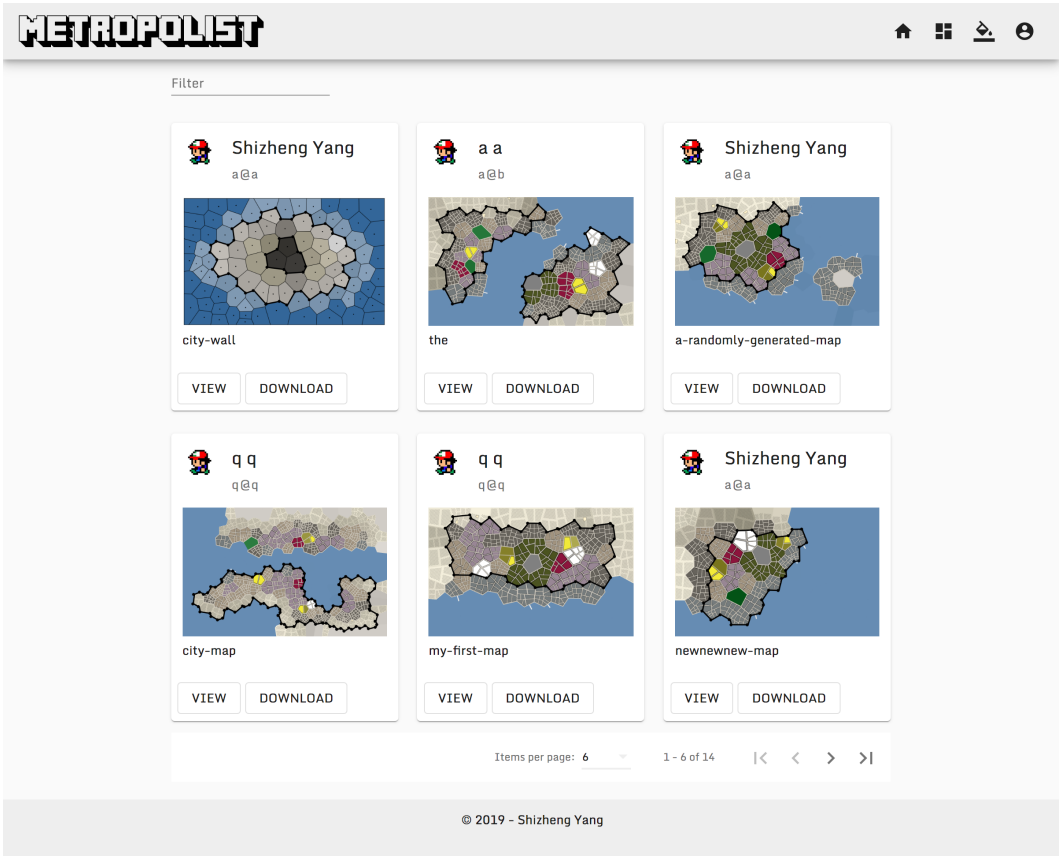


Figure 11. Community page

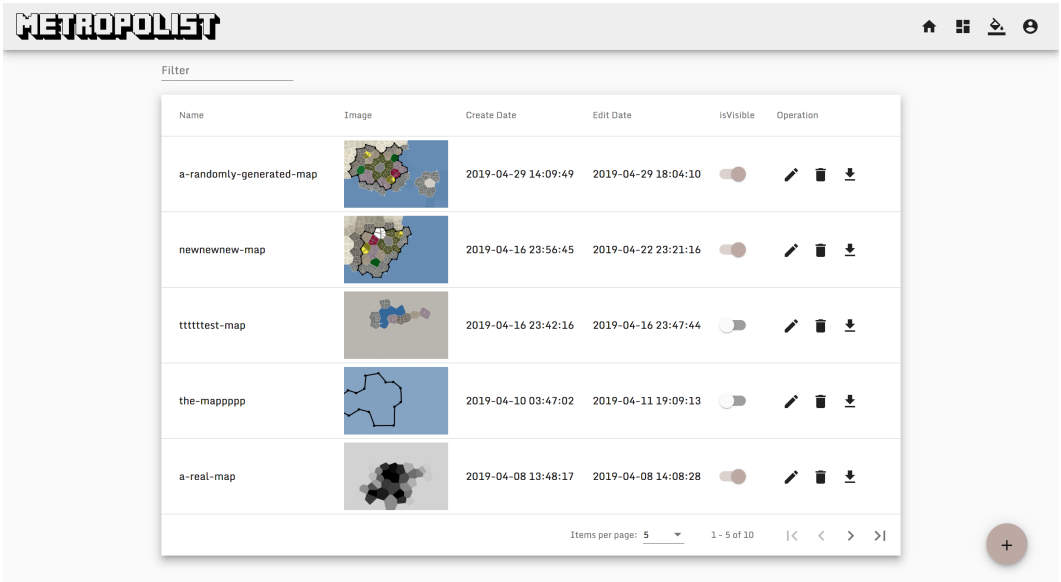


Figure 12. User dashboard page

page.

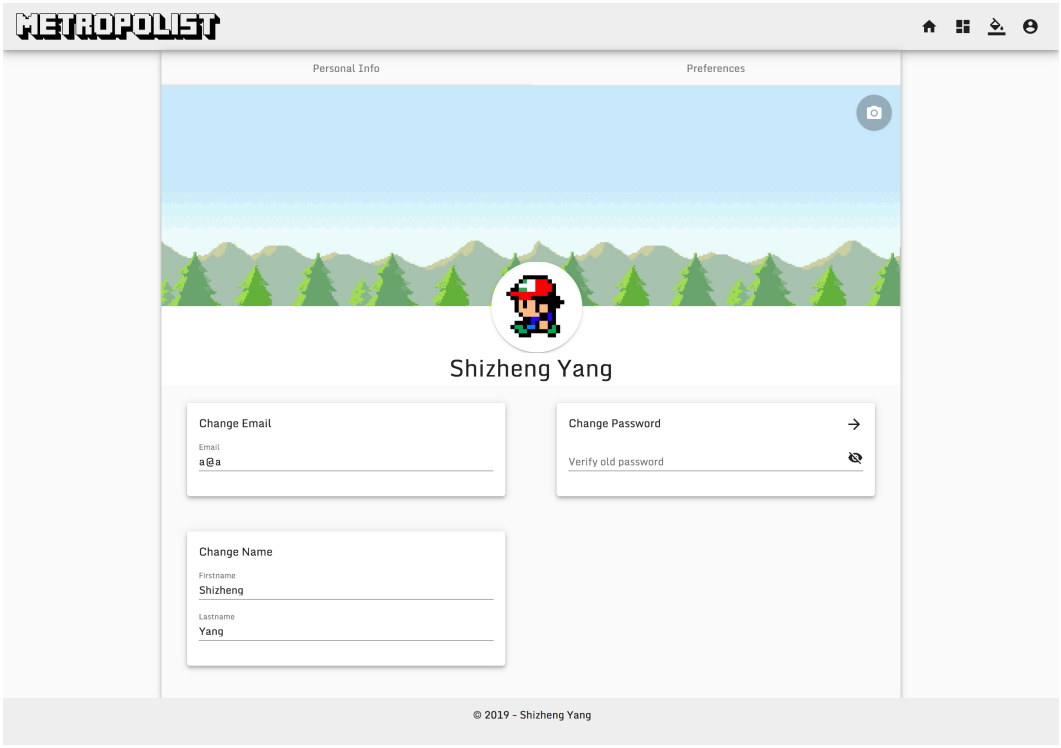


Figure 13. Profile page

4.2. Backend

The backend consists of the server, database, and APIs. The server running on the developer'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.

Figure 14 shows the final structure of the backend application powered by Node.js.

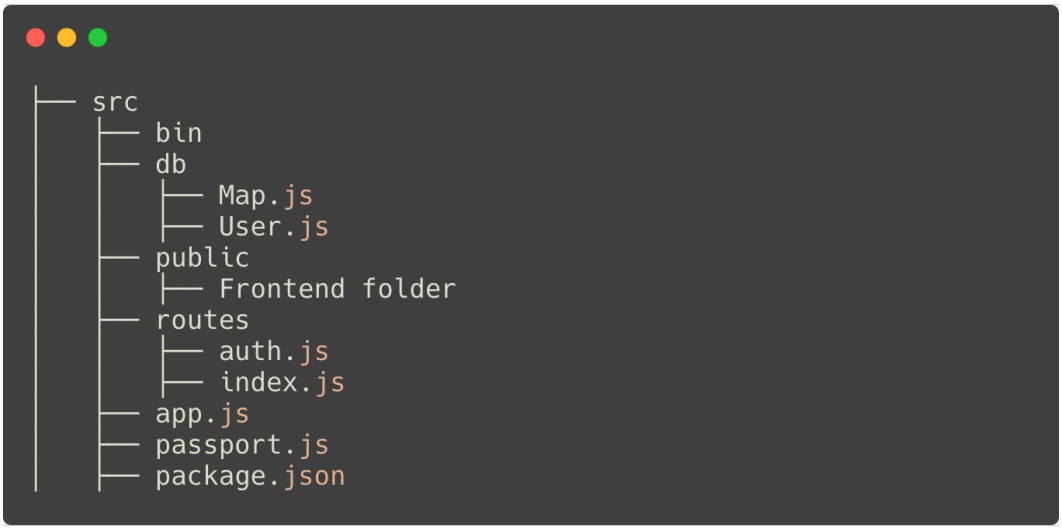


Figure 14. Backend Folder Structure

Figure 15 shows a part of implemented endpoints, which are located in src/routes/index.js.

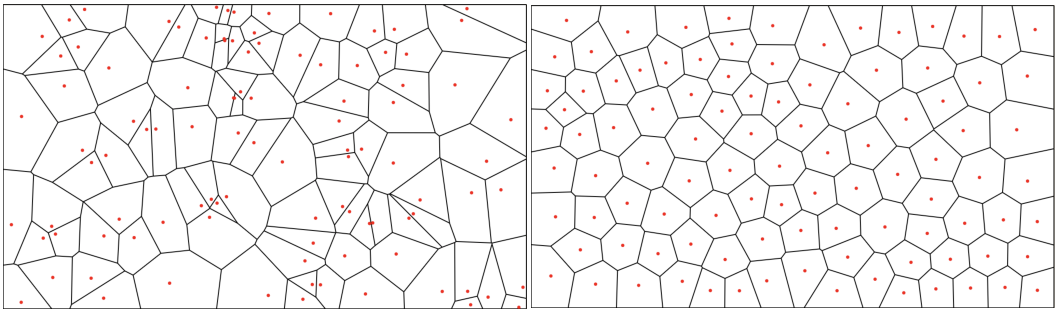
```
/*============================================================================
                                User API
============================================================================*/
router.get('/metro/api/v1/users/:uid', (req, res, next) => {...});
router.get('/metro/api/v1/users', (req, res, next) => {...});
router.patch('/metro/api/v1/users/:uid/password', (req, res, next) => {...});
router.put('/metro/api/v1/users/:uid/email', (req, res, next) => {...});
router.put('/metro/api/v1/users/:uid/name', (req, res, next) => {...});
router.put('/metro/api/v1/users/:uid/password', (req, res, next) => {...});
router.put('/metro/api/v1/users/:uid/enabled', (req, res, next) => {...});

/*============================================================================
                                Map API
============================================================================*/
router.get('/metro/api/v1/maps', async (req, res, next) => {...});
router.get('/metro/api/v1/maps/:mid', (req, res, next) => {...});
router.get('/metro/api/v1/users/:uid/maps/', (req, res, next) => {...});
router.post('/metro/api/v1/maps', (req, res, next) => {...});
router.put('/metro/api/v1/maps/:mid', (req, res, next) => {...});
router.delete('/metro/api/v1/maps/:mid', (req, res, next) => {...});
```

Figure 15. Endpoints

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 triangu-



(a) Before relaxation **(b)** After 5 times of relaxation

Figure 16. Voronoi polygons before relaxation and after relaxation

lation of a set of two-dimensional points. Figure 16a is an example of 100 dots and the corresponding 100 polygons:

As we can see the tessellation of these random points is irregulars. In the sense that ratio between the largest and smallest region is overly large. To reduce this variance and obtain a more regular partition, we used a variant of Lloyd relaxation, also known as Voronoi iteration or relaxation, which has the effect of making the polygons evenly distributed. Lloyd relaxation replaces each point by the centroid of its polygon. The more iterations, the more regular the polygons get. Figure 16b 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 include a height layer, so we introduced the concept of "elevation." We called the 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 is normalized to the range 0..1. Before making the height map, we defined the waterline, whose value is also normalized to the range 0..1. If the value of elevation is greater than the waterline, the polygon represented by the site is a land element. Otherwise, it is water. Also, for viewing the map, darker colors denote higher elevations. As the waterline rises, the land with lower altitudes are flooded and become water. So in the beginning, we set all the polygons to be land and have an initial elevation that is 0.35. If the user wants to create water, then he should manually edit them. Figure 17 is an example that divides the world into the land and water polygons:

4.3.3. Contour Lines

We calculated the contour lines of the map, which are determined by the site elevations. The calculation of the contour lines is based on the Delaunay triangulation, which is related to the Voronoi diagram. Every triangle in

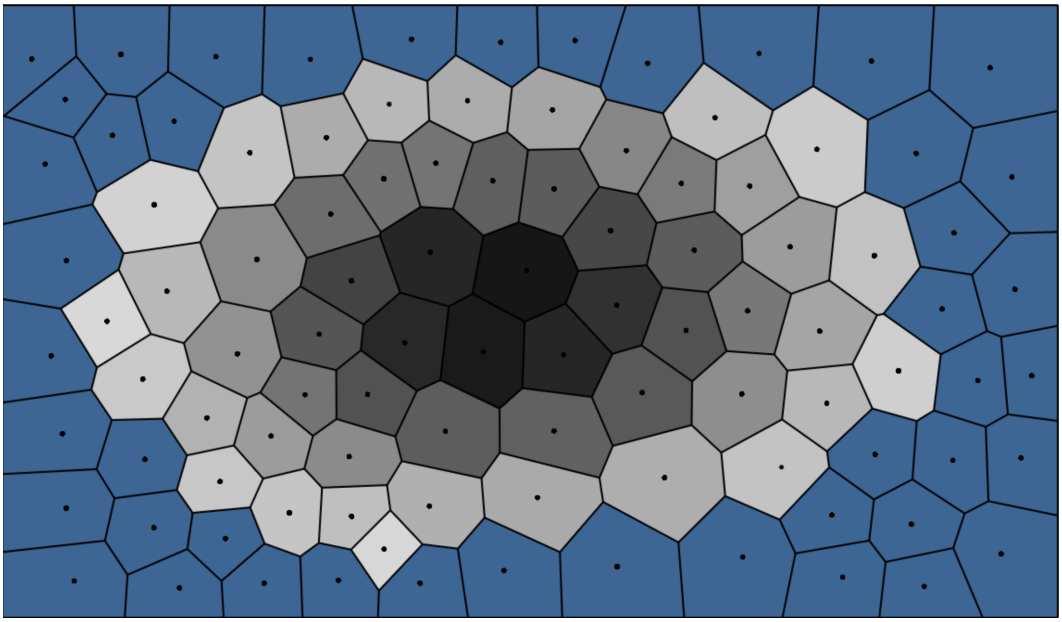


Figure 17. The land surrounded by water

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 computed these triangles directly from d3.voronoi.triangles(). Figure 18 is the Delaunay triangulation graph based on the Voronoi diagram after relaxation.

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 over 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. This process is given in pseudocode form in Algorithm 1.

Algorithm 1 Draw contour lines for elevation X

Require: $X \geq 0 \wedge X \leq 1$
  for every triangle T **do**
    for every edge E_1, E_2 in T **do**
      if $X \geq$ **elevation**$(E_1)\&\&X \leq$ **elevation**(E_2) **then**
        Draw a line from E_1 to E_2
      end if
    end for
  end for

We expected the user to edit the terrain contour lines by using the tools we provided. Figure 19 gives an example that draws four contour lines based on the value of waterline (blue) and several elevation values: 0.25 (red), 0.5

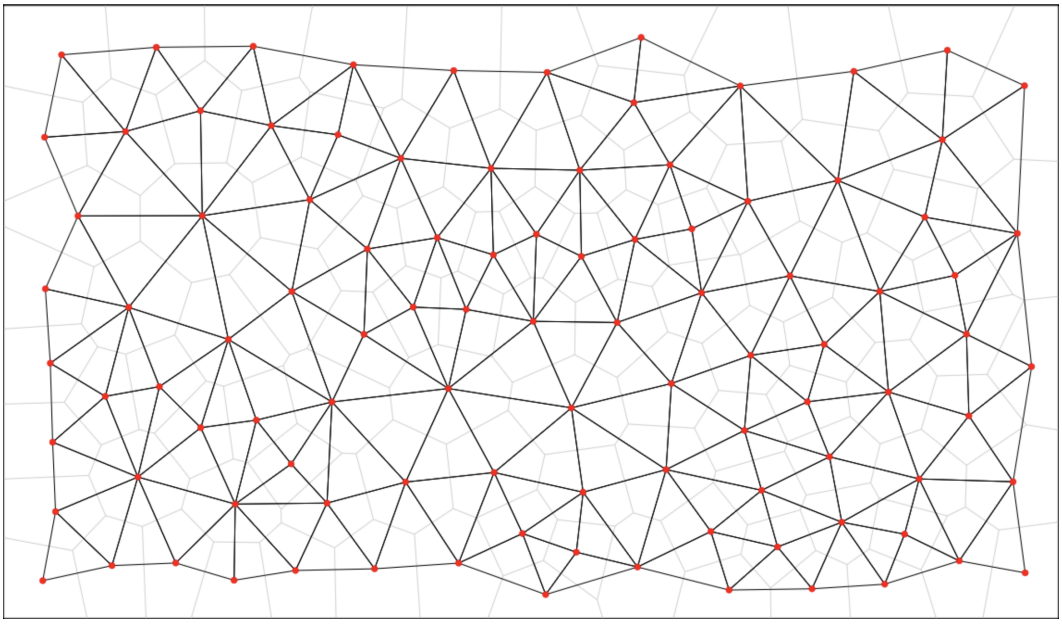


Figure 18. Delaunay triangulation

(green), and 0.75 (yellow).

4.3.4. Warp Tool

As mentioned in the previous section, the user needs to edit the map manually. So we provided several interactive methods, the most important of which is the warp tool. The warp tool is rendered as 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, we assigned a property called "delta" for each site object, which is the factor that determines the elevation. The user can click and drag the warp tool to change the elevation of the selected sites accordingly. In general, the longer the distance is dragged, The more values are changed. The value of elevation is the product of the delta and the increment value manually set by the user. Within the radius of the warp tool, the elevation value of sites in the center position changes more than the edge position.

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 warp tool. Also, the size of the warp tool 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

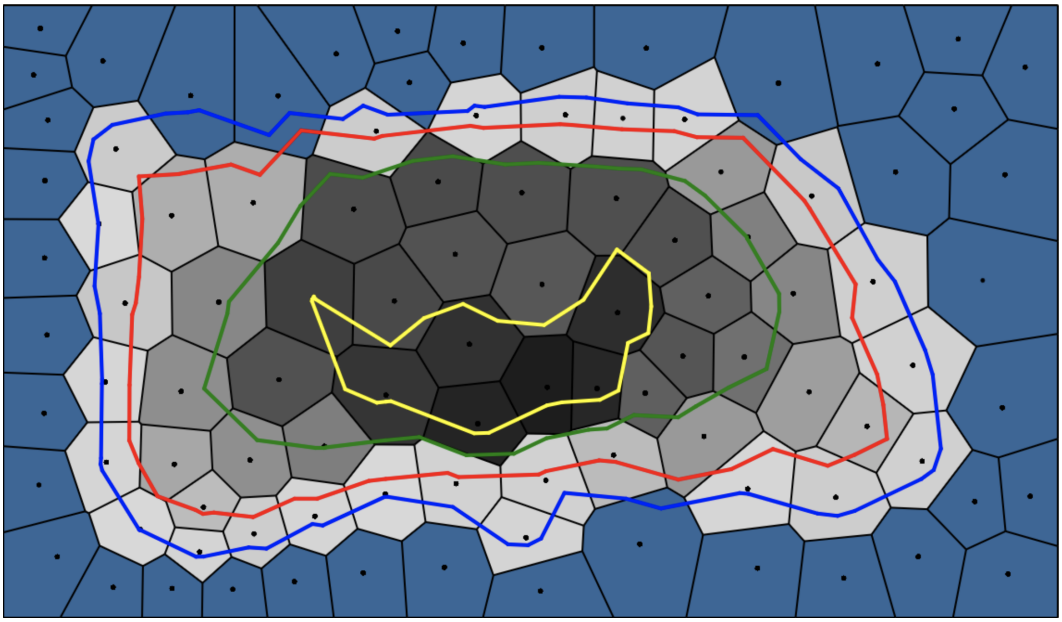


Figure 19. A map with contour lines

layer is unchecked, its edit option would also be unchecked. We always keep at least one layer viewable to ensure the normal display of the map if the user unchecks all of them. 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.

Just like introducing the concept of elevation and assigning it to sites, we then assigned "affluence", "desirability", "district", "buildings" as attributes to each site.

4.3.6. City Wall

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 warp tool 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 20.

4.3.7. Types

As we mentioned in the design phase, the city map was subdivided into 13 types, and we assumed that the value of desirability determines most of them. The equation and the assignment conditions are listed below:

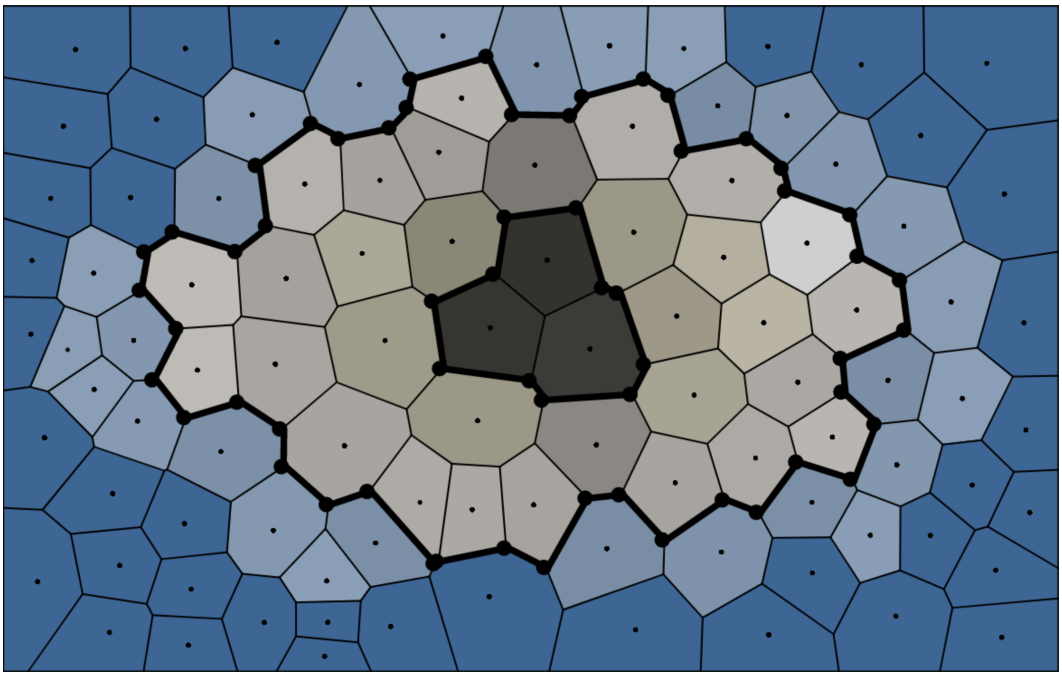


Figure 20. A map with walls and classification

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

1. rich: $0.8 \leq desirability \geq 1$
2. medium: $0.6 < desirability > 0.8$
3. poor: $waterline < desirability \geq 0.6$
4. university: $\{rich, rich, rich, rich\} \subset \{neighbours\} \wedge \mathbf{count}\{university\} < 2.5/100 * \mathbf{count}\{polygons\}$
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: $\mathbf{max}\, desirability$
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. Blocks

One of the most important parts is the generation of buildings, which is procedural. 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.

The main idea of creating buildings is polygon splitting. Because the time to build the wheel was too long, we used a third-party library called poly-split-js.js written by Kladess [4]. It implemented the polygon-splitting algorithm written by Sumit Khetarpal [3]. 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.

We split all polygons (except water, park, and castle) into blocks, to obtain the preliminary version of the city map. Figure 21 is an example of this technique.

4.3.9. Streets

A feature of this map is that of 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.

Considering 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.

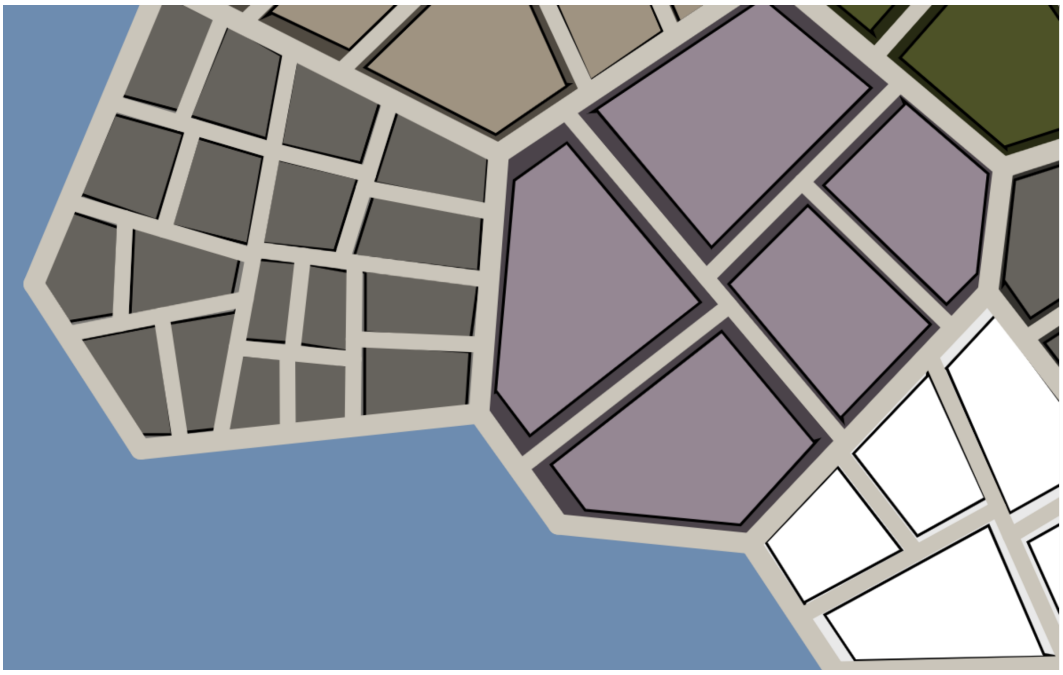


Figure 21. A city map with each district divided into sub-blocks

Each street has its own street name. First, we obtained a collection of 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 22 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.

4.3.10. Buildings and Details

After getting blocks, we split them again as before. The small polygons returned this time are treated as buildings, which are located in blocks. We have previously defined several types of districts, some of which do not need to be split, for example: water, castle, and park. Some of them require unique segmentation methods, such as the farm.

For farms, it only needs to be split 2 or 3 times, unlike other ones that have been split a dozen times. These split larger polygons are pieces of the farm, we randomly made some diagonal lines to be treated as fields, and then changed the background color to the wheat color that echoes the farm. With the field, of course, the farmer's house is indispensable. When we divide the buildings, we only need to keep a small polygon as a house. And not every farm needs a house, so we reduce the probability of splitting farms, let it occasionally do a building-level segmentation.

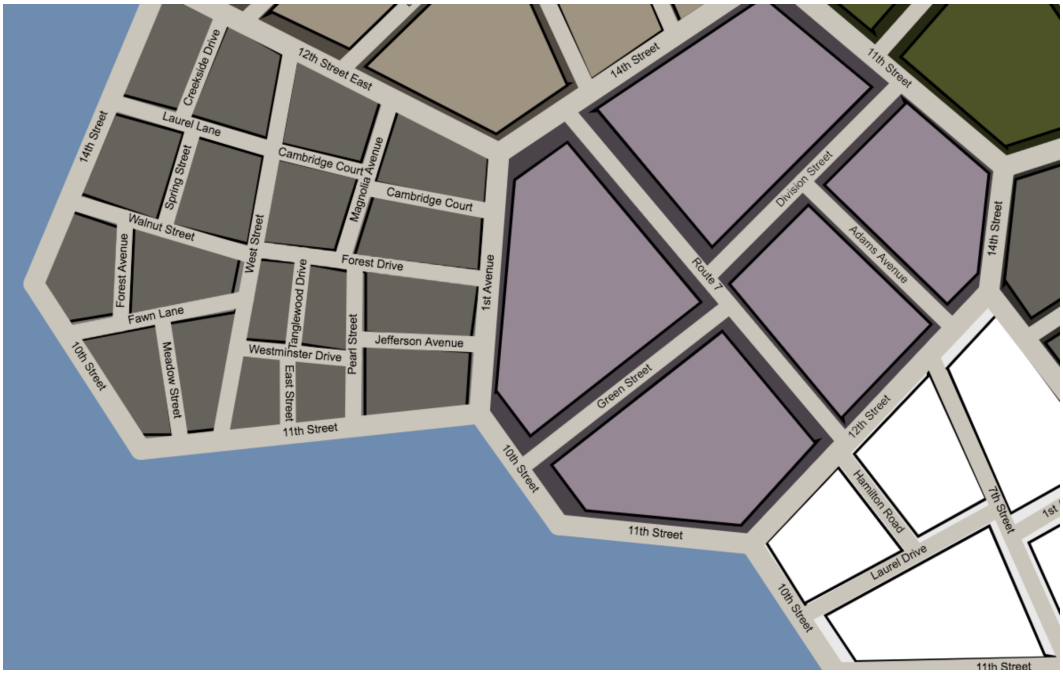


Figure 22. A city map with streets

Besides, for the type harbor, we got the inspiration from the Medieval Fantasy City Generator and introduced the concept of deck, which is randomly generated and of moderate length. Figure 23 is the implementation of buildings and details.

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. For example, when the map is enlarged to 2 times, the street names will be automatically displayed, when it is enlarged to 3 times, all the buildings will be rendered. 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.

4.3.11. Graph Data Structure

Figure 24 is the core code that generates the graph and shows the graph data structure.

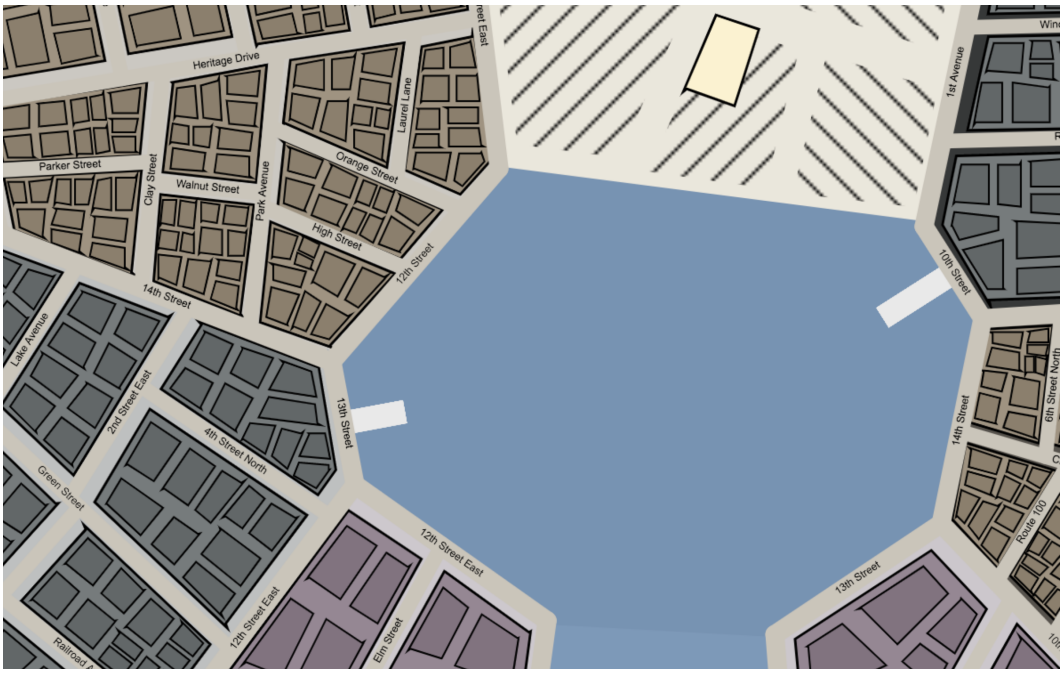


Figure 23. A city map with buildings and details

```
function Graphics( ) {
  this.sites = d3
    .range(state.N)
    .map(( ) => [
      Math.random( ) * (MAX_WIDTH - state.MIN_WIDTH) + state.MIN_WIDTH,
      Math.random( ) * (MAX_HEIGHT - state.MIN_HEIGHT) + state.MIN_HEIGHT,
      0
    ]);
  this.voronoi = d3
    .voronoi()
    .extent([[state.MIN_WIDTH, state.MIN_HEIGHT], [MAX_WIDTH, MAX_HEIGHT]]);
  this.diagram = this.voronoi(this.sites)
  // relax sites in using Lloyd's algorithm and delete duplicated data
  for (let n = 0; n < 5; n++) {
    this.sites = relax(this.diagram)
    this.diagram = this.voronoi(this.sites)
  }
  this.edges = this.diagram.edges.map(makeEdge);
  this.links = this.diagram.links();
  this.triangles = this.diagram.triangles();
  this.polygons = makePolygons(this.sites, this.diagram);
}

```

Figure 24. Core code that generates the graph and shows the graph data structure

5. References

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

[2] Kenny Hunt. Metropolist. https://charity.cs.uwlax.edu/projects/metropolist/metropolist.html, 2018. [Online].

[3] Sumit Khetarpal. Dividing A Polygon In Any Given Number Of Equal Areas. http://www.khetarpal.org/polygon-splitting/, 2014. [Online; accessed 24-December-2014].

[4] Kladess. poly-split-js. https://github.com/kladess/poly-split-js, 2016. [Online; accessed 1-January-2018].

[5] Leif Lindbäck. Introduction to Non-Functional Requirements on a Web Application - Internet Applications, ID1354. https://www.kth.se/social/files/54351342f276543b77c90951/non-func.pdf. [Online].

[6] Victor Osetskyi. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. https://medium.com/existek/sdlc-models-explained-agile-waterfall-v-shaped-iterative-spiral-e3f012f390c5/, 2017. [Online; accessed 29-August-2017].

[7] RESTfulAPI.net. REST Architectural Constraints. https://restfulapi.net/rest-architectural-constraints/, 2018. [Online].

6. Appendices