

TERRATINKER

Crafting Playful Visualizations
from Geospatial Data

Master Thesis

Jonáš Rosecký

MUNI
FACULTY
OF INFORMATICS

Masaryk University
Faculty of Informatics
Brno, Spring 2024

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used GitHub Copilot for advanced code auto-completion. I declare that I used this tool in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Supervisor: RNDr. Vojtěch Brůža

Consultant: doc. RNDr. Barbora Kozlíková, Ph.D.

Acknowledgements

First of all, I would like to thank my supervisor **Vojtěch Brůža** for his idea of visiting Minecraft from the informatics point of view and supervising my HCI project two years ago, that started my passion for Minecraft development. Furthermore I would like to thank him for all his guidance during the development and writing of this thesis.

I would like to thank **Bára Kozlíková** for her kindness, her great feedback, support and for dedicating her time towards making this thesis worth writing.

I would like to thank my **family**, my **close friends** and my **loving girlfriend**, who supported me through my studies, helped me enjoy every moment I did not spend behind the screen, provided me with a lot of hugs and forcefully dragged me out of my chair to get some fresh air, when needed.

Thanks to all the amazing people at **VisitLab** and **HCI**, that helped with the project in any way and most importantly helped me build my love for coffee. I am probably craving one as you are reading this.

Furthermore, I must thank all the people involved in the **Craft-my-Street project** for providing me with an amazing opportunity of working on an international project with a real world impact.

Abstract

Despite the popularity of video games and the importance of visualizations, surprisingly little research has been conducted on using video games for interactive data visualizations. Such approach seems to combine the benefits of interactive visualizations with the engagement brought by serious games. The objective of this thesis was to design and develop TerraTinker — a tool that transforms geospatial data into interactive visualizations within the Minecraft video game. The tool allows users to generate Minecraft maps using datasets sourced from OpenStreetMap or provided by the users themselves. During the creation process the users can customize the resulting map using node-based editor that offers options for altering shapes and block materials. The thesis includes an evaluation of the tool's functionality and usability. Overall, TerraTinker provides an interactive way to generate playful visualizations of the geospatial data within Minecraft worlds.

Keywords

Geospatial Data, Minecraft, Visualization, Web Application, Node Graph

Contents

| | | |
|----------|---------------------------------------|-----------|
| 1 | Introduction | 1 |
| 1.1 | Specifications | 2 |
| 1.2 | Target Groups | 3 |
| 2 | Geospatial Data | 5 |
| 2.1 | Data Types | 5 |
| 2.2 | Coordinate Systems | 8 |
| 2.3 | Working with Geospatial Data | 9 |
| 3 | Game Selection | 11 |
| 3.1 | Game Selection Criteria | 11 |
| 3.2 | Analysis of Games | 16 |
| 4 | Technical Aspects of Minecraft | 25 |
| 5 | Existing Tools | 29 |
| 5.1 | FME by SafeSoftware | 29 |
| 5.2 | Alternatives | 31 |
| 6 | System Design and Requirements | 35 |
| 6.1 | Requirements | 35 |
| 6.2 | Our Approach | 37 |

| | | |
|-----------|-------------------------------------|-----------|
| 7 | Technology | 47 |
| 7.1 | Web Client | 49 |
| 7.2 | Server | 50 |
| 7.3 | Package | 51 |
| 8 | User Interface | 53 |
| 8.1 | Region Selection | 55 |
| 8.2 | Layers Design | 56 |
| 8.3 | Preview and Publish | 59 |
| 8.4 | Documentation | 61 |
| 9 | Evaluation Algorithm | 63 |
| 9.1 | Handling the Execution Flow | 68 |
| 9.2 | Stale Data and On-Demand Evaluation | 71 |
| 10 | Usage and Results | 73 |
| 10.1 | Geospatial Datasets | 74 |
| 10.2 | Mathematics and Geometry | 81 |
| 11 | Evaluation | 87 |
| 12 | Future Work | 91 |
| 13 | Conclusion | 93 |
| | Electronic Attachments | 95 |
| | Bibliography | 97 |

1 Introduction

Video games have become an integral part of the digital world. Thanks to their popularity, they have transcended their traditional role as sources of entertainment and have evolved as tools for education and conveying information. The so-called serious games have the purpose of leveraging the power of computer games to captivate and engage end-users for a specific purpose, such as to develop new knowledge and skills [5]. Such engagement is useful in many fields and could be advantageous for effective communication of information to young consumers.

One of the use cases of the serious games can be in visualizations. We define visualization as the communication of information using graphical representations [1]. Traditionally, pictures have been used to convey a message in visualizations. With home computers becoming increasingly more common, visualizations were expanded onto their virtual, interactive canvases.

Despite the popularity of video games and the importance of visualizations, surprisingly little research has been conducted on using video games for interactive data visualizations. Such approach seems to combine the benefits of interactive visualizations with the engagement brought by video games.

Addressing this gap, the goal of the thesis was to create TerraTinker — a tool for generating visualizations of real world objects and phenomena inside a virtual environment of a video

game world. Such visualizations can be usually created manually at small scale, but creating larger maps or repeating the same process for multiple areas is expensive and time-consuming. TerraTinker aims to automate the process of designing and expanding such visualizations while maintaining the freedom of customization.

The maps generated by TerraTinker can be used directly for the exploration of the used geospatial datasets and engagement of public with a specific problem (such as noise level in the city). If needed, the creators can further expand the map by building custom scenarios to highlight the focus points or gather users opinions.

TerraTinker will be developed in a close collaboration with the team of the Craft-my-Street project. The Craft-my-Street project [28] of the University College Dublin aims to provide resources for the public sector to implement digital spaces in Minecraft for young people to interact with the urban environment and with each other. One of the targets of the Craft-my-Street project is to develop a platform that would streamline the creation of such digital spaces for less technically proficient users and TerraTinker should be a part of this platform in the future.

1.1 Specifications

To ensure the alignment of personal ideas for the application with the objectives of the Craft-my-Street project, several meetings over the course of six months were arranged with the participation of Ph.D. and postdoctoral researchers from the faculty of Architecture, Planning & Environmental Policy at the University College Dublin and a Ph.D. researcher from the Faculty of Informatics at the Masaryk University Brno. The meetings took place at the University College Dublin as well as online, and they provided an opportunity for dialogue and aided deeper understanding of the problem and the target groups of the application.

According to the meetings, TerraTinker should be an application that allows personnel from the public sector to visualize geospatial datasets by transforming them into virtual worlds within a video game. Those worlds should serve a goal of fostering interaction among young individuals within the digital urban landscapes. The suggested video game for the visualizations was Minecraft, but research and evaluation of alternative video games should be performed to make sure the most suitable game was selected.

The application should allow its users to select a real-world region of interest for the transformation. Furthermore, the users should be able to change the scale of the real world projection inside the virtual world. The users should be able to use their own geospatial datasets on top of the publicly available ones and for each dataset define how it should be displayed in the game. The application should allow for semi-realistic depiction of the real-world geospatial features (buildings, streets, lamp posts) as well as visualization of virtual events and phenomena (noise level, bus routes, solar irradiance). The output of the application should be a static map that contains the user-defined features and can be imported into the selected video game.

1.2 Target Groups

When discussing the target user groups of the TerraTinker application, we consider two disjoint groups that will interact with the finished product — the creators and the players.

The **creators** are local authorities personnel, schools, museums, and other workers within the public sector. The creators are the direct users of TerraTinker. The application should allow them to generate digital environments within a video game, adapt it to their needs and (outside of the scope of the project) publish it for the players to interact with.

The term **players** refers to the target group of the generated visualization. Players will not use the system directly, but rather interact with the visualization generated by the creators. This group mainly consists of digital natives that might find interacting with data through an immersive video game more entertaining than with traditional approaches. Digital natives are children and young adults who have grown up in the information age. They have spent their entire lives surrounded by and using computers, video games, digital music players, video cameras, cell phones, and all the other toys and tools of the digital age. Today's average college graduates have spent fewer than 5,000 hours of their lives reading, but over 10,000 hours playing video games [6].

2 Geospatial Data

Before the implementation of the TerraTinker application can be discussed, it is important to understand the basics of working with geospatial data. Geospatial data is different from other types of data as it describes objects (roads, buildings, lamp posts) or phenomena (car crashes, bus routes, solar irradiance) with a specific location in the real world [1]. Geospatial data typically combines location information with attribute information — the characteristics of the object, event, or phenomena concerned.

2.1 Data Types

Geospatial data comes in many forms and file formats, however, most of them can be categorized into two general groups — raster and vector datasets. Both groups can contain two or three-dimensional data.

Raster datasets assign each point in a given region one or more values. As a simple example, any raster image placed on the surface of Earth can be considered a raster dataset with red, green, and blue components being the assigned values. Most of geospatial raster data types, however, provide inbuilt description for projecting their contents onto the Earth's surface. The most common example of such data type is GeoTIFF (as seen in Figure 2.1).



Figure 2.1: Solar Irradiance dataset of Brno (in grayscale) on top of OpenStreetMap layer (available under the Open Database License), screenshot from QGIS

Vector datasets contain a set of features. Features can represent points, lines, polygons, or 3D objects. Every point of every shape holds its geolocation for mapping it onto the Earth globe. Features can additionally contain attributes with arbitrary values. Those can describe, for example, type of the feature. An example of a vector file format can be GeoJSON that encodes the data in a human-readable format. Vector data can also be obtained from online services, such as OpenStreetMap [39] – an open-source mapping project used by many map providers (depicted in Figure 2.2).

Way: Faculty of Informatics Masaryk univerzity (761057736)

Version #6

Improve a geometry

Edited 2 months ago by xpisar

Changeset #147561348

Tags

| | |
|------------------|--|
| addr:city | Brno |
| addr:housenumber | 554/68a |
| addr:postcode | 60200 |
| addr:street | Botanická |
| amenity | university |
| name | Fakulta informatiky Masarykovy univerzity |
| name:en | Faculty of Informatics Masaryk univerzity |

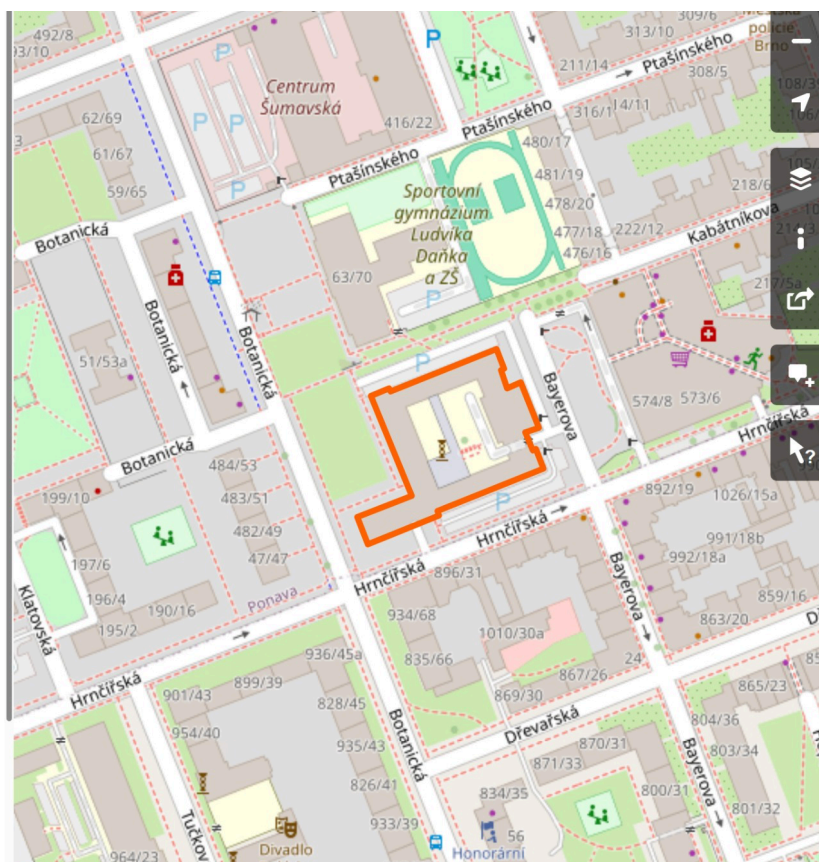


Figure 2.2: Vector features with additional attributes in the OpenStreetMap editor portal

2.2 Coordinate Systems

Even if some people disagree, the Earth is a globe and this fact makes working with geospatial data very complicated. Since it is impossible to project a sphere onto a flat plane without major distortions, we use **Geographic Coordinate Systems** (GCS) for describing geolocation of real-world objects. GCSs use angular measurements to describe points on Earth's surface. The coordinate system used by the Global Positioning System (GPS) is World Geodetic System (WGS 84) [14]. Thanks to the GPS, WGS 84 is the most common used GCS for most use cases. It describes locations with a pair of latitude and longitude in decimal degrees.

As GCSs use angular measurements, most people do not get intuitive sense for distance and size just from looking at the angles. To gain one degree of longitude, one would need to walk approximately 1.8 kilometers on the equator (0° of latitude), but less than a single step at the poles ($\pm 90^\circ$ of latitude).

Projected coordinate systems (PCS) are planar systems that use linear measurements for the coordinates rather than angular units [15]. We can transform data from a GCS to a PCS using a process called **projection** and vice versa by so-called **inverse projection**.

A map projection is the means by which you display the coordinate system and your data on a flat surface, such as a piece of paper or a digital screen. Mathematical calculations are used to convert coordinates from the coordinate system used on the curved surface of earth to the one for a flat surface [15]. When such a transformation is made, some features will always be distorted. Some distances, shapes, and areas will be stretched, and others will be compressed [2]. The elimination of such distortions is impossible. However by choosing a suitable projection, they can be mitigated. The selection of an appropriate projection for TerraTinker is discussed in Section 6.2.1.

2.3 Working with Geospatial Data

Geography is very old and important science field drawing a lot of research towards improving and streamlining the processing of geospatial data. Since the mathematics behind the geographic coordinate systems and their projections is usually complicated, the power of computers was utilized and in the late 1960s the first Geographic Information System (GIS) did emerge [3].

There are currently two most commonly used GIS applications for creating and modifying geospatial data – commercial ArcGIS [30] and open-source QGIS [31]. For professional use, most businesses utilize ArcGIS for its better stability and wider range of features. QGIS is, however, the go-to solution for home users and tinkerers as well as educators.

For processing geospatial data, merging datasets and producing different data formats, applications like Carto [32] (Figure 2.3) or FME by SafeSoftware [33] exist. Both utilize node-based data processing and provide a wide range of node types. Both tools are commercial with only a short trial period.

Another group worth mentioning are visualization tools like Tableau [34] (Figure 2.4) or Microsoft Power BI [35] that also have support for geospatial data. Such tools do not provide so much in terms of data processing, but focus more on visually pleasing interactive visualizations.

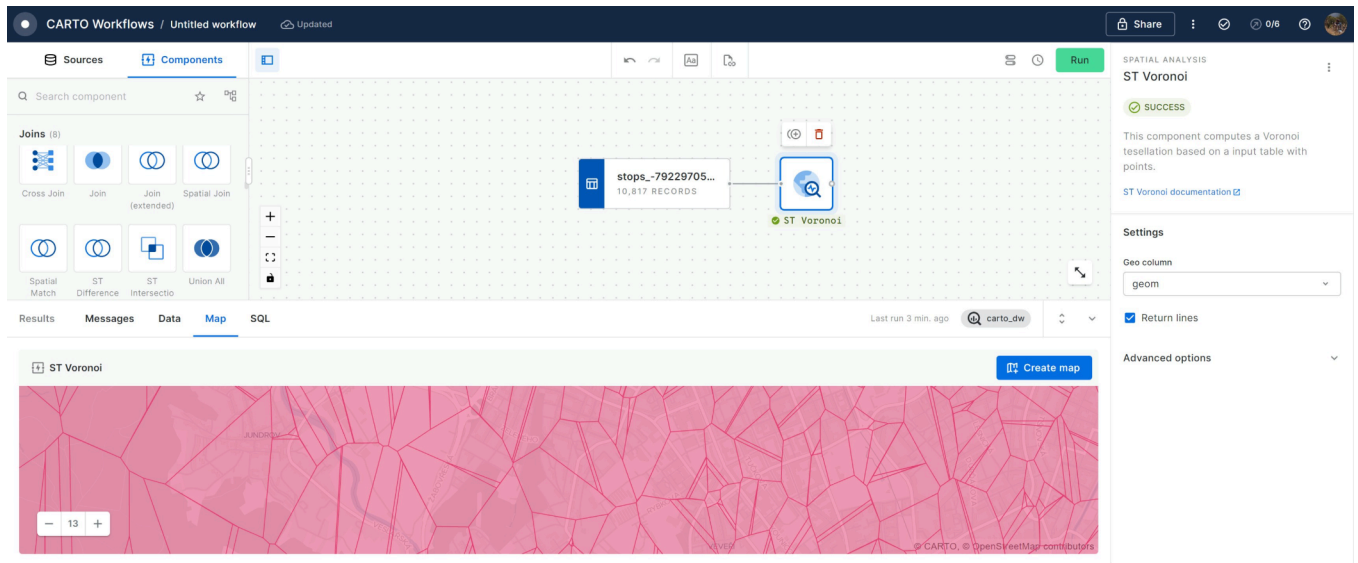


Figure 2.3: Screenshot of Carto Workflow web interface with an example of processed geospatial dataset — a Voronoi diagram of the nearest bus stop in the South Moravian region [70]

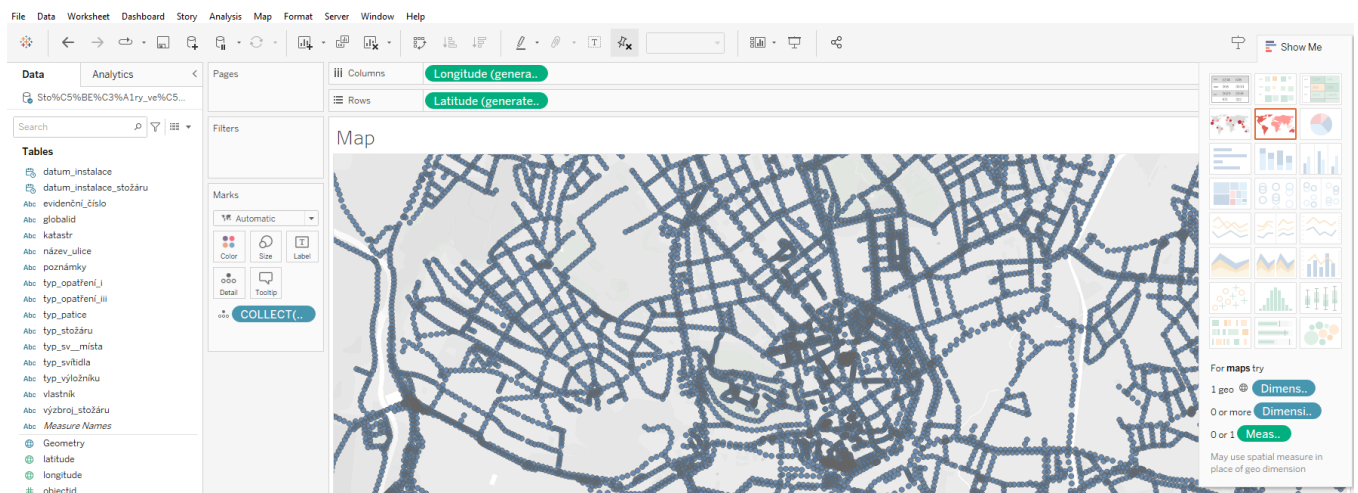


Figure 2.4: Screenshot of Tableau visualization tool with lamp posts vector dataset [69]

3 Game Selection

As mentioned in Section 1.2, the target player group are game natives that might find the in-game world worth exploring and the visualization should spark their interest and a desire for deeper understanding of the data (defined as *players* throughout the thesis). Before the tool is implemented, the suitability of Minecraft for this kind of data visualization among the vast landscape of video games should be assessed.

3.1 Game Selection Criteria

The spectrum of games is large and the fact that around 40 new games get released on Steam (the largest video game store) every day [73], makes it impossible to analyze every single existing video game. To have a measure of game suitability, a set of criteria was established. The criteria are not strict nor exhaustive. They also do not all have the same weight in the decision process. The criteria were, however, used as an initial filter for the game selection.

3.1.1 Technical Criteria

Large player base (G1) — Choosing a game with a large player base is beneficial, as a significant number of people from the target group already have the game set up and ready to play. Thanks to that, the created visualizations can reach a large group of people without additional effort. Apart from the reach, the players are also already familiar with the controls and inner-workings of such game, making the onboarding process easier.

The large player base is also beneficial for the developers and the content creators as older games with an active community tend to have better documentation and plethora of downloadable content, usually free of charge.

Download and play (G2) — To reach the broadest audience possible, even outside of the current player base, the game must be straightforward to set up. No game modifications (mods) should be required as the process of installing such mods can be complicated and could cause player drop-off during the setup.

The ideal game would be free-to-play to attract any player interested in this type of visualization. However, given game development's complexity and time-consuming nature, it is reasonable to consider the free-to-play requirement nonessential.

Generating the visualizations (G3) — The manual setup is applicable for a single small-scale visualization but would not be scalable for larger or repetitive visualizations. This thesis aims to automate this process and provide a simple tool for creating the visualizations fast.

In order to generate the visualizations automatically, the game needs to provide one of the following:

Allow scripting — Scripts in this context are pieces of code that can modify the game's behavior, allow the modification of the in-game maps, and add or change the behavior of particular objects.

Documented and modifiable world files — This allows us to modify the files containing the information about the game world saves externally and then load the modified files in the game. The format can be either documented by the developer or reverse-engineered by the community.

3.1.2 Gameplay Criteria

Interactivity and engagement (G4) — According to Ward et al. [1], interaction in geovisualization is crucial. Exploring the visualization from different perspectives might allow the users (in our case the players) to see connections and understand the data more deeply. Especially for geospatial data, navigation is very important. It overcomes the limitations of space allowing users to see the details and overview within the same visualization [4].

In video games, interactions with the environment form the central part of the experience. When combined with the navigation, the player suddenly becomes a part of the visualization, is encouraged to interact with it, and get immersed in the virtual world. It is also beneficial for the players to be able to modify the virtual world, observe the changes and see the impact of their actions.

Grid-based world (G5) — Games that provide complete freedom of object and terrain placements (Astroneer [71], Roblox [72], etc.) might appear complex and hard to grasp. The modification of such worlds by the players is usually impossible or very imprecise. For those reasons, the spectrum of the games is narrowed to grid-based games only.

Grid-based game worlds use a regular grid and a predefined set of tiles placeable on the grid. There are three main categories of tile-based worlds depicted in Figure 3.1:

- top-down 2D games have tiles placed along the horizontal axes,
- side-view 2D games have tiles placed on the grid oriented along the up and right directions,
- 3D grid games have tiles placed on the grid in all three dimensions.

We can omit the side-view games since we want to visualize geospatial data with two horizontal input dimensions.

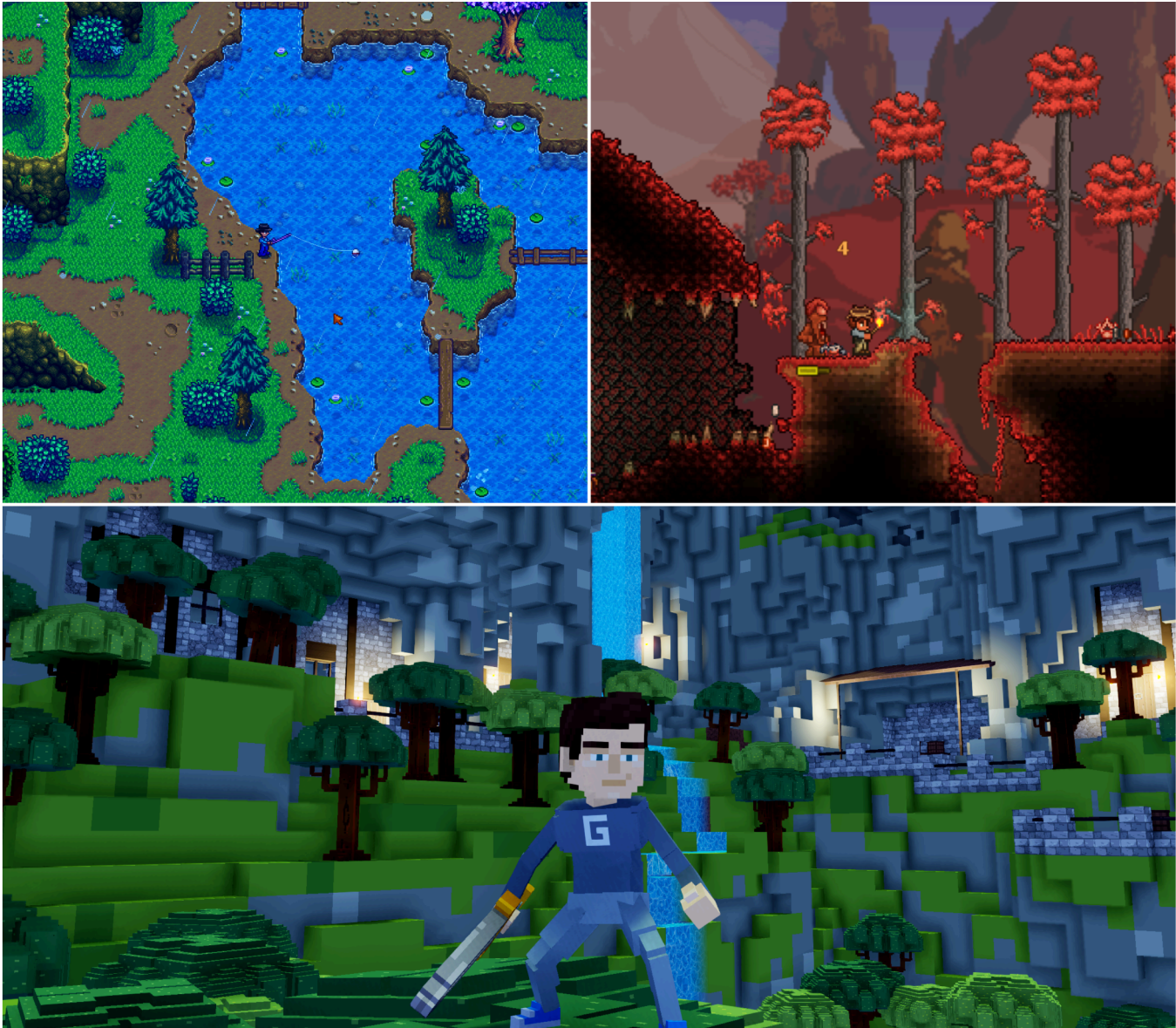


Figure 3.1: Comparison of the tile-based worlds — top-down 2D: Stardew Valley (top left), side-view 2D: Terraria (top right), 3D grid: Sandbox editor (bottom)

3.2 Analysis of Games

Given the criteria defined in the previous section, three strongest candidates are described below. Apart from Minecraft, two other games — Factorio and OpenTTD — were judged to be the most compliant with the rules. After a thorough evaluation, the initial choice of the Minecraft video game was approved mainly for its large player base and modification options.

Even though this thesis does not extend the visualizations to other video games, the visualization of data in video games is a topic worth exploring, as shown by Cardoso, 2021 [7] and Šťastná, 2023 [13].

3.2.1 Factorio

Factorio [37] (Figure 3.2) is a Czech video game about designing, building, and maintaining factories. After crash landing on an alien planet the player has to launch a rocket back to space in order to win the game. To achieve this, it is required to automate various processes, which comes with its own challenges. These range from space occupying trees, over logistical problems, to hostile, native inhabitants, which are not happy about the increasing pollution levels caused by the ever growing factory [9].

Factorio provides an extensive map editor and the option to write custom game scripts (G3). It is also natively multiplayer, providing more immersive group experiences (G4). Its main downside is the selection of the tiles. Most of the provided tiles represent a single separate building or machinery. This prevents the visualization of custom polygons or linear data. Furthermore, the lack of a third dimension within the game world narrows the visualization options, making it hard to present additional value related to specific geographic features.

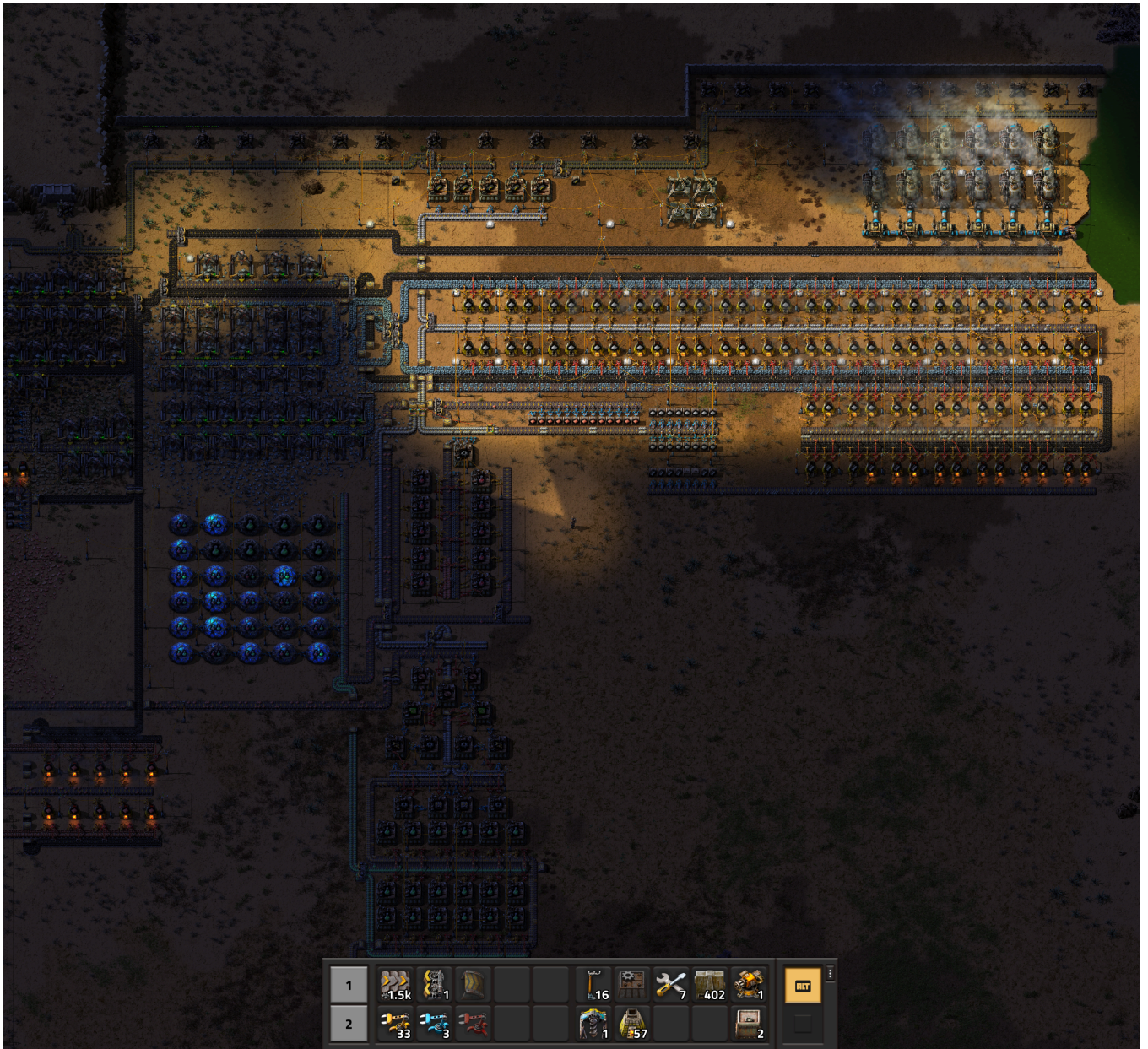


Figure 3.2: Example of a Factorio map



Figure 3.3: New OpenTTD world

3.2.2 OpenTTD

OpenTTD [38] (Figure 3.3) is an open-source simulation game based on Transport Tycoon Deluxe. The main goal of the game is to build and develop your own transport company in such a way that it gets the highest possible revenue, thanks to an optimal transport organization, transporting as much cargo as possible in the most efficient way [10]. The world of OpenTTD is 2D rendered in pixel graphics using an isomeric camera.

The open source aspect of the game is beneficial in multiple criteria (G2, G3). Since the source code is provided, multiple game modifications and tools exist for manipulating the save files externally. In addition, the game provides inbuilt tools for map creation and a direct option to import a height map from the file.

As this game focuses on transportation, it contains detailed mechanics for traveling, roads, railways etc. However, similarly to Factorio, it contains only non-combinable tiles, making it suitable only for geospatial visualizations with small scale.

3.2.3 Minecraft

Minecraft [36] (Figure 3.4) is a multiplayer sandbox game focused on creativity, building, and survival. It can be played in singleplayer and multiplayer mode. The game world is procedurally generated with cubes arranged in a fixed grid pattern. Cubes are made of different materials, such as dirt, stone, etc. Players can gather these materials and place them in the world to create various constructions or use them to craft other items [8].

With over 300 million sold copies as of 2024, Minecraft is the best-selling video game ever (G1). In part, thanks to that, there is already a large number of resources and an ecosystem of modifications around the base game.

As stated in requirement G2, we want the game to be “download and play,” meaning no mods on the client side. Fortunately, thanks to the community, the game has a plethora of custom server implementations allowing the modification of the game behavior using server-side only scripts (more about game modifications is described in Chapter 4). Scripts can be used to alter the gameplay, enhance the immersion (G4), and generate the visualizations directly in the game (G3).

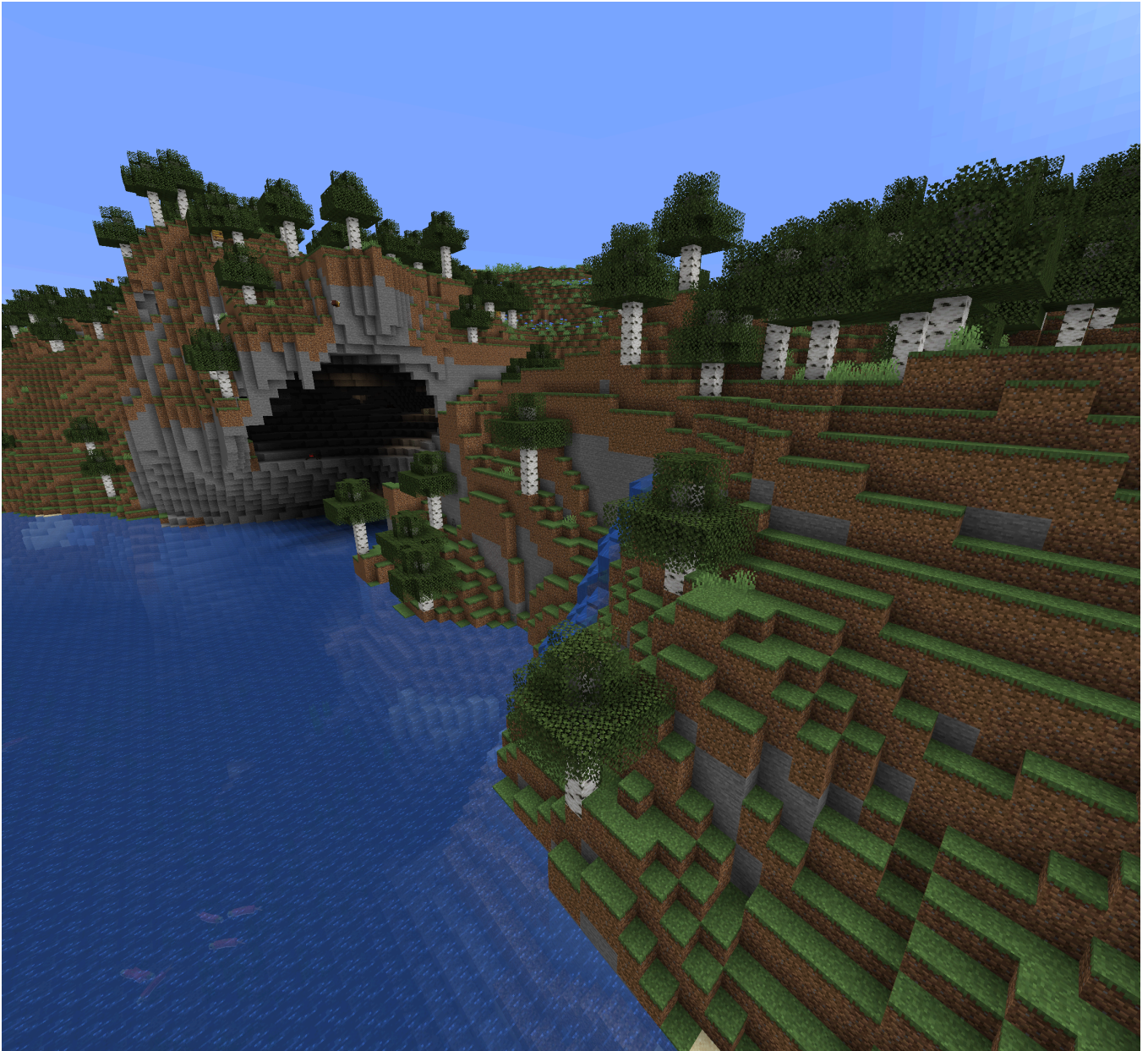


Figure 3.4: Screenshot of Minecraft world generated by the native Minecraft world generator

3.2.4 Other Analyzed Games

Apart from those mentioned above, five more games were considered in the selection process. **Stardew Valley** [68], **Heroes of Might and Magic III** [40], and **Warcraft III** [41] are popular top-down grid-based video games. All three of them provide an extensive map editor for editing the existing maps or creating new ones. Stardew Valley and Warcraft III provide tools for scripting, which could help generate the map (G3). However, all three games provide only premade tiles, which cannot be customized or changed, similarly to Factorio and OpenTTD.

Heroes of Might and Magic III is turn-based, unlike the others mentioned. Turn-based game style provides interesting gameplay options, however, it is not ideal for exploring the visualizations in the game (G4). Both Heroes of Might and Magic III and Warcraft III have a save-file format that has been reverse-engineered, which, as explained in G3, enables development of an external generator.

Two relaxing games that were analyzed are **Townscaper** [29] and **Dorfromantik** [42]. Both games supply little in terms of gameplay possibilities (G4). However, they provide simple tools for modifying the visualization (G5). In either game, you can zoom out and have an overview of the whole map and zoom in to see the details of each tile.

Townscaper allows importing and exporting from text files, which makes it great for the external generation of visualizations. Dorfromantik does not have such a feature, and its save file contains only a seed for each tile preventing creation of custom tiles with an external tool.

4 Technical Aspects of Minecraft

Minecraft is an open-world, voxel-based video game. Its game world is built out of unit cubes (blocks) that can represent different materials. Players can destroy (mine) the blocks to obtain various items. Those can be placed back in the world or transformed using a process called crafting.

As of 2024, three official versions of Minecraft exist [16] — Minecraft Java Edition, Minecraft Bedrock Edition, and Minecraft Education Edition. The Java Edition of the game is the original Minecraft version. It was created in Java and still has the largest player base above the other two mentioned. Thanks to the large player base, there is already an extensive library of user content and game modifications.

The Bedrock Edition and Education Edition are both written in C++. They try to closely imitate the behavior of the Java Edition and provide some additional features [17]. The Java and Bedrock editions are separate programs, but often come bundled with a single purchase. The Education edition [43] is sold as a part of the Office 365 suite for schools, making it a popular tool for education as many schools already have access to it.

Minecraft Java Edition was selected for this project for its rich library of resources and ease of extendibility. If not stated otherwise, this thesis uses the term Minecraft to describe the Java Edition of the game.

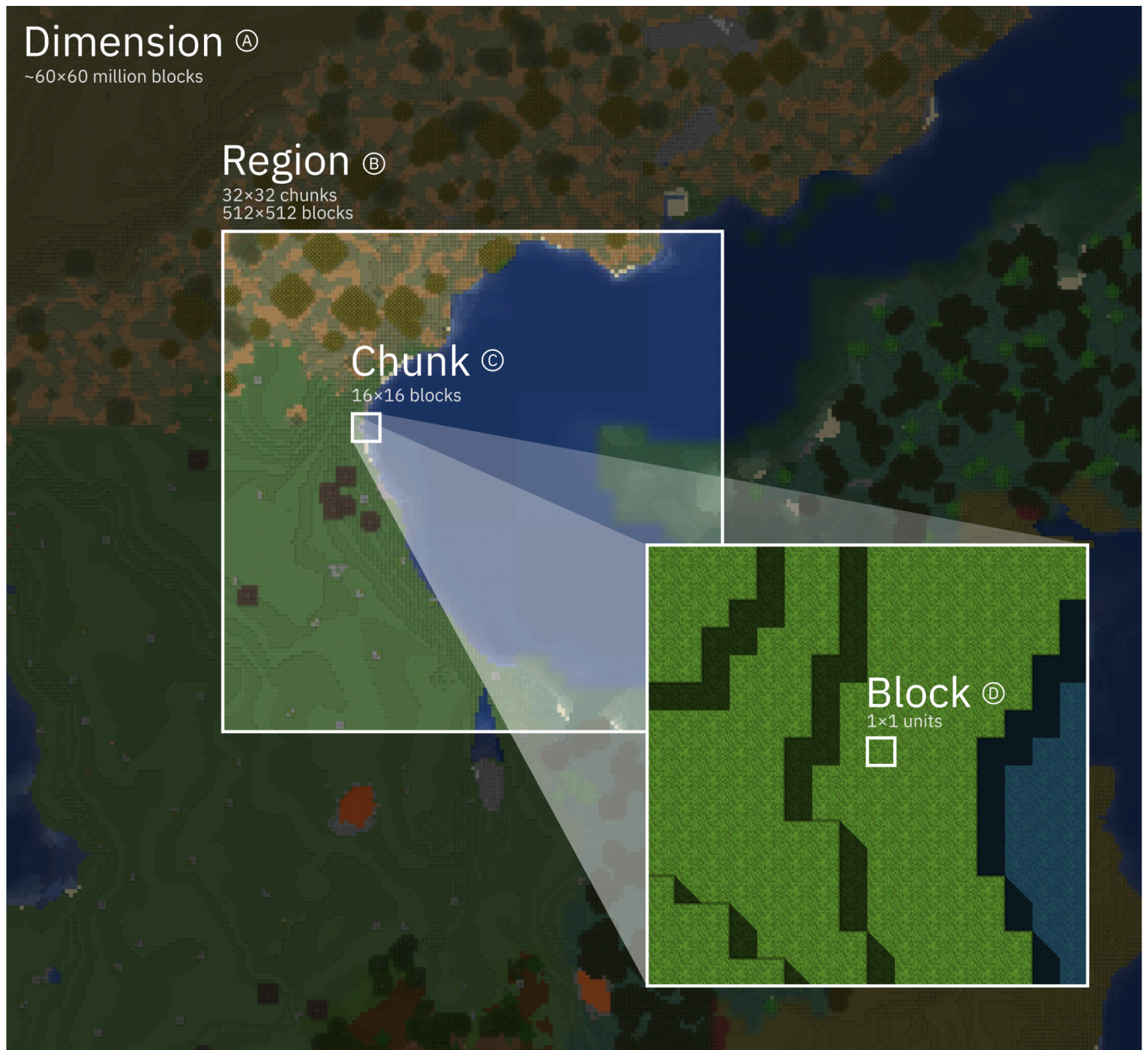


Figure 4.1: Visualization of the Minecraft chunks and regions

A single Minecraft game save can contain a virtually unlimited amount of dimensions (worlds) stored in separate folders. Only three basic dimensions (Overworld, Nether, and End) can be traversed by default within the game.

As depicted in Figure 4.1, blocks ④ within each world ① are separated into chunks ③. A chunk is a column of 16 by 16 blocks reaching from the lower to the upper build limit (-64 to 320 for Minecraft version 1.19) within each world. Chunks are used for efficient loading of the world. The chunks are grouped into regions ② stored as separate files within the game save. A single region file contains 32 by 32 chunks (512×512 blocks). Since the game version 1.2.1, Minecraft has used a binary save-file format called Anvil [18].

Each block or entity (animal, item, player, ...) has their location specified by three coordinates: X, Y, Z. The X and Z coordinates are horizontal and the Y coordinate is vertical. A size of each block corresponds to a unit within the coordinate system and should resemble a meter in real world.

Minecraft game mechanics run on a server that can be either located away from the player's computers as a dedicated server or started inside the game client for singleplayer or LAN experience. The original — Vanilla — game mechanics and client behavior can be customized using mods and plugins.

Mods are game modifications that usually require mutation of all the clients playing together as well as the server they are connected to. They require altered client and server implementations like Forge [44], Fabric [45], or Quilt [46]. Mods offer more options than plugins, allowing developers to create custom blocks, entities, and user interfaces.

Plugins do not require modification on the client side as they are server-side only. They do not allow developers to add blocks or change the default interfaces. However, they provide the op-

tion to modify the game's behavior and inner workings. As all the mechanics are evaluated on the server, they can be altered or replaced by a custom implementation. The most common dedicated server implementation with plugin support is Bukkit [47] and its modern forks — PaperMC [48] and Purpur [49].

5 Existing Tools

A part of this thesis is an analysis of the existing solutions that can be used for transforming geospatial data into a Minecraft map. After an extensive search, surprisingly few possible solutions were discovered. As this tool should be accessible even for creators with no coding skills and experience, only solutions that do not require coding were considered.

5.1 FME by SafeSoftware

The closest alternative to the goals of TerraTinker is FME by SafeSoftware [33]. FME (Feature Manipulation Engine) is a geospatial extract, transformation, and load software platform that provides a no-code solution for the abovementioned tasks. FME Workbench, the editor for the FME platform, allows the users to create node-based diagrams for repeatable geospatial data transformations. On top of an extensive set of transformations, the FME Workbench provides tools for converting a point cloud into a Minecraft map.

As shown in Figure 5.1, the Craft-my-Street project initially used FME to generate the maps. Even though FME was capable of fulfilling the requirements of the project, it has proven to be problematic for the future usage for the following reasons. The FME suite is **very complex** and extensive, which on one hand provides advanced tools for data processing and analysis. On the other hand, due to this complexity, the application feels overwhelming and hard to un-

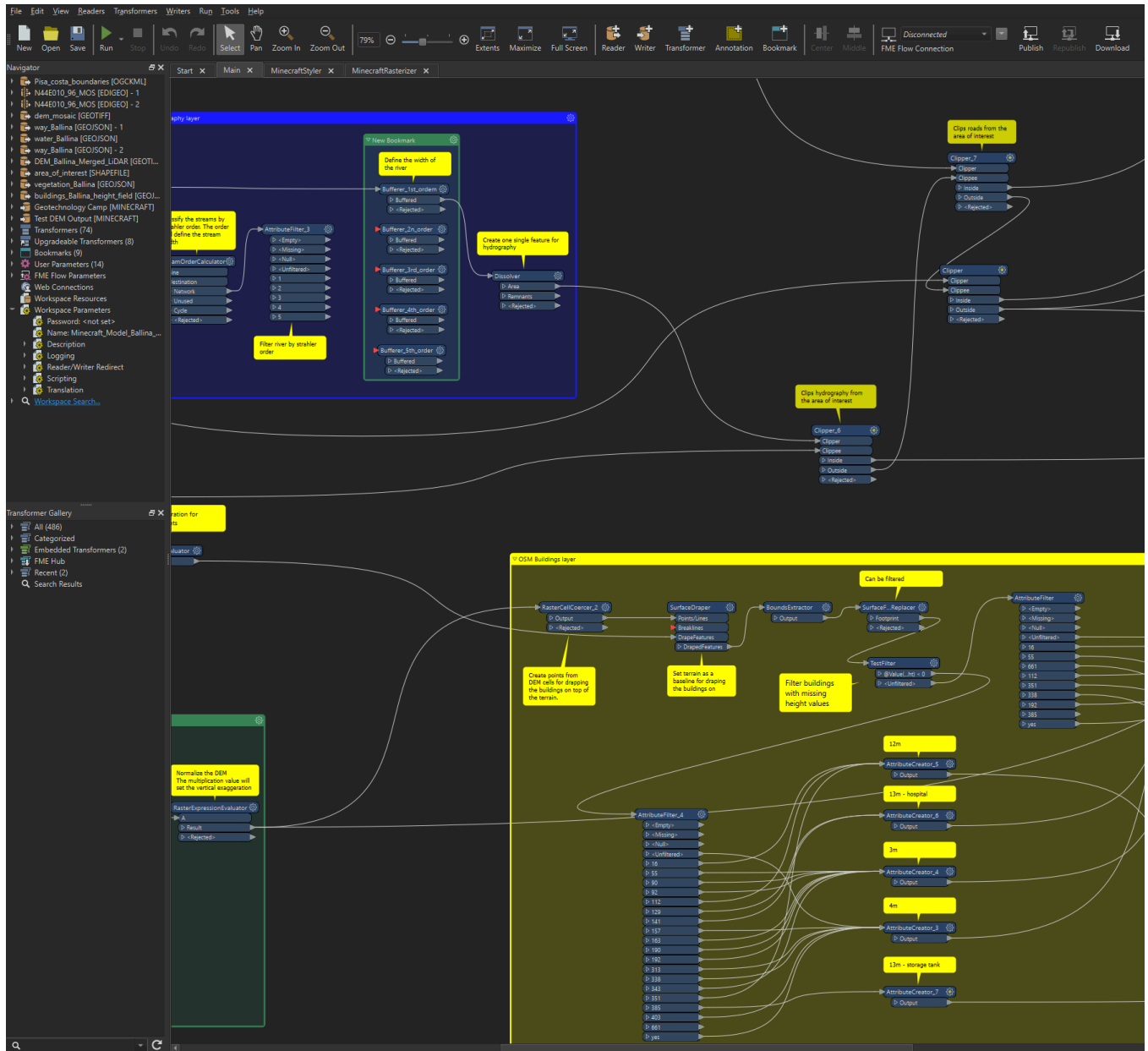


Figure 5.1: FME Workbench with the original project used for GeoMinasCraft [12], later adapted for the Craft-my-Street project

derstand for the creators. Furthermore, FME is a commercial software with very **expensive license**, which is not viable for most creators.

Due to the abovementioned reasons, the leadership of the Craft-my-Street project have decided to implement a custom solution that would replace the FME software in their workflow. Despite the problems, FME was an inspiration that helped form the TerraTinker application and pinpoint some concepts that would help improve it over FME for the selected use case. These concepts are:

- abstract the creators from the coordinate systems and provide unified, user-friendly interface for manipulating geolocations,
- no errors while running the generator, all problems need to be indicated to the creators before the transformation is started,
- all parameters should be visible, so it is possible to understand what a node is doing just from looking at it (unlike in FME where configuration of nodes is hidden inside a menu).

5.2 Alternatives

The most out-of-the-box solution available for generating virtual world of a selected region is a commercial **WorldBoxer** [50] online service. WorldBoxer allows users to select a region of interest and a visual style from a set of templates. There are unfortunately no more customization options – mainly the user cannot provide their own dataset for advanced visualizations.

Alternative solution worth-mentioning is a two step process – first generate a 3D model and then transform it into Minecraft. For generating the 3D model from real world data a suite of tools from ArcGIS includes the commercial **ArcGIS CityEngine** [51]. CityEngine is a tool



Figure 5.2: Model of a building from the ArcGIS CityEngine (left) and the model voxelized by FME (right), adapted from [11]

for transforming real GIS data into procedurally-generated 3D cities that can be exported into various 3D formats. Such model can be then transformed to Minecraft using abovementioned FME (Figure 5.2), or free online tools, such as [ObjToSchematic](#) [52] or [Online Voxelizer](#) [53].

Unlike the WorldBoxer, CityEngine allows users to supply custom datasets. However, the described two-step approach is more complicated. Furthermore, CityEngine is a very expensive commercial tool not suitable for most use cases of TerraTinker.

The visual results of the CityEngine are very realistic and visually pleasing. On the contrary, it is impossible to include intangible phenomena or alternative datasets into such a generated 3D model. Since the abovementioned solution requires two independent steps, it can be harder to replicate for less technically proficient creators. Furthermore, it might be difficult to enforce a selected scale to the real world (such as 1 meter ~ 1 block) through this multi-step process.

6 System Design and Requirements

In the previous chapters, the main ideas have already been described, the suitability of Minecraft for the visualization has been validated, and the existing tools have been analyzed. In this chapter, we define the requirements, discuss the coordinate projections, and start to design the mechanics of the application.

6.1 Requirements

Before the design and implementation can be described, it is important to establish the formal requirements for the application. As the assignment left a lot of freedom for the interpretation and implementation, the following requirements have been designed to narrow the possibilities and to help steer the ideas towards a consistent result. The listed requirements are intended to be followed alongside the initial specifications described in Section 1.1.

Configure the region (R1) — The user must be able to select the region of interest for the visualization. This selection should be visual and intuitive. TerraTinker must be able to generate small visualizations with the size up to at least 500×500 Minecraft blocks. It is hard for a player to comprehend a larger map and be able to get an overview of such visualization. Generating larger regions can be added to the system during the future work as it might be useful in some scenarios.

Choose the datasets (R2) — The creators must be able to choose, which datasets they want to use. Both raster and vector geospatial datasets must be accepted. The application must accept data sources in WGS 84, as it represents the standard among GCS formats (as described in Section 2.2). The support for other GCSs and PCSs is optional. On top of user-provided datasets, the application should allow fetching data directly from the OpenStreetMap or alternative open-data source.

Customize the visualization (R3) — Each creator should be able to tweak and adapt the Minecraft map that will get generated, for their personal needs. This includes choosing a way each dataset will be represented in the final in-game map as well as selecting the surrounding environment. The representation can change in shape (outlines or filled buildings, building height or underlying height map) as well as block material used.

Generate the result (R4) — Output of the TerraTinker application must be a valid Minecraft map. It must be possible to import the generated map into the game for either singleplayer or multiplayer experience. This process should be simple and straightforward for the creator and the player to perform.

Usability (R5) — A creator with a basic knowledge of GIS should be able to independently use the system without a personal guidance. This implies that the system should be intuitive enough for such users or provide documentation that can help them complete their task.

Web-based (R6) — The system should be implemented as a web application to mitigate the need of installation on the users computers. As a web application, it can also be easily extended into a larger cloud solution in the future.

6.2 Our Approach

The main idea of the application is to allow creators to transform their geospatial data into a Minecraft map to form a geospatial visualization. There are multiple steps to such transformation. As the creator is working with geospatial data, a projection from real-world to in-game coordinates must be established. Furthermore, the creator should be able to specify, which datasets they want to use within the given project and define their transformation.

6.2.1 Coordinate System Transformation

As mentioned in Section 2.2, map projection is the process of making a portion of a globe fit into a flat plane — in our case, a flat Minecraft world. This step is required since it is not possible to represent a globe inside a flat world with 1:1 accuracy. Since the visualization will only contain a small area of the globe, the projection can approximately preserve distance, area, and local shape. A simple planar projection visualized in Figure 6.1 and defined in Equation 6.1 makes a good fit in this case.

$$\begin{aligned}x &= R \cos \varphi \sin (\lambda - \lambda_0) \\y &= R(\cos \varphi_0 \sin \varphi - \sin \varphi_0 \cos \varphi \cos (\lambda - \lambda_0))\end{aligned}$$

Equation 6.1: Formula for orthographic planar projection from longitude and latitude (λ, φ) to flat coordinates (x, y) where R is the radius of Earth and (λ_0, φ_0) is the center longitude and latitude of the projection (the tangent point)

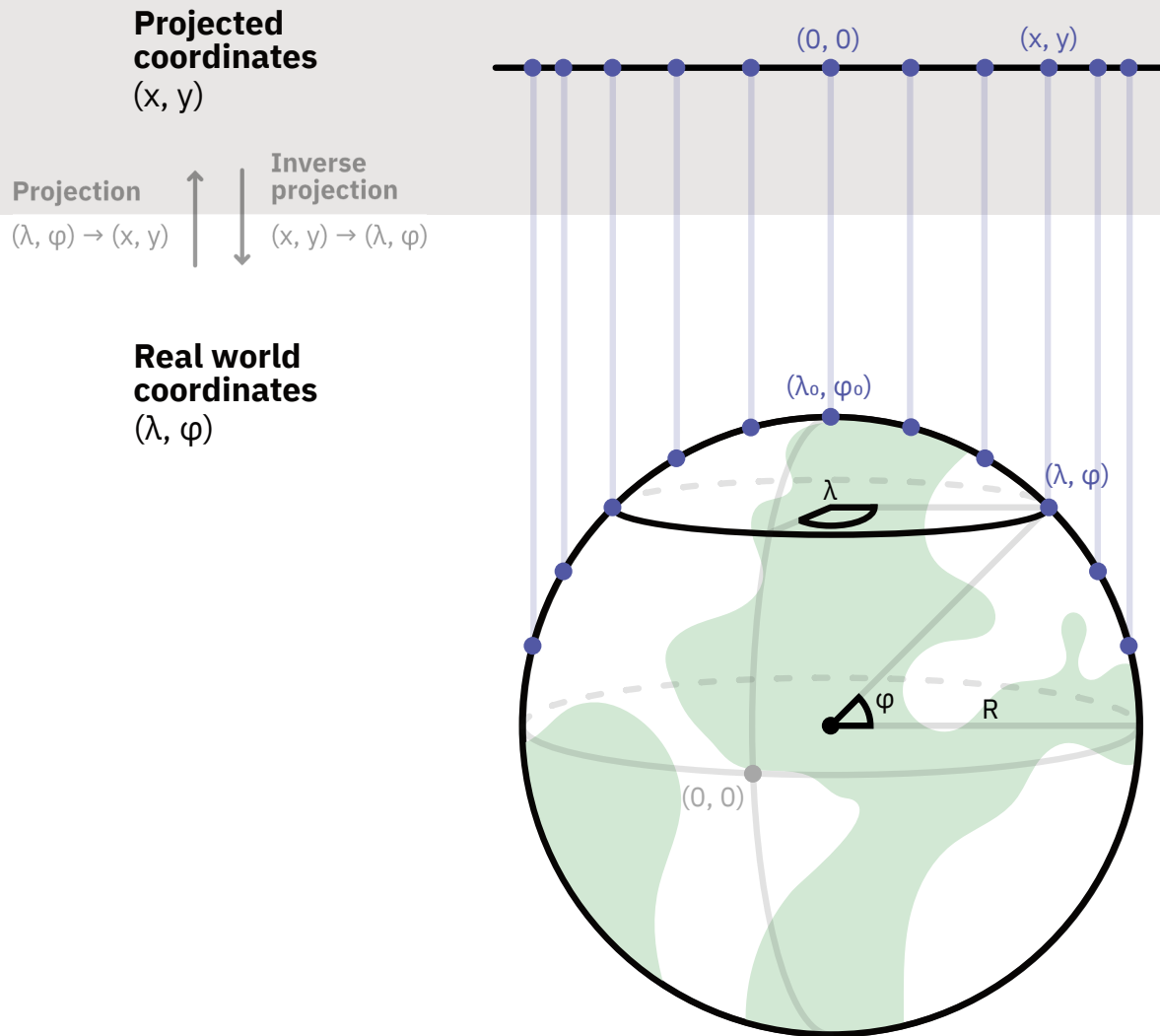


Figure 6.1: Scheme of the planar orthographic projection and inverse projection

Planar projections map the surface of the sphere to a plane that is tangent to the sphere, with the tangent point corresponding to the center of the projection [1]. The focal altitude of the projection can further differentiate planar projections. For the simplicity and quality of the results, the orthographic planar projection with the focal distance in the infinity has been selected.

For the TerraTinker application, we want to choose a projection that feels close to reality for the player. When they walk through the world in the first person, the scale of the features and their shape should match the real world. This should hold for the regions up to 50 km (R1, 500×500 blocks at the scale of 1:100). To test that the selected projection matches this requirement, a comparison between the two following coordinate calculations (depicted in Figure 6.2) has been performed:

- A circle with radius δ inverse projected from the selected PCS with tangent point (λ_0, φ_0)
- Points reachable by walking the distance δ from the point (λ_0, φ_0) in every direction along the surface of the Earth

The second mentioned approach is close to real world and uses the spherical law of sines and cosines to calculate the new latitude and longitude respectively based on the given distance δ and bearing θ from the starting point (λ_0, φ_0) . The exact formula is described in Equation 6.2.

$$\begin{aligned}\varphi' &= \arcsin(\sin(\varphi_0) \cos(\delta) + \cos(\varphi_0) \sin(\delta) \cos(\theta)) \\ \lambda' &= \lambda_0 + \arctan2(\sin(\theta) \sin(\delta) \cos(\varphi_0), \cos(\delta) - \sin(\varphi_0) \sin(\varphi'))\end{aligned}$$

Equation 6.2: Formula for calculating coordinates in the experiment

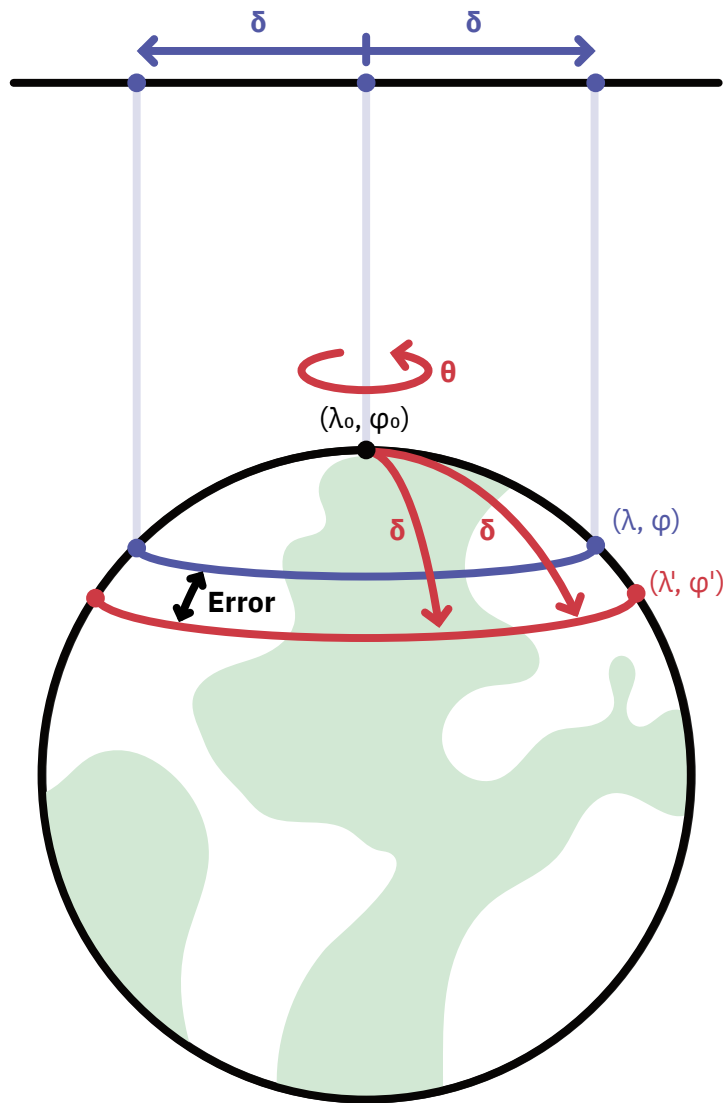


Figure 6.2: A comparison of a circle obtained by the inverse projection (blue) and by the spherical law of sines (red), the measured value is the Error between the mentioned approaches

This formula is applied for a set of 3 600 bearings between 0° and 360° and compared to 3 600 values obtained from the inverse projection. The spherical distance is then calculated from all the tuples of points $((\lambda, \varphi), (\lambda', \varphi'))$ and the maximum of those distances is returned. This value is considered the error in the selected projection.

When comparing the resulting distance of the points, the error of only 0.5 meters can be observed on circles with the radii $\delta = 50 \text{ km}$. This result is more than sufficient for the needs and requirements of the application as the resolution of the Minecraft world is basically only one meter.

TerraTinker aims to isolate creators from the specifics of each dataset and provide a common format for the data, especially coordinates. All the values of coordinates throughout the system are already projected into a custom PCS. The benefit of such approach is twofold:

- The creator does not have to consider hundreds of different coordinate systems. All the data is in the same centralized projected coordinate system.
- The PCS uses units corresponding to meters similarly to Minecraft's coordinate system. This similarity makes it simpler to understand than angular measurements.

As requested in the requirement R1, the creator is free to manipulate certain parameters of the projection to best fit their needs. The most important parameter is the tangent point corresponding to the origin (0, 0) of the local coordinate system of the Minecraft map. Furthermore, the creator may want to scale the world in either horizontal or vertical direction to either fit a larger area of the map or highlight a detail in a specific smaller region. The system also needs the minimum altitude of the world since the size of a Minecraft world is vertically limited. The listed parameters are exposed to the creator as a part of the region selection step described in Section 8.1.

6.2.2 Transforming the Data

Similarly to graphics editors, the TerraTinker application allows creators to define multiple layers that are independent and get merged into a final Minecraft map at the end. Despite this core idea holding through the design process, the principle of creating layers within the application has significantly evolved in time.

According to the requirements defined in Section 6.1, the TerraTinker application is supposed to be accessible for creators without previous experience with similar tools (R5), yet provides advanced options for customization and linking data sources (R2, R3). Such a combination is hard to achieve and almost contradictory. During the approach design, the application leaned towards the freedom of transformations compensated by comprehensive documentation, samples, and help.

The initial approach (Figure 6.3) included a set of static *generators* and *data providers* that could be linked to define individual data layers. The *data providers* included loading from GeoTIFF, OpenStreetMap API [19], or other geographic data files and sources (R2). *Generators* were complex actions, such as “Box draw,” rendering a hollow box on each input geometry, or “Surface painter”, which colors the surface in different blocks according to the input raster dataset (R3).

This approach is straightforward and usable for any creator without experience with similar tools (R5). The main flaw of this approach was the specificity of each *generator*. Since the generator actions were complex and fixed, the desire for more flexibility in the visual output emerged.

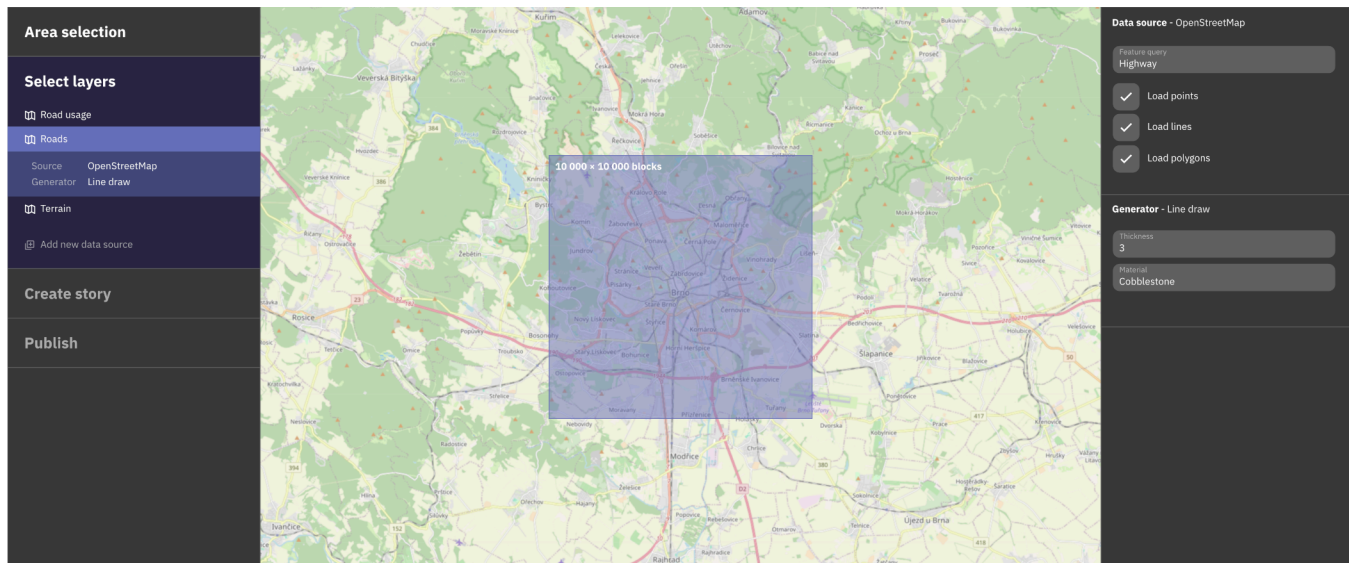


Figure 6.3: Original design file of the initial approach – in the right column is the configuration of the individual transformations, the rest of the UI is similar to the one described in Chapter 8

Some datasets require preprocessing, transformation, or filtering before they can be used for the visualization. With the initial approach, modifying the input data before feeding it into the generator was impossible.

The second design iteration (Figure 6.4) addressed the latest stated issue. An essential change in this version was the introduction of a node graph for defining the layers of the map. The node graph provided the needed freedom of data transformation between the data providers and the generators. The addition of the node graph, however, introduced more complexity and created an entry barrier for new users. This approach also did not solve the issue of predefined complex generators.

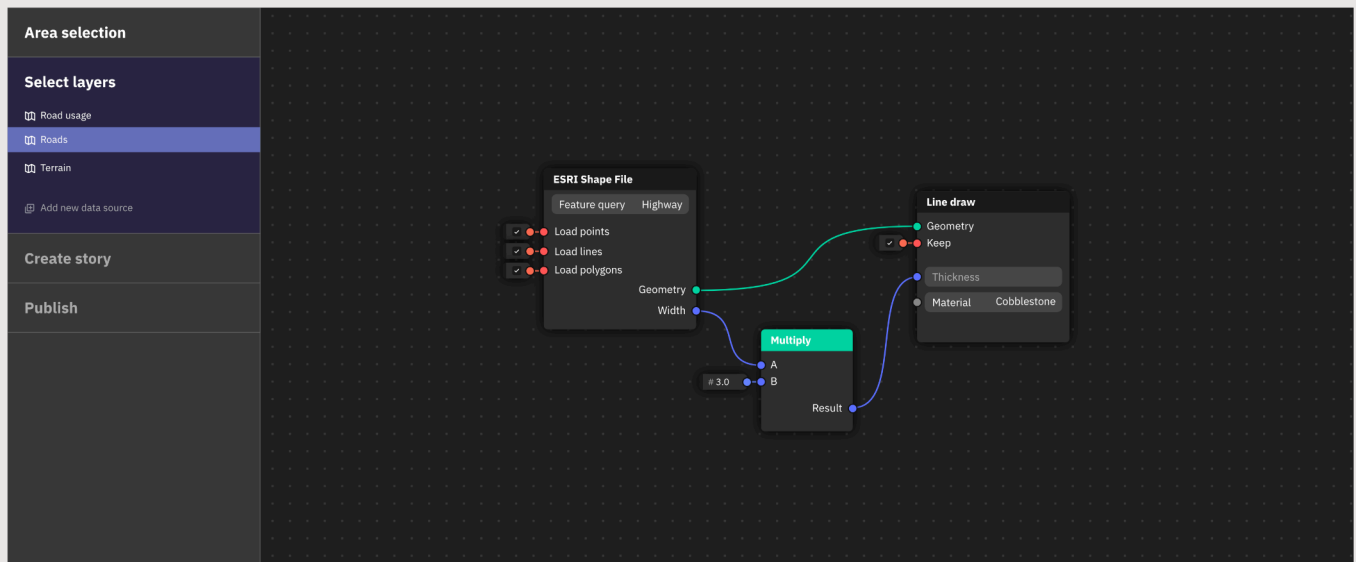


Figure 6.4: Design file of the second design iteration with the removal of the right configuration column and addition of the node graph, which allows for the transformation of the data between the data source and a generator (more about the node graph is in Section 8.2)

Version three (Figure 6.5) removed the idea of complex generators and looked at the data similarly to vertex and fragment shaders on a GPU. The “vertex” part of the node graph loaded vector data and manipulated the shapes. The “fragment” part of the node graph manipulated individual columns of blocks (set of Minecraft blocks that share the same X and Z coordinates) of each shape. The two parts of the node graph were connected with a “Rasterize” node. This node turned the vector shape from the “vector part” into a set of (X, Z) coordinate pairs that were then used in the “fragment part” of the graph.

When inspecting the third approach, the separation of the node graph appeared arbitrary as a single graph can easily contain multiple Rasterize nodes. Instead **the final approach** (Figure 6.6) joined both sides together with the “Rasterize” node becoming a regular node that can be used anywhere in the node graph. More details about the current implementation can be found in Chapter 8.

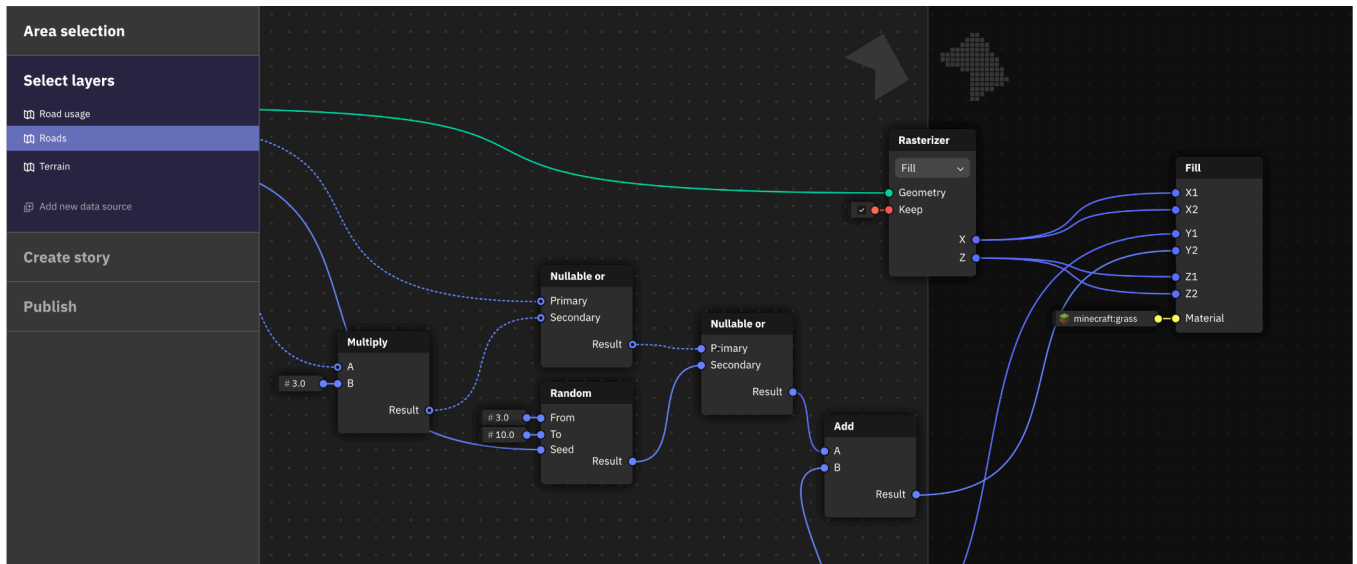


Figure 6.5: Design file of the version three with simple actions and the separation of the vector (left lighter part) and the raster (right darker part) parts joined with a Rasterize node

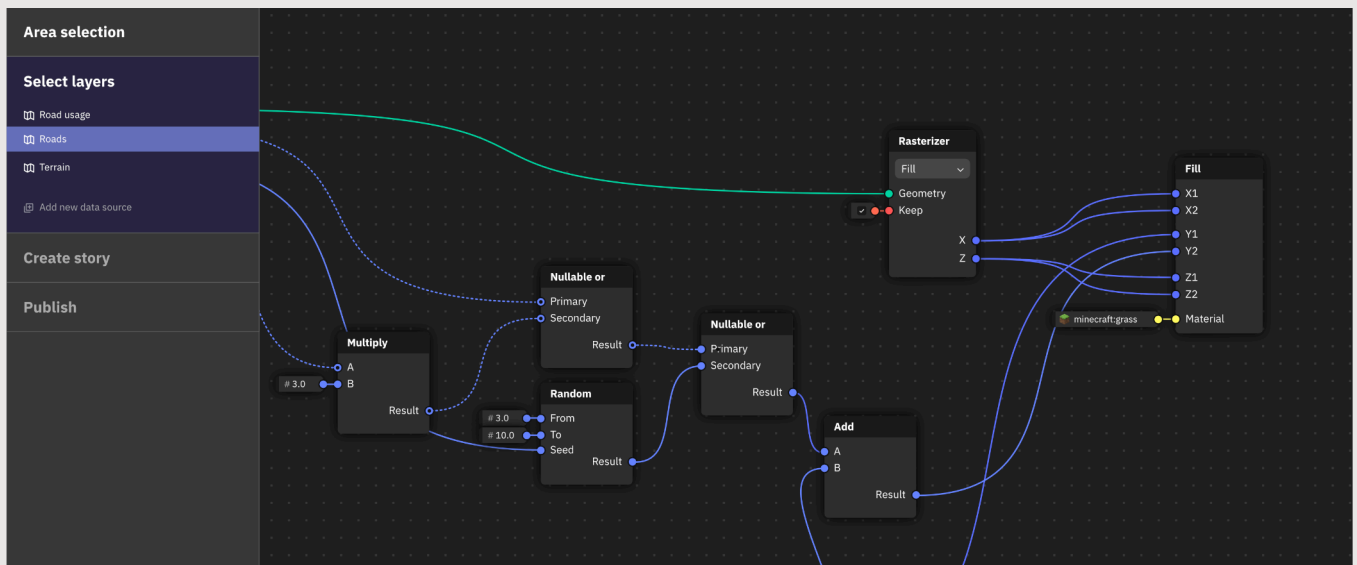


Figure 6.6: Final approach with the separation of the raster and vector parts removed

7 Technology

As mentioned in the requirement R6, the front-end of the system must be served as a web application. The computation will be performed on a back-end supplying its functionality through a REST API. This creates a natural segregation of the two parts of the application (visualized in Figure 7.1):

- **Client browsers:** User interface, layer creation, configuration
- **Minecraft server:** Configuration evaluation, world generation

In addition, a middle **Manager** layer could be added in the future (Figure 7.2). Such an addition would open options for creating multiple server instances, allowing for load balancing or creating dedicated servers for specific users. The manager would also provide missing account management functionality.

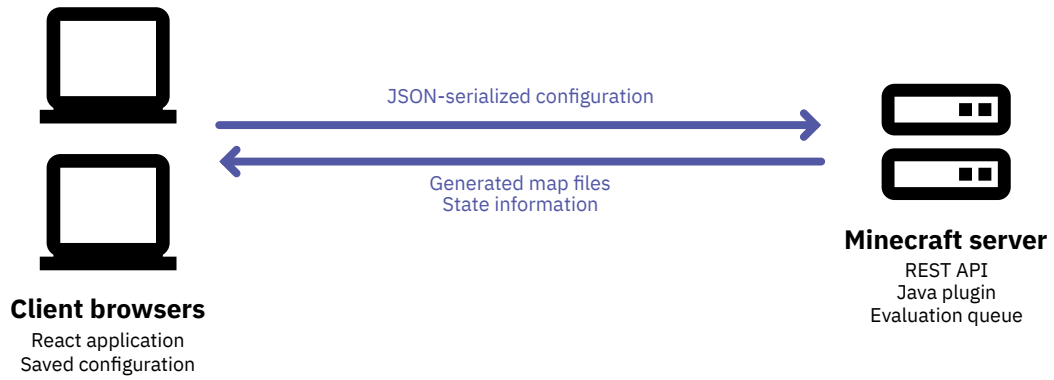


Figure 7.1: Current system architecture

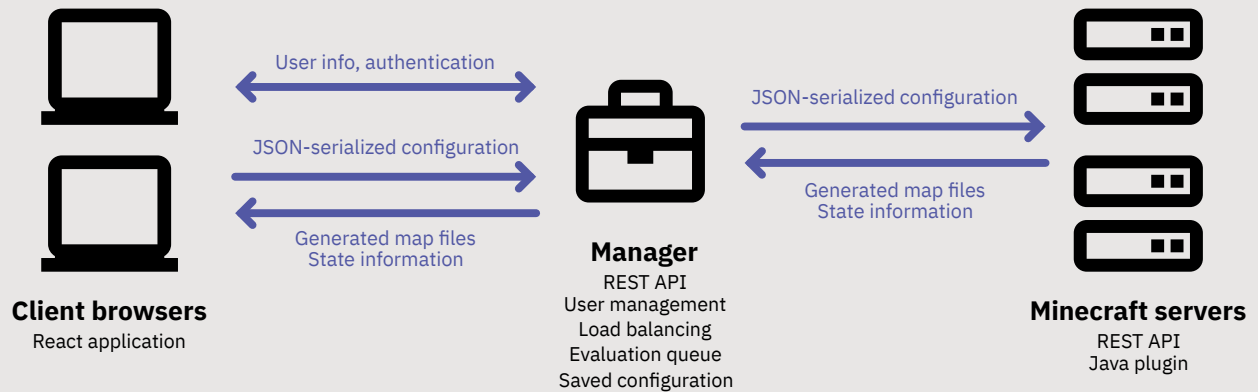


Figure 7.2: Extended system architecture with the addition of Manager

7.1 Web Client

The front-end of the application provides a User Interface (UI) for setting the coordinate transformation parameters, including the center of the map and scale. It also provides tools for node-based layer creation, as mentioned in Section 6.2.2. It additionally presents a preview of the generated Minecraft world directly in the browser without the requirement to start the game.

The application is created using **TypeScript** [54] programming language for future extendibility and readability. The **React JS** [55] front-end framework was used as a component-based front-end framework. Components from **ChakraUI** [56] component library were utilized for the primary user interface elements, such as buttons, tooltips, and drawers.

To allow selecting the size and center of the area that a creator is about to generate the application needed a component for displaying a map with a preview of the selected region. The **Leaflet** [57] library for rendering maps was adopted to display the main map component, including the interactivity and animations.

React Flow [58] library was included in the project to provide support for node-based graphs. React Flow is a customizable React component for building node-based editors and interactive diagrams [58]. It provided a strong foundation, and the implementation of the desired functionality would be very complicated without its inclusion. However, as the library did not provide all the functionality required by the application, it had to be heavily extended. The most important change was a custom implementation of the data storage system, because the state store included with the library was not sufficient.

For the preview of the Minecraft map, I created **react-minecraft-viewer** [59], a custom library for rendering a part of Minecraft region. More information about the library is in Section 8.3.

7.2 Server

The server implementation must be able to parse the user-created configuration, execute it, and, at the end, create a Minecraft world save. The last mentioned task is the trickiest part. Minecraft is a closed-source game that is (at the time of writing) under active development. Not only are new versions with new block types constantly added, but some technicalities of the game might change, and even the current save format might become deprecated. Multiple projects try to manipulate the save files externally using Python, JavaScript, or other languages [20]. However, they are community-based projects that do not guarantee any future development.

For the best forward compatibility, the back-end of the application has been implemented as a **plugin for the PaperMC** game server (see Chapter 4 for more information). Because PaperMC uses the original provided binaries of Minecraft and is currently the most commonly used Minecraft server with more than 100,000 running instances [24], Paper quickly ports its code to every new Minecraft version. By using it we can pretty much guarantee forward-compatibility while being sure the save files are handled correctly. On the contrary, such an approach is wasteful of computation resources as the default game mechanics still need to run behind our custom plugin.

The PaperMC plugins are written in Java and have to be compiled into a shadowed JAR [60] archive containing all the dependencies required by the plugin. Such an archive can then be inserted into the server folder, from where it gets loaded on the server startup alongside the Vanilla code.

Since the server must contain a simple REST API, the **Javalin** [61] library provided required functionality. It brings a functional approach to creating APIs and is more than sufficient for the needs of the project.

Handling the geographical data is complicated and implementing such functionality from scratch would be unwise. **GDAL** [62] is a system library that handles raster and vector geospatial files and thanks to the **proj library** [63], it supports a wide range of geographical projections as well. It is distributed as a standalone tool but contains official bindings for Java [21] and multiple other languages [22]. Thanks to that, the application can work with different data types without additional struggle with parsing.

7.3 Package

Running the application requires the setup of an extensive set of tools:

- **Web:** Node.JS, PNPM, dependencies, build, serving
- **Server:** Java, Gradle, GDAL, PaperMC server

This process makes the application cumbersome to set up for an average user and time-consuming to start using on a new machine. To mitigate this problem, the whole system has been packaged into two Docker [64] images and set up such that a single **Docker Compose** command can install all the dependencies, build and run the whole application. It also provides additional functionality for switching the Minecraft version across the system. As the Minecraft End User License Agreement (EULA) [23] does not allow redistribution of the server implementation, the Minecraft server itself gets installed after the container is started and the creator has agreed to the Minecraft EULA.

8 User Interface

From the creators point of view, the whole application is controlled through a web interface. The interface provides a wizard-like experience leading the steps of a user from the generic setup of the region to previewing and exporting the final map. The wizard consists of four individual steps described in the following sections.

The user interface is divided into two columns, as shown in Figure 8.1. In the left smaller column ①, a creator can switch between steps and configure the current step. The right column ② contains a visualization or graphical configuration related to the given step. More details are described in the following sections.

The top bar above the two columns contains a status indicator ③ that informs the user about the current status of the back-end server as well, as a pair of buttons for saving and loading the current configuration ④. The top right corner of the UI holds two more buttons for opening the documentation (more details about the documentation are described in Section 8.4).

The current configuration of the application is stored to browser's local storage providing an easy way to continue, where the user left off. It can be also downloaded as a JSON file and transferred to a new machine or browser.

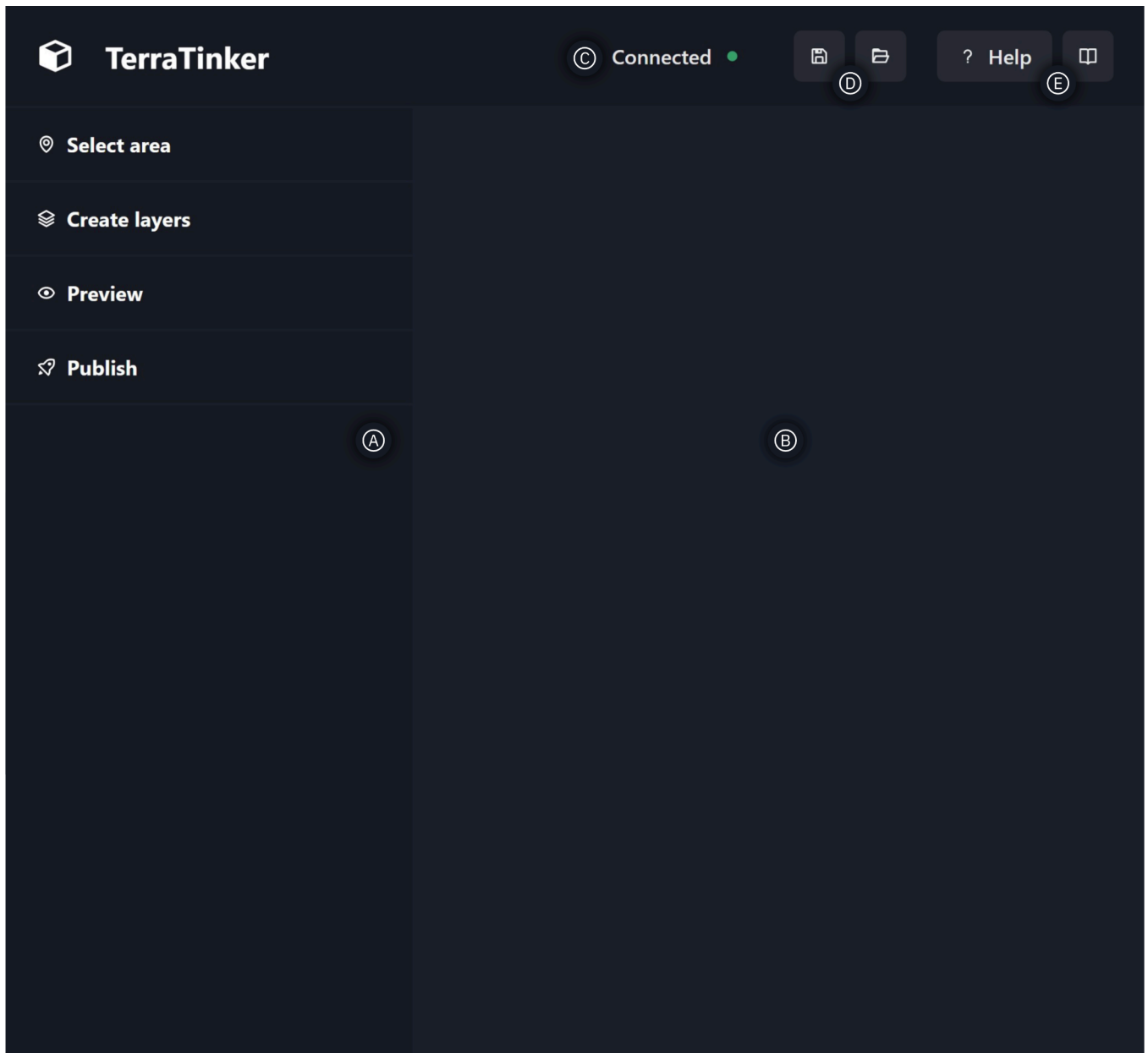


Figure 8.1: The overall layout of the user interface of TerraTinker

8.1 Region Selection

The first step in the creation process is the region selection (R1) depicted in Figure 8.2. In this step, the creator is presented with a map (A) and a set of options for configuring the map projection (B) (described in Section 6.2.1).

Live visualization of the selected region is displayed in the map window on the right side (C). The creator can see a real-time impact of changing the parameters of the projection on the size of the selected region. They can also drag-and-drop the region directly in the map, improving the user experience.

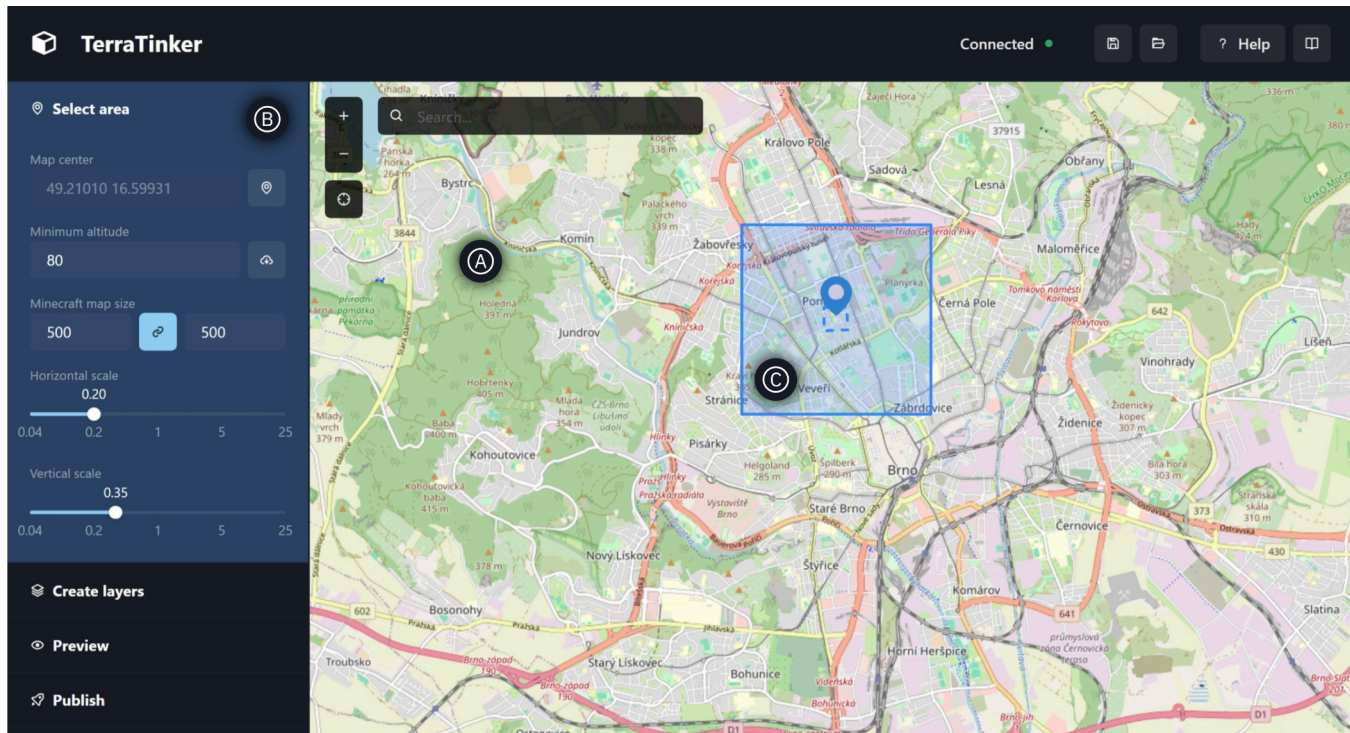




Figure 8.2: Region selection step with the selected region around FI MUNI

8.2 Layers Design

The layer design shown in Figure 8.3 is the core of the whole process. It empowers a creator to define multiple independent transformers (layers) ① using a node graph. The node graph ② represents a series of operations applied to the input data. Each layer independently transforms input geospatial data into a common Minecraft map. Layers are evaluated in a bottom-up manner, meaning that the bottom-most layer is evaluated first and the top-most layer is evaluated last. This behavior is similar to how layers behave in a graphic editors like Photoshop. This empowers the creators to define layers that overwrite the results of the previous ones.

A single **node** ③ in the graph represents a basic operation applied to the input data of the node. Inputs can either be values or links to an output of other nodes. The application contains three general types of nodes (indicated with an icon in the top-right corner):

- **Generic node** (no icon) ④ performs a generic transformation of data, for one set of inputs it produces one set of outputs.
- **Fork node**  ⑤ can produce multiple sets of outputs for one set of inputs.
- **Action node**  ⑥ performs an action, like placing a block, and usually does not have any outputs.

The application currently contains 45 different nodes. This includes data loaders required by R2 (GeoJSON, GeoTIFF, ESRI Shapefile, OpenStreetMap Overpass API), nodes for working with numbers, strings, and boolean values, processing vector and raster files, and manipulating the Minecraft world (R3). All the nodes contain documentation (R5) with example usage (where it is necessary for understanding how the node works). Furthermore, the application contains a Comment node ⑦ that describes the thought processes within the examples and allows the creators to document the layers by themselves.

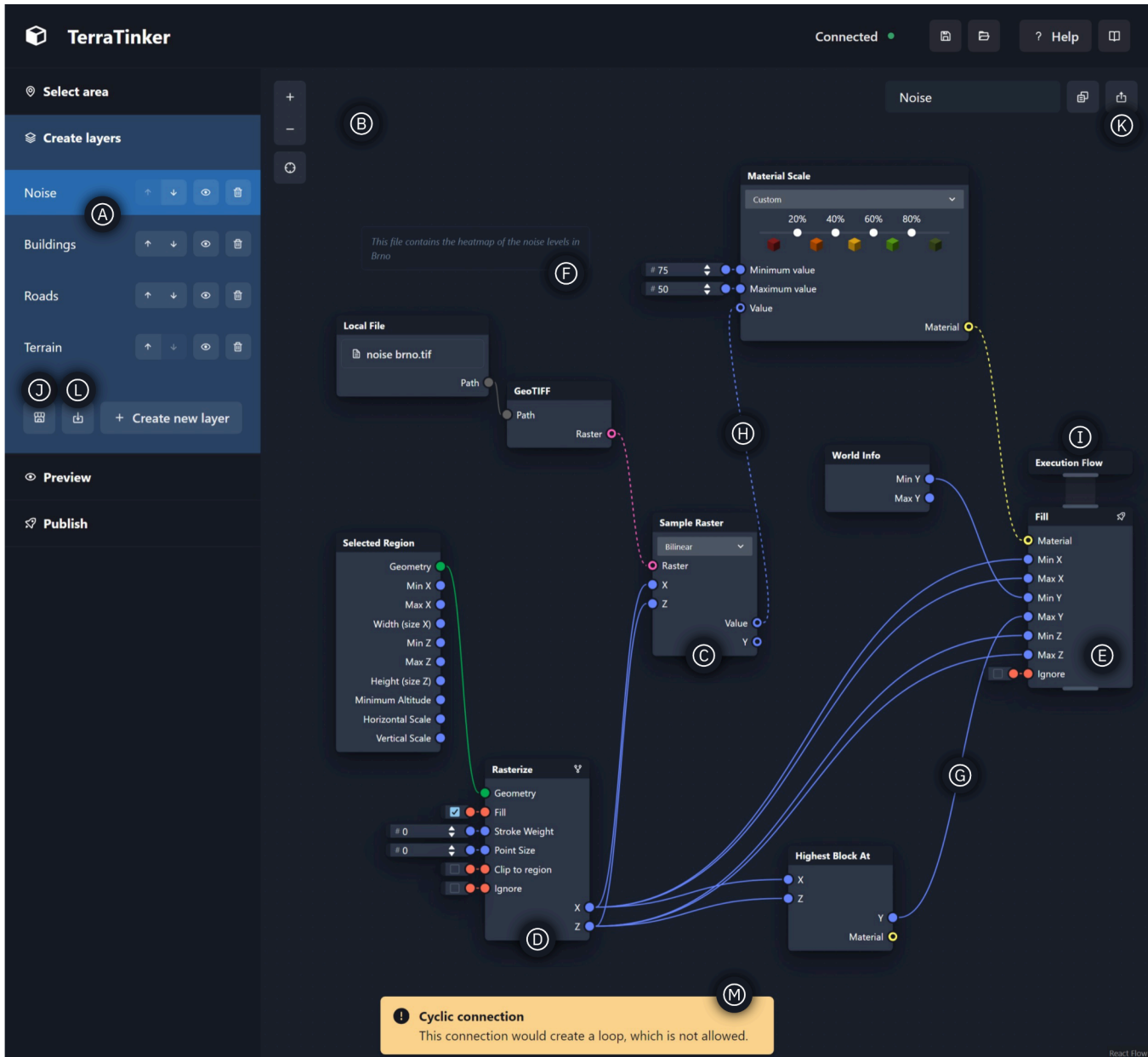


Figure 8.3: The layer creation step with four layers after an attempt to create an invalid connection

Links ③ connect inputs and outputs of nodes and represent the flow of the data. Links can be connected only between compatible inputs and outputs and must not form cycles. When creating a connection, the graph is traversed using the Depth-first search (DFS) to detect any cycles, prevent them, and report them to the creator ④. Each data type (● number, ● boolean, ● string, ● geometry, ● raster, and ● material) carried by a link is represented with a different color of the connection.

If the node has a possibility of outputting a null value, the corresponding output shows a black dot ⑤ and its links are rendered using a dashed line ⑥. The indication is included to ensure correct null value handling, to inform the user about a potential problem and simplify its elimination. This is especially important if a node is unable to work with null values, since such node outputs null upon receiving null on any of its inputs.

Execution flow ⑦ is used to define the order of execution of action nodes. Unlike generic or fork nodes, action nodes have an additional input and output at the top and bottom respectively. Those are used to connect the nodes into the execution flow and specify their order of execution. The evaluation algorithm described in Chapter 9 guarantees that every action node is executed before its successors. Similarly to the layer order, this can be useful for overriding outputs of a previous action.

The creator is encouraged to browse provided **templates** ⑧ and use them as a scaffolding for generating a simple scenery. The templates currently provide layers for buildings, streets, and terrain but can be expanded in the future to provide more options for less experienced users. Each layer can be also exported ⑨ and imported ⑩ for sharing of the creations between the creators.

8.3 Preview and Publish

The last step of the creation process is generating the Minecraft map. On the front-end side, the creator requests a generated map from the back-end and after the evaluation is finished, they can download a ZIP archive containing the generated world save. This save can then be directly extracted into Minecraft's saves directory or provided to a server for multiplayer hosting (R4).

However, generating the map for the whole selected region every time the creator wants to preview and validate the visualization is very inefficient, time-consuming, and importing it into the game every time is also inconvenient. For such situation, the creators can use the preview step (Figure 8.4), which generates only 4×4 chunks (64×64 blocks) in the center of the selected region and provides the creator with a preview directly in the browser.

The preview uses `react-minecraft-viewer` [59] package I created for TerraTinker. React Minecraft Viewer is a simple component for displaying a part of Minecraft region file with React.js. It is based on the `Deepslate` [65] library with a majority of the code extracted from the NBT viewer for VS Code extension [66]. The original code from the extension was extracted to a standalone package with additional support for touch controls and loading multiple chunks. The library is published on the NPM package registry [67] under the MIT License.

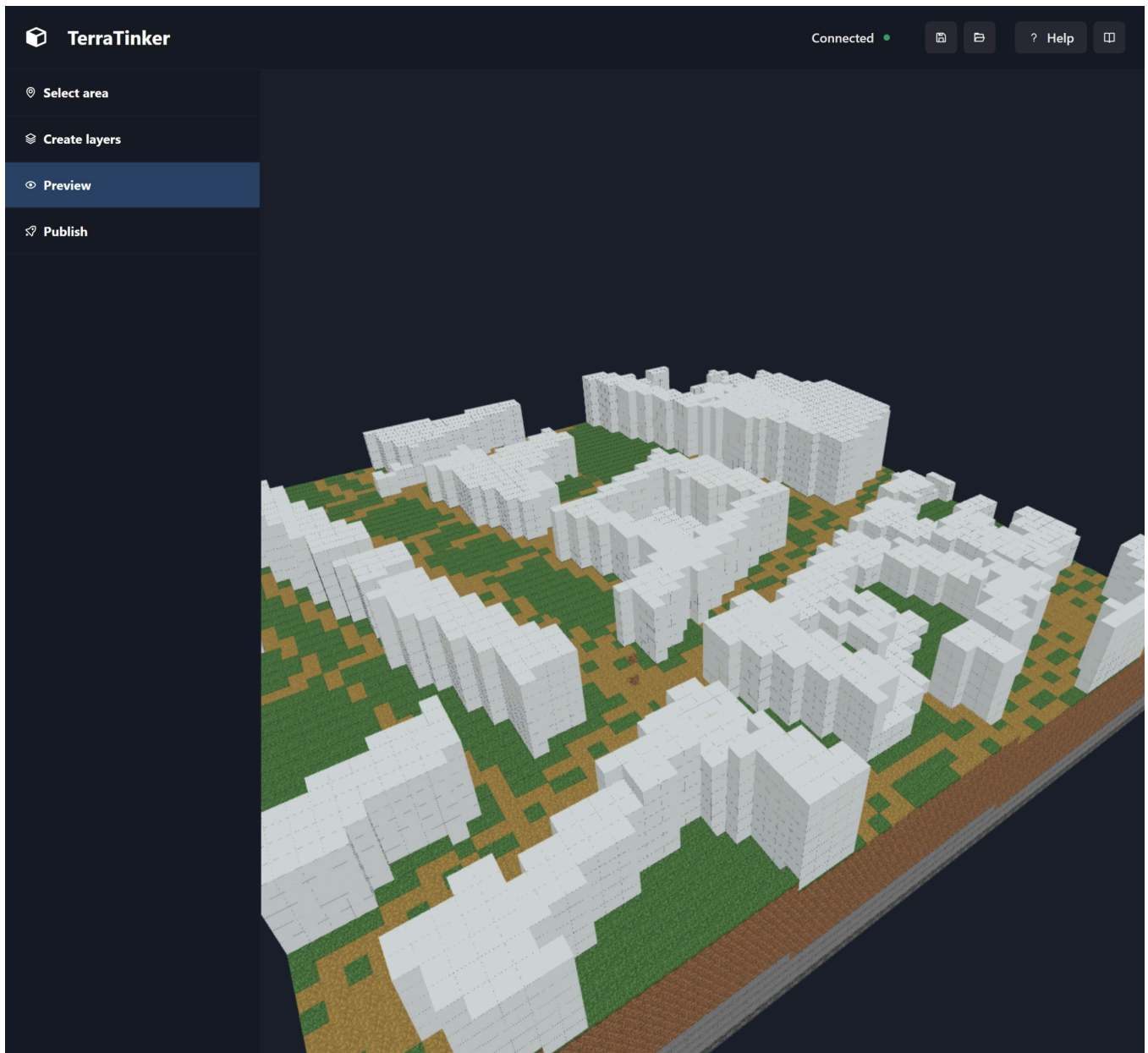


Figure 8.4: The preview step with the model of Brno around FI MUNI (datasets from Section 10.1.1)

8.4 Documentation

As mentioned in Section 6.2.2, the final application had to be accompanied with comprehensive documentation (R5). A user can either open the last visited page and browse the documentation (Figure 8.5) through the file tree in the left sidebar ④, or use context help (Figure 8.6) throughout the application. By clicking the help button ⑤, an overlay is displayed and a user can open the documentation directly for any on-screen component by clicking the blue icon in its top right corner ⑥. To ensure the best experience for the creators, all nodes include context help that should simplify its understanding and aid the user during the layer creation process.

The documentation pages can contain formatted text as well as individual nodes ⑦ or even examples of node graphs. It is structured into chapters with links ⑧ between them. A user can return to a previous page with the back button in the top-left corner of the right panel ⑨.

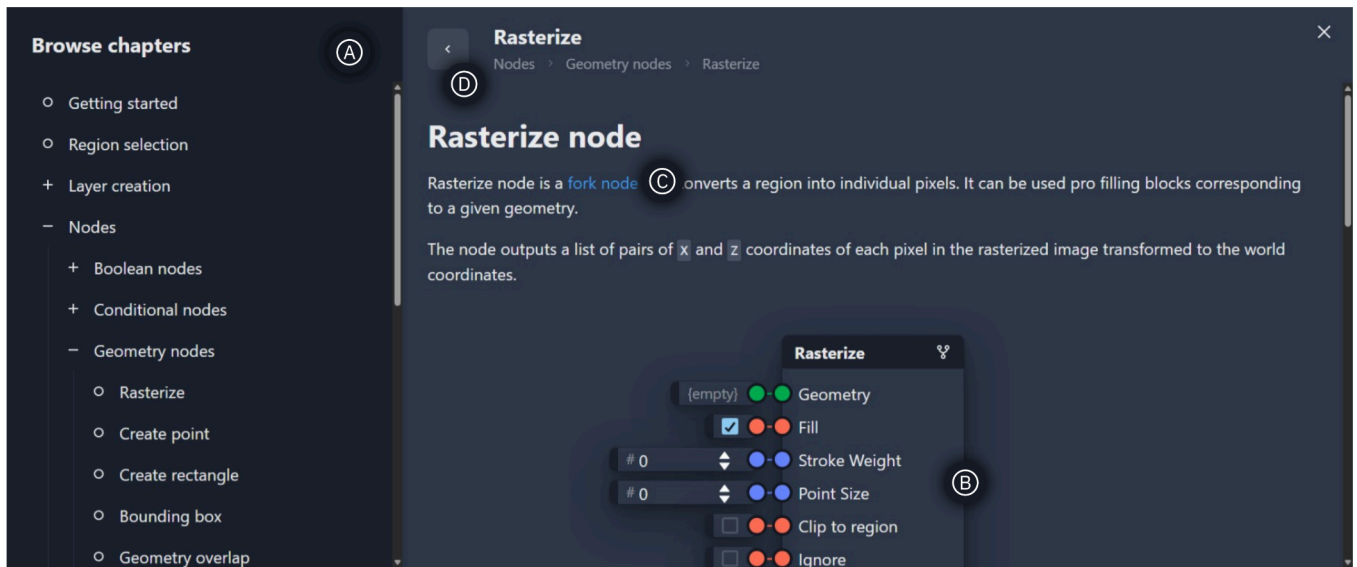


Figure 8.5: The documentation of the Rasterize node



Figure 8.6: The help overlay open on the layer creation step

9 Evaluation Algorithm

After a creator is satisfied with the selected parameters and layers, as defined in Chapter 8, the configuration is serialized into a JSON (JavaScript Object Notation) file and an evaluation is requested on the back-end. In the current implementation, there is a single server that handles all requests sequentially, however, this approach can be extended in the future (see Chapter 7). Upon receiving a request, the configuration is enqueued for evaluation and when the server is idle, the oldest configuration is dequeued and evaluated.

During the evaluation, each node can receive multiple sets (rows) of inputs and return multiple sets of outputs. A row of inputs contains values for all input fields of a given node. For example, for a Math node, a row of inputs is a pair of numbers and a row of outputs is a single number calculated from the two inputs (depending on the operation specified). For most nodes, the number of input and output rows matches. This condition does not have to hold for the fork nodes (see Section 8.2 for more details), where the number of output rows can be larger (often) or smaller than the number of the input rows.

Using a simple naïve algorithm, each node would receive an array with all the rows for the inputs and would directly output an array of all its output rows. This algorithm is common for similar tools, such as Carto. It requires only a single pass through the layer tree, however, it uses a very large amount of memory as it needs to store all the resulting values of all the nodes at the same time. This algorithm could also cause problems when the number of input and output rows of any node do not match (which is common for fork nodes in TerraTinker). Such situ-

ation is depicted in Figure 9.1, where the inputs of the action node do not match in length. With this naïve approach, it is very hard to match the corresponding rows together and this situation usually results in an error.

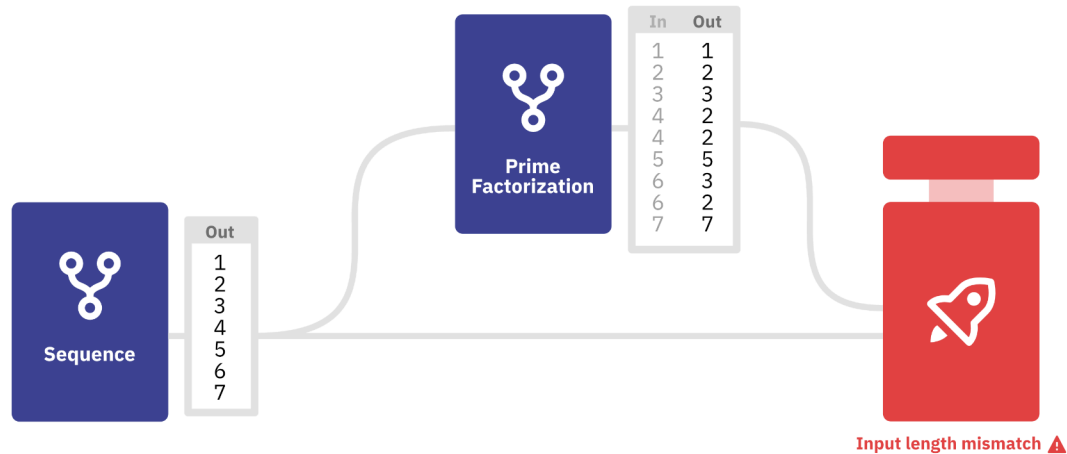


Figure 9.1: An example of the problematic scenario for the naïve algorithm — the action node has mismatched lengths of the inputs, because the Prime Factorization node outputs multiple primes for a single non-prime number

For the listed reasons, an alternative algorithm following the main idea described in Pseudocode 9.1 was considered and later selected for a better fit for the application. The program [A](#) including all the nodes and their connections gets recursively traversed [C](#) from a selected start node [B](#) towards its prerequisites. Upon returning from the recursion, each node is evaluated using the values of the prerequisites [D](#) and its result is stored in a common dictionary called `tree`. If a node is a fork, it cannot be evaluated immediately. Rather it returns a reference to itself [E](#) indicating that a fork has been visited and the rest of the tree needs to be evaluated for all the possible outputs of such node [F](#). An example of evaluation using this algorithm is depicted in Figure 9.2.

```

# The root function, that evaluates the program
# - program: A Dictionary with the individual nodes from the graph
# - startNode: ID of an action node, that the evaluation starts from
function EvaluateProgram(program @, startNode @)
    tree <- {}
    EvaluateForTree(program, startNode, tree)
end

# A function, that starts the evaluation for a specific result tree, the tree can be empty
# (when calling from EvaluateProgram) or partially full (when calling from EvaluateFork)
# - tree: A dictionary with the results of individual nodes
function EvaluateForTree(program, startNode, tree)
    fork <- program[startNode].Evaluate(program, tree)
    if fork != null then
        EvaluateFork(program, startNode, tree, fork)
    end
end

# Evaluate the program for all rows of a specific fork node
# - fork: ID of a fork node, for which the node graph will be completely evaluated
function EvaluateFork(program, startNode, tree, fork)
    # While the specified fork node has more values to return, continue the evaluation
    while program[fork].EvaluateNext(program, tree.Copy()) do
        EvaluateForTree(program, startNode, treeCopy)
    end
end

```

Pseudocode 9.1: Pseudocode that served as a base for the algorithm implementation (continued on page 66)


```

# A function of fork nodes, sets its next outputs into the tree
# If all values were already evaluated, return false to indicate it
function Node.EvaluateNext(program, tree) ⑥
  if not evaluated all do
    tree[this.id] <- result
    return true
  end
  return false
end

# Evaluate a specific node
function Node.Evaluate(program, tree)
  # Traverse all the prerequisites and evaluate them, if they are not already
  for input in this.input do
    if !tree.contains(input) then
      fork <- program[input].Evaluate(program, tree) ③
      if fork != null then
        # If the prerequisites contain an unevaluated fork node,
        # we return its ID
        return fork
      end
    end
  end
end

# If this node is a fork, it should be evaluated individually, so we return its ID
if this.isFork do
  return this.id ⑤
end

# Evaluate the node here using the values stored in the tree (tree[input])
tree[this.id] <- result ④

# This node is not a fork, so we return null
return null
end

```

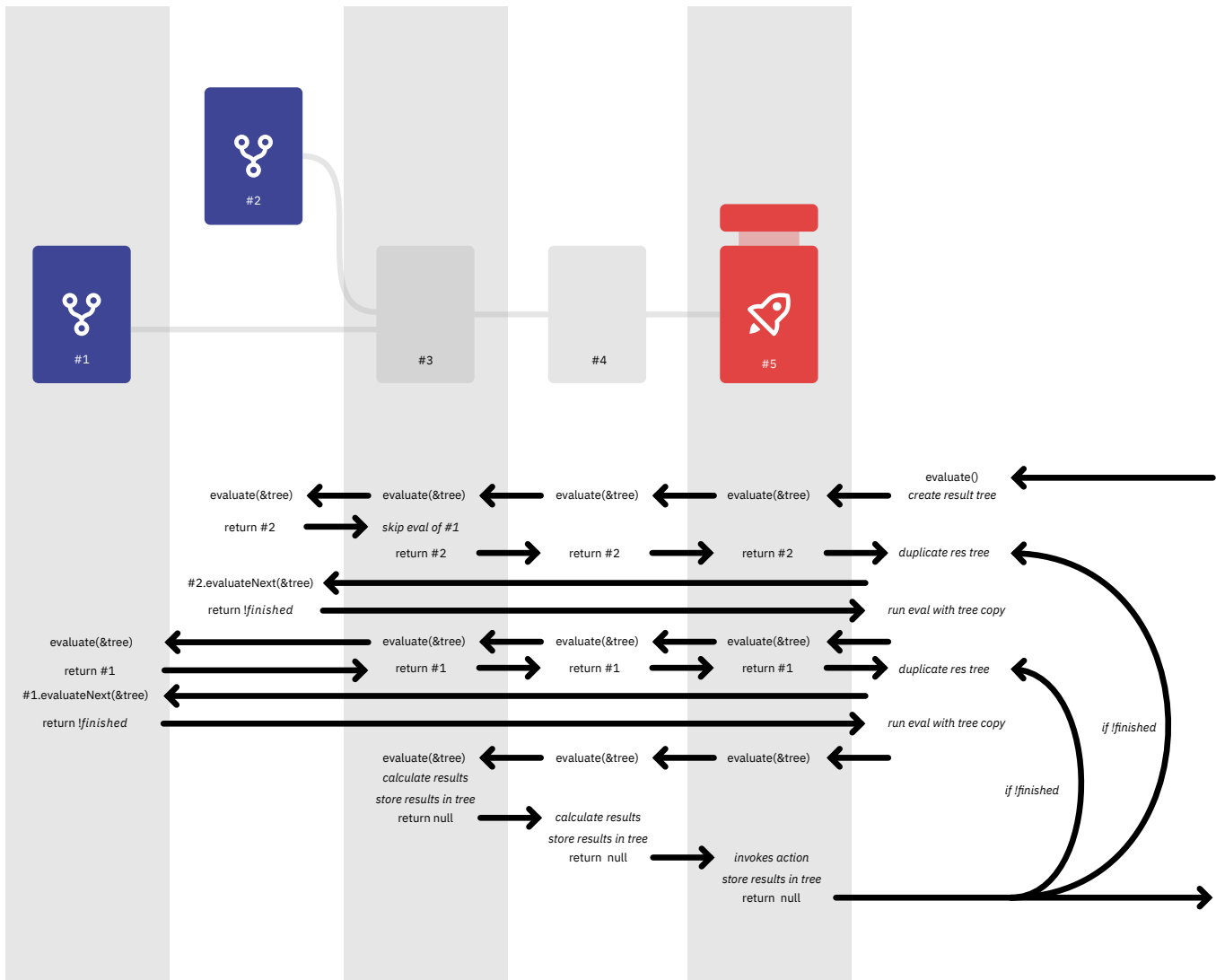


Figure 9.2: Visualization of the algorithm described in the Pseudocode 9.1 for an example with two fork nodes (#1, #2) and one action node (#5)

By implementing the described algorithm, all action nodes get executed for all the combinations of fork node outputs within its prerequisites (cartesian product of all rows outputted from the connected fork nodes). This approach might seem inefficient, however, it is the key that unlocks the freedom of dataset merging. Furthermore, a set of optimizations was implemented to increase the performance.

The algorithm allows for merging of different datasets, their filtering and transformation. Merging is performed automatically on any node that has two (or more) different fork nodes as its prerequisite. Such node is then executed for a cartesian product of its input rows. Filtering is applied on the action and fork nodes (that have any input) by an additional input field called `Ignore`. If the value of `Ignore` is true, the evaluation of that node is preliminarily skipped saving expensive computations of values that would be unused at the end.

9.1 Handling the Execution Flow

Event though the evaluation follows strict rules, there are still different approaches to handling multiple action nodes within the same layer. Each approach has different efficiency and each one is more suitable in a different situation. In this section, two different variations of the same algorithm are explained and compared. Both of those were considered for the final implementation and as both are valid in certain scenarios.

Approach A: Each action is independent — As the name suggests, in this approach, each action node is evaluated as if no other action is present in a given layer. The whole program is evaluated independently for each action node from the top to the bottom, guaranteeing the order of the actions.

Approach B: Actions are prerequisites of each other — Using this approach, only the last node in the execution flow is evaluated. Each action has then one additional prerequisite being the previous node in the flow. This way the order of action nodes is guaranteed and the results from the rest of the tree are shared between the actions.

We can compare the listed approaches in two problematic scenarios:

Scenario 1

In this scenario, the program contains a single data source that is processed by a computationally expensive node and then used by two separate action nodes, as shown in Figure 9.3.

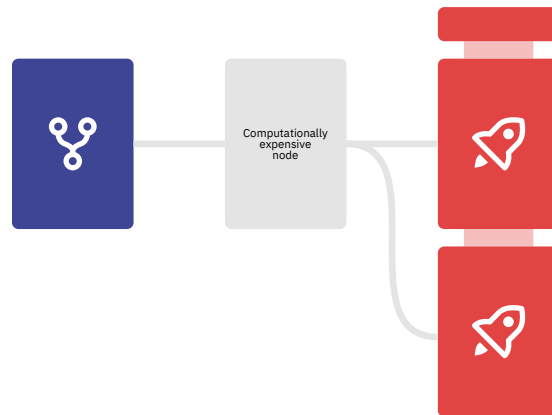


Figure 9.3: Scheme of Scenario 1 with a single fork node used by two different actions

Approach A performs poorly in this situation as the computationally expensive operation gets performed twice for each row of the data source (once for each action). On the contrary, using **Approach B**, the expensive operation is performed only once for each row as the results of it are shared between both actions. The Approach B is thus a better fit for this scenario.

Scenario 2

The contrasting scenario uses two independent data sources and two actions, where each one uses only a single data source, as shown in Figure 9.4.

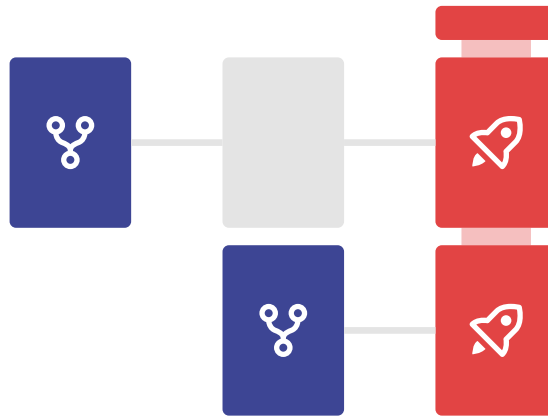


Figure 9.4: Scheme of Scenario 2 with two fork nodes used each by one action node

Approach B considers the upper action as the prerequisite of the lower one thus performing the lower action for each combination of the input rows, as described in the algorithm. That is very inefficient, as the bottom action is not dependent on any value of the upper data source. **Approach A** considers each action independently, thus eliminating the mentioned problem.

When using TerraTinker as intended and defining separate layers for separate data sources, **Scenario 2** is basically eliminated making **Scenario 1** more problematic. Due to that, the implementation of **Approach B** was selected for the current implementation of the evaluation algorithm. However, it might be beneficial to give creators ability to switch between them in the future.

9.2 Stale Data and On-Demand Evaluation

Another problem with the described algorithm is that some nodes cannot be evaluated in advance. Consider a situation when a creator wants to place 100 blocks randomly inside a given region. They can use a Random number node to generate the coordinates, such as in Figure 9.5. However, this node could be evaluated in advance, before any fork node would cause the program to split (the Sequence node in our example). In such case, the block would get placed on the same coordinates 100 times.

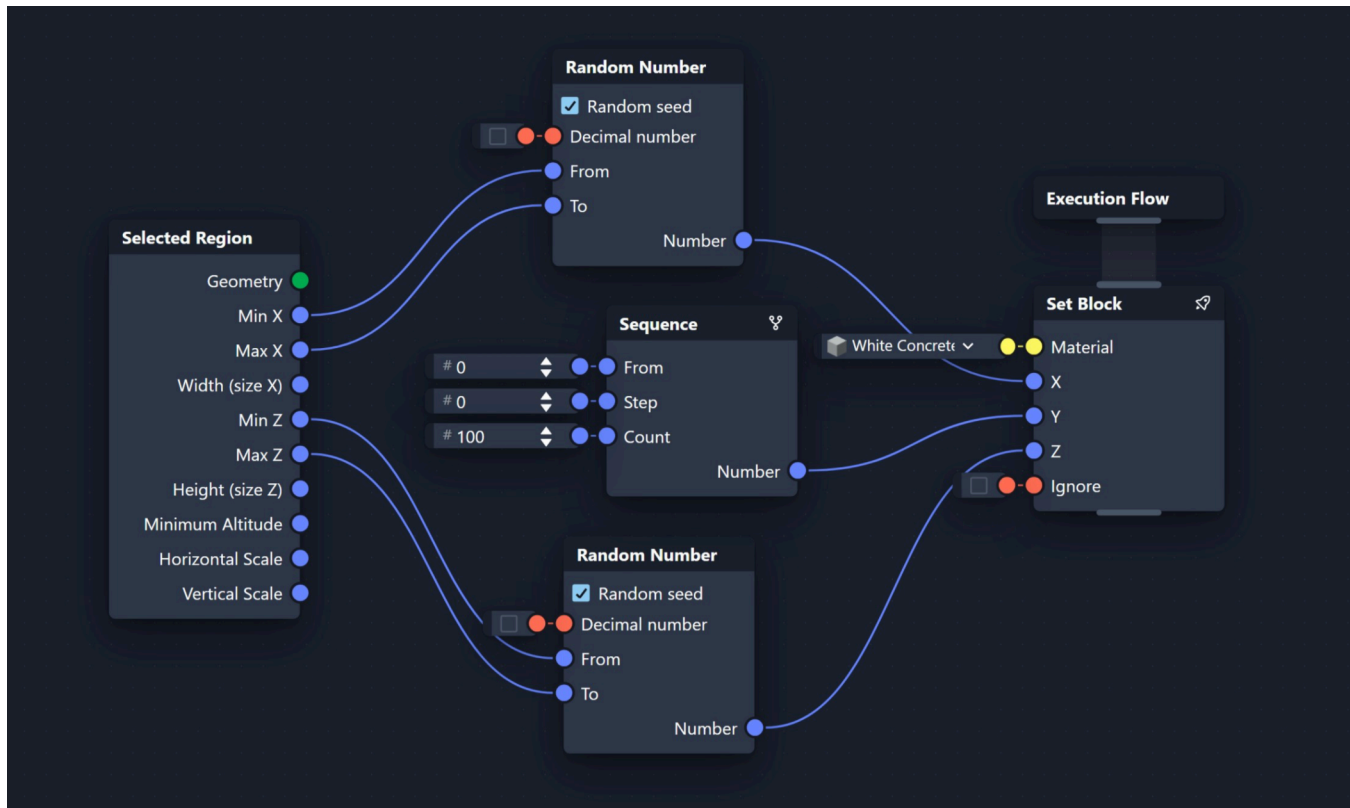


Figure 9.5: A layer for placing 100 blocks randomly in the region

To mitigate this problem, TeraTinker does not evaluate the nodes immediately, but rather stores lambda functions containing the evaluation algorithm. With this approach, the actual output values will be obtained only when it is necessary – when an action node or a fork node will get evaluated.

This workaround is unfortunately also not perfect. The creator would expect that for a single execution of an action node an output of any node would be consistent. That is not true with this approach, as the value is obtained on demand for every input field of each node separately. This problem can be solved by adding a cache during the evaluation step, but as so little nodes are non-deterministic in this way, the implementation of such cache will be left for future work.

10 Usage and Results

TerraTinker at its early stage has been used by the Craft-my-Shore project at the Faculty of Architecture, Planning & Environmental Policy at the University College Dublin to generate virtual version of multiple areas of coastal Dublin and neighboring towns (Figure 10.1). This project tries to demonstrate the impact of the water level rising, embrace young players to improve the situation and shape the coast to mitigate the effect of flooding.

10.1 Geospatial Datasets

As already stated, the primary usage and the goal of the TerraTinker application is to visualize geospatial datasets inside a Minecraft world. This section contains examples of such Minecraft maps generated with TerraTinker. Each example includes a short description of the configuration and layers and the list of used datasets.

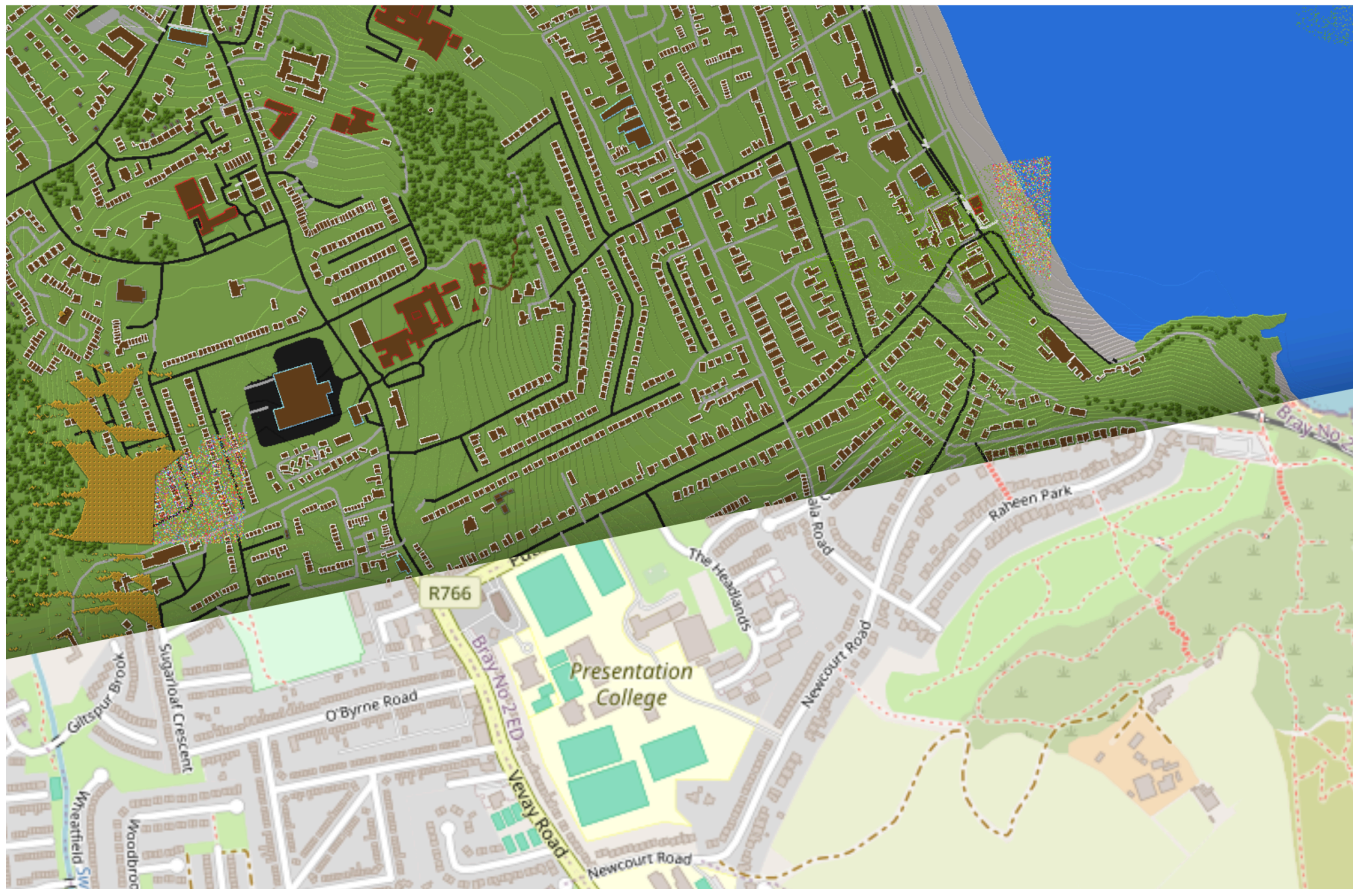


Figure 10.1: The town of Bray generated by the early stages of the TerraTinker application (top), comparison with the OpenStreetMap of the same area (bottom)



Figure 10.2: The base model generated with the terrain, building and streets layers

10.1.1 Base Model

This model utilizes three layers available as the templates. The first layer uses a GeoTIFF file to generate the terrain. The two additional layers both use OpenStreetMap to visualize roads and buildings (Figures 10.2 and 10.3).

Datasets

- **MERIT DEM** (Multi-Error-Removed Improved-Terrain Digital Elevation Model) by the Institute of Industrial Sciences, The University of Tokyo [27] – used for the terrain data
- **OpenStreetMap** (available under the Open Database License) [39] – streets and buildings

Configuration

This configuration without the datasets is available in the source code under `samples/Configurations/base.json`.

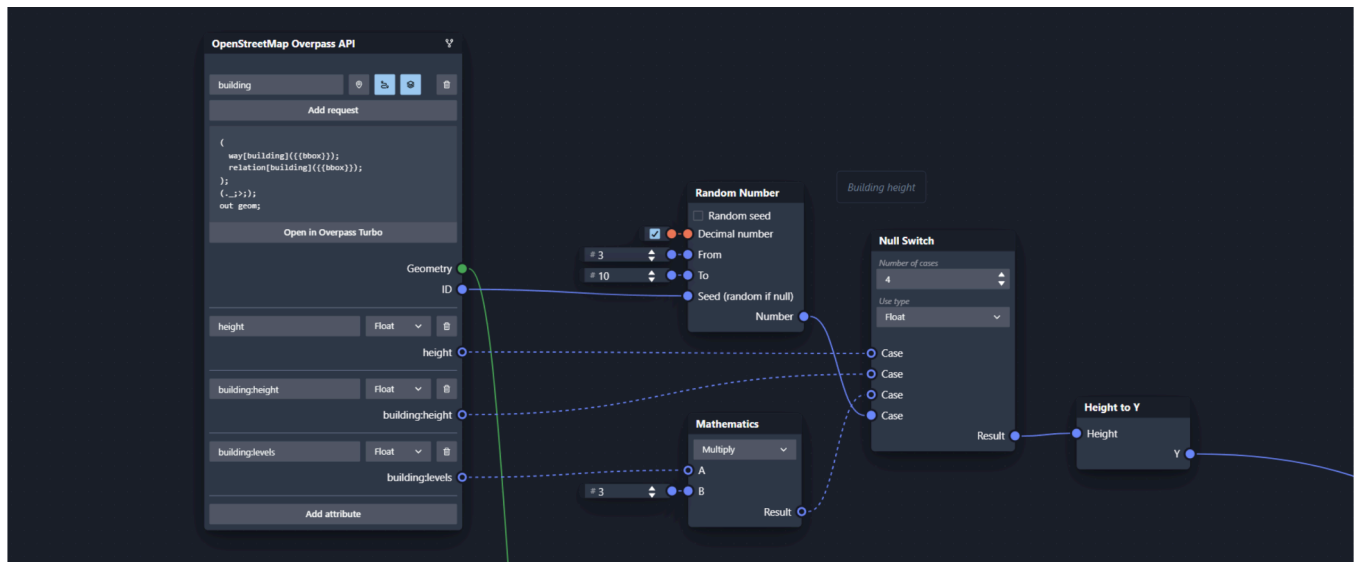


Figure 10.3: The part of the layer for the building generation that handles calculation of the height of buildings

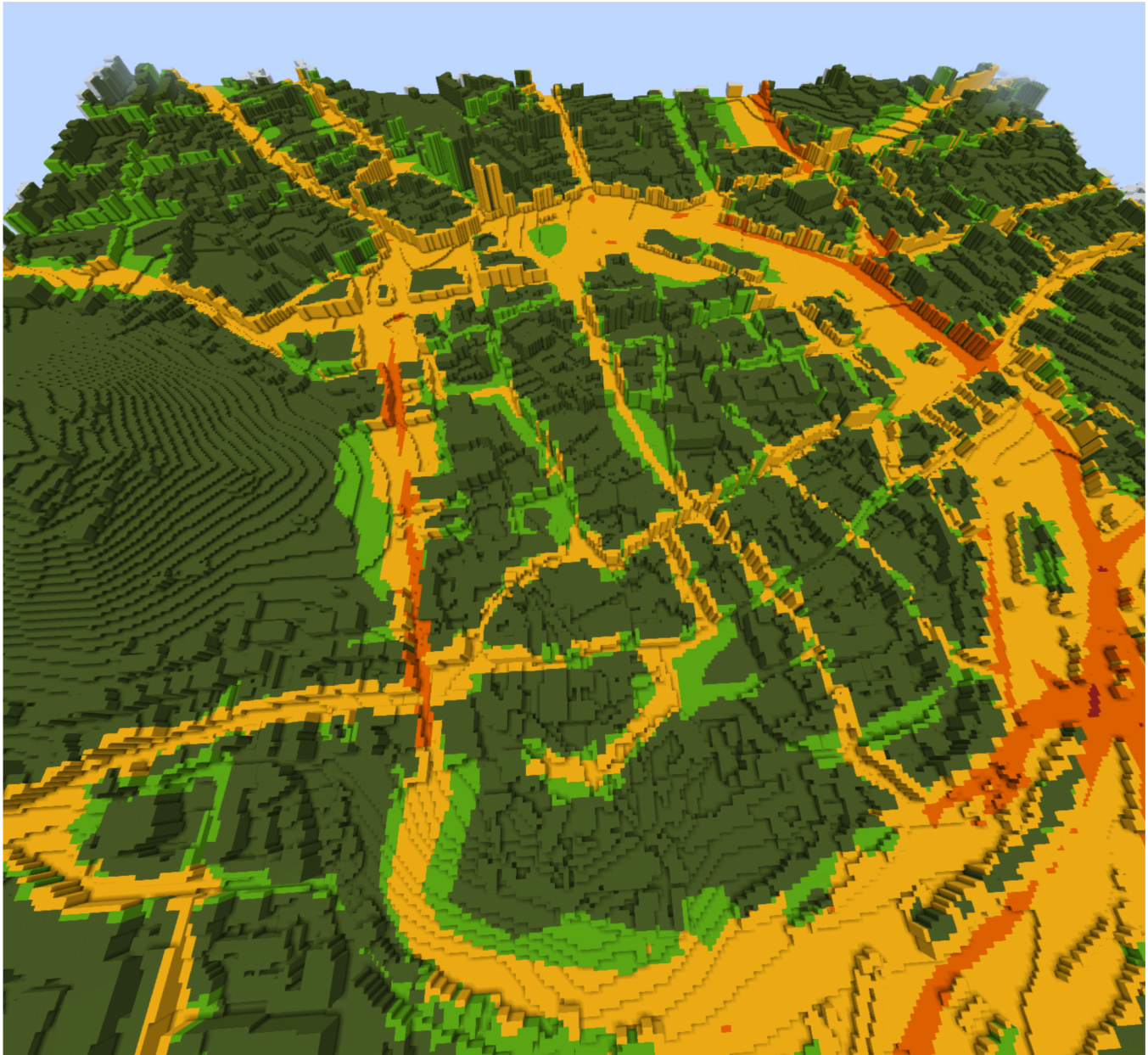


Figure 10.4: A map visualizing the noise levels in the Brno city center

10.1.2 Noise Levels

This map visualizes the noise levels in the Brno city. The model utilizes the base model described in the previous section. The map is then colored according to a raster containing the noise heatmap (Figures 10.4 and 10.5).

Datasets

- **Noise level 2022** [26] available under the public license by the data.Brno portal
- **MERIT DEM** [27] – used for the terrain data
- **OpenStreetMap** [39] – used for the buildings

Configuration

This configuration without the datasets is available in the source code under `samples/Configurations/noise_level.json`.

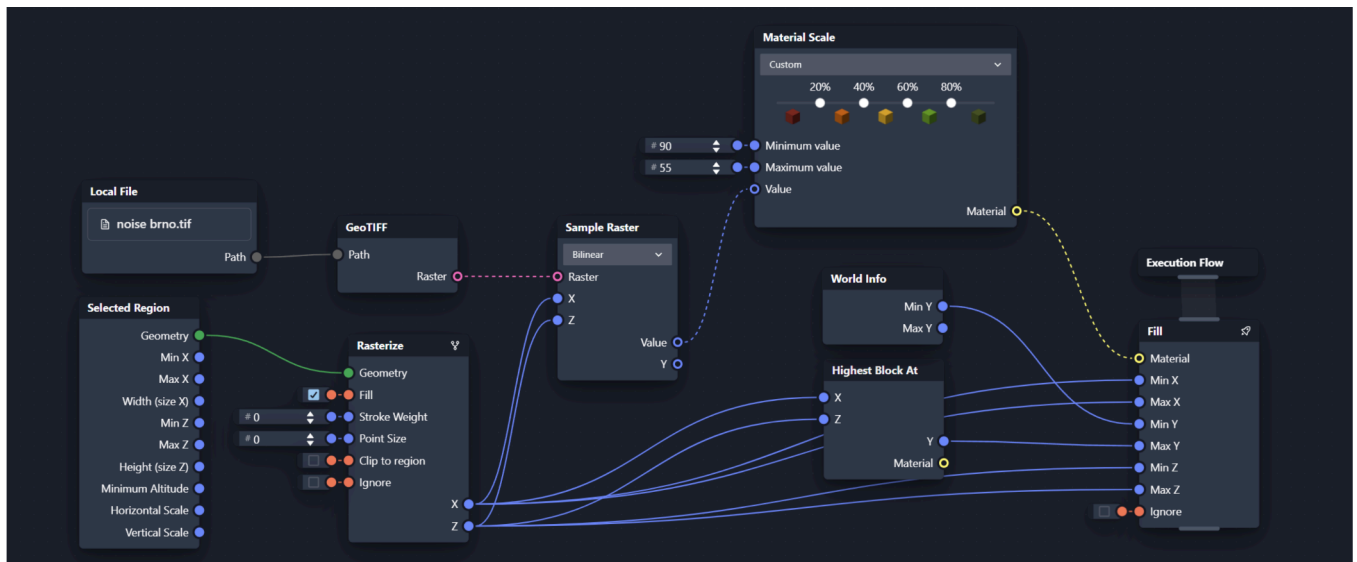


Figure 10.5: Layer for coloring the existing map according to the raster dataset containing the noise data

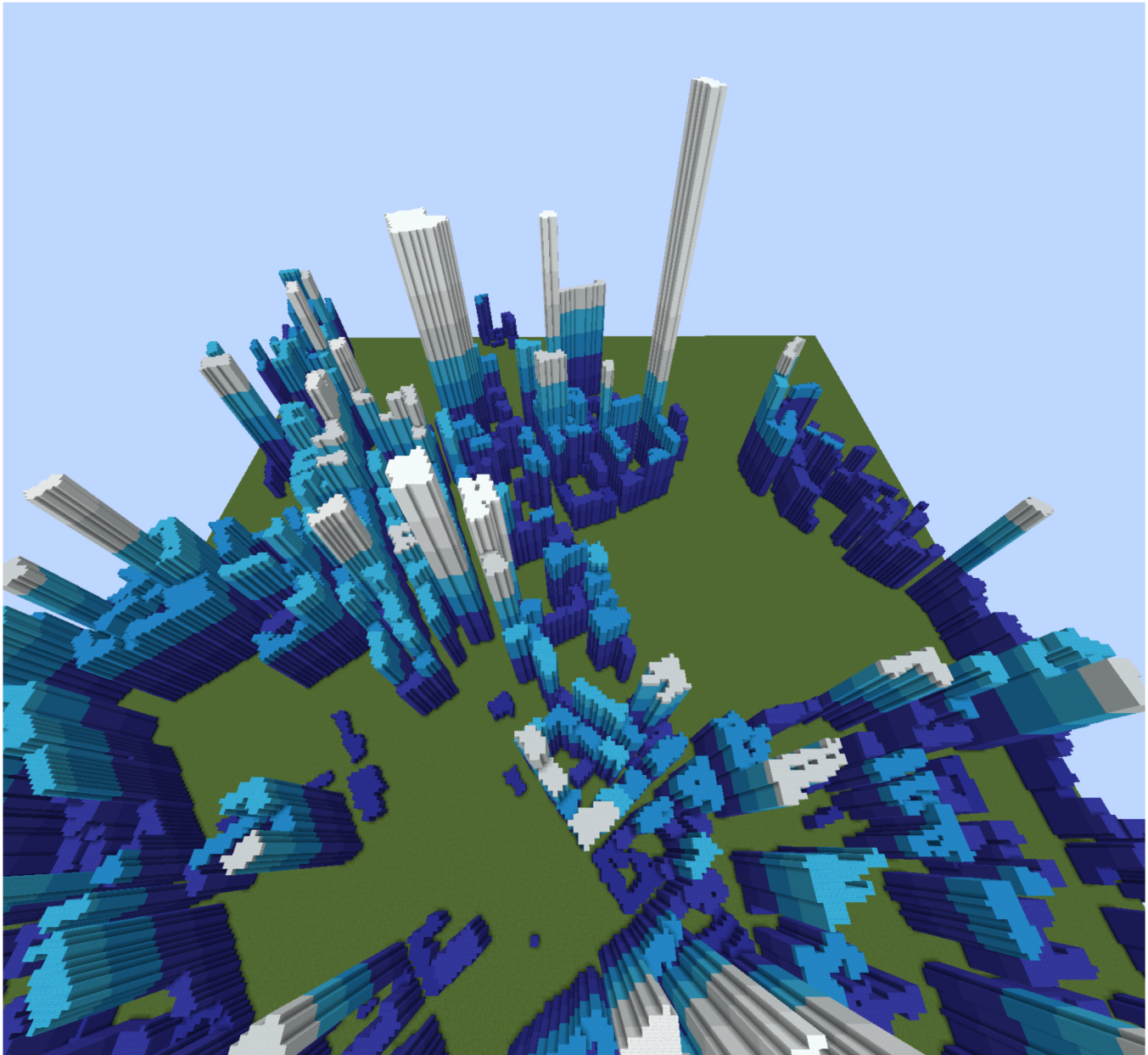


Figure 10.6: A visualization of the number of people living at addresses around the Lužánky park, Brno

10.1.3 Number of Citizens

This model visualizes the number of people living at addresses throughout Brno. The number of people is displayed as height of the different buildings. The terrain layer is not utilized to provide clear comparison between the building heights (Figures 10.6 and 10.7).

Datasets

- **Number of people living at the addresses** [25] available under the public license by the data.Brno portal
- **OpenStreetMap** [39] – used for the building outlines

Configuration

This configuration without the datasets is available in the source code under `samples/Configurations/number of citizens.json`.

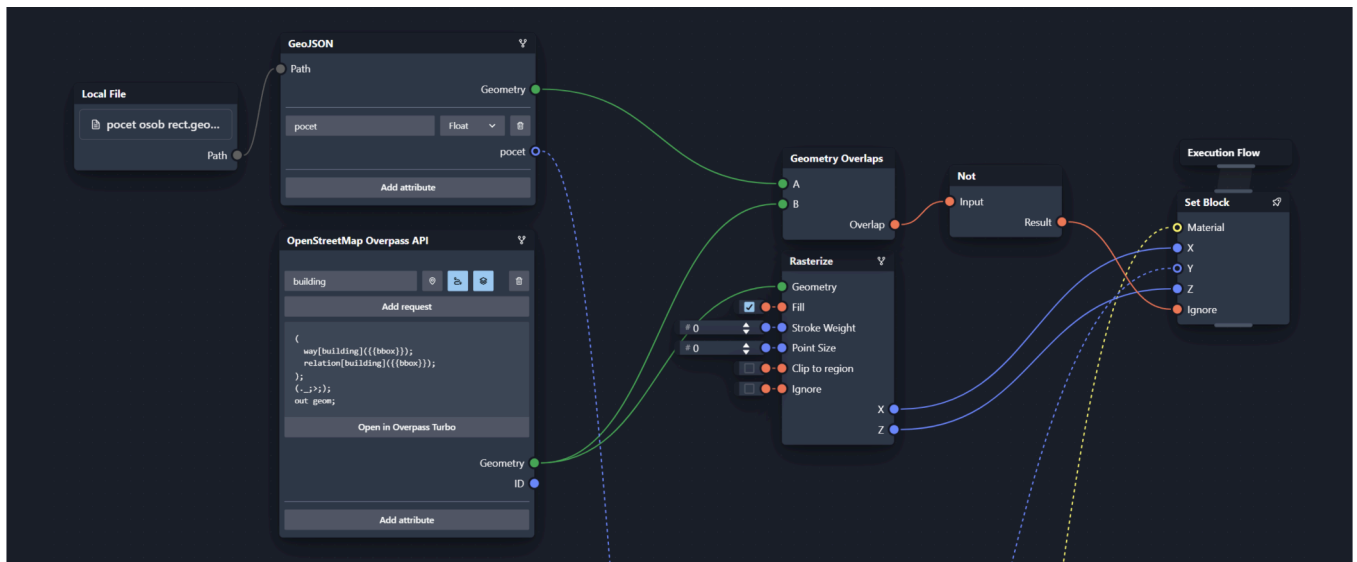







Figure 10.7: Section of the layer, that determines, if an address from the GeoJSON containing the number of people corresponds to location of a building from the OpenStreetMap

10.2 Mathematics and Geometry

Apart from the geospatial data visualization, the TerraTinker application can be useful in other education fields. In particular, you can see examples of usage for plotting functions and visualizing mathematical definitions of geometric bodies. Even though such use case was not intended, it can be very useful and definitely worth exploring.

For the described use case, one can use Sequence nodes , which generate an arbitrary sequence of numbers. Multiple Sequence nodes can be mapped to individual axes of the coordinate space and their combination can then be inputted to a mathematical formula. The formula generates either value for the remaining unused axis  or it can be used for filtering the placed blocks .

The values from a Sequence node usually require scaling  , as most of the functions are not plotted nicely with a step of one. An example of a configuration with two axis input (X, Z) and a single output (Y) is depicted in Figure 10.8.

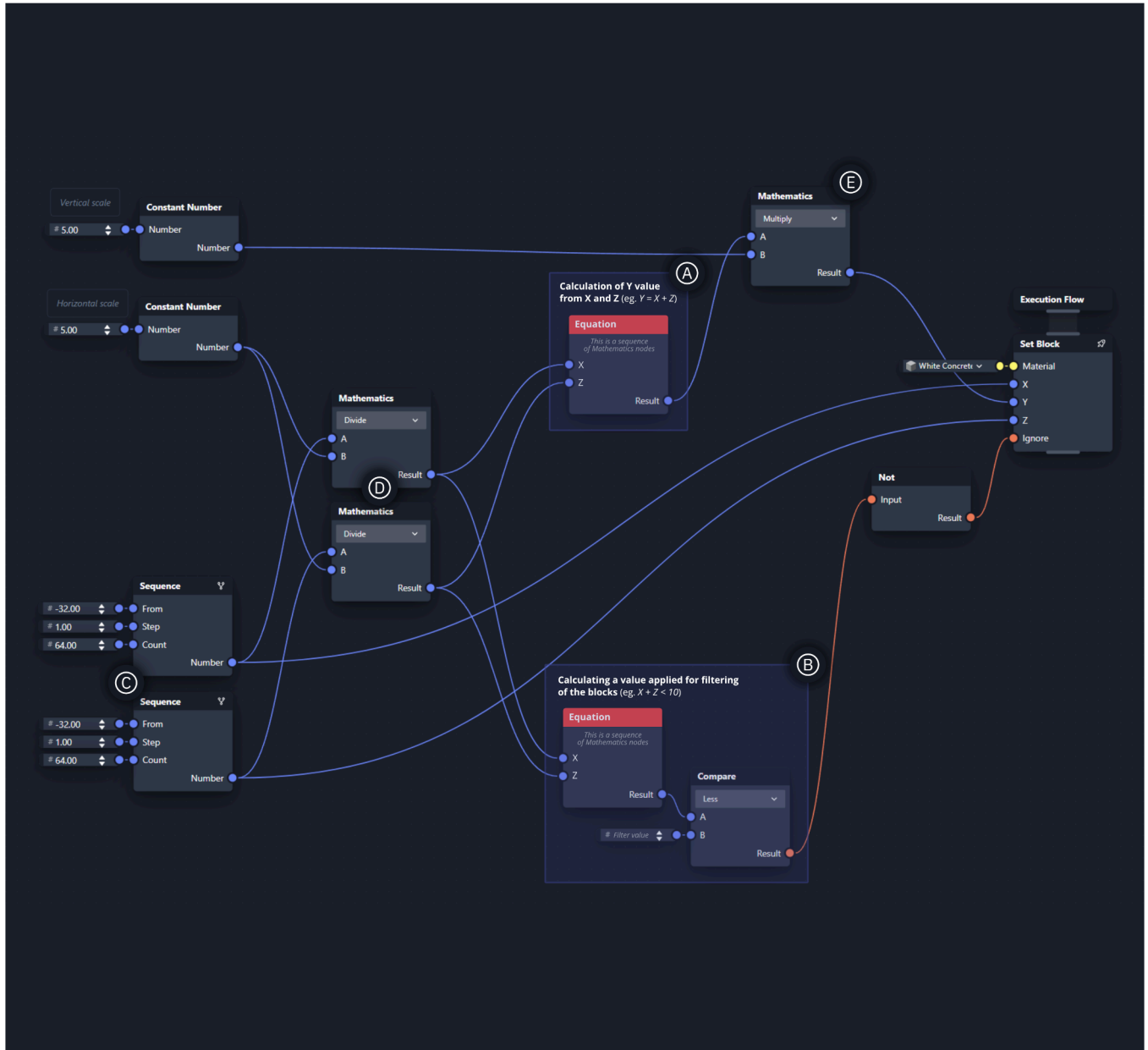


Figure 10.8: Scaffolding for a mathematical and geometrical functions in TerraTinker. The red nodes can be replaced with an arbitrary amount of number-processing nodes.

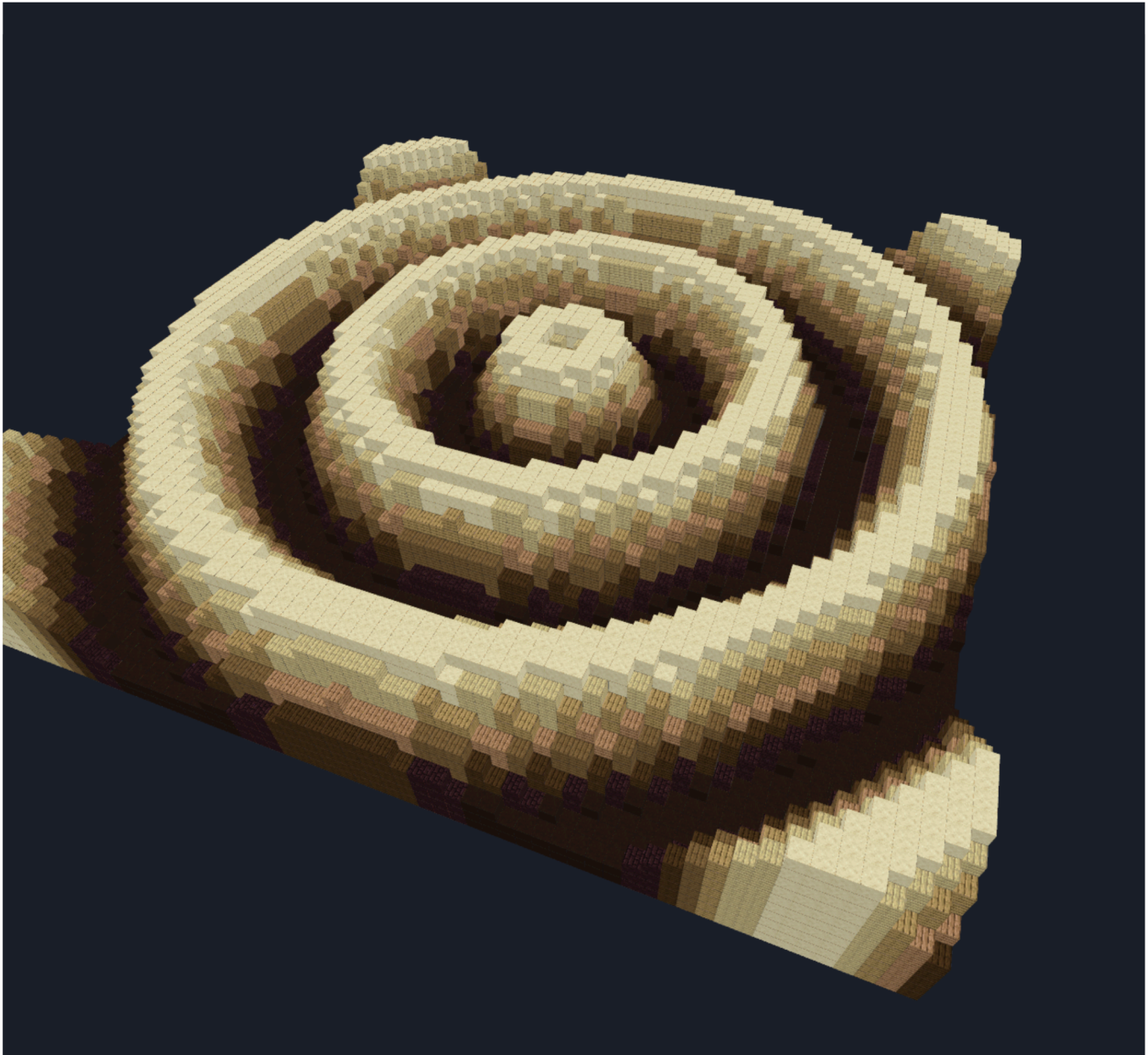


Figure 10.9: The function $y = \sin \sqrt{x^2 + z^2}$ plotted using TerraTinker

10.2.1 Function Plots

TerraTinker can be used to plot 2D and 3D functions. In this example, we can see a plot of a 3D function $y = \sin \sqrt{x^2 + z^2}$. This function creates rippled surface from the origin. The approach in TerraTinker is very simple as for every pair of (x, z) coordinates, the application can calculate the remaining y coordinate and use it to plot a 3D function (Figures 10.9 and 10.10).

Configuration

This configuration without the datasets is available in the source code under `samples/Configurations/math.json`.

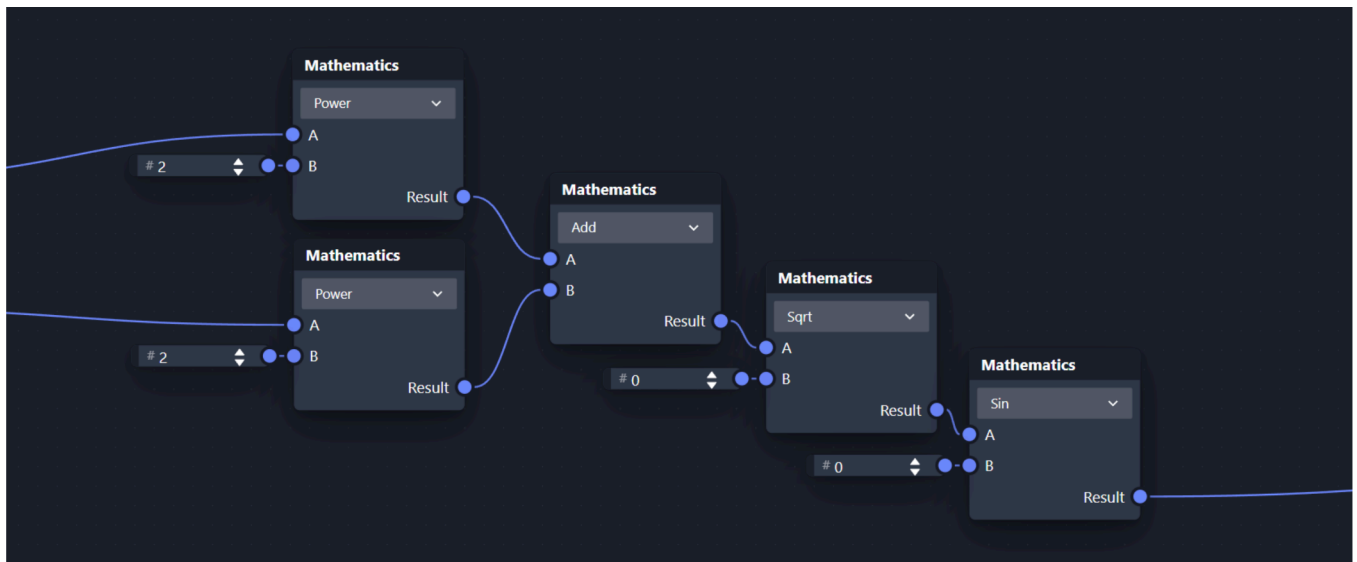


Figure 10.10: The equation defining the plotted function

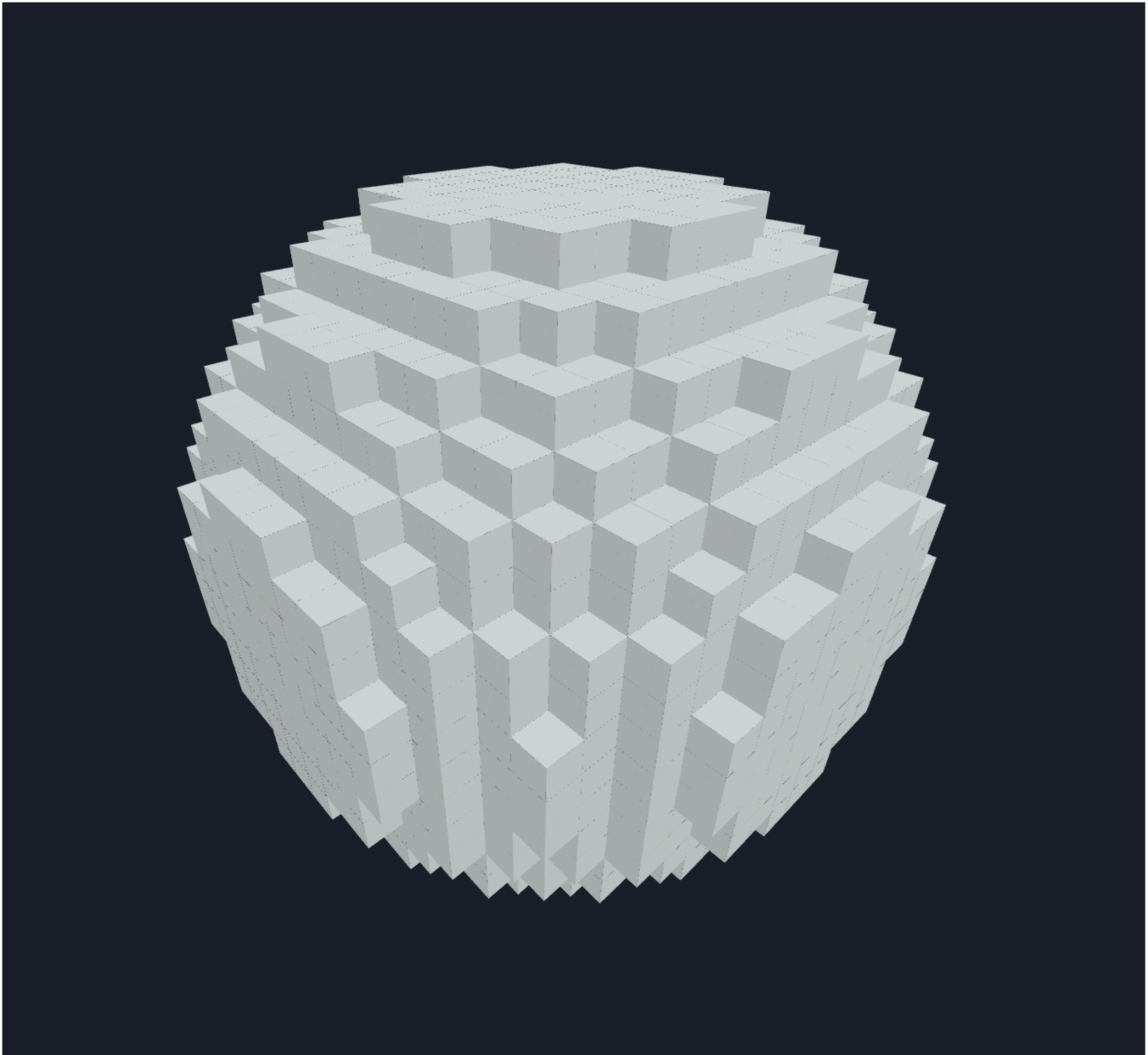


Figure 10.11: A sphere generated by the described layer

10.2.2 Definition of Geometric Bodies

Thanks to the Ignore parameter of the action nodes described in Chapter 9, we can only set blocks from a given region that fit within specified rules and ignore the rest. As an example of such rules, we can use the equation of a sphere $\sqrt{x^2 + y^2 + z^2} \leq r$. By rewriting this equation in TerraTinker layer, we can create mathematically defined spheres, however, still built from cubes (Figures 10.11 and 10.12).

Configuration

This configuration without the datasets is available in the source code under `samples/Configurations/geometry.json`.

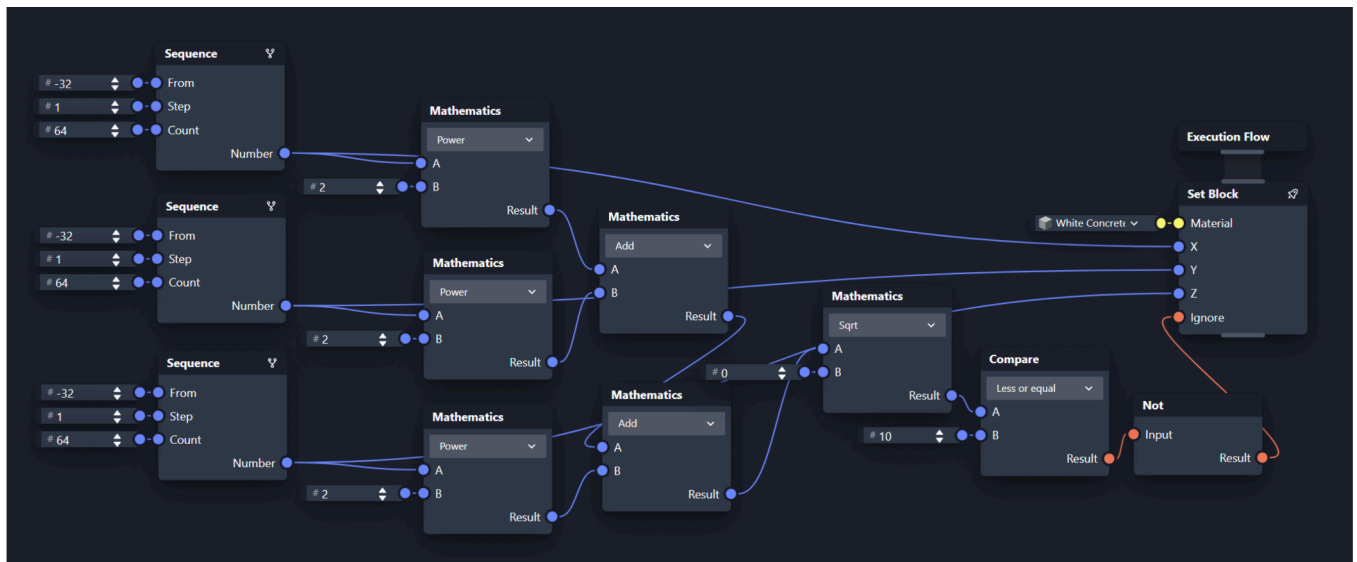


Figure 10.12: The layer mathematically describing the sphere

11 Evaluation

To pinpoint the problematic parts of the thought process during the map generation as well as test the functionality of the implemented features, a small-scale user evaluation was performed at the University College Dublin (UCD) and the Masaryk University Brno (MUNI). In total, the application was evaluated with five individual participants with different backgrounds and education levels, described in the Table 11.1. The study has been anonymized, but all participants agreed with publishing all the details from the study. The audio and screen capture from each session was recorded for future reference.

At the start of the evaluation, the basic concept of the application was explained to the participants (similarly to Section 1.1). They were then walked through the creation steps, each step was briefly described, and participants were pointed to the documentation for further reference. During this phase, the participants were encouraged to manipulate the UI and ask questions.

In the following part, basic concepts were slowly introduced to the participants through a set of simple tasks. The participants worked on a prepared layers and configuration that they were asked to modify according to the tasks. The structure of the evaluation was the following (in the brackets are the concepts that the given steps are supposed to introduce and test):

| Participant ID | Faculty | Current post | Experience with GIS | Experience with node-based editors | Experience with Minecraft |
|-----------------------|---|-------------------------|----------------------------|---|----------------------------------|
| Participant A | Faculty of Informatics, MUNI | Ph.D. student | No | Blender, Scratch | Basic (Player) |
| Participant B | Faculty of Architecture, Planning & Environmental Policy, UCD | Postdoctoral researcher | Yes | FME, Carto, Blender | Advanced (Server management) |
| Participant C | Faculty of Architecture, Planning & Environmental Policy, UCD | Ph.D. student | No | No | None |
| Participant D | Faculty of Architecture, Planning & Environmental Policy, UCD | Postdoctoral researcher | Yes | FME, Blender, Scratch | Advanced (Server management) |
| Participant E | Faculty of Informatics, MUNI | Ph.D. student | No | Unity Shader Graph, Scratch, ... | Expert (Plugin development) |

Table 11.1: A list of participants in the user evaluation and their background

Model 1 - Roads

Starting with a simple configuration that loads roads from OSM and draws them at $Y = 0$.

- Select different area (*region selection, understanding map*).
- Change the scale so the preview fits more roads (*region selection*).
- Change the block the roads are built from (*parameter modification, understanding the graph*).
- Change the thickness of the roads (*parameter modification*).

Model 2 - Terrain

Starting with a simple configuration that renders a single layer of blocks on a terrain loaded from a GeoTIFF.

- Change the scale so that the variations in terrain are more prominent (*region selection*).
- Fill the rest of the map under the terrain with dirt (*execution flow*).
- Modify the “Roads” layer so the buildings are placed on the terrain (*adding multiple nodes, complex graph*).

Model 3 - Buildings

Depending on the time, add buildings on your own, using the provided steps or use a template.

- a. Load terrain from GeoTIFF and place buildings on top of it.
- b. Buildings should be loaded from OpenStreetMaps using “building” key.
- c. Buildings should be drawn as boxes with height 5 relative to the lowest point of the terrain.

During the evaluation, the functionality of the system was validated. The system is capable of generating Minecraft maps from user-provided datasets as well as with features from the OpenStreetMap. Users did not discover any major flaws with the design nor the implementation and they were seriously interested in the future of the project. Members of the Craft-my-Street project were part of the testing and they were very satisfied with the resulting application.

The additional goal of the evaluation was to pinpoint the problematic concepts with the selected design process. No major flaws were discovered, however, it could be observed that the following four concepts took more time to grasp than other tasks:

- The concept of the rasterization — users must get used to looking at 2D raster and vector shapes in the 3D world.
- The vertical scale — this concept was not used in the first model.
- Adding the first ever node on their own — this is to be expected with any node-based programming tool.
- The technicalities of Minecraft, especially the fact that the lower building limit of Minecraft is not at $Y=0$ — this was more a thing users did not realize.

None of the abovementioned concepts are very problematic and all of them can be explained by extending specific documentation sections. After completing all the previous tasks, every participant managed to complete Model 3 without any assistance by using one of the possible approaches. The evaluation validated the functionality of TerraTinker and provided valuable feedback for future development.

12 Future Work

The next step in the near future is to extend the library of available nodes. This can be achieved in two ways — expanding the core library and allowing creators to create custom nodes. The **core library of nodes can be expanded** to provide more transformations and to accept more types of input files, such as CSV and JSON. The tool could be simply adapted to even accept input files that do not include geospatial data at all.

The second way is to **allow creators to define their own custom nodes**. Such nodes would group multiple operations together providing advanced functionality wrapped inside a single node. The grouping would not provide new functionality, but rather make advanced transformations more approachable for less technically proficient users. The created custom nodes could be shared between creators virtually expanding the library of available templates.

As intended by the Craft-my-Street project, TerraTinker should be a part of a larger cloud solution in the future. The resulting application should allow the creators to create virtual education scenarios based on the real-world data. TerraTinker currently represents the first step of the creation process. At the moment, however, the application does not include any **user authentication, project management, or data retention** across multiple devices. The resulting platform for the Craft-my-Street project must include those features to be deployable for the general public.

As mentioned in Chapter 7, the evaluation is currently handled by a single back-end server. To increase the performance, **additional servers** can be deployed. Such servers would have to be connected by a Manager that would handle the load balancing.

13 Conclusion

This thesis aimed to create a web-based tool for transforming geospatial data into playable Minecraft worlds. The goal was to create a universal tool that provides freedom in both the dataset selection and the final visualization.

In the beginning, a set of meetings with the Craft-my-Street project management was organized. The meetings provided needed insight into the problem and offered a platform for the alignment of ideas and understanding of the target groups. As this project heavily focuses on working with the geospatial data, further research was conducted in this field including the analysis of the dataset types and the geographic coordinate systems.

The usage of Minecraft for the project was not compulsory, so a set of criteria was established and seven alternative video games were evaluated. After detailed analysis of the selected video games, Minecraft was confirmed to best fit the needs of the project.

Not to reinvent the wheel, the analysis of the existing tools was performed, yielding FME by SafeSoftware as by far the most comparative tool. Even though FME does not fulfill all the needs of the Craft-my-Street project, it provided valuable insight for defining formal requirements and designing TerraTinker.

The TerraTinker application consists of separate front-end and back-end parts. The back-end is built as a Minecraft server plugin with a Rest API. The front-end provides a wizard-like user experience leading the user through the process of designing their map. The users make use

of node-based design tool for defining the data transformations. Furthermore, the front-end allows users to preview the generated map using a created library. Both parts of the application can be easily deployed using Docker.

To test the TerraTinker application, the evaluation with five users was performed. During the evaluation, the functionality of the system was validated and the evaluation provided valuable feedback for future development.

TerraTinker is capable of generating playful visualizations of geospatial data in the form of Minecraft maps. The application fulfills all the requirements setup for the project and the Craft-my-Street project members are very satisfied with the resulting application. The quality-of-life features, such as the preview, make the application pleasant to use. The inclusion of documentation helps users getting acquainted with the design process.

TerraTinker has been successfully used in real-world applications, as demonstrated by the Craft-my-Street project at the University College Dublin. The tool was used to generate a virtual version of the Dublin Docklands area, showcasing the potential impact of rising water levels.

The application of TerraTinker extends beyond just geospatial data visualization. It can also be utilized in other educational fields, such as plotting functions and visualizing mathematical definitions of geometric bodies. Despite these applications not being the primary focus during the tool's development, they offer great potential and are definitely worth exploring in the future.

Electronic Attachments

The archive in the Masaryk University Information System attached to this thesis contains:

- The **source code** of the application attached as a ZIP file
- Two **Docker container images** of the front-end and the back-end
- A **Docker compose file** for launching the application automatically with the docker images uploaded to the GitHub repository of TerraTinker
- The **Minecraft world saves of the examples** described in Chapter 10

Running the Application

The application can be run using multiple approaches. While the repository with the images is available at GitHub, the suggested way of launching the application is using the provided Docker compose file. If the repository is unavailable, the source code contains a development version of a Docker compose file intended for local build.

Using Prebuilt Images (recomended)

This project uses Docker and Docker Compose to run the application. You need to install and enable Docker from the official website. Download the provided `docker-compose.yml` file from the thesis archive.

Before running the application, you need to agree to Minecraft EULA [23]. To do so you need to create a file `.env` in the root of the project with the following content:

```
EULA=true
```

To start the application, run the following command:

```
docker compose up
```

Using the Source Code

Similarly to the previous option, you need to instal Docker and agree to Minecraft EULA.

To build and run the application using the provided source code, run the following command:

```
docker compose -f docker-compose.dev.yml up --build
```

Developing Individual Applications

The guide for running the front-end and the back-end separately is in the *readme* of the individual applications in the source code archive.

Bibliography

1. WARD, Matthew O., GRINSTEIN, Georges and KEIM, Daniel. *Interactive data visualization: Foundations, techniques, and applications*. 2nd ed. CRC PRESS, 2021. ISBN 9781482257373.
2. CAIRO, Alberto. *The Truthful Art: Data, Charts, and Maps for Communication*. New Riders, 2016. ISBN 9780321934079.
3. BERNHARDSEN, Tor. *Geographic Information Systems: An Introduction*. 3rd ed. Wiley, 2002. ISBN 9780471419686.
4. KIRK, Andy. *Data Visualisation: A Handbook for Data Driven Design*. 2nd ed. SAGE Publications Ltd, 2019. ISBN 9781526468925.
5. CORTI, Kevin. Games-based Learning; a serious business application. *Informe de PixelLearning*. 2006. Vol. 34, no. 6, p. 1–20.
6. PRENSKY, Marc. Digital natives, digital immigrants. *On the horizon*. 2001. Vol. 9, no. 5, p. 1–6.
7. CARDOSO, Bruno, COHN, Neil, TRUYEN, Frederik and BROSENS, Koenraad. Explore Data, Enjoy Yourself - KUbism, A Playful Approach to Data Exploration. In: 2021. p. 43–64. ISBN 9783030856120.

8. CANOSSA, Alessandro, MARTINEZ, Josep B. and TOGELIUS, Julian. Give me a reason to dig Minecraft and psychology of motivation. In: *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. 2013. p. 1–8. DOI 10.1109/CIG.2013.6633612.
9. LEUE, Andre. *Verification of Factorio belt balancers using Petri nets*. Bachelor thesis. 2021. Available from: <https://d-nb.info/123465766X/34>
10. WISNIEWSKI, Maciej. *Artificial intelligence for the OpenTTD game*. Master thesis. 2011. Available from: <https://www2.imm.dtu.dk/pubdb/edoc/imm6091.pdf>
11. BAGH, Dmitri, DE VOGEL, Brian and LUTZ, Dale. *Esri CityEngine & Minecraft: Engaging Citizens in 3D City Planning*. Online. 2015. Available from: https://proceedings.esri.com/library/userconf/proc15/papers/997_696.pdf. Esri User Conference
12. SENA, Italo, POPLIN, Alenka and ANDRADE, Bruno. *GeoMinasCraft: A Serious Geogame for Geographical Visualization and Exploration*. In: 2021. p. 613–632. ISBN 9783030760588.
13. ŠŤASTNÁ, Tereza. *Playful visualization: Evaluating data visualization in video games*. Online. Brno, 2023. Available from: <https://is.muni.cz/th/td5tv/>
14. World Geodetic System (WGS84). Online. 2024. [Accessed 5 May 2024]. Available from: <https://gisgeography.com/wgs84-world-geodetic-system/>
15. Coordinate systems, map projections, and transformations. Online. 2023. [Accessed 5 May 2024]. Available from: <https://pro.arcgis.com/en/pro-app/3.1/help/mapping/properties/coordinate-systems-and-projections.htm>
16. *Minecraft: Java & Bedrock Edition Deluxe Collection | Compare Key Features of Minecraft Java and Bedrock Editions*. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.minecraft.net/en-us/store/minecraft-deluxe-collection-pc#title-486066bf42>

17. Differences Between Minecraft: Bedrock Edition and Minecraft: Java Edition. Online. 2024. [Accessed 5 May 2024]. Available from: <https://learn.microsoft.com/en-us/minecraft/creator/documents/differencesbetweenbedrockandjava>
18. Anvil file format. Online. 2018–2023. [Accessed 5 May 2024]. Available from: https://minecraft.fandom.com/wiki/Anvil_file_format
19. Overpass API. Online. 2009–2023. [Accessed 5 May 2024]. Available from: https://wiki.openstreetmap.org/wiki/Overpass_API
20. Tutorials/Programs and Editors/Mapping | Map Editors. Online. 2012–2024. [Accessed 5 May 2024]. Available from: https://minecraft.fandom.com/wiki/Tutorials/Programs_and_editors/Mapping#Map_editors
21. GDAL: Java bindings. Online. 1998–2024. [Accessed 5 May 2024]. Available from: <https://gdal.org/api/java/index.html>
22. GDAL: GDAL/OGR in Other languages. Online. 1998–2024. [Accessed 5 May 2024]. Available from: <https://gdal.org/api/index.html#gdal-ogr-in-other-languages>
23. Minecraft End-User License Agreement. Online. 2023. [Accessed 5 May 2024]. Available from: <https://www.minecraft.net/en-us/eula>
24. Paper. Online. 2024. [Accessed 5 May 2024]. Available from: <https://bstats.org/plugin/server-implementation/Paper/580>
25. Number of people living at the addresses. Online. 6 February 2024. [Accessed 5 May 2024]. Available from: https://data.brno.cz/datasets/923cbe09f11c4999bedeb0bc20905964_0/explore
26. Noise level 2022. Online. 8 May 2024. [Accessed 5 May 2024]. Available from: https://data.brno.cz/datasets/a5ad8e597ff64bfb8a0d794e78c10512_0/explore

27. YAMAZAKI, Dai, IKESHIMA, Daiki, TAWATARI, Ryunosuke, YAMAGUCHI, Tomohiro, O'LOUGHLIN, Fiachra, NEAL, Jeffery C., SAMPSON, Christopher C., KANAE, Shinjiro and BATES, Paul D. A high-accuracy map of global terrain elevations. *Geophysical research letters*. 2017. Vol. 44, no. 11, p. 5844–5853. DOI 10.1002/2017GL072874.
28. Craft my Street. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.craftmystreet.com/>
29. GOG:Townscaper. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.gog.com/en/game/townscaper>
30. ArcGIS. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.arcgis.com/>
31. QGIS. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.qgis.org/en/site/>
32. Carto. Online. 2024. [Accessed 5 May 2024]. Available from: <https://carto.com/>
33. FME by SafeSoftware. Online. 2024. [Accessed 5 May 2024]. Available from: <https://fme.safe.com/>
34. Tableau. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.tableau.com/>
35. Power BI. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.microsoft.com/en-us/power-platform/products/power-bi>
36. Minecraft. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.minecraft.net/en-us>
37. Factorio. Online. 2015–2024. [Accessed 5 May 2024]. Available from: <https://factorio.com/>

38. OpenTTD. Online. 2005–2024. [Accessed 5 May 2024]. Available from: <https://www.openttd.org/>
39. OpenStreetMap. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.openstreetmap.org/>
40. GOG: Heroes of Might and Magic® 3: Complete. Online. 2024. [Accessed 5 May 2024]. Available from: https://www.gog.com/game/heroes_of_might_and_magic_3_complete_edition
41. Warcraft III: Reforged. Online. 2024. [Accessed 5 May 2024]. Available from: <https://warcraft3.blizzard.com/en-us/>
42. GOG: Dorfromantik. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.gog.com/en/game/dorfromantik>
43. Minecraft Education. Online. 2024. [Accessed 5 May 2024]. Available from: <https://education.minecraft.net/en-us>
44. Downloads for Minecraft Forge. Online. 2018–2024. [Accessed 5 May 2024]. Available from: <https://files.minecraftforge.net/net/minecraftforge/forge/>
45. Fabric Loader. Online. 2024. [Accessed 5 May 2024]. Available from: <https://fabricmc.net/>
46. The Quilt Project. Online. 2024. [Accessed 5 May 2024]. Available from: <https://quiltmc.org/en/>
47. Bukkit. Online. 2021. [Accessed 5 May 2024]. Available from: <https://dev.bukkit.org/>
48. PaperMC. Online. 2024. [Accessed 5 May 2024]. Available from: <https://papermc.io/>

49. Purpur. Online. 2019–2024. [Accessed 5 May 2024]. Available from: <https://purpurmc.org/>
50. WorldBoxer. Online. 2018. [Accessed 5 May 2024]. Available from: <https://geoboxers.com/worldbloxer/>
51. ArcGIS CityEngine. Online. 2020–2024. [Accessed 5 May 2024]. Available from: <https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview>
52. ObjToSchematic. Online. 2024. [Accessed 5 May 2024]. Available from: <https://objtoschematic.com/>
53. Online Voxelize. Online. 2023. [Accessed 5 May 2024]. Available from: <https://drububu.com/miscellaneous/voxelize/?out=min>
54. TypeScript. Online. 2012–2024. [Accessed 5 May 2024]. Available from: <https://www.typescriptlang.org/>
55. React. Online. 2024. [Accessed 5 May 2024]. Available from: <https://react.dev/>
56. chakra. Online. 2024. [Accessed 5 May 2024]. Available from: <https://v2.chakra-ui.com/>
57. Leaflet. Online. 2010–2024. [Accessed 5 May 2024]. Available from: <https://leafletjs.com/>
58. React Flow. Online. 2024. [Accessed 5 May 2024]. Available from: <https://reactflow.dev/>
59. react-minecraft-viewer. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.npmjs.com/package/react-minecraft-viewer>
60. Gradle Shadow Plugin. Online. 2024. [Accessed 5 May 2024]. Available from: <https://imperceptiblethoughts.com/shadow/>
61. javalin. Online. 2024. [Accessed 5 May 2024]. Available from: <https://javalin.io/>

62. GDAL. Online. 1998–2024. [Accessed 5 May 2024]. Available from: <https://gdal.org/index.html>
63. PROJ. Online. 1983–2024. [Accessed 5 May 2024]. Available from: <https://proj.org/en/9.4/>
64. docker. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.docker.com/>
65. deepslate. Online. 2024. [Accessed 5 May 2024]. Available from: <https://github.com/misode/deepslate>
66. vscode-nbt. Online. 2024. [Accessed 5 May 2024]. Available from: <https://github.com/misode/vscode-nbt/tree/master>
67. NPM. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.npmjs.com/>
68. Stardew Valley. Online. 2016–2024. [Accessed 5 May 2024]. Available from: <https://www.stardewvalley.net/>
69. Street lights. Online. 14 March 2024. [Accessed 5 May 2024]. Available from: https://data.brno.cz/datasets/462d3e4e68984538a29e715a885d770e_0/explore
70. Public transport stops. Online. 1 September 2020. [Accessed 5 May 2024]. Available from: <https://data.brno.cz/maps/747a824783044377b6d07a8060e7769d>
71. Astroneer. Online. 2024. [Accessed 5 May 2024]. Available from: <https://astroneer.space/>
72. Roblox. Online. 2024. [Accessed 5 May 2024]. Available from: <https://www.roblox.com/>
73. Games Released in Previous Months. 2024. [Accessed 5 May 2024]. Available from: <https://steamspy.com/year/>



is.muni.cz/th/ru3xf