UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 283

RURAL AREA MODEL BASED ON REAL DATA

Borna Cafuk

Zagreb, June 2021

UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 283

RURAL AREA MODEL BASED ON REAL DATA

Borna Cafuk

Zagreb, June 2021

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 12 March 2021

BACHELOR THESIS ASSIGNMENT No. 283

Student:	**Borna Cafuk (0036513396)**
Study:	Electrical Engineering and Information Technology and Computing
Module:	Computing
Mentor:	prof. Željka Mihajlović

Title:	**Rural Area Model Based on Real Data**

Description:

Investigate available public sources of spatial cartographic information related to the development of terrain models. Develop possibilities for using this information in creating a three-dimensional model of an arbitrary part of a defined cartographic area. Implement the visual representations of the terrain model based on real data. Show examples of results. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms. Implement the appropriate software solution. Use technologies: WebGL and three.js. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Submission date: 11 June 2021

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 283

Pristupnik:	**Borna Cafuk (0036513396)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Željka Mihajlović

Zadatak:	**Model ruralnog područja temeljen na realnim podacima**

Opis zadatka:

Proučiti izvore javno dostupnih prostornih kartografskih informacija vezanih uz izradu modela terena. Razraditi mogućnosti korištenja tih informacija u izradi trodimenzionalnog modela proizvoljnog dijela definiranog kartografskog područja. Ostvariti prikaze modela terena temeljenog na realnim podacima. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti grafičku programsku knjižnicu Three.js za prikaz rezultata. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 11. lipnja 2021.

Contents

INTRODUCTION

Computer graphics are a powerful tool for depicting locations, both fictional and real. The former have to be modelled by artists or generated procedurally from a mathematical model. While depictions of the real world can also be created by hand, the possibility of generating them from collected cartographic data exists as well. Utilising real data to render depictions of terrain allows for greater realism where desired and lets artists spend their time on tasks which require creativity and can thus not yet be automated.

Such realistic depictions of locations can then be used in films and video games set in the real world, as well as in various visualisations. One recent example of a video game relying heavily on real data is the 2020 release of Microsoft Flight Simulator. It uses the same data as Bing Maps to faithfully represent buildings and terrain on Earth.[6] Applications of such technology encompass not only flight simulators intended for civilian use, but also military simulations and data visualisation tools for fields like civil engineering.

Creating a model of a real-world rural area has applications in the aforementioned flight simulators, products like Google Maps and Bing Maps, and films with computer-generated imagery, as well as potentially in boundary disputes, agriculture, logging, and drainage modelling. Additionally, a model created from previously collected data from an area can be used to plan further surveying of the area.

1. Tools and technologies used

1.1. The TypeScript programming language

TypeScript[1] is a programming language developed by Microsoft and licensed under the Apache License 2.0. It builds upon JavaScript by allowing the programmer to define data types for variables and have them statically checked. TypeScript compiles to regular JavaScript and can thus be run in the browser. Type checking is done only at compile time, so no extra code for type checking is emitted.

TypeScript code resembles JavaScript, but has added type annotations. This helps mitigate type errors, which in JavaScript only become apparent at run time because the language is dynamically- and weakly-typed. Instead, when using TypeScript, these errors are most likely to be detected early. The type annotations also aid in documenting the source code by explicitly defining the types of variables, function arguments, object properties, etc. The availability of visibility modifiers for class fields and methods allows for easier encapsulation.

Looking at listings 1.1 and 1.2, one can see that the TypeScript compiler has kept the program structure intact and that it has preserved the comment from the original source code. The type annotations and visibility modifiers have been stripped from the code, as has the property declaration for `name`.

Because all type checking is done statically and no type-checking code is present in the resulting JavaScript, the type checking is not bulletproof. Runtime type errors are possible in a variety of situations. The programmer should be aware of this, even if such errors are not a common occurrence when using TypeScript.

[1]`https://www.typescriptlang.org/`

```typescript
class Student {
  private readonly name: string;

  public constructor(name: string) {
    this.name = name;
  }

  public get greeting(): string {
    return "Hello, " + this.name + "!";
  }
}

function printGreeting(student: Student) {
  // Greet the student on the console
  console.log(student.greeting);
}

const susan = new Student("Susan");
printGreeting(susan);
```

Listing 1.1: An example TypeScript program.

```javascript
class Student {
    constructor(name) {
        this.name = name;
    }
    get greeting() {
        return "Hello, " + this.name + "!";
    }
}
function printGreeting(student) {
    // Greet the student on the console
    console.log(student.greeting);
}
const susan = new Student("Susan");
printGreeting(susan);
```

Listing 1.2: The example program from listing 1.1 after compilation to JavaScript.

1.2. The WebGL API

WebGL[2] is a graphics JavaScript API developed by the Khronos WebGL Working Group and supported in all major browsers.

The WebGL specification "describes an additional rendering context and support objects for the HTML 5 canvas element. This context allows rendering using an API that conforms closely to the OpenGL ES 2.0 API."[5, Abstract] As such, WebGL allows for displaying three-dimensional graphics in the browser using a programming interface that is a subset of modern desktop OpenGL. Vertex attribute data is held in buffer objects (VBOs)[5, Section 5.14.5] and the pipeline is programmable. Vertex and fragment shaders are written in GLSL ES.[5, Section 5.14.9]

1.3. The Three.js library

Three.js[3] is a JavaScript library for three-dimensional graphics. It uses WebGL internally to render the graphics in the browser, but provides a higher-level interface to the programmer. WebGL is a low-level API and Three.js abstracts away many of its intricacies. As a result, most users of Three.js do not need to write their own shaders or make calls to WebGL functions, using the objects Three.js provides instead.

Three.js also contains implementations of numerous features useful for computer graphics. These include a scene graph, shading, shadow maps, an animation system, mouse and keyboard input, procedural noise, and loading assets like textures and models, just to name a few.

Type definitions[4] for Three.js are available, so TypeScript code using the library can be type checked. Both Three.js and the type definitions are available under the MIT license.

1.4. The Webpack module bundler

Webpack[5] is a module bundler used to bundle and package assets for the web, primarily JavaScript files. Projects are easier to maintain if they are logically divided into separate files instead of having all the code in one single file. However, requesting

[2]`https://www.khronos.org/webgl/`

[3]`https://threejs.org/`

[4]`https://www.npmjs.com/package/@types/three`

[5]`https://webpack.js.org/`

many files at once via HTTP slows the web page down by introducing overhead on both the client and the server. It is thus useful to have a tool bundle the separate JavaScript source files into one file for distribution. This is the primary motivation for using bundlers like Webpack.

Additionally, bundlers make it easy to use libraries, as they bundle the library code with the user code so it can be served to the client together, instead of having to be separately downloaded from a content delivery network (CDN). The code of such libraries ("vendor code") can be bundled into a separate JavaScript file (the "vendor bundle"). The contents of the user code bundle change every time the application is updated, but the vendor code changes much less often, only when a dependency is updated. This improves cache performance, as it allows for the vendor bundle to be cached separately in the user's browser and re-downloaded rarely.

The TypeScript compiler when used by itself translates each source file into its own JavaScript file. Webpack can be used in conjunction with TypeScript using the ts-loader[6] package. This avoids having to first compile all TypeScript source code into JavaScript files on disk before bundling them. Webpack and ts-loader are available under the MIT license.

1.5. The Blender suite

Blender[7] is a software suite used to produce three-dimensional computer graphics. Perhaps its most famous use is that of 3D modelling, though its features extend far beyond that. Some of its other applications include ray-traced rendering, rigging of models, animation using forward and inverse kinematics, motion tracking, and simulation of smoke, fire, fluids, cloth, hair, etc.[7] Blender is licensed as GNU GPL Version 2 or later.

[6]https://www.npmjs.com/package/ts-loader
[7]https://www.blender.org/

2. Rendering a scene in Three.js

2.1. Setting up

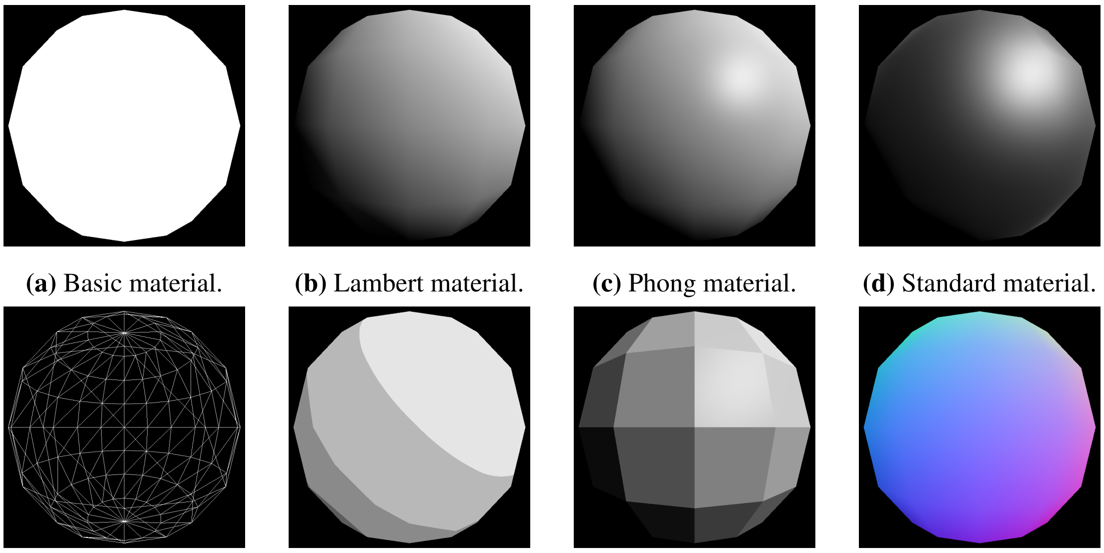
In order to begin rendering in Three.js, three objects are required: a renderer, a camera, and a scene. The renderer contains the browser's rendering context, as well as information about the clipping planes, shadow map, scissor test, etc. The camera contains its own position, orientation, and the projection to be used when rendering, among other information. The scene contains all objects to be rendered, all lights, and describes the fog and background to use when drawing.

Three.js's `WebGLRenderer` requires an HTML `<canvas>` element, for which it then creates a WebGL 2 rendering context. After the renderer is created, its `render` method can be used to draw a scene to the canvas from the viewpoint of a camera. The scene and the camera are passed to the method as parameters. This method needs to be called every frame in order for changes to be immediately visible to the user. This is possible using the standard JavaScript function `requestAnimationFrame`.

When the canvas is resized, the renderer and the camera need to be updated with the new size. This is required because the camera needs to update its projection matrix, and because the renderer needs to call the `viewport` method of the WebGL rendering context in order to have WebGL draw the scene to the entirety of the canvas.

2.2. Adding objects

Objects can be added to the initially blank scene using the scene's `add` method. A kind of object which can be added are meshes, which "[represent] triangular polygon mesh based objects." When using meshes, the object's structure is defined by an instance of the `BufferGeometry` class, and its appearance is defined by one or more instances of the `Material` class.[4]



(a) Basic material. **(b)** Lambert material. **(c)** Phong material. **(d)** Standard material.

(e) Wireframe. **(f)** Toon material. **(g)** Flat shading. **(h)** Normal material.

Figure 2.1: A white, low-poly sphere shaded with different materials and settings.

2.2.1. Mesh geometry

The geometry of a mesh describes the triangles that make it up. Though typically loaded from a model file, it can also be generated in code. The geometry consists of buffers which closely correspond to the underlying WebGL buffer objects. Each buffer is assigned to an attribute, corresponding to attributes in the GLSL shaders. Some of the attributes used by default materials are:

`position` The position of the vertex in model space.
`normal` The normal vector associated with a vertex.
`uv` The texture coordinates of the vertex.
`color` The vertex colour.

Indexed rendering can be used by providing vertex indices.

2.2.2. Mesh materials

Three.js offers several materials by default and allows the user to define their own using shaders. Some of the materials for meshes available out-of-the-box are:

- A solid-coloured material without any shading. (figures 2.1a and 2.1e)
- Gouraud (per-vertex) shading with Lambertian (diffuse-only) reflectance. (figure 2.1b)

  – Phong (per-fragment) shading with Phong (diffuse and specular) reflectance. (figure 2.1c) This material can also be used for flat shading. (figure 2.1g)
  – Physically-based materials: without (figure 2.1d) and with a clear coat.
  – A material for cartoon-like cel shading. (figure 2.1f)
  – Several specialised materials for rendering data (e.g. normals: figure 2.1h) to textures.

Meshes can be rendered as wireframes, as seen in figure 2.1e.

2.3. Instanced rendering

Three.js supports instanced rendering, which can be used when one mesh needs to be drawn many times, each time differing only in its transformation matrix, and optionally its colour. In the example shown in figure 2.2, each monkey head has a distinct transformation matrix and colour.

Instanced meshes, like regular meshes, require geometry and one or more materials. However, they also require as an argument the maximum number of instances they will hold. After having been created, the number of actual instances can be set, as can be the matrices and colours for the instances.

Without instanced rendering, one would have make a rendering call for each instance. Additionally, a new matrix and colour uniforms would have to be passed to the shader before rendering each instance. When using instanced rendering, one call is used to draw all instances, with the matrices and colours passed to the shader all at once as an array.

Instanced rendering improves performance, but when using it, some features are not available. For example, shadow maps are not supported by the default instanced rendering shaders provided by Three.js. Meshes with multiple levels of details (LODs) are also not supported.

2.4. Loading assets

Three.js offers loader classes for many formats used in three-dimensional graphics and game development. These loaders are used to easily retrieve assets from a given URL. Notable loaders are:

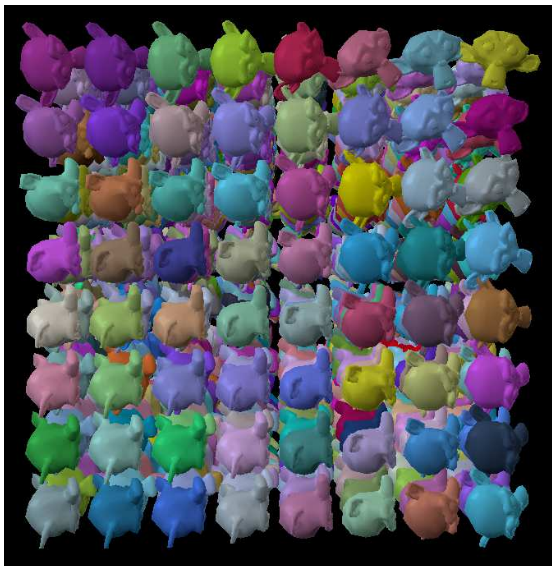


Figure 2.2: A Three.js example demonstrating instanced rendering.

`ImageLoader` loads an image file and provides it as an `HTMLImageElement`. This loader is used internally by other loaders, but can also be used on its own where necessary.

`TextureLoader` loads an image file and makes it available as an instance of the `Texture` class. Instances of this class can be used in Three.js as texture maps for materials, e.g. as diffuse maps.

`MTLLoader` loads material descriptions from an MTL file and provides an instance of the `MaterialCreator` class, which can be later used when loading an OBJ file.

`OBJLoader` loads model geometry from an OBJ file and creates an `Object3D`. This object can then be added to a scene.

3. The developed application

The web application developed as part of this thesis is codenamed RAM-BReaD, which is an acronym of *Rural Area Model Based on Real Data*. It allows the user to specify the coordinates of point on Earth's surface. A three-dimensional model of the surrounding area will then be generated based on aerial imagery. This model contains an approximation of the plants present in the area. The user can move the camera around the scene in real time using the mouse and keyboard.

This application was written in TypeScript, uses Three.js for graphics, and uses Webpack for bundling.

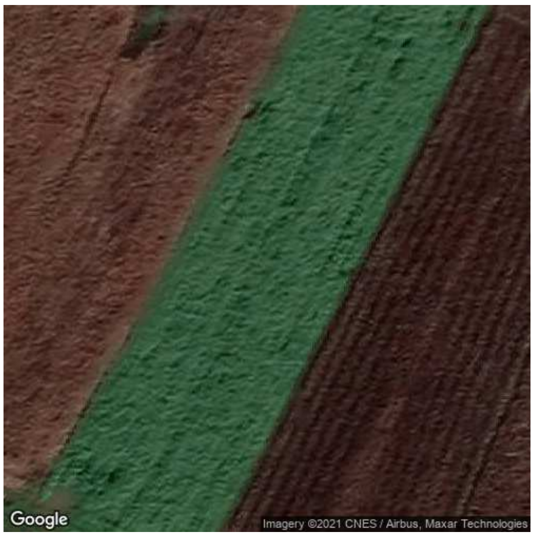
3.1. Running the application

To build the application, Node.js and NPM are required. The Google Maps API key to be used for getting the aerial imagery should be specified as per the instructions in the file `src/apiKeys.ts_TEMPLATE`.

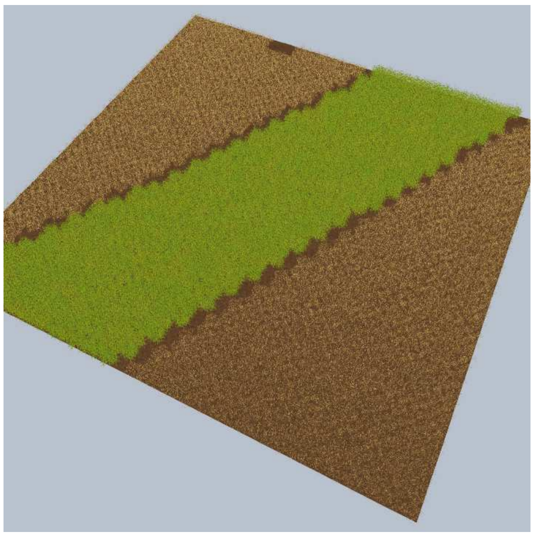
In the root of the repository, the command `npm install` should be run in order to download and install dependencies. The application can then be built using the command `npm run all`. This will create the `build` directory and build the application in it.

Finally, to run the application, an HTTP server should be run from the `build` directory. A simple way to accomplish this is using the http-server package from NPM by running the command `npx http-server`.

The location to show in the application is defined using the `lat` and `long` query parameters in the URL. These values represent the latitude and longitude of the centre of the area, respectively. Both values are in decimal degrees. For example, if the application is hosted at `http://127.0.0.1/`, then the location at 45°52′12.25″N, 16°52′57.91″E can be viewed by accessing the URL `http://127.0.0.1/?lat=45.870070&long=16.882753`.



(a) Aerial imagery retrieved from the Static Maps API.



(b) The location from figure 3.1a rendered in the application.

Figure 3.1: Aerial imagery and generated model of fields.

3.2. Data fetching

Numerous providers of cartographic data exist. The OpenStreetMap Overpass API provides read-only access to data from OpenStreetMap. This data includes buildings, marked places, public transit stops, streets and roads, etc. Open Data on AWS (Amazon Web Services) contains datasets available to anyone, including geospatial ones. This includes imagery from multiple satellites, air quality data, elevation, and more.

Google Maps Platform provides access to much of Google Maps and Google Street View functionality via APIs. It includes APIs like static and dynamic maps, marked places, elevation, directions, Street View panoramas, etc. Although use of the API is not free, users receive $200 in credit every month[1], which has been more than enough for the requirements of this thesis.

The data used to construct the rural landscape model is retrieved from the Google Maps Platform. The Maps Static API is used to obtain aerial or satellite imagery of the area for which the model is being generated. For the square 100 m × 100 m area, a 500 pixel × 500 pixel image is fetched. This allows for a resolution of up to 5 pixels per metre. However, the actual resolution of the imagery available is several times lower in many areas.

The Static Maps API is available at the URL `https://maps.googleapis.com/maps/api/staticmap` using the GET method. The application uses the following query parameters when fetching the imagery:

`center` Coordinates of the centre of the depicted area.

`zoom` The zoom level of the map.

`maptype` The type of the map, the value `satellite` is used to get satellite or aerial imagery.

`size` The size of the resulting image.

`format` The file format of the resulting image.

`key` The Maps API key of the application.

3.3. Image sampling

The application takes a sample every 2.5 metres. Sampling is done by computing a weighted average of the colour in a square around the sampled point. Every sample is taken from a 7.5 m × 7.5 m area centred on the centre of the sample. The value of each pixel is weighted according to its distance from the centre point of the sample. The weight $w(P)$ of the pixel at the point P is calculated as follows:

$$w(P) = \frac{1}{1 + d_{PC}}$$

d_{PC} is the distance in metres between the point P and the centre of the sample, C.

This kind of weighting function ensures that pixels which are further away from the centre contribute less to the overall average, while also preventing division by zero when the distance is zero. Each component of the colour is averaged individually. The components are treated as linear, that is, no gamma correction is performed. Listing 3.1 shows this sampling function in its entirety.

The average colour of the sample is then converted into its HSL (hue, saturation, lightness) representation. Then, depending on these components, a plant type is chosen for the current sample.

```
private static readonly METERS_PER_SAMPLE = 2.5;
private static readonly SAMPLE_RADIUS = 3.75;
// ...
private readonly width: number;
private readonly height: number;
// ...
private sampleAverageColor(
  imageData: ImageData,
  x: number,
  z: number
): [number, number, number] {
```

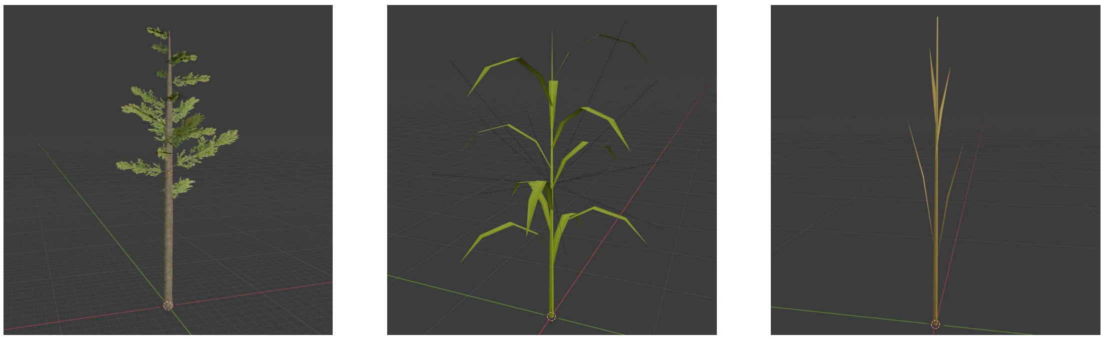
```
const pixelsPerMeterWidth = imageData.width / this.width;
const pixelsPerMeterHeight = imageData.height / this.height;

const pixelXMin =
  (x + this.width / 2 - RuralAreaModel.SAMPLE_RADIUS) *
  pixelsPerMeterWidth;
const pixelYMin =
  (z + this.height / 2 - RuralAreaModel.SAMPLE_RADIUS) *
  pixelsPerMeterHeight;

const pixelXMax =
  pixelXMin +
  2 * RuralAreaModel.SAMPLE_RADIUS * pixelsPerMeterWidth;
const pixelYMax =
  pixelYMin +
  2 * RuralAreaModel.SAMPLE_RADIUS * pixelsPerMeterHeight;

const average: [number, number, number] = [0, 0, 0];
let scalingFactor = 0;

for (
  let pixelY = Math.max(pixelYMin, 0);
  pixelY < Math.min(pixelYMax, imageData.height - 1);
  pixelY++
) {
  for (
    let pixelX = Math.max(pixelXMin, 0);
    pixelX < Math.min(pixelXMax, imageData.width - 1);
    pixelX++
  ) {
    const pixel = RuralAreaModel.getPixel(
      imageData,
      pixelX,
      pixelY
    );
    const distance = Math.hypot(
      pixelX / pixelsPerMeterWidth - x,
      pixelY / pixelsPerMeterHeight - z
    );

    const weight = 1 / (1 + distance);
    scalingFactor += weight;
    for (let i = 0; i < 3; i++) average[i] += weight * pixel[i];
    }
  }

  for (let i = 0; i < 3; i++) average[i] /= scalingFactor;
  return average;
}
```

Listing 3.1: The method used to calculate the average colour of a sample.

3.4. Plant models

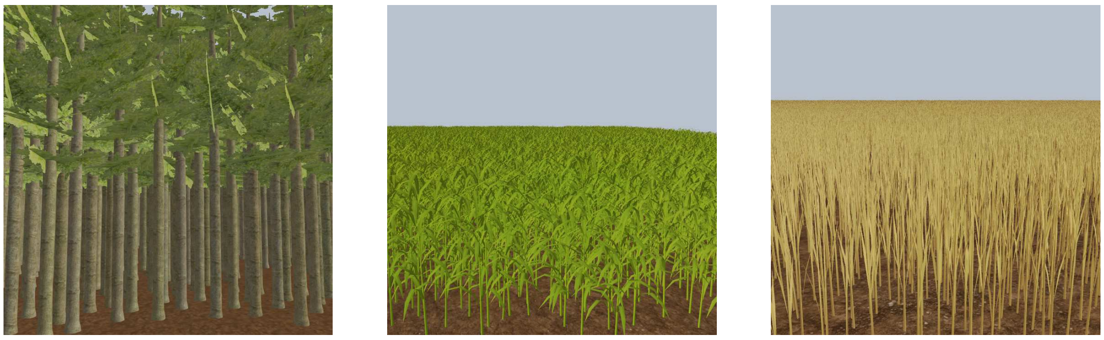
Three plant models were created for the application:

- European beech, *Fagus sylvatica* (figure 3.3a),
- maize, *Zea mays* (figure 3.3b), and
- wheat, mostly based on common wheat, *Triticum aestivum* (figure 3.3c).



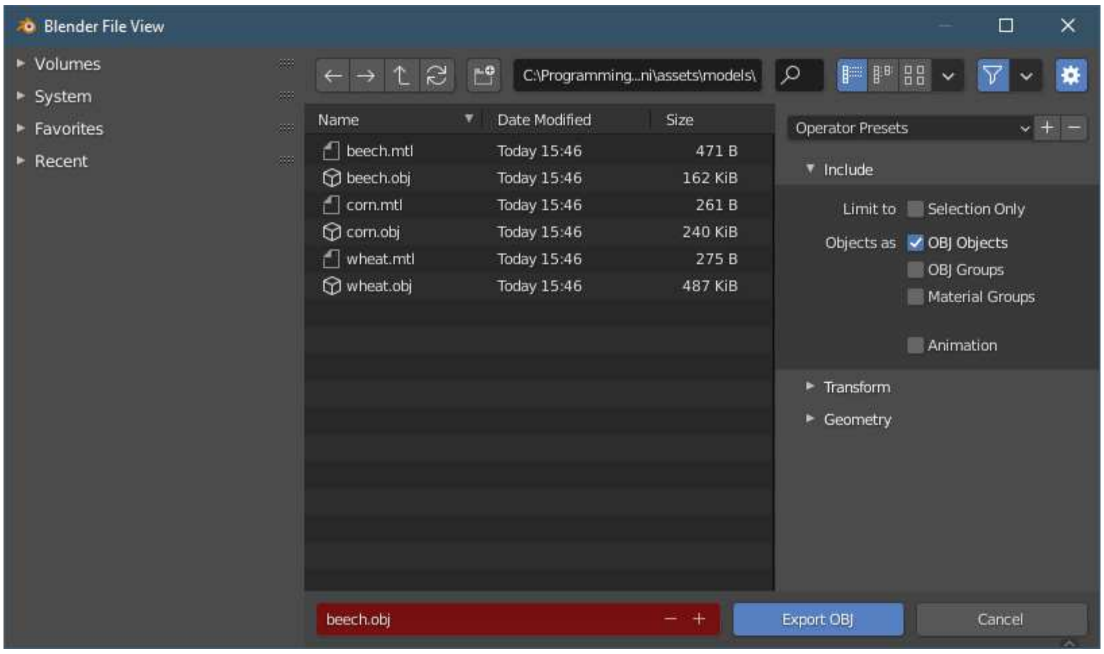
(a) Beech. **(b)** Maize. **(c)** Wheat.

Figure 3.2: Screenshots of individual plant models in Blender.



(a) A beech forest. **(b)** A maize field. **(c)** A wheat field.

Figure 3.3: The different types of plants, rendered in the application.



Name	Date Modified	Size
beech.mtl	Today 15:46	471 B
beech.obj	Today 15:46	162 KiB
corn.mtl	Today 15:46	261 B
corn.obj	Today 15:46	240 KiB
wheat.mtl	Today 15:46	275 B
wheat.obj	Today 15:46	487 KiB

Limit to [ ] Selection Only
Objects as [✓ OBJ Objects] [ ] OBJ Groups [ ] Material Groups
[ ] Animation

beech.obj

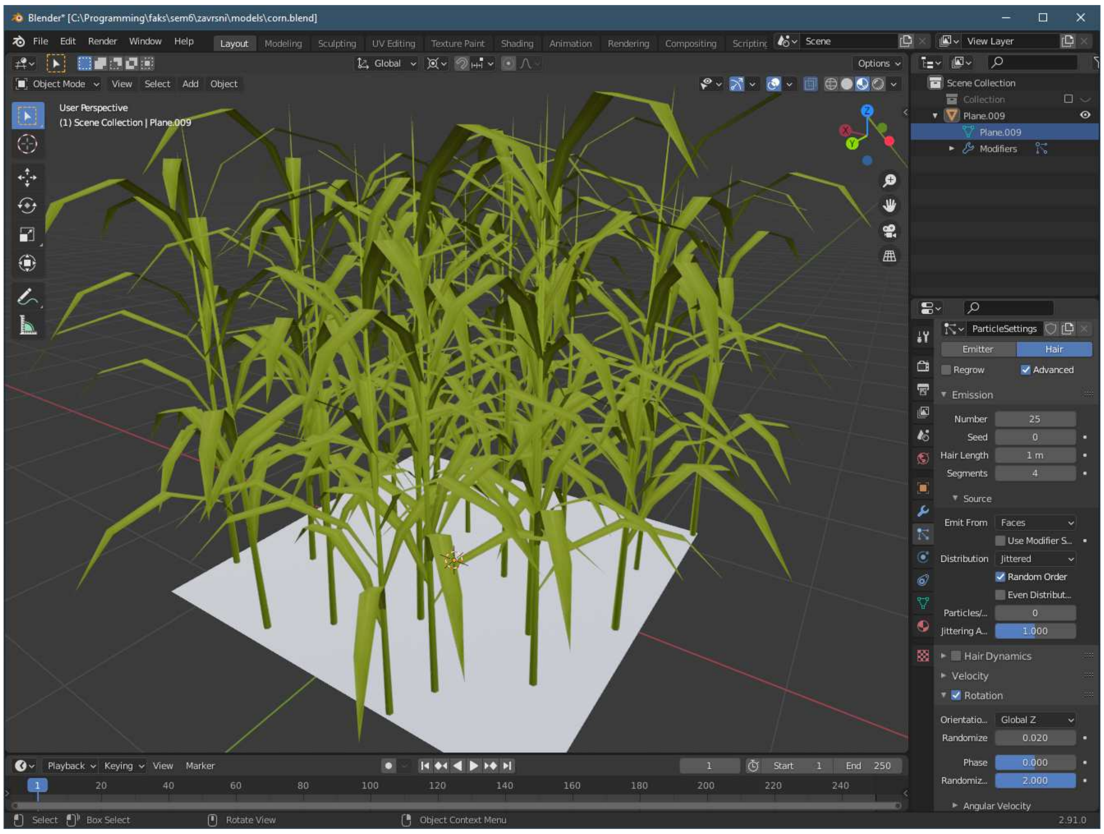
Figure 3.4: The OBJ export dialogue in Blender.

These models were created in Blender and exported as OBJ files. By default, Blender exports objects from the project as distinct objects in the OBJ file (figure 3.4). This then causes Three.js to import the model as multiple meshes with different geometries. This is not suitable for instanced rendering, as the instanced mesh requires exactly one geometry. Therefore, it is necessary to uncheck the option “Objects as OBJ Objects” if one wishes to use these models for instanced rendering.

Because each sample uses only one instance of a model, the models for maize and wheat have to contain multiple plants. This could be done manually, copying and pasting each plant the desired number of times, perturbing each plant’s rotation and scale slightly each time.

However, this can be done much more easily using the particle system in Blender. When using the *hair* option in the particle toolbar (figure 3.6), Blender will place “hairs” on the surface of an object. These “hairs” can be made to look like any object or collection. A plane the size of a single sample was created and made a particle emitter. The model of a single maize plant was made into a collection and used as the “hairs” for the emitter (figure 3.5). By using this system, parameters like the number of plants or the random distribution of their scale and rotation can be easily adjusted.

It is worth noting that the particle system expects the collection or object used as the “hairs” to be pointing in the positive X direction, which means that the template model of a single plant needs to be laid on its side in order for the hairs to be pointing



ParticleSettings
Emitter [✓ Hair]
Regrow [✓ Advanced]
Emission
Number 25
Seed 0
Hair Length 1 m
Segments 4
Source
Emit From Faces
Use Modifier S...
Distribution Jittered
✓ Random Order
Even Distribut...
Particles/... 0
Jittering A... 1.000
Hair Dynamics
Velocity
✓ Rotation
Orientatio... Global Z
Randomize 0.020
Phase 0.000
Randomiz... 2.000
Angular Velocity

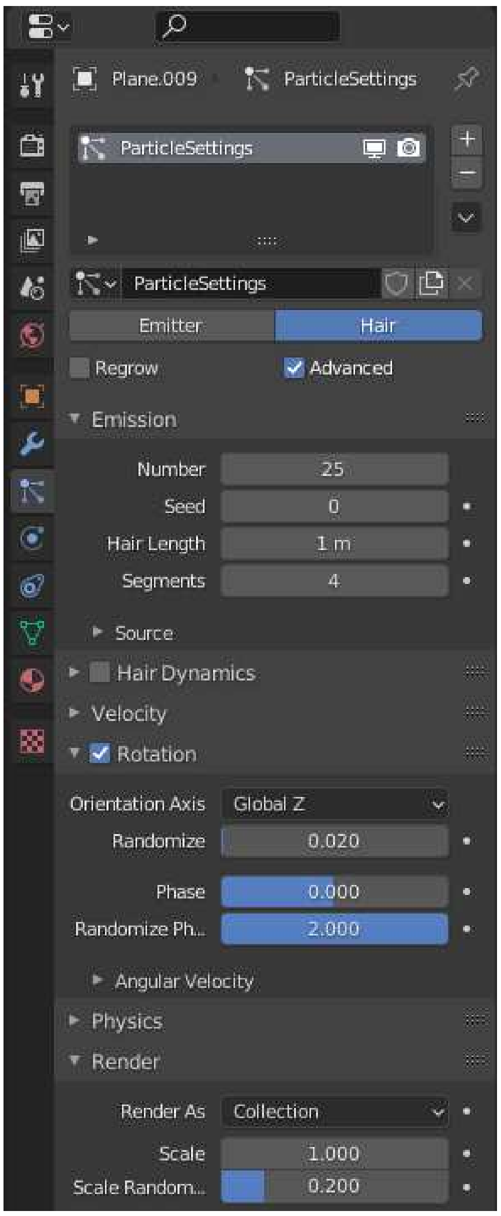
Figure 3.5: The maize model being created in Blender.

in the correct direction.

These models are intentionally simple, seeing as they will be drawn many times in a single frame. A complex model containing more polygons would adversely impact performance. An entire area model contains 1600 samples, each with 200 individual wheat plants, resulting in up to 320,000 wheat plants on screen at once. At 16 triangles per plant, this adds up to over five million triangles on screen. Without levels of detail (LODs), it is infeasible to have more realistic models.

3.5. The ground

The ground is made up of a mesh with a soil texture. The height of the ground is slightly randomised using simplex noise in order to introduce detail. Simplex noise is a gradient noise developed by Ken Perlin as an improvement upon his earlier Perlin noise.[2] This noise is used as a base for octave noise. Octave noise is obtained by repeatedly adding the base noise to itself with increasing frequencies and decreasing amplitudes. Listing 3.2 contains the method used to do this.



Plane.009 › ParticleSettings

ParticleSettings

Emitter | [✓ Hair]

Regrow
[✓] Advanced

Emission

Number	25
Seed	0
Hair Length	1 m
Segments	4

Source

Hair Dynamics

Velocity

[✓] Rotation

Orientation Axis	Global Z
Randomize	0.020
Phase	0.000
Randomize Ph...	2.000

Angular Velocity

Physics

Render

Render As	Collection
Scale	1.000
Scale Random...	0.200

Figure 3.6: A close-up on the particle toolbar in figure 3.5.

```
private readonly octaveCount: number;
private readonly baseFrequency: number;
private readonly persistence: number;
private readonly lacunarity: number;
// ...
private applyOctaves(
  noiseFunction: (...baseNoiseCoords: number[]) => number,
  ...coords: number[]
): number {
  let cumulativeAmplitude = 0;
  let sum = 0;

  let currentAmplitude = 1;
  const scaledCoords: number[] = coords.map(
    (c) => c * this.baseFrequency
  );

  for (let i = 0; i < this.octaveCount; i++) {
    sum += noiseFunction(...scaledCoords) * currentAmplitude;

    cumulativeAmplitude += currentAmplitude;
    currentAmplitude *= this.persistence;

    for (let j = 0; j < scaledCoords.length; j++)
      scaledCoords[j] *= this.lacunarity;
  }

  return sum / cumulativeAmplitude;
}
```

Listing 3.2: The method used to evaluate octave noise.

3.6. Performance

The application was developed and tested in Mozilla Firefox 89 on a computer with an AMD Ryzen 5 2500U processor at 2 GHz with integrated Radeon Vega 8 Graphics, running Windows 10 Pro, version 20H2.

The scene depicted in figures 3.8 and 3.9 contains 3,956,650 triangles. Running at a resolution of 1280×720 pixels, it is displayed at frame-rates between 20 and 40 FPS, depending on which parts of the scene are visible. With antialiasing disabled in code, the frame-rate never dropped below 30 FPS, at the cost of introducing a lot of visual noise in the wheat fields.

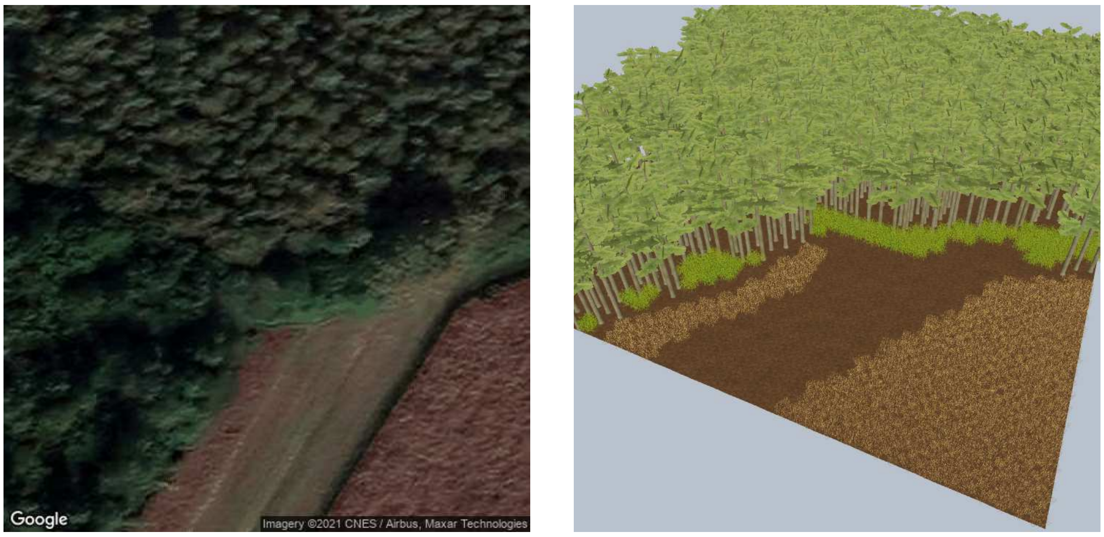


Figure 3.7: Aerial imagery and generated model of the edge of a forest.

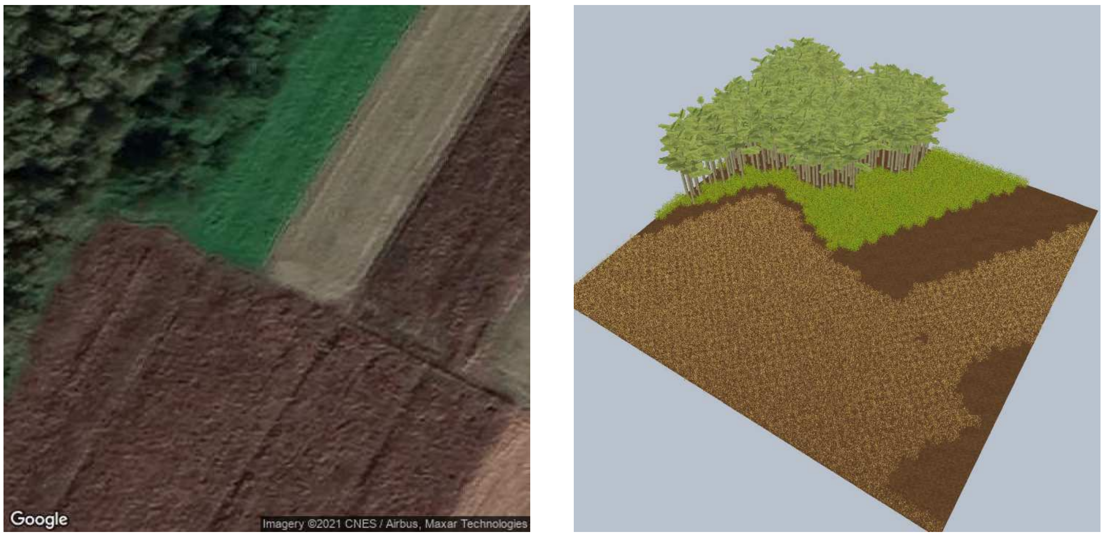


Figure 3.8: Aerial imagery and generated model of a forest and fields.

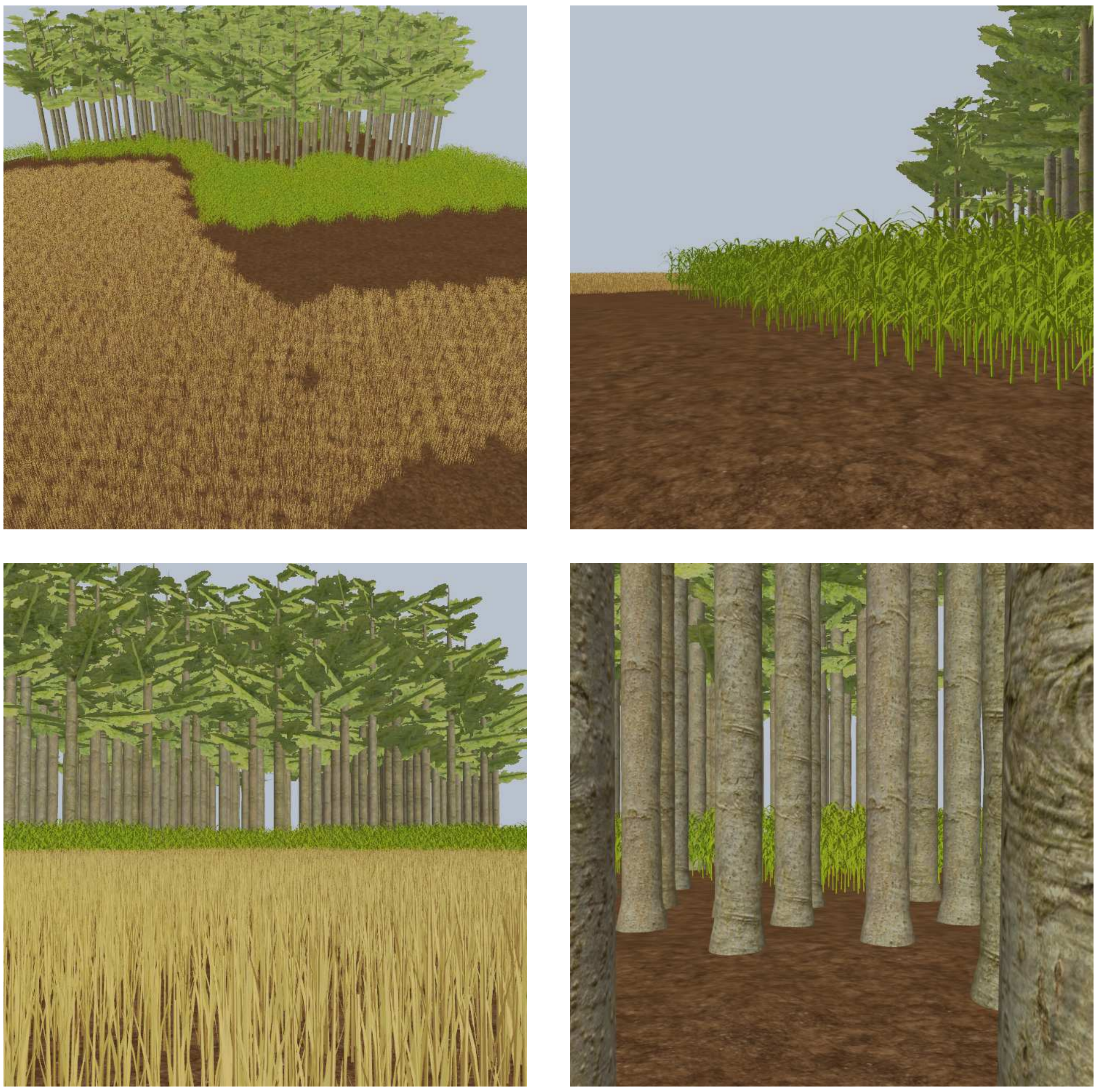


Figure 3.9: Closer images of the model from figure 3.8.

CONCLUSION

The application created as part of the thesis allows the user to specify coordinates on Earth, defining the centre of a square area. Aerial imagery will be retrieved for this area. A three-dimensional model of the area will be populated with three different kinds of plants, according to the imagery. This creates an approximate model of the area in question. The developed implementation has several flaws which could be improved upon.

The method used to map aerial imagery samples to plant types is very simple and was devised from looking at imagery from inland Croatia, combined with trial and error. There exist datasets which could potentially be used with machine learning in order to derive a better, more realistic method to determine plant types from imagery. Even with an improved classifier, mistakes will be inevitable. Identifying plant species from cartographic data is a difficult problem, one not even Microsoft can solve perfectly[3]. Still, even a minor improvement would likely lead to models with vastly better representation of reality.

Additionally, more plant types could be modelled and added to the application. The application is currently limited by the choice of plant species to the region where these plants grow in the wild. By increasing the number of plant species, more diverse areas could be modelled. To aid in this, additional data, like mean temperatures and yearly rainfall, could be fed into the plant species classification algorithm along with the imagery. The time of year when the imagery was taken would help make the application capable of recognising the same plant at different points in its lifecycle.

Looking into writing a custom shader in order to support shadow maps with instanced meshes in Three.js would be worthwhile. Additionally, it might be possible to use levels of detail with instanced rendering. This would allow for more detailed and realistic models to be used closer to the camera and for simpler models with fewer triangles to be used farther from the camera. The result would be greater realism with potentially better performance.

Integrating more data into the model should be investigated. Elevation data might be a good start for this. A different approach to sampling could yield more natural-looking results. For example, a grid based on triangles instead of squares, or different sampling resolutions for different plant types.

The arrival of new technologies like WebGPU has the potential to allow improvements to the quality and fidelity of models in web applications utilising computer graphics while achieving the same or better level of performance.

Standardisation of new APIs has led to the deprecation of proprietary plug-ins like Adobe Flash and Java applets, therefore leading to a safer and more secure Web, since the track record with security of such plug-ins has been less than stellar. In the past decade, several new major web technologies have been standardised, of which WebGL is just one. These have played a major role in providing users with novel experiences on the Web, as well as also opening the gates to making previously native-only applications available in the browser. I believe that these are of tremendous value to users of the Web and I am hopeful for what the future may bring.

References

[1] *Billing Account Credits, Google Maps Platform.* Google, 2021. URL `https://developers.google.com/maps/billing-credits?skip_cache=true#monthly`, accessed: 10 June 2021.

[2] Stefan Gustavson. *Simplex noise demystified*, 2005. URL `https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`, accessed: 10 June 2021.

[3] Charlie Hall. *The biggest surprise of Microsoft Flight Simulator: it makes Bing cool.* Polygon, 2019. URL `https://www.polygon.com/2019/9/30/20885197/microsoft-flight-simulator-bing-maps-hands-on-demo#UbpJ1Q`, accessed: 10 June 2021.

[4] Michael Herzog, Mr.doob, Amit Nambiar. *Mesh – three.js docs*, 2021. URL `https://threejs.org/docs/index.html?q=mesh#api/en/objects/Mesh`, accessed: 6 June 2021.

[5] *WebGL Specification Version 1.0.3.* Khronos WebGL Working Group, 2014. URL `https://www.khronos.org/registry/webgl/specs/1.0.3/`, accessed: 1 June 2021.

[6] *Partnership Series: Bing Maps.* Microsoft, 2020. URL `https://www.youtube.com/watch?v=fjMKLGe1dd8`, accessed: 31 May 2021.

[7] *Features — blender.org.* The Blender Foundation. URL `https://www.blender.org/features/`, accessed: 3 June 2021.

LIST OF FIGURES

Model ruralnog područja temeljen na realnim podacima

Sažetak

Ovaj rad istražuje generiranje trodimenzionalnih modela proizvoljnih ruralnih područja korištenjem knjižnice Three.js. Rad diskutira o izvorima kartografskih podataka. Predstavljen je kratak uvod u korištenje Three.js-a i objašnjene su neke od njegovih poveznica s WebGL programskim sučeljem. Razvijena je web aplikacija u programskom jeziku TypeScript radi istraživanja načina generiranja takvih modela iz podataka dostupnih korištenjem programskog sučelja Google Maps Platform. U radu je opisana ta aplikacija. Modeli biljaka izrađeni za aplikaciju su prikazani. Zaključno, diskutira se o rezultatima rada, utjecaju novih tehnologija na Web, kao i potencijalnim poboljšanjima razvijene aplikacije.

Ključne riječi: računalna grafika, ruralno područje, Three.js, instancirano crtanje, zračne snimke

Rural Area Model Based on Real Data

Abstract

This thesis explores generating three-dimensional models of arbitrary rural areas using the Three.js library. Sources of cartographic data are discussed. A short introduction to using Three.js is presented and some of its links to the underlying WebGL API are explained. A web application was developed using the Typescript programming language in order to explore the generation of such models from data available using Google's Maps Platform API. This application is described in the thesis. The plant models created for the application are displayed. Finally, the results of the thesis are discussed, along with potential improvements to the application developed, and the effect of new technologies on the Web.

Keywords: computer graphics, rural area, Three.js, instanced rendering, aerial imagery