Abstract

Voxelization is the process of converting 3D models into volumetric data. The main goal of this thesis is to improve the the open-source Voxelizer engine, which is written in JavaScript. To make voxelization easy and available, a complementary cross-platform desktop application is also developed, making use of the Voxelizer engine. Further, to make the software secure and easy to maintain, this thesis also focus on automation. A GitHub Action named JSDoc Action is developed for the purpose of automating the generation of JSDoc documentation. The results of this thesis includes a maintainable and easy to use collection of high-quality voxelization software, in addition to several automation tools.

Sammendrag

Voxelisering er prosessen å konvertere 3D-modeller til volumetrisk data. Hovedmålet med denne oppgaven er å forbedre open-source Voxelizer-motoren, som er skrevet i JavaScript. For å gjøre voxelisering enkelt og tilgjengelig, er det også utviklet en plattformuavhengig desktop-applikasjon som benytter seg av Voxelizer-motoren. For å gjøre programvaren sikker og enkel å vedlikeholde, fokuserer denne rapporten også på automatisering. En GitHub Action kalt JSDoc Action er utviklet for å automatisere genereringen av JSDoc dokumentasjon. Resultatene fra denne oppgaven inkluderer en vedlikeholdbar og brukervennlig samling av høykvalitets voxeliseringsprogramvare, i tillegg til flere automatiseringsverktøy.

Acknowledgement

I wish to express my deepest gratitude to my supervisor, Professor Ricardo da Silva Torres, for all the help and guidance throughout the entire project. I also want to acknowledge all the love and support from my family - my parents, Synnøve and Ove; and my sisters, Maria, Viktoria and Helene. They all kept me going, and this work would not have been possible without them.

André Storhaug, Ålesund 20.05.2020

Contents

List of Figures

List of Tables

Listings

Terminology

Glossary

Cross-platform software Computer software that can be run on multiple computing platforms.

Open-Source Software Software that is available to the public and can be freely used or modified.

Polyfill Code for providing modern functionality on older browsers that do not support it natively.

Voxelization Process for transforming a polygon mesh into voxels.

SemVer Versioning scheme.

Library A collection of data and programming code that is used to develop software.

Voxel Three-dimensional analogue of a pixel, representing a value on a regular grid in three-dimensional space.

Notation

GB Gigabyte

MB Megabyte

sec System International unit for Second

Abbreviations

API Application Programming Interface

GUI Graphical User Interface

OSS Open-Source software

JSX JavaScript XML

JSON JavaScript Object Notation

NPM Node Package Manager

VCS Version Control System

RTF Rich Text Format

JS JavaScript

CG Computer Graphics

ES ECMAScript

ES5 ECMAScript 5

ES6 ECMAScript 2015

CSS Cascading Style Sheets

AMD Asynchronous Module Definition

WebGl Web Graphics Library

JIT Just In Time

UUID Universally unique identifier

IO Input Output

IPC Inter-process Communication

GC Garbage Collector

CPU Central Processing Unit

CLI Command Line Interface

UML Unified Modeling Language

Chapter 1

Introduction

1.1 Background

Computer graphics (CG) studies methods for digitally synthesizing and manipulating visual content. An important part of CG is the study of algorithms. This forms the basis for many of the powerful graphics technologies available. One of these technologies is WebGL, a relatively new technology for displaying graphics in web browsers. Since its introduction, the graphical user-experience for the large user mass of the web has been significantly expanded. It allows for almost desktop like graphics performance.

WebGl has especially expanded the abilities regarding the creation of web browser games and simulations. Creating high performance graphics applications in the browser has never been easier. It has opened up for a sea of possibilities where only one's imagination is the limit.

In a lot of simulation software, and even some games, volumetric information plays a crucial part. With everything from fluid dynamics to voxel games like Minecraft, volumetric information is often a key component [1]. One way to acquire such volumetric data is through voxelizing a 3D model (polygon mesh). Voxelization is the process of converting 3D models into a volumetric data. However, to the best of my knowledge, it does not exist any easy-to-use open-source voxelization software written in JavaScript. In order to obtain such volumetric data, developers are therefore forced to go through tedious preprocessing steps, often involving old and complex, hard to use, platform specific tools (for example binvox [2]).

This was a problem I encountered myself a year ago in 2019. In connection with an assign-

ment in a simulation course at the Norwegian University of Science and Technology (NTNU), I needed to be able to easily generate some volumetric data based on 3D models. I was using web-technologies, so I was looking for a simple solution in plain JavaScript. However, I was not able to find such a solution. I therefore decided to make one myself. The result was an open-source voxelization engine, written entirely in JavaScript. It was named Voxelizer. The engine is able to load a 3D model and voxelize it.

However, the Voxelizer engine carries strong signs of the limited amount of time allocated for creating the software. By improving and expanding the capabilities of the project, it could serve to be a valuable open-source asset to the web based game- and simulation-development ecosystem, providing easy access to voxelization.

1.2 Problem Formulation

Problems to be addressed

There exists an open-source JavaScript voxelization engine for voxelizing 3D models, named Voxelizer. The software faces several issues and is lacking important features. It does not produce accurate and representative results. The output sometimes contains holes and a lot of artifacts. Importing and exporting support is extremely limited. Documentation is lacking, and the coding is of poor quality. The project needs to be professionalized, and made easy to both use and maintain. A more complete description of the problems Voxelizer v0.1.3 faces are described in Section 2.14.

Packaging and publication of new releases, as well as documentation, are tedious and manual procedures. This workflow is prone to human errors, potentially introducing critical bugs. These processes could be automated with modern continuous integration and continuous deployment tools, effectively eliminating these vulnerabilities.

1.3 Objectives

The main goal of this thesis is to improve and extend the open-source JavaScript Voxelizer engine, turning it into a maintainable and high-quality open-source project. A second goal is to

develop complimentary software for the Voxelizer project. This will be in the form of a cross platform desktop application and command line interface (CLI), based on the Voxelizer engine, making it easy to voxelize 3D models.

In order to ensure the maintainability of the various software projects, automation is a critical component. Therefore a third goal is to automate the various software projects as much as possible. This also includes the develop a GitHub Action for automating the API documentation generation process.

1.4 Scope

The main purpose of this project is to make it easy to conduct high quality voxelization of 3D models. Its scope is limited by the requirements specification defined in the Preliminary report. Complementary to these requirements, a backlog with user-stories has been created. See Appendix D.

It is important to note that the project does not primarily focus on speed of the voxelization algorithm. The targeted systems often run in an environment where resources are scarce. Performance is therefore of course important. However, usability is also extremely important. This thesis will mainly focus on providing easy access to high-quality voxelization, with reasonable performance. If speed is of the main concerns, it would be better to do the extra work of setting up a native solution (for example binvox [2]), often written in C/C++.

1.5 Systems overview

This section provides an overview of the different repositories developed in connection with this thesis.

1.5.1 Voxel systems

The diagram in Figure 1.1 shows how the different software repositories regarding voxelization interconnects. The green boxes represents the main software projects developed in conjunction with this thesis. During development, some components were generalized and extracted into a

separate repository. This side project is represented by the blue box. The white box represents a third party library.

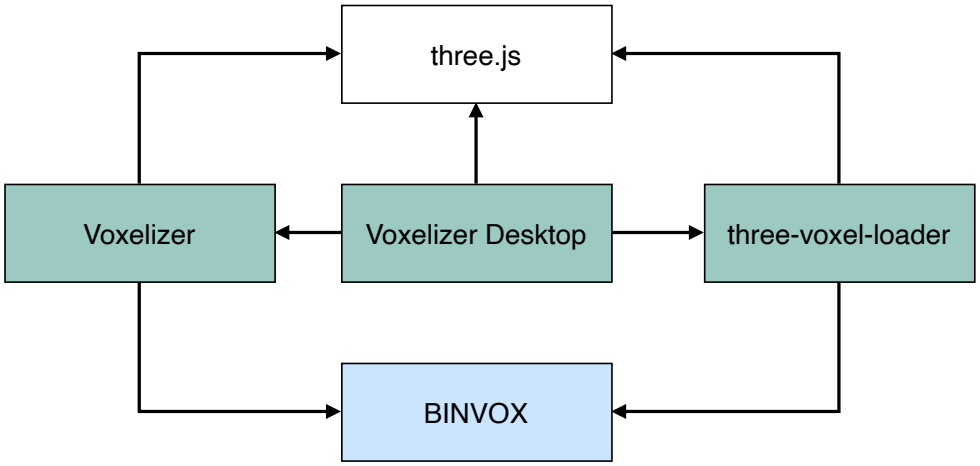


Figure 1.1: Voxel systems overview.

1.5.2 Automation systems

Figure 1.2 shows a diagram of the various automation repositories. This is mainly GitHub Actions, published to the GitHub Marketplace. The yellow box represents a main project. Throughout the project, it also became clear that some supportive actions needed to be created. These side projects are represented by the blue boxes. The white box represents third party software.

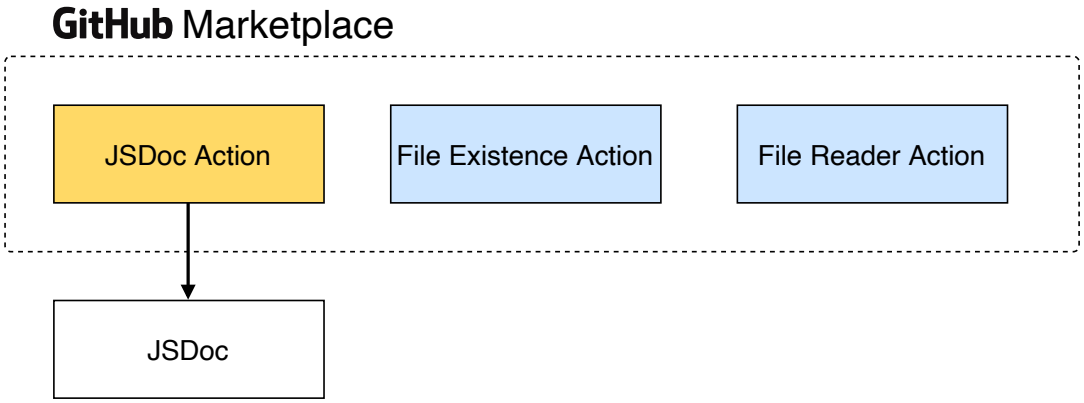


Figure 1.2: Automation systems overview.

1.6 Outline

The rest of the report is structured as follows.

Chapter 2 - Theory: Gives an introduction to the theoretical background that lies the foundation of this thesis.

Chapter 3 - Method: Contains a description of the methodology and materials used throughout the project.

Chapter 4 - Result: Contains a description of the completed works.

Chapter 5 - Discussion: Discusses the achieved results, the execution of methodologies, and how the projects could be further improved.

Chapter 6 - Conclusions: An overall conclusion of the project is presented, reviewing the objectives and the progress made.

Chapter 2

Theory

This chapter includes the theory that lies the foundation of this thesis. The chapter consists of several parts. First, working methodologies are presented, followed by various programming tools and languages. Then, a section about 3D computer graphics is included. Acceleration data structures are then explained. Lastly, a through description of the Voxelizer engine version v0.1.3 is presented.

2.1 Agile methods

2.1.1 Scrum

Scrum [3] is an agile methodology. It is a lightweight, iterative and incremental framework for managing complex work. Scrum is one of the most popular agile methodologies used in the software business. The framework is mainly intended for developing information systems. Scrum defines several roles and different processes. Table 2.1 lists the various roles. An illustration of the Scrum process is shown in Figure 2.1.

At the hart of Scrum, there is a development team. This team is comprised of around three to nine people. The team collectively breaks down project tasks (user-stories) from the backlog into projects. These are then to be completed within a given time frame, so-called "sprints". A sprint usually lasts for about two weeks, to a month. Scrum is also highly focused in the communication between the involved persons, defining several short planning- and review-meetings.

Table 2.2 lists a more detailed description of the various processes in Scrum.

Table 2.1: Roles in Scrum.

Role	**Description**
Product owner	Responsible for representing the stakeholders interests, and ensuring the product success.
Development team	The persons actually implementing the project tasks. They are also responsible for setting up the sprints and having daily stand up meetings.
Scrum Master	Person within the agile development team. The Scrum Master is to serve as a facilitator for the development team. A good Scrum Master should make himself/herself superfluous.

Table 2.2: Scrum processes.

Process	**Description**
Sprint planning	The team selects user-stories from the product backlog. Normally, story points are assigned to the user stories, based on the effort needed to implement it.
Sprint	The actual implementation of the selected user stories. This should result in the next increment (or version) of the product.
Stand-up meeting	Each day, the team then has a 10-15 minute long "stand-up" meeting. This gives the team an opportunity to plan for the day, as well as catching up on the progress done by the other team members.
Sprint review	An evaluation process of what was and what was not finished in the last sprint. The results of the sprint are also presented to the product owner and the various stakeholders.
Sprint retrospect	The team will have a meeting for assessing the completed sprint. This is an important step, as this presents an opportunity to find out how to improve the process of the next sprint.

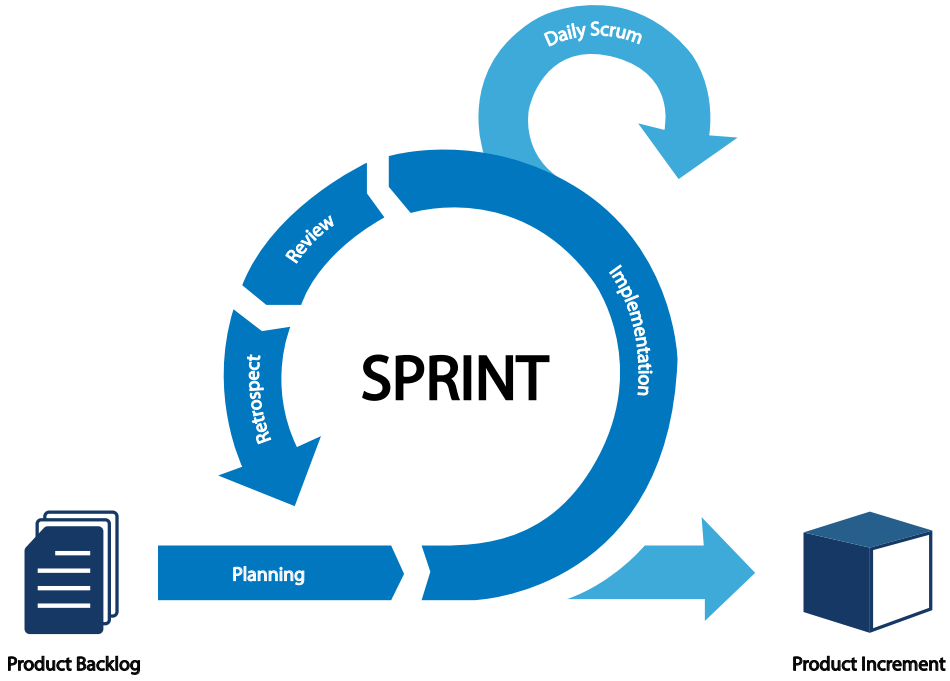


Figure 2.1: Scrum workflow.

2.1.2 Kanban

Kanban [4] is a lean software development methodology, based on the lean methodologies. These have been highly successfully in manufacturing processes. At the center of Kanban is a just-in-time (JIT) process. No new tasks are to be started unless there is a need for it. Further, once a capacity limit has been reached, no further tasks can be started on. When a task has been resolved, a new one can be started. This can simply be described as a pulling workflow. Kanban also often focuses on the visualization of tasks. This is normally done with a Kanban Board. In contrast to Scrum, Kanban allows for the software to be developed in one large development cycle. It is much more flexible, and does not define any roles.

2.2 Git

Git [5] is a type of distributed version control system (VCS), originally created by Linus Torvalds in 2005. Git is free and open-source, and is today the most popularly used VCS. It is fast and efficient, able to handle everything from small hobby projects to giant projects like the Linux kernel. Every Git directory is a complete repository with history and full version-tracking abili-

ties. It does not need access to internet, nor a central server in order to work.

2.2.1 GitFlow

GitFlow is a popular branching model for Git, created by Vincent Driessen [6]. Figure 2.2 displays an example of git history, adhering to the GitFlow branching style. GitFlow truly excels in parallel development of software features. It is extremely well suited for collaboration and scales well. It also provide an efficient and predictable merging flow, making it easy to customize workflows for various needs.

Development of new features are done in **feature branches**. These branch off from the **development branch**, often named **develop**, reflecting the current state of "development". When a feature is done, it is merged back into the **development branch**. When it is time for a new release, a **release branch** is created based on the **development branch**. In this branch, finishing touches can be made, like bumping up the version numbers, etc. When approved, the **release branch** is then merged into a **master branch**, and also back into the **development branch**. The **master branch** only contains released code. In the event of an emergency, a **hotfix branch** can be used. This provides a shortcut for implementing critical fixes. A **hotfix branch** branches directly from master. When finished, the **hotfix** is merged into both **master** and **develop**.

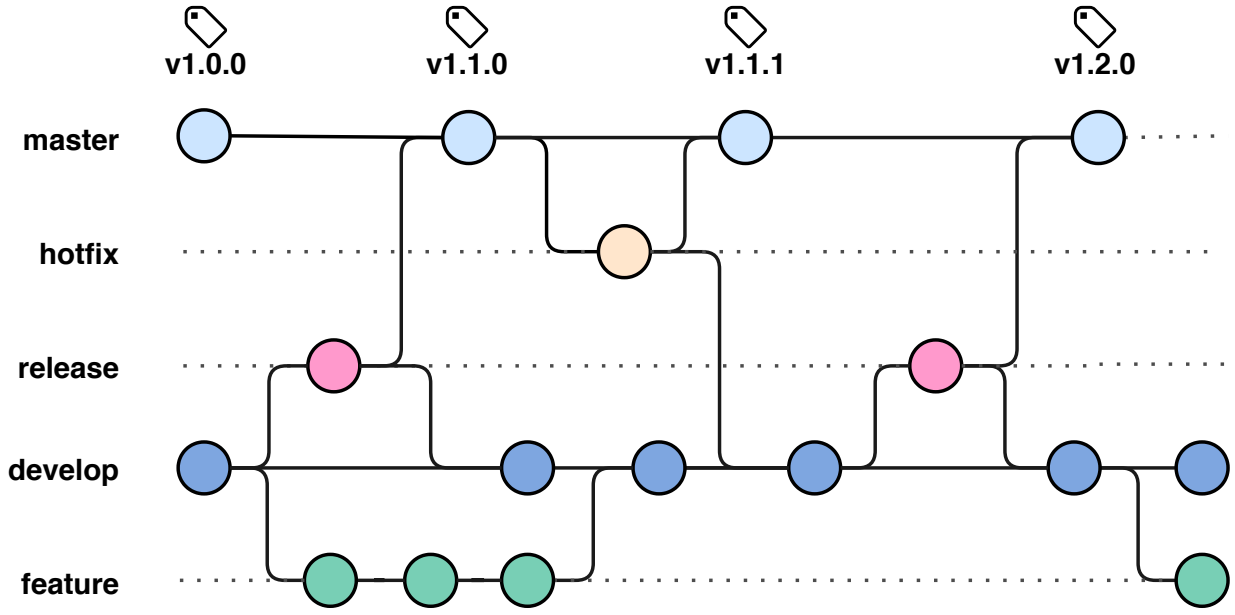


Figure 2.2: GitFlow branching model example.

2.3 GitHub

GitHub is primarily a hosting service for Git repositories. In addition to repository hosting, GitHub provides a range of different services through it's web-based GUI. This includes both wikis, access controls, simple task management tools, statistics, automation capabilities and websites hosting.

2.3.1 GitHub Actions

GitHub Actions [7] is a fairly very new service provided by GitHub. It enables users to automate software workflows, effectively providing a high quality CI/CD for free [1]. It is also possible to set up self-hosted runners. GitHub Actions makes it possible to build, test and deploy code directly from within GitHub.

To setup and configure an automated process, so-called workflows needs to be defined. These are made up of one or more jobs. The actual workflow has to be defined in a YAML file. This file needs to be created and placed in a repository hosted on GitHub. The reader may consult the documentation available at GitHub for being aware of the appropriate workflow syntax for creating such YAML files [8].

2.3.2 GitHub Pages

Github provides its users with a public webpage hosting service. This is named GitHub Pages [9]. User are able to serve static websites directly from their repository hosted on GitHub. Normally, a GitHub Pages site is published by pushing static files to a specific branch named *gh-pages*.

2.4 HyperText Markup Language (HTML)

HTML is a markup language, originally defined by Tim Berners-Lee and Robert Caillau in 1989. It is primarily used for documents on the web, intended to be displayed in a web-browser. It is used for structuring and formatting information. HTML can be used in conjunction with other

[1] GitHub Actions usage is completely free for public repositories. For private repositories, depending on the subscription plan, some thousand minutes of free usage is provided each month.

web technologies such as Cascading Style Sheets (CSS) and scripting languages like JavaScript, in order to either style, or dynamically change and alter the contents of a web page. The currently latest release of the language is HTML5.

2.5 Cascading Style Sheets (CSS)

CSS is short for Cascading Style Sheets. CSS is a programming language in order to describe how HTML elements in a HTML file are to be rendered. Everything from the boldness of a headline, to the background of the entire page.

2.6 JavaScript

JavaScript is a lightweight interpreted programming language. The language is prototype-based, a type of object-oriented programming where properties and methods are added to an instance of an implicitly defined class [10]. JavaScript does not provide any type-checking.

JavaScript was developed by Brendan Eichand and released in 1995 [11]. Initially, it was designed to be a small scripting language for enabling interaction with web pages. A standardization effort of JavaScript, led by Ecma International [12], lead to the ECMAScript specification that the modern JavaScript language conforms to. Since then, the language has evolved rapidly and gained massive in popularity. It has become the de-facto language for adding dynamic behavior to HTML. As of may 2020, JavaScript is among the top ten programming languages according to the TIOBE index [13].

Originally, JavaScript engines were mainly used in browser environments. However, with the development of Node.js in early 2009, this was dramatically changed. Node.js provides a runtime environment for executing JavaScript outside of a web browser. It set the stage for server-side JavaScript programs. In January 2010, a package manager named npm [14] (originally short for Node Package Manager) was released for Node.js. This made it easy for developers to share and reuse source code.

The size of JavaScript programs has increased massively in size. With increased size, so does the complexity of the code. However, JavaScript had very limited functionality in terms of split-

ting a program up in smaller modules for use with the browser. Maintaining large codebases was a nightmare. It therefore became apparent that a way of breaking down a JavaScript program into smaller modules was needed. Several open-source module systems were therefore developed by the community in order to tackle this problem.

2.6.1 Module systems

- **CommonJS** is a module specification meant for JavaScript outside the browser. It is mainly used in Node.js, and hence it is one of the most popularly used module definitions. The modules are mainly imported and exported with the keywords "require" and "module.exports".
- **Asynchronous module definition**, or more commonly known as AMD, is a JavaScript module definition intended for the browser. It defines an API for defining code as modules, including their dependencies. AMD also has the capability of loading modules asynchronously. The most popular AMD module loader is named RequireJS [15].
- **Universal Module Definition**, abbreviated UMD, is a module definition wrapper to be able to use various module systems [16]. Be it in the browser or in Node.js. It is compatible with both CommonJS and AMD.
- **JavaScript modules**, or ES Modules, are a language native module system, introduced with ECMAScript 2015 (ES6) in 2015. The implementation is relatively new, so a lot of libraries, frameworks and packages does not support this yet. Still, most browsers have already implemented support for this [17].

2.6.2 Transpilation

Due to the many versions of JavaScript, or more specifically ECMAScript, like ES5, ES6 and ES7, compatibility is an issue. Not all browsers and environments support the latest ECMAScript versions. Tools like Babel [18] has been developed in order to transpile JavaScript to a specific version. The most common transpilation target, supported by all the major browsers is ECMAScript 5 (ES5).

2.6.3 Bundling

JavaScript bundling is an optimization technique to combine separate resource files into one file. This is done in order to reduce the number of HTTP requests required for a page to load. Several bundlers are able to do so called tree-shaking, dramatically reducing the size of the finished bundle. In addition to the performance gain, bundling is also often done in order to develop a JavaScript application in separate files, effectively employing a form of module system. The bundlers often use one or more of the popular module systems like UMD and ES modules. There are three main actors in terms of bundling JavaScript.

- **Webpack** [19] is a module bundler for JavaScript. However, it is also able to transform front-end assets like HTML, CSS, and images. Webpack is mostly used for bundling JS applications, and is highly extendible. It also provides a way to bundle an application for Node.js to be used in the Browser [2]. However, as of may 2020, it does not support exporting ES Modules.

- **Rollup** [20] is a module bundler primarily focusing on JavaScript libraries. It has a lot of similarities with Webpack. However, it is a bit lighter and provides exporting support for ES Modules.

- **Browserify** [21] is a lightweight module bundler for enabling the use of CommonJS syntax in the Browser.

2.7 TypeScript

TypeScript [22] is a typed superset of JavaScript that compiles to plain JavaScript. It is open source, and primarily developed and maintained by Microsoft [23]. TypeScript provides optional static typing to the JavaScript language. The TypeScript compile also includes support for the latest ECMAScript features.

[2] This requires that no Node.js specific APIs are used. Alternatively, polyfills could be supplied.

2.8 JavaScript Object Notation (JSON)

JavaScript Object Notation, better known as JSON, is a lightweight data interchange format. It is easy for both humans and machines to read and write. The data is stored as attribute–value pairs. An example is shown in Listing 2.1. Here, the the first key is "name", and the corresponding value is "A. Storhaug".

Listing 2.1: Example JSON data

```
1 {
2     "name": "A. Storhaug",
3     "age": 22,
4     "email": "andr3.storhaug@gmail.com",
5     "url": "https://github.com/andstor"
6 }
```

2.9 JSDoc

JSDoc is markup language for annotating JavaScript source code files. The JSDoc specification was released in 1999. Today it has become the de-facto JavaScript documentation language. It is for example used in projects like the Google Closure Compiler [24] by Google. Since JavaScript has no type-checking, JSDoc is able to patch some of this inconvenience. Figure 2.2 shows an example of JSDoc code, describing a soda bottle class implementation. By using various tools, one is able to generate documentation in formats like HTML and RTF. JSDoc 3 is the current version of the original companion documentation generation tool for JSDoc. JSDoc 3 [25], also referred to as just JSDoc, is the most used tool for programmatically generating JavaScript documentation. It has a wast feature set, even allowing users to create customized themes, known as templates. Currently, JSDoc is used by more than 38.800 public projects [26], and has over 10.600 stars on GitHub [27].

Listing 2.2: JSDoc example code

```
1 /**
2  * Represents a soda bottle.
3  * @constructor
```

```
4   * @param {string} brand The brand of the soda.
5   * @param {number} size The size of the soda in deciliters.
6   */
7  function SodaBottle(brand, size) {
8  }
```

2.10 Tools and libraries

2.10.1 WebGL

WebGL (Web Graphics Library) [28] is a JavaScript API for rendering interactive high-performance 3D and 2D graphics in a web browser. WebGL uses OpenGl ES [29], a subset of OpenGL. WebGL makes it possible to take advantage of hardware graphics acceleration provided by the user's device. For actually displaying the graphics in the browser, HTML5 <canvas> elements are used.

2.10.2 three.js

three.js[30] is a cross-browser JavaScript library for creating and displaying 3D computer graphics in a web browser. It is open-source and licensed under the MIT license. three.js uses WebGL under the hood. The library abstracts away a lot of tedious manual labour, like the setup of WebGL, construction of vertices, faces, etc. three.js provides the user with an easy API for directly constructing three-dimensional objects like boxes, spheres and toruses, as well as easy camera controls. The library also includes a vast set of shaders, making it very simple to make use of high quality materials. three.js is one of the most popular 3D graphics JavaScript library for use in the browser, as can be seen on its GitHub repo [30].

2.10.3 ndarray

ndarray [31] is an open-source JavaScript package providing modular multidimensional arrays, written by Mikola Lysenko in 2013. In short, ndarray implement a higher dimensional views of 1D arrays. The 1D array can either be a normal JavaScript Array, or a JavaScript typed array.

MDN defines typed arrays as “array-like objects that provide a mechanism for reading and writing raw binary data in memory buffers.” (Mozilla Developer Network [32]). Mainly, they are used for maximizing efficiency and reducing memory footprint. However, typed arrays are normally quite difficult to work with in JavaScript. Multidimensional typed arrays even more so. ndarray provides a simple but powerful API, making use of multidimensional typed arrays easy.

2.10.4 Electron

Electron [33] is an open-source framework developed and maintained by GitHub [34]. It allows one to build cross-platform desktop applications with JavaScript, HTML and CSS. Electron is used by thousands of people, and apps like Visual Studio Code [35], Facebook Messenger [36] and Microsoft Teams [37] are all made with Electron.

In short, the Electron Application architecture is as follows. An Electron application starts by running a package.json’s main script. As shown in Figure 2.3, this creates a process that is called the main process. From within this process, a GUI can be displayed by creating web pages. Electron uses Chromium for creating web pages Google [38]. This means that Chromiums multi-process architecture is also available. Every generated web page is therefore run in its own process. These processes are called renderer processes. Communication between the main process and a rendering process is done using inter-process communication, or IPC. Electron also makes it possible to use Node.js APIs, effectively allowing lower level operating system interactions.

2.10.5 React

React [39] is a JavaScript library for building user interfaces, created by Facebook [40]. It is based around components, where each component manage its own state. These are then composed together, enabling the creation of intricate and complex UIs. The library also implements its own syntax extension to JavaScript. It is named JavaScript XML, or the more popular used term - JSX. In addition to this, there exists a vast ecosystem of plugins for React, greatly simplifying the implementation of everything from localization to state management.

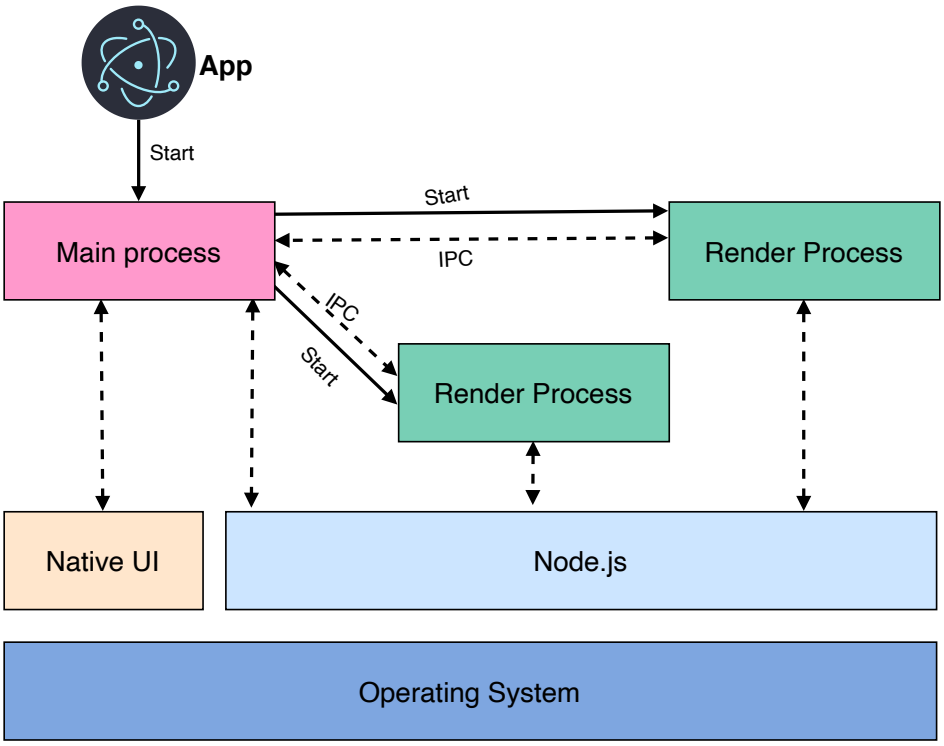


Figure 2.3: Electron architecture.

2.10.6 Semmle LGTM

LGTM by Semmle [41] is a web service providing code security analysis. The service is free for open-source projects. LGTM integrates with sites like GitHub and Bitbucket, and is able to analyze projects written in Java, Python, JavaScript, TypeScript, C#, Go, C and C++. It seeks out to combat the manual process of finding vulnerabilities. By catching them at an early stage, one can prevent vulnerabilities from reaching production. LGTM is based on large community of top security researchers, making it possible to help developers ship secure code [42].

2.10.7 Coveralls

Coveralls [43] is a web service for code testing coverage. It enables one to track a projects code coverage over time, providing valuable insight in a projects testing suite. Coveralls also features close integration with GitHub, enabling pull request coverage reviews.

2.11 3D computer graphics

3D computer graphics refers to three-dimensional representation of geometric data in computers, normally to be rendered into a two-dimensional image. The finalized render may be saved or displayed on a screen in realtime. The geometric data is usually a 3D model, stored in an appropriate file format. The most basic polygon primitives in computer graphics includes vertices, edges and faces. A vertex is simply point in space. An edge is a connection between two vertices. A face is a closed set of edges. Figure 2.4 shows a yellow triangle with the appropriate vertex, edge and face labels. These primitives together defines a polyhedral. Polyhedrons can then be further grouped together into a mesh. The most common type of polygon mesh is a triangular mesh. This is a mesh comprised of only triangles. Figure 2.4 shows an illustration of a section of a triangular mesh.

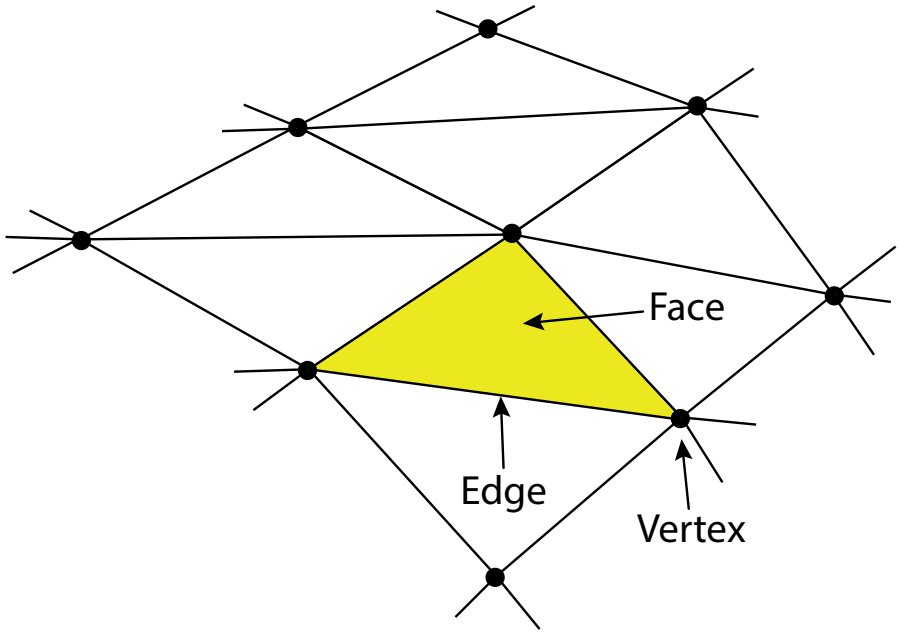


Figure 2.4: Triangular mesh

2.11.1 Texture maps

A texture map is an image which is applied, or mapped, onto the surface of a geometry. The images is often in the form of a bitmap image or a procedural texture. Texture mapping, or

UV mapping, is the process of projecting an actual 2D image onto a 3D model. The technique was initially developed by Edwin Catmull in 1974 [44]. UVs are two-dimensional texture coordinates, assigned to every vertex in a polygon. They are essential in terms of describing how an image gets applied onto a geometry. Figure 2.5 shows an illustrative example of how a 2D image gets "wrapped" around a 3D model. A lot of 3D modeling software are able to do the UV unwrapping automatically, for example Blender [45]. It is also possible to map a finalized render into a surface texture, a process known as baking [46]. This is primarily used as an optimization technique.

Figure 2.5: Texture mapping illustration.

2.11.2 Ray casting

Ray casting is the concept of use of ray–surface intersection tests to solve a variety of problems in 3D computer graphics and computational geometry. The first use of the term ray casting was made by Scott Roth, in a paper from 1982 titled "Ray casting for modeling solids" [47]. Raycasting is demonstrated in Figure 2.6. A ray is directed towards an object. If it crosses a face, an intersection is registered.

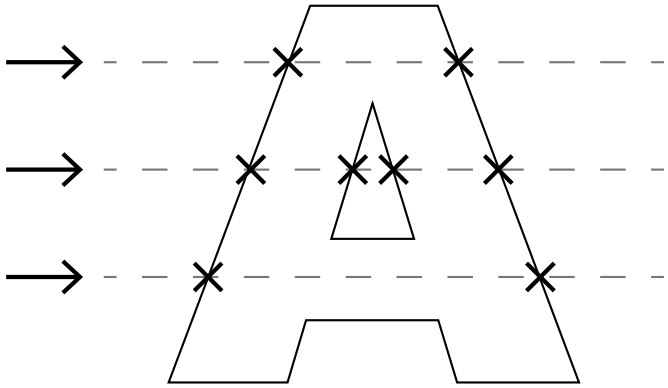


Figure 2.6: Raycasting intersections example.

2.12 Acceleration data structures

2.12.1 Octrees

An octree is a type of tree structure. Each internal node of an octree has exactly eight children. An octree is most commonly used for partitioning three-dimensional space. This is done by recursively subdividing the space into eight octants. Note that depending on the number of recursive subdivisions, an octree may contain multiple objects in its leaf nodes. Figure 2.7 shows an example of an octree with three levels. Octrees are very commonly used in 3D computer graphics. Another common use case of octrees is for storing voxel data.

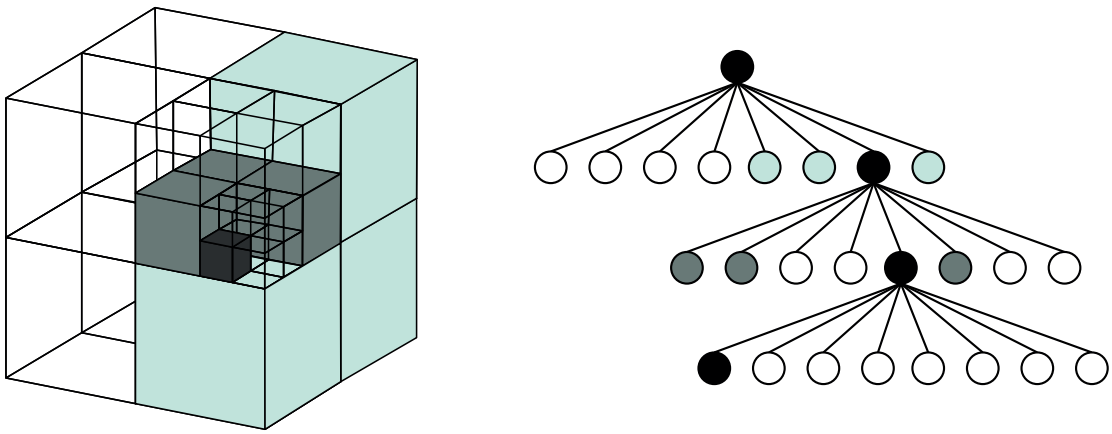


Figure 2.7: Example of an octree with three levels.

2.12.2 Bounding volume hierarchy

A bounding volume hierarchy, abbreviated BVH, is a tree structure on a set of geometric objects. A BVH construction algorithm partitions the actual objects. The objects are wrapped in a so-called bounding volume, forming the leaves of the tree. These are then grouped together into a larger bounding volume. This process is then repeated in a recursive manner. The result is a tree structure with one single bounding volume as the root node. An example of a BVH is shown in Figure 2.8. BVHs are often used for accelerating collision detection and raytracing.

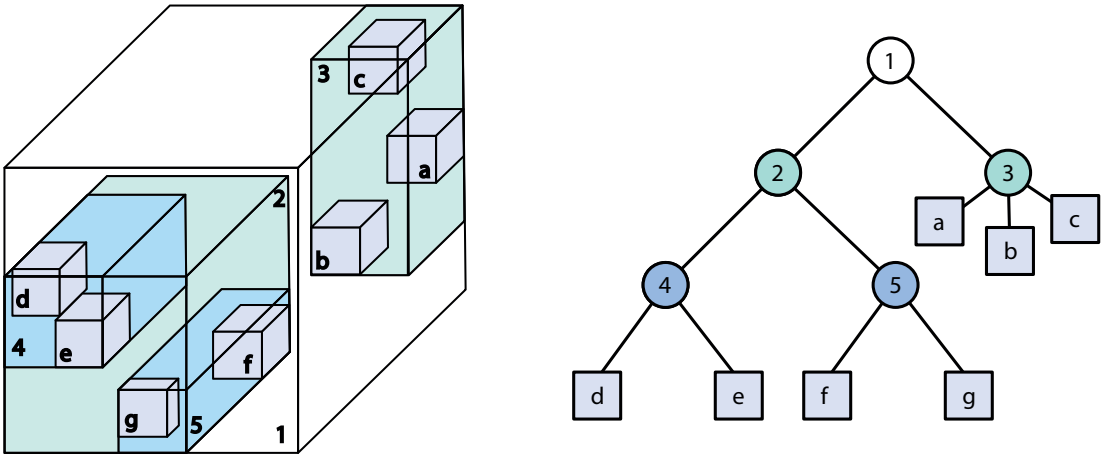


Figure 2.8: Example of an BVH. Replica of figure from MacDonald [48].

2.13 Voxel

A voxel is the three-dimensional analogue of a pixel [49]. It represents a single data point in a regularly spaced three-dimensional grid. Figure 2.9 shows an illustration of three voxels, where one of the voxels are marked with blue color. A very common use of voxels are in medical imaging, for example datasets produced by a CT scan. Other areas where voxels are commonly used includes simulations and for representing terrain in games.

2.14 Voxelizer v0.1.3

Voxelizer v0.1.3 [50] is a JavaScript engine (or library) for conducting voxelization of 3D models. It was written by me, André Storhaug, in 2019. Version 0.1.3 features a relatively simple voxeliza-

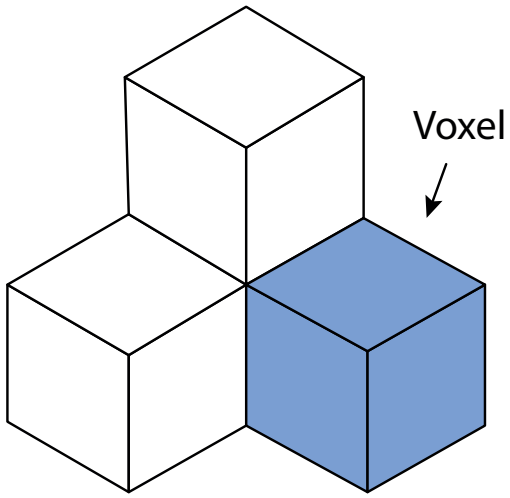
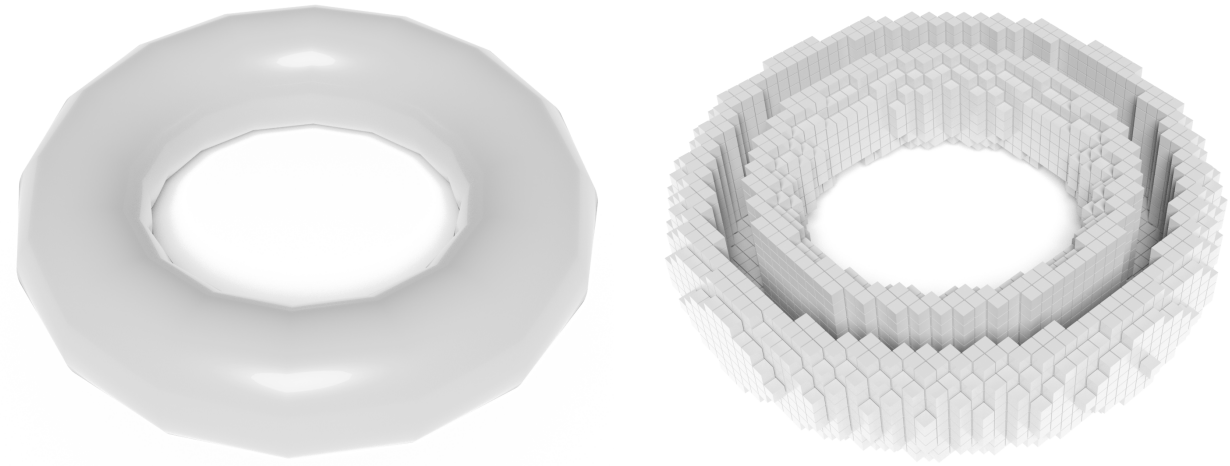


Figure 2.9: Three voxels.

tion algorithm which is based on raycasting. The sampling algorithm tries to produce a filled volume-representation of a supplied 3D model. It samples the front and the back of the model, combines the two results together and tries to fill the in-between gap. The 3D model loading capabilities of the program is limited to plain OBJ files. In terms of exporting, the software is able to output a 3D JavaScript array (nested arrays).

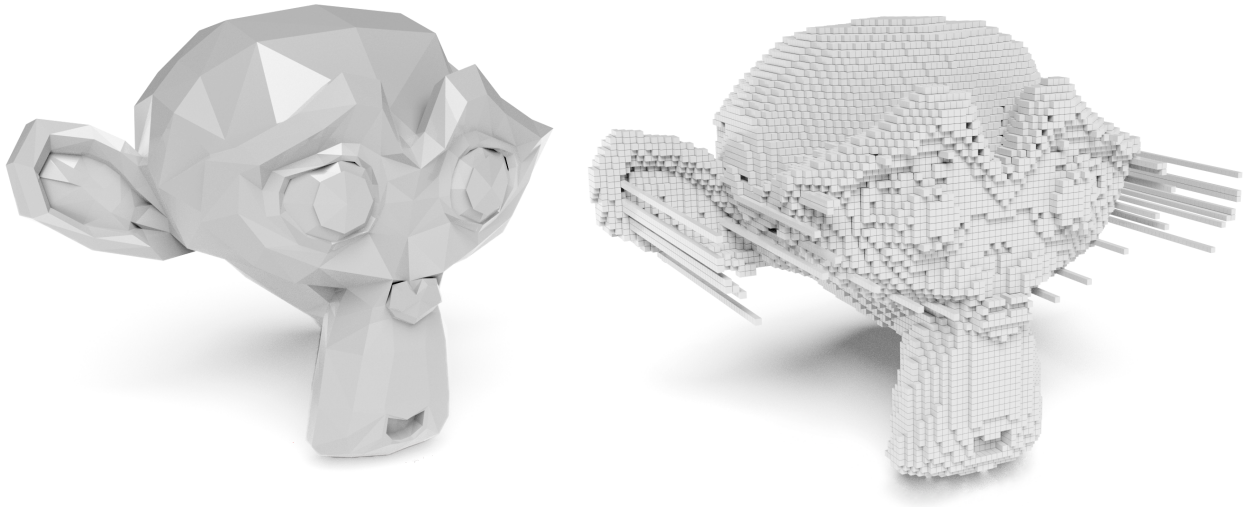
The engine is using ES6 features, hence it is transpiled with Babel (see Section 2.6.2). However, is not bundled. It is therefore not possible to use the program out of the box in a browser. One is limited to Node.js, or setting up a build system involving a module bundler like Webpack or Rollup, as described in Section 2.6.3. The source code is messy, and it is very hard to extend functionality. Especially due to a severe lack of documentation.

Voxelizer v0.1.3 produces unsatisfactory voxelization results. Firstly, several of the voxelizations contains holes. This can be clearly seen in Figure 2.10. A voxelization with holes often renders the voxelization useless. Secondly, a lot of artifacts are often generated, severely degrading the results. This is shown in Figure 2.11, where long strains of voxels appear in the front of the model. This is especially pronounced around the ears of the monkey. Thirdly, the software is only able to produce a filled voxelization result. This is mainly because the model is only sampled from the front and back. The other sides of the model are not taken into account. This means that shell voxelization is not an option, and details may be lost. Lastly, the algorithm used is unstable, often failing in the filling prosess. This problem can be seen in Figure 2.12.



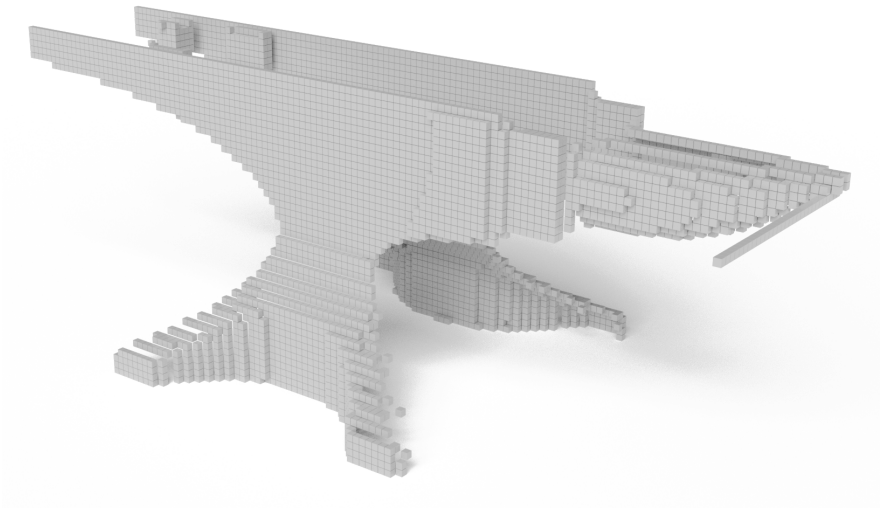
(a) Original torus mesh. (b) Voxelized result.

Figure 2.10: Voxelization of a torus with Voxelizer v0.1.3. The voxelization is done with a resolution of 40.

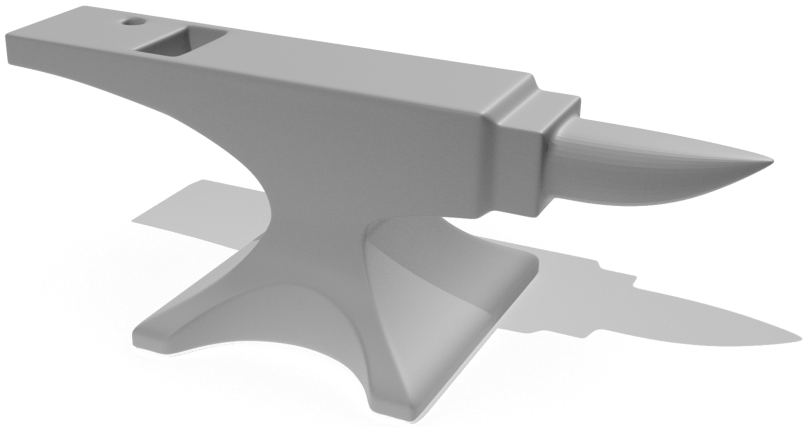


(a) Original monkey mesh. (b) Voxelized result.

Figure 2.11: Voxelization of a monkey with Voxelizer v0.1.3. The voxelization is done with a resolution of 100.



(a) Voxelized anvil with Voxelizer v0.1.3.



(b) Original anvil 3D model.

Figure 2.12: Voxelization of a monkey with Voxelizer v0.1.3. The voxelization is done with a resolution of 2^7.

Chapter 3

Materials and methods

This chapter presents the materials and methods used in this thesis. The chapter is organized as follows. Section 3.1 lists the various tools used in this project. Section 3.2 describes the working methodology used. Then, the following five sections presents how the different main projects are implemented. This includes the three-voxel-loader plugin, the Voxelizer engine, the BINVOX package, the Voxelizer Desktop application and the JSDoc Action. Section 3.8 presents how automation for the projects is implemented. Section 3.9 and 3.10 presents two additional GitHub Actions needed in connection with the automation. Finally, Section 3.11 provides a description of how the 3D models used in this thesis are created.

3.1 Tools and libraries

This section provides a describes the various tools, libraries and services used.

3.1.1 JavaScript

JavaScript has mainly been chosen as the base implementation language for all the projects because of its cross platform compatibility and popularity.

3.1.2 npm package manager

npm is used as the main package tool. Also, all published packages are published to the npm package registry. Alternatively, the new GitHub package registry could be used instead. However, npm is the both the most stable and popular JavaScript package registry available.

3.1.3 GitHub Actions

For automation, several platforms are available. One of the more popular ones includes Travis CI [51]. However, GitHub Actions is used instead. This is because GitHub Actions are deeply integrated into GitHub. It also has a unique concept of Actions, as described in Section 2.3.1.

3.1.4 Build tools

Several build tools are used. This includes:

- **Rollup** for bundling the three-voxel-loader and BINVOX projects.
- **Webpack** for bundling the Voxelizer engine.
- **Babel** for transpiring JavaScript to ES5.
- **electron-builder** for building and preparing Electron apps for distribution.

3.1.5 Third-party libraries

Several third-party libraries and frameworks are used. Following is a short summary:

- **three.js** provides a simple and powerful 3D graphics library. It also includes raycasting functionality.
- **sparse-octree** is used for generating octrees.
- **three-bvh-mesh** is used for improving the raycasting functionality of three.js.
- **format-vox** provides tools for parsing VOX files.
- **dat.gui** makes it easy to create input controls.

- **ndarray** provides multidimensional views of 1D typed arrays.
- **Jest** is used as testing framework.

3.2 Working methodology

This section presents the working methodology used throughout the project.

3.2.1 Scrum

Even though this is a one-man project, I have tried to adapt the scrum methodology. Alternatively, the much more flexible Kanban methodology could also be a good fit. However, Scrum is chosen for its very good framework and supporting tools like Jira and Confluence. Scrum helps to keep the pace up, and stay on track with progress.

The main adaptations made are mainly regarding the various meetings defined by Scrum, as described in Section 2.1.1. Sprint planning is done as normal. User stories are selected from the backlog, and story points are assigned. The sprint is then started, and lasts for two weeks. Instead of daily stand-up meetings with a team, I have have set aside 10-15 minutes for planning every day. After a sprint is finished, a sprint review is done. Then, a sprint retrospect is conducted, in order to be able to improve the next sprint. This includes investigating both sprint burn-down charts and other statistics generated with Jira.

3.2.2 GitFlow

The branching strategy used during development is GitFlow, as described in Section 2.2.1. This creates a consistent development and merging flow, making it easy to implement automation in the various project. Since GitFlow is very popular, a lot of potential contributors are already familiar with the strategy. Providing familiar guidelines helps to keep the projects tidy and secure.

3.2.3 Semantic versioning

All the created projects are enforcing Semantic Versioning. This makes the use of the software predictable in terms of compatibility and potential breaking changes.

3.3 three-voxel-loader

The main goal for the three-voxel-loader is to generate a 3D mesh based on voxel data. The next subsections will present a walkthrough of how the plugin is implemented, including the various design choices made. Figure 3.1 shows a simplified UML diagram of the three-voxel-loader.

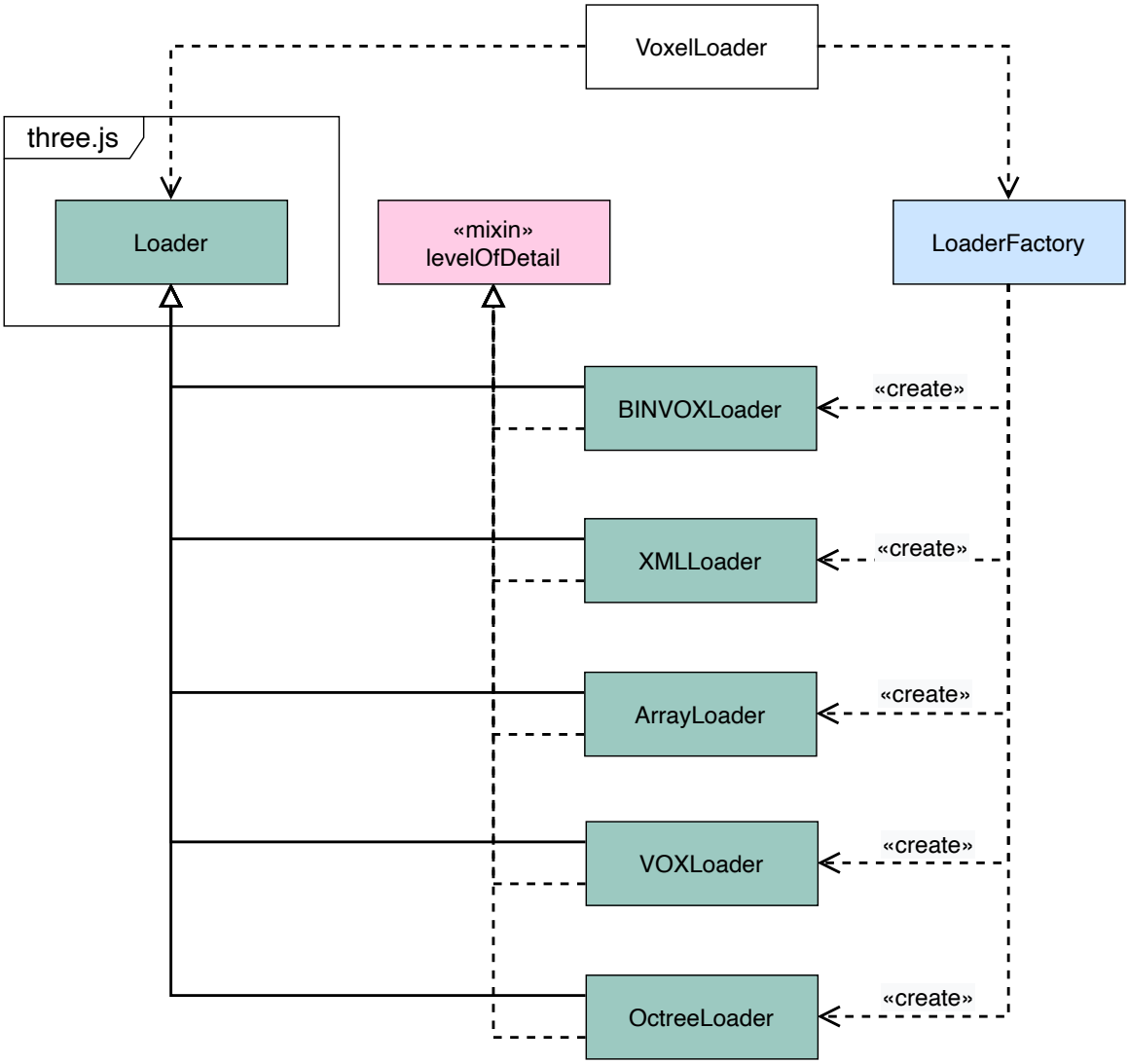


Figure 3.1: UML class diagram of the three-voxel-loader.

3.3.1 Internal data structure

For the plugin to be able support loading of various voxel data formats, an internal data structure is used. This serves as an interface for the processing of various voxel data internally in the

plugin. The chosen data structure is an octree, as described in Section 2.12.1. The actual implementation of the octree is done with the sparse-octree library by Raoul van Rüschen [52]. There are several reasons behind the choice of using an octree.

Voxel data normally consists of very large amounts of data. One of the main limitations for how big a dataset can be, is the amount of available computer memory (RAM). The memory footprint of the plugin is therefore a big concern, especially since the targeted runtime environments are often placing further restrictions on the available memory resources. Voxel data normally contains very large amounts of empty space, or "air". This data is not needed for generating the polygon mesh. Only the data about the actual voxels and their locations are of interest. The location of the voxels are normally not stored explicitly, but rather derived by their relative location to neighboring voxel cells. An octree is especially well suited for this purpose. Since an octree is based on partitioning of space, large amounts of this empty space in the voxel data can be discarded. This works especially well if the voxel data is clustered.

Octrees also makes it easy to implement a Level of Detail (LOD) mechanism. By determining the desired depth of the octree, one are able to simplify the detail of the voxel data. This is very valuable, as generating mesh geometry for every voxel is placing high stress on the available hardware resources. Being able to control the LOD, this can be very effective in terms of simplifying the resulting mesh.

3.3.2 Loading voxel data

In order to actually load the voxel data (by generating an octree), several loader classes have been created. These classes all extend the *loader* class defined in three.js, providing both consistency and tight integration with library. It also ensures that extending the support for more file loaders in the future is easy. Finally, a factory pattern has been implemented for getting and instantiating the desired voxel loader. This makes it able to define an easy-to-use API, where the user only needs to supply the actual voxel data and the corresponding format.

The currently implemented loaders supports several file formats, including XML, VOX and BINVOX. It is also possible to import plain 3D arrays. Several of the loaders also supports color of the voxels. Following is a brief description of these loaders.

- **XML** is a versatile file format, which is easy to manipulate and customize. For implementing the XML loading, the native JavaScript DOM parser is used for the actual parsing of the XML data. The format supports color data. The required format of the XML document structure is described on the GitHub wiki page for the plugin. PROVIDE LINK HERE!!!!! ACTUAL http//::.

- **VOX** file format is provided by MagicaVoxel, a popular voxel graphics editor. The file format also supports color data. VOX files are loaded with a third party package named *format-vox* [53].

- **BINVOX** [54] is one of the more popular voxel data file formats. BINVOX is the file format used by the binvox [2] voxelization software. A separate repository named binvox [55] has been created for handling BINVOX files. See Section 3.5

- **Arrays** are normal datatypes in JavaScript. For plain voxel data, a 3D array needs to be supplied. This is simply several nested arrays. A filled voxel is represented with a truthy value, whereas a void/empty voxel is a falsy value. For loading color data, a 4D array with RGB values has to be supplied along with the 3D array. The array loading support is implemented by simply iterating the multidimensional array(s).

- **octrees** are used as the internal data structure for the three-voxel-loader, as discussed in Section 3.3.1. The three-voxel-loader support loading an octree of this type. This makes it simple to save and load the internal octree data.

3.3.3 Visualization

The most intuitive way to visualize a voxel is in the form of a cube. BoxBufferGeometry from three.js has therefore been used to generate the individual 3D visualization of the voxels. However, one BoxBuffer geometry consists of no less than twelve triangles. The number of triangles generated for visualizing the mesh is therefore twelve times the number of voxels. One of the more time-consuming operations in terms of actually displaying the 3D graphics, is the number of draw calls made to the graphics API. In order to limit this, all the generated box meshes are merged into one big mesh. For actually coloring the voxels, color is applied to the vertices of the generated box meshes.

3.3.4 Debugging

For actually developing and testing the plugin, a HTML page was created. The page includes basic setup of three.js, alongside various input controls for inspecting and testing the three-voxel-loader plugin. The input controls are provided by a lightweight JavaScript controller library named dat.gui [56]. In the end, the debugging solution was polished and deployed to GitHub Pages, serving as an example for the various functionality the plugin provides.

LINK TO GITHUB PAGES EXAMPLE

3.3.5 Building

The plugin is bundled with Rollup. This produces excellent bundles, with support for both UMD and ES Modules. See Section 2.6.3 for more details on Rollup, and Section 2.6.1 for UMD and ES Modules. three-voxel-loader makes use of ES6 features. All source files are therefore transpiled to ES5 with babel.

3.4 Voxelizer

The Voxelizer engine version v0.1.3 needs to be improved. The next subsections will present a walkthrough of how the plugin is reimplemented and improved, including the various design choices made.

3.4.1 Systems overview

UML diagram of OLD system here? The system is broken down in several modules/systems (folders). Figure 3.2 displays a UML diagram of the Voxelizer engine v1.0.0. The white box is a **core system** class. Blue is the **algorithms system**, and the **exporters system** is colored green. The **color system** is purple. Finally, the pink box is a **volume system** class. Following is a short description of the various modules:

- **core** - The core module contains core APIs. This provides the main user API for conducting the voxelization.

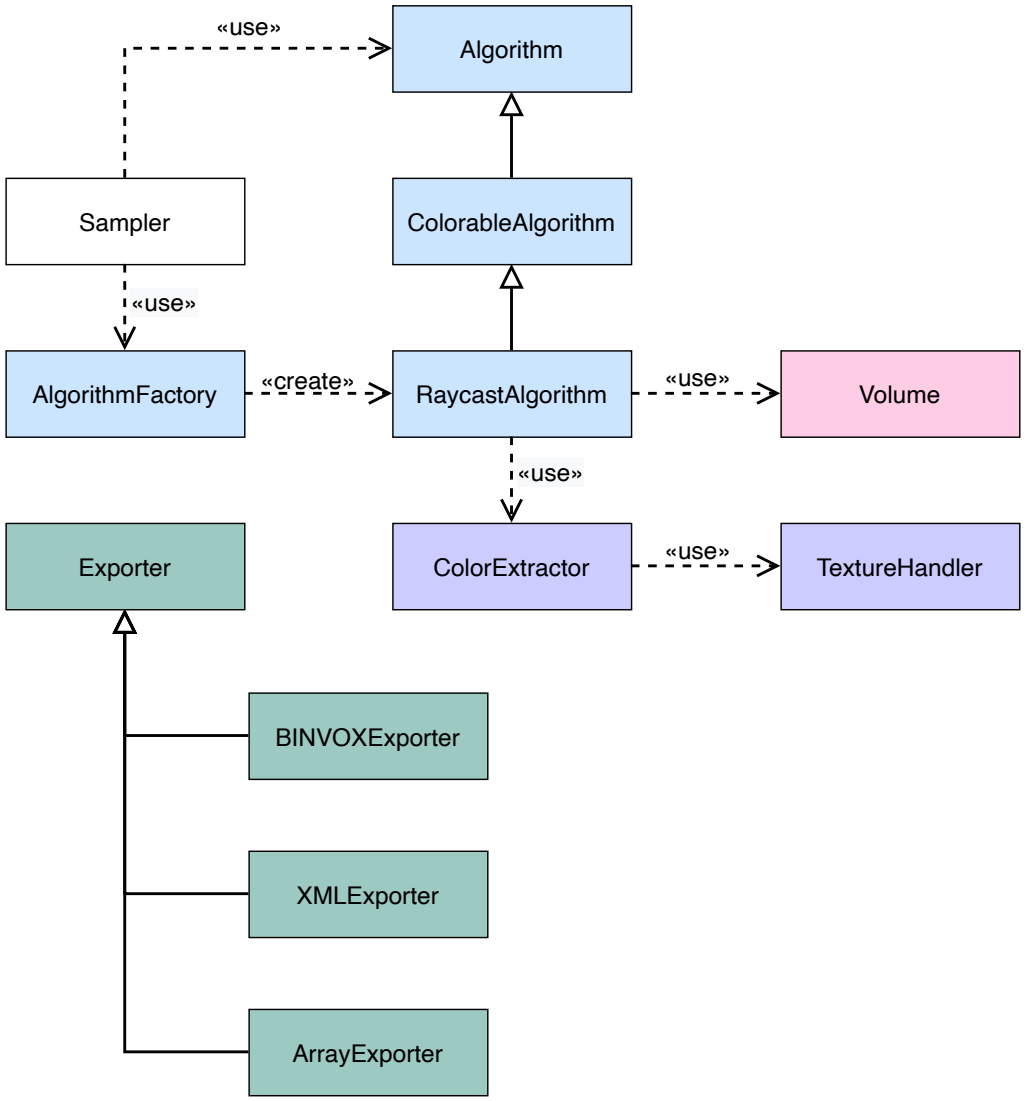


Figure 3.2: UML class diagram of the improved Voxelizer engine v1.0.0.

- **algorithms** - The algorithm module defines the algorithm system. This is in charge for the actual sampling of the 3D models.
- **color** - The coloring system is found in the color module. This manages the color extraction for supporting color voxelization.
- **volume** - The volume module mainly contains a wrapper class for providing a consistent interface for interacting with volumes throughout the application.
- **exporters** - The exporters module is made up of various exporter classes. These enables the engine to export the voxel data into many different formats.

- **utils** - This module contains various utility functions.

3.4.2 Algorithm system

Voxelizer implements an algorithm system that makes it easy to extend the voxelization support for new algorithm types, or adding new options. The system mainly consists of two base abstract algorithm classes. One for plain voxelization, and another is colorable. By simply extending the appropriate base class, a new algorithm can be defined. Further, a factory pattern has been implemented for retrieving the appropriate algorithm.

Since the generated voxel data can be huge, an efficient internal data structure needs to be used. Two main concerns need to be taken into account. The first is the memory footprint. In order to be able to do high resolution voxelizations, a limiting factor is the amount of available system memory. A second concern is speed. The JavaScript engines are able to do quite a lot of optimization. By using the JavaScript language in clever ways, quite high processing speeds can be achieved.

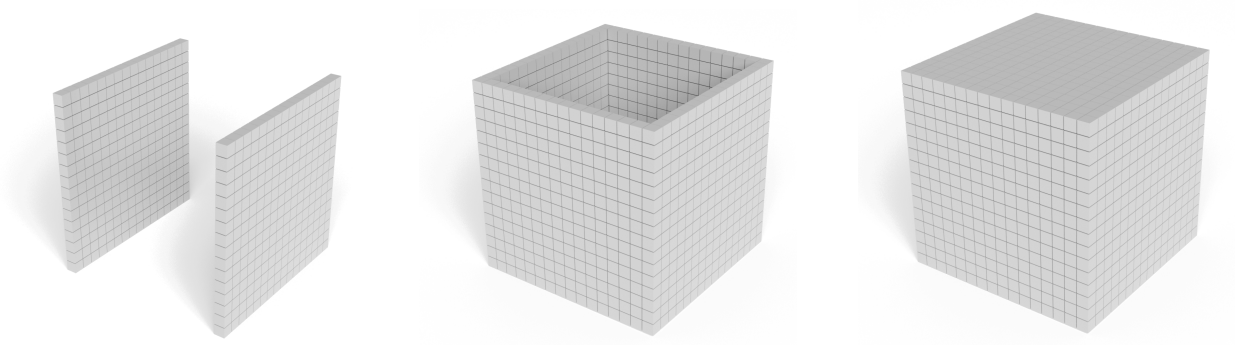
The old Voxelizer v0.1.3 used normal JavaScript arrays which was nested. These arrays grows and shrink dynamically, potentially resulting in slow performance. Although, the JavaScript engines are often able to optimize the execution quite a bit, resulting in decent speeds. However, in order to comply with both the memory and performance requirements, typed arrays have been used instead. More specifically, only one large one-dimensional typed array is used. This is a more low level data type than normal arrays, providing mechanism for reading and writing raw binary data in memory buffers. In order to support multidimensionality, the efficient third party library ndarray is used. See Section 2.10.3 for more details on ndarray.

3.4.3 Raycasting algorithm

The new and upgraded voxelization algorithm is mainly based on raycasting (see Section 2.11.2). Next, a description of how the algorithm works is provided.

First, several preparations are made on the mesh. Firstly, the mesh is centered at the origin. Then the mesh is manipulated to include both front and back faces. This ensures that raycasting against the mesh will result in a hit against both front and back sides of the model. This means

that it is not needed to raycast from all 6 sides of the mesh. Hence, the algorithm samples the mesh from the front, left and top sides. Previously, this was only done from the front and back. These results are then merged together. Further, the result is wrapped inside a Volume class and returned to the user. Figure 3.3 illustrates how these three results are merged, resulting in a complete surface shell representation.



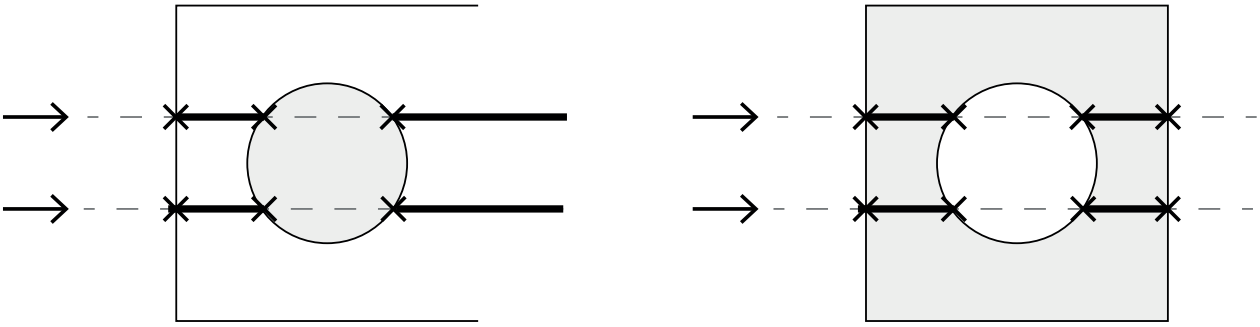
(a) Front side sample.

(b) Front and left side samples merged together.

(c) Front, left and top side samples merged together.

Figure 3.3: Merging of voxel samplings.

The algorithm also has an option for producing filled, or solid, results. This is achieved by interpreting the first raycast intersect as the surface of the object. From this point on, everything will be considered "inside" the object. When a second intersect is detected, the state is changed to be "outside" the object. A new hit would indicate "inside", and so on. This works very well with a watertight 3D model, as can be seen from Figure 3.4a. However, when trying to fill an object which is not watertight, this can result in severe inaccuracies. This can be seen in Figure 3.4b, where no boundary is provided for the rays exiting after a third hit.



(a) Non-watertight 3D model cross section.

(b) Watertight 3D model cross section.

Figure 3.4: Solid (voxelization) filling of 3D model cross section.

3.4.4 three.js optimization

The raycasting functionality, used by the raycasting algorithm described in Section 3.4.3, is supplied by the three.js library. The library provides a throughly tested and accurate raycasting solution. However, it is CPU bound and iterates every face of the mesh. This gives each raycasting operation a time complexity of $\mathcal{O}(n)$, where n is the number of triangles in the mesh. If the 3D mesh is highly detailed, containing a large amount of polygons, the raycasting will take a long time to perform. After a careful assessment of potential solutions, the three.js plugin named three-bvh-mesh [57] is used to improve this problem. This plugin provides a BVH implementation in order to speed up the raycasting against three.js meshes. See Section 2.12.2 for details on BVHs. By using this plugin, the time complexity for a single raycasting operation decreases from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$, where n is the number of triangles in the mesh. The plugin's dynamic tree generation is also used. This means that the BVH tree is gradually built. The voxelization of large resolutions would be faster if the entire tree is generate at the beginning. However, the time taken to generate the tree would outweigh the benefits if the resolution is to low. Dynamically generating the tree is therefore a good tradeoff. Figure 3.5 shows an example visualization of a BVH tree applied to a 3D model, which is generated by the three-bvh-mesh plugin.

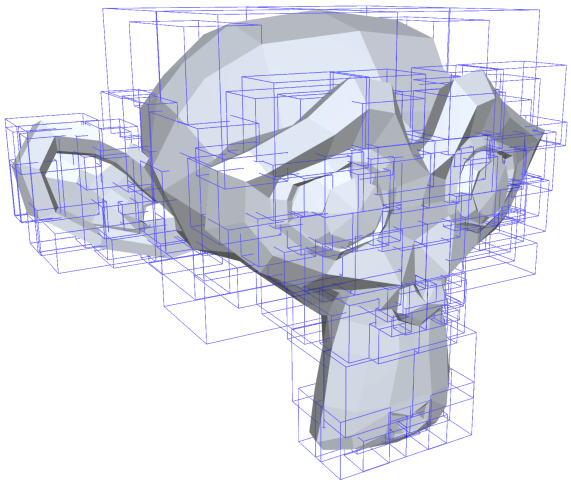


Figure 3.5: Visualization of BVH applied to 3D model of a monkey face.

3.4.5 Color system

The color system of Voxelizer handles the storing and extraction of color from polygon meshes. This enables the engine to extract color information from the 3D mesh, and apply it to the voxels. The system mainly comprises of a ColorExtractor class and a TextureHandler class. The TextureHandler class stores a copy of all the texture maps associated with a 3D object. These are stored in a hashmap. The key used for each hashmap entry is a universally unique identifier (UUID), generated by three.js for all three.js objects. Further, it is able to look up a UV coordinate of a texture map in its store, and retrieve the color. See Section 2.11.1 for more details on texture maps and UV coordinates. The ColorExtractor class provides various methods for extracting colors from from an raycast intersect. It uses the TextureHandler as internally texture storage, for fast lookup of texture colors.

During raycasting with the raycasting algorithm described in Section 3.4.3, each intersect produces an UV coordinate of the associated texture map, along with its UUID. This info is then fed to a ColorExtractor class, returning the appropriate RGB color of the intersect. This color data is then stored in a four-dimensional view of a large ndarray. For maximum efficiency and safety, the actual datatype supplied to ndarray is an Uint8ClampedArray [58] (typed array).

3.4.6 Loading

The Voxelizer engine previously made use of a wrapper OBJ loader. This resulted in very limiting compatibility with other file formats. This support has been dropped in favor of the new ES6 JS loader modules introduced by three.js. three.js supports around 40 different file formats for loading 3D models. All three.js objects inherits from a base class named Object3D. This includes meshes. By ensuring compatibility with three.js meshes, any loader compatible with three.js can be used.

3.4.7 Exporting

As described in the Sections above, the engine normally outputs ndarrays with color and voxel information, wrapped in a Volume class. However, the engine also comes with several exporter classes. This includes:

- **XML** provides a versatile and flexible file format. The native JavaScript DOM parser is used for the actual parsing of the XML data. The outputted format of the XML document structure is described on the GitHub wiki page for the engine. PROVIDE LINK HERE !!!
- **BINVOX** BINVOX [54] is one of the more popular voxel data file formats. BINVOX is the file format used by the binvox [2] voxelization software. A separate repository named binvox [55] has been created for handling BINVOX files. See Section 3.5.
- **Arrays** - An array exporter is implemented for exporting the voxel data as several nested JavaScript arrays (3D array). If the export includes color data, this is exported as a 4D JavaScript array.

3.4.8 Testing

Several unit tests are created for testing the different parts of the voxelization system. This ensures correct operation of the voxelization process, and protect against introducing new bugs. Jest [59] has been chosen as the testing framework provider. Jest also provides coverage reports. These are very valuable in terms of analyzing what parts of the system is and is not tested.

3.4.9 Migration

The previous version of Voxelizer was version v0.1.3. Following Semantic Versioning [60], or SemVer, the old version of Voxelizer is defined as still in Beta. Introducing a new Major version of the library with breaking functionality is therefore no problem. Still, a very simple migration guide is provided on the Wiki of the Voxelizer engine repository on GitHub LINK TO GITHUB WIKI HERE!!!!.

3.4.10 Debugging and Profiling

During development, the three-voxel-loader plugin was used to visualize the actual voxel outputs. This made it easy to visually inspect the results of the voxelization algorithm. A similar solution to the debugging setup used for the three-voxel-loader plugin was used. Likewise, this also resulted in an example usage of the engine, and is deployed to GitHub Pages. LINK TO VOXELIZER EXAMPLE PAGE.

For assessing the memory consumption and speed of the engine, the performance tool [61] in the Google Chrome Developer Tools is used. This helped removing some CPU-heavy bugs, memory issues and other performance bottlenecks. Figure 3.6 shows a screenshot of the performance tool in action. In the middle section of the image, the execution time for the various methods in the main thread are displayed. In the lower part of the screenshot, a memory consumption graph is displayed.

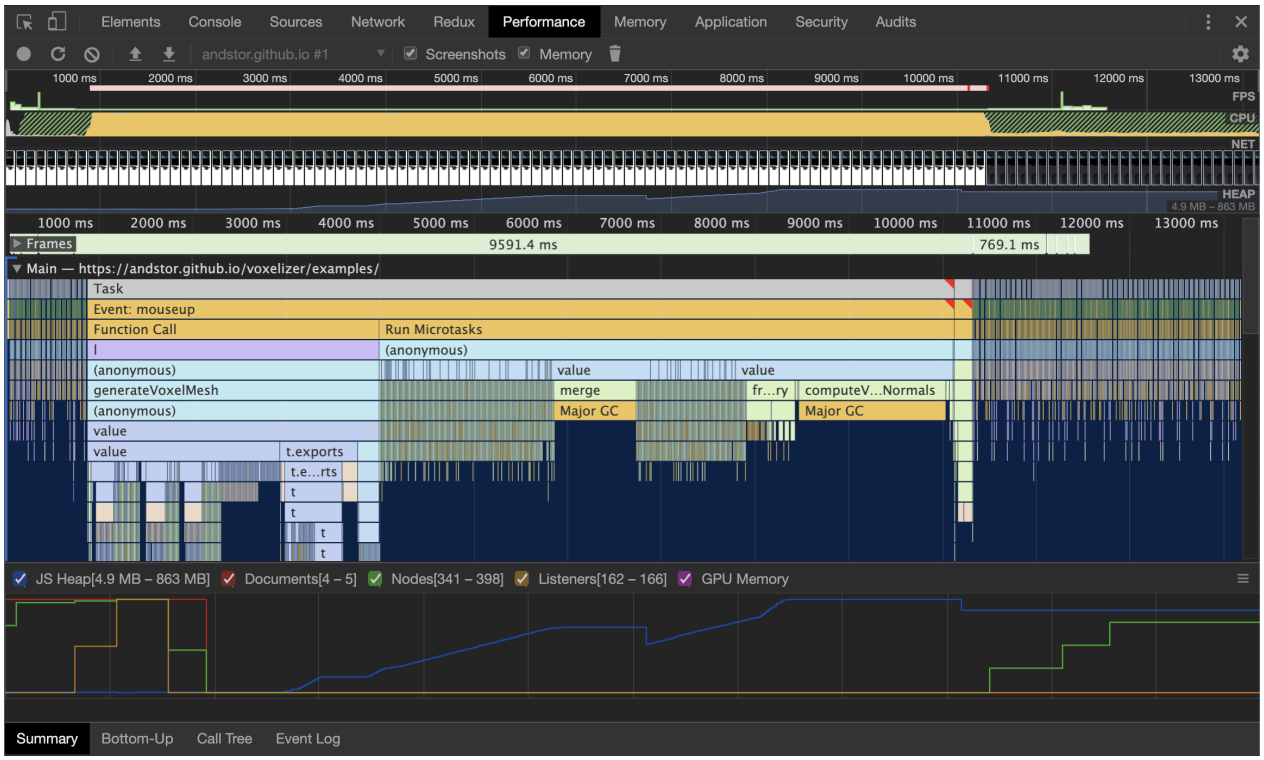


Figure 3.6: Screenshot of performance profiling with Google Chrome Developer Tools.

3.4.11 Building

The engine is bundled and built with Webpack. This gives great control over the building process. See Section 2.6.3 for more details on Webpack. Voxelizer also makes use of ES6 features, so all source files are also transpiled to ES5 with babel. The main output of the library is UMD, as described in Section 2.6.1. This makes the engine compatible with a range of other module systems. However, since Webpack does not support ES Modules right out of the box, a better alternative would be to use Rollup 2.6.3. Unfortunately, this was not possible due to a circular

dependency in one of the dependencies of Voxelizer. Therefore, in order to use Voxelizer as a native ES Module, a project also needs to use a module loader like Webpack.

3.5 BINVOX

A separate open-source repo for building and parsing BINVOX file formats were created during refactoring of the Voxelizer engine and the three-voxel-loader plugin. It is named BINVOX and licensed under the MIT license. The BINVOX file format consists of a header in plain ASCII, followed by binary data. The binary data is compressed using run-length encoding. From the BINVOX specification: "The binary data consists of pairs of bytes. The first byte of each pair is the value byte and is either 0 or 1 (1 signifies the presence of a voxel). The second byte is the count byte and specifies how many times the preceding voxel value should be repeated (so obviously the minimum count is 1, and the maximum is 255)." (Min [54]).

The new binvox package provides tools to both parse and construct files according to the BINVOX file specification [54]. Parsing is done on the individual set of two bits. The data is uncompressed and stored in a JSON format. An example of the parsed JSON result can be seen in Listing 3.1. Similarly, a user can supply the same JSON data structure for constructing a BINVOX file resource. Hence, parsing and building commute.

Listing 3.1: Example BINVOX data in JSON format

```
{
    "dimension": {
         "depth": 32, "width": 32, "height": 32
    },
    "translate": {
        "depth": 11.81, "width": 21.39, "height": -1.69
    },
    "scale": 30.206,
    "voxels": [
        { "x": 0, "y": 2, "z": 3 },
        { "x": 0, "y": 3, "z": 3 },
        { "x": 0, "y": 4, "z": 3 },
        ...,
    ]
}
```

3.6 Voxelizer Desktop

Voxelizer Desktop is a desktop application for voxelizing 3D models. The next few sections goes through how the application is developed.

3.6.1 Design

Wireframes diagrams are created for planning the application's GUI. Figure 3.7 shows a wireframe diagram over the start screen. Here, a drag and drop interface is presented. The user should be able to just drag and drop the files he/she wants to voxelize into the application. Then, the application will need to load the 3D file(s). Figure 3.8 shows how this should be conveyed to the user, by using some form of loading graphic. Finally, Figure 3.9 shows the main screen. This interface is divided in two parts. In the lower half of the application window, various controls are presented. This is further divided into a *Settings* section, and a *Exporting* section. The settings section provides controls for the actual voxelization settings. This includes settings for resolution, coloring and solid/shell voxelization. The exporting section provides a UI for selecting

the exporting file format, and saving it as a file. In the upper half, an interactive widow should display the voxelized result of the voxelization.

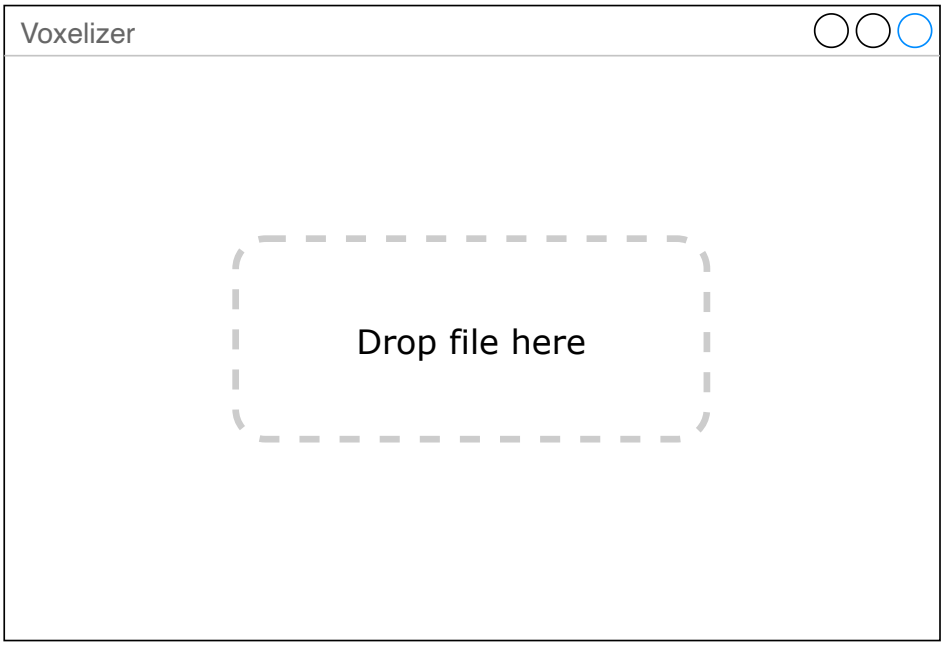


Figure 3.7: Wireframe diagram of drag and drop start screen.

3.6.2 Implementation

The Voxelizer Desktop application is built with the Electron framework, described in Section 2.10.4. Alongside the bare Electron framework, the electron-builder [62] package is used. This package provides a complete solution for packaging and building a distribution ready Electron app for macOS, Windows and Linux. It also provides "auto update" out of the box. For creating the actual GUI, the React library is used. The create-react-app [63] is used for bootstrapping the react project.

For providing the voxelization, the Voxelizer engine is used. For visualization of the voxelized result, the three-voxel-loader plugin is used to generate a 3D mesh. This is then passed on to three.js.

In order to make the application user-friendly, a translation system is also implemented. This system is mainly backed by the react-intl package [64]. During application startup, the language setting (locale) on the users computer is read in the application's main process. This is

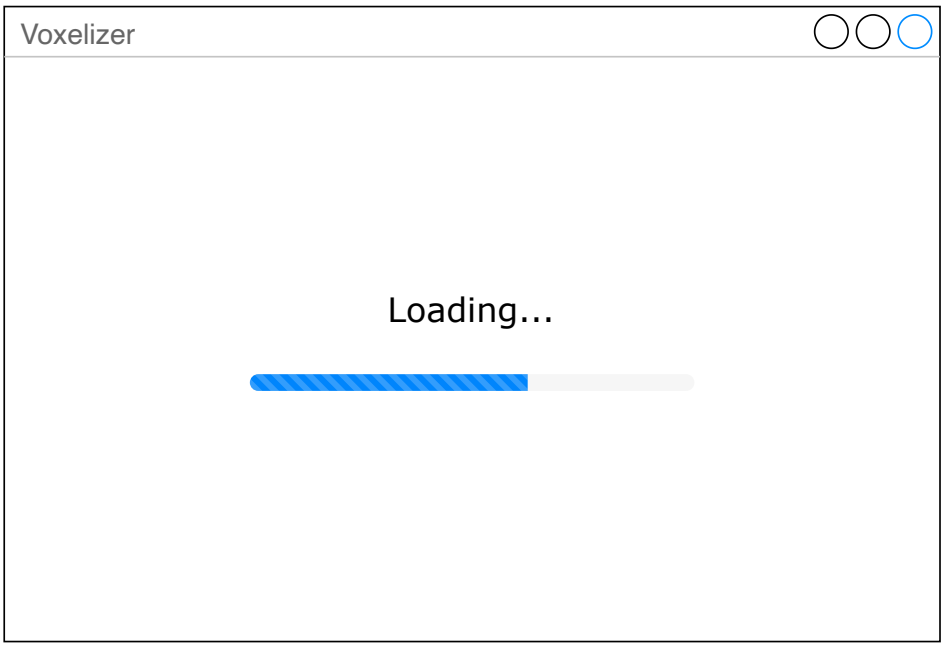


Figure 3.8: Wireframe diagram of loading screen.

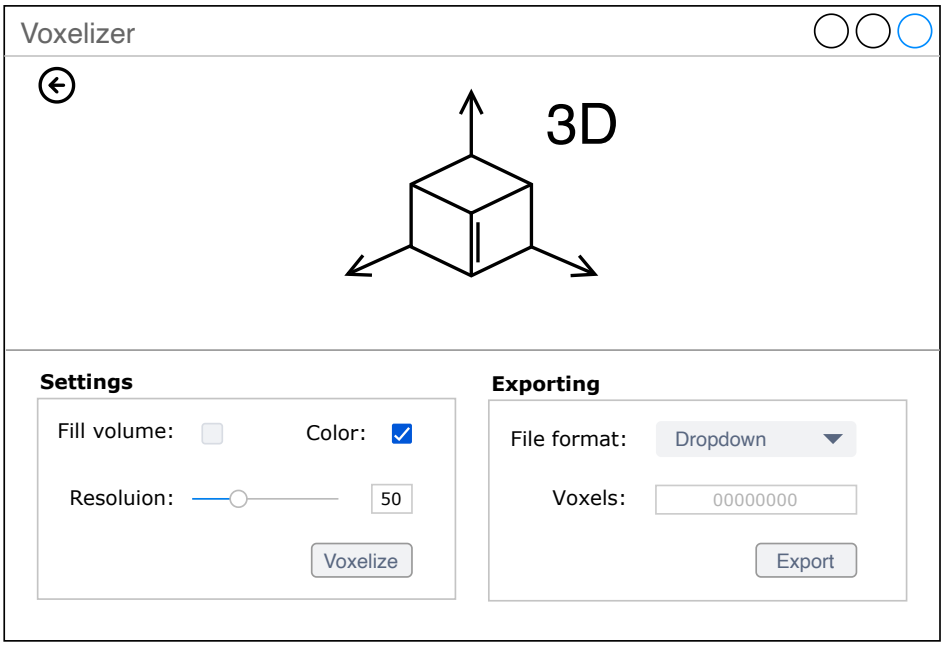


Figure 3.9: Wireframe diagram of the main screen.

then passed on to the instantiated renderer process through IPC. Based on this locale, the GUI is displayed in the appropriate language is presented.

For presenting warning or error messages to the users, modal windows are used. However, these can only be generated from the main process. An IPC system for generating modals from a renderer process is therefore implemented.

3.6.3 Releases

When it is time for a new release, application builds for both macOS, Windows and Linux are generated with the help of the electron-builder package [1]. These are then uploaded to GitHub as "release assets". This enables the auto updating functionality to check for new versions. If there exists a newer version than currently installed, the application will download the newest version from GitHub. Then, the next time the application starts up, the downloaded update will be installed.

3.7 JSDoc Action

In the next couple of subsections, the implementation and design decisions of the JSDoc Action are be presented.

3.7.1 Implementation

For creating a GitHub Action, several mandatory files and configurations has to be created. This documentation is available at GitHub. GitHub Actions provides two main types of action. One is JavaScript based. The other is based on Docker. They both have their advantages and disadvantages. A Docker container action provides an isolated environment, providing extremely flexible and stabile solutions. However, GitHub only supports running docker actions on Linux. On the other hand, a JavaScript GitHub Action can be run directly on a runner machine. This makes it a lot faster than a Docker based Action. This is because of latency due to retrieve and build the container. A JavaScript action is also cross platform compatible, meaning it can run on both

[1] Some build targets can only be generated on specific platforms.

a Linux, Windows or MacOS operating system. In order to provide a fast and cross compatible solution, the JavaScript action type is chosen for the JSDoc Action.

The JSDoc Action is made up of mainly two parts. One is the template installation system, and the other is the actual execution of the JSDoc tool. For providing maximum flexibility, all functions of the JSDoc tool is made available as input configurations through the GitHub Actions Workflow API. The action is also able to use a JSDoc configuration file [65]. The reader can consult JSDoc Action's README.md file for further details on the available input variables. If a user provides a template to be installed, the action will first install this. This is done with the help of the node package manager (npm). The supplied template can be everything from a GitHub repository, to an npm package. The installed template files are then processed. The action does all of its input/output (IO) operations asynchronously, ensuring fast execution speed of the action. When finished, a JSDoc CLI command is then formed based on the various user inputs and (if provided) config file. This command is then executed, effectively telling JSDoc to generate the documentation. The result is a user-defined output folder with the generated API documentation. The entire flow of the action can be seen in Figure 3.10 which provides a flowchart diagram of the JSDoc Action.

3.7.2 Usage

For actually using the action in a Workflow, the action makes a couple of assumptions. Firstly, the actual source files to generate documentation from, needs to be supplied. This is normally solved by using the Checkout Action [66] made by GitHub. Secondly, the JSDoc Action simply generates an output directory with files. This means that nearly any deployment action can be used for upload the files the desired service, for example GitHub pages. The deployment action supplied as example in the README.md file of the repository is named GitHub Pages action [67].

3.7.3 Feedback

The JSDoc Action generated quite a lot of feedback from eager users of GitHub Actions. Several wanted to test the action, and multiple issues were filed in the issue tracker on GitHub [68]. See for example issue #20 [69]. Since maintenance is essential to the success of an open-source

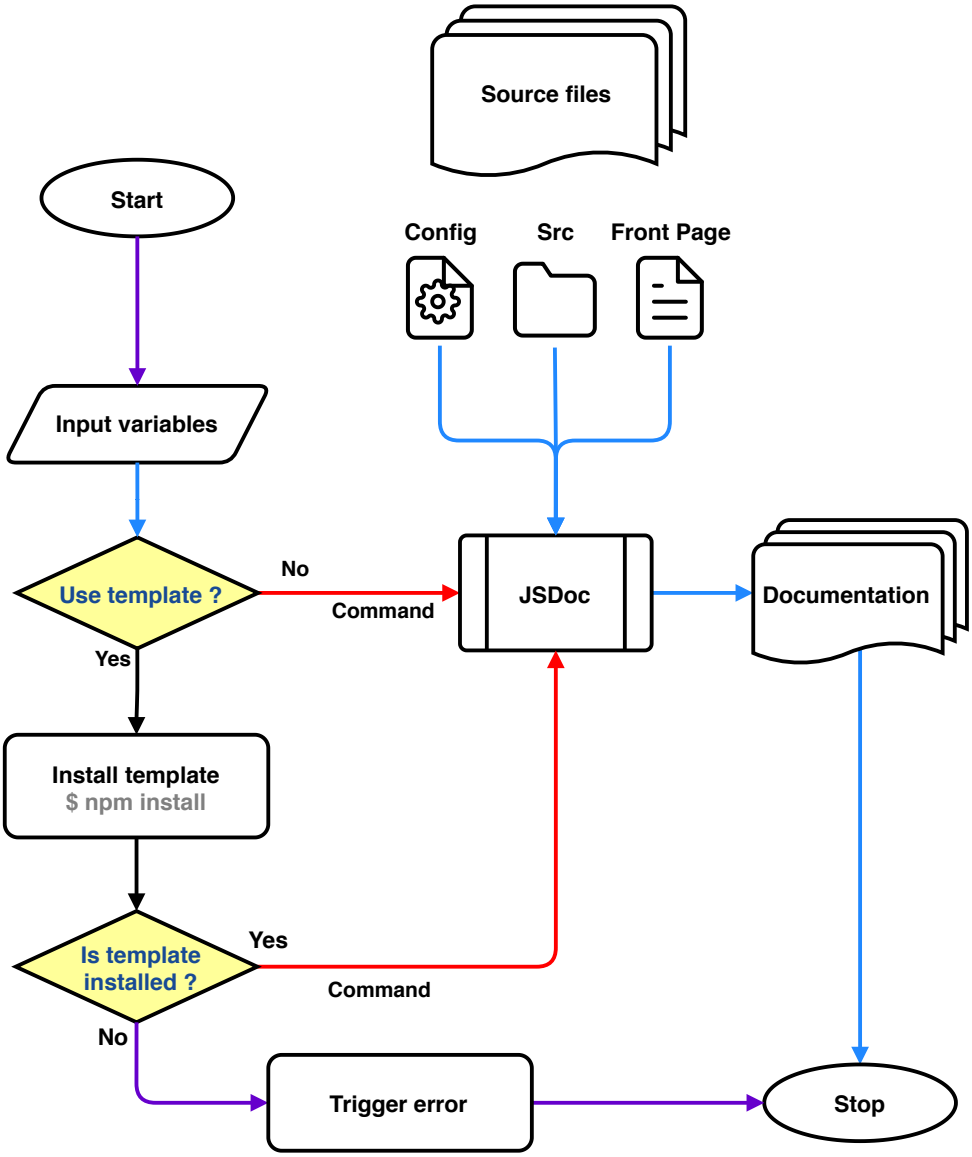


Figure 3.10: JSDoc Action flowchart diagram.

project, all feedback were responded to and handled accordingly. Alongside the development of the other main projects, many bug-fixes were made to the JSDoc Action. Eventually, all issues were resolved, resulting in several happy users.

3.8 Automation

Automation is an important part for ensuring both efficiency and security. The next subsections will contain a description of how the various open-source systems developed in connection with this thesis have been automated.

3.8.1 JavaScript package workflows

GitHub provides continuous integration (CI) and continuous development (CD) as a part of its GitHub Actions system. This system is heavily used in this thesis. Several workflows are created in order to automate various tasks like testing, building, documentation generation and publishing. The workflows have been made so that they should work out of the box for similar projects. See Section 2.3.1 for a description of GitHub Actions. Do note that the default behavior of Workflows is to terminate if any errors are encountered.

Figure 3.11 shows a simplified diagram of the CI/CD automation pipelines created for the JavaScript packages which are to run on new contributions to the codebase. This system mainly consists of two workflows. One for building the package, and one for generating and publishing API documentation. A simple step by step walkthrough of the system is now presented.

1. **Pull request** - The pipelines are mainly triggered by a pull request. This starts up both a security analysis by LGTM and a build workflow.

2. **Security analysis** - As described in Section 2.10.6, LGTM provides a security analysis.

   (a) **Checkout** - First, the repository in which the automation is done on is cloned.

   (b) **Build** - Then, the JavaScript project is built.

   (c) **Test** - If the project provides tests, these are also run.

   (d) **Coverage** - Coverage reports are created with Jest.

3. **Coveralls** - The coverage report is then published to Coveralls.io. See Section 2.10.7 for details on Coveralls.

If the workflow finishes, the security analysis comes out clear, and the coverage percentage is not decreased, the pull request is approved for merging. When a user with the appropriate privileges approves the pull request, the code is merged into the base branch. If the base branch is the master branch, a second workflow is started. This is a workflow for generating/updating the API documentation.

1. **Checkout** - The source repo is cloned once again.
2. **JSDoc Action** - The JSDoc Action is then used for generating JSDoc API documentation.
3. **GitHub Pages** - The outputted documentation is then finally deployed to GitHub Pages with an action called GitHub Pages action [67].

In order to automate the release process of a JavaScript package, a third workflow is used. This is shown in Figure 3.12. It mainly involves packaging and publication of the software to the npm registry [70]. The workflow functions like this:

1. **Trigger release** - First, a user with the appropriate privileges needs to create a release manually on GitHub. This generates a git tag. Further, the release starts up the package workflow.
2. **Package workflow** - The package workflow has four steps.
3. (a) **Checkout** - The source repo is cloned once again.
   (b) **Build** - The JavaScript project is then built.
   (c) **Test** - If the project provides tests, these are also run.
   (d) **Package** - If all the above tests are completed without errors, the project is then packaged.
4. **Publish package** - If the package workflow completes, the new package is then published to the npm registry.

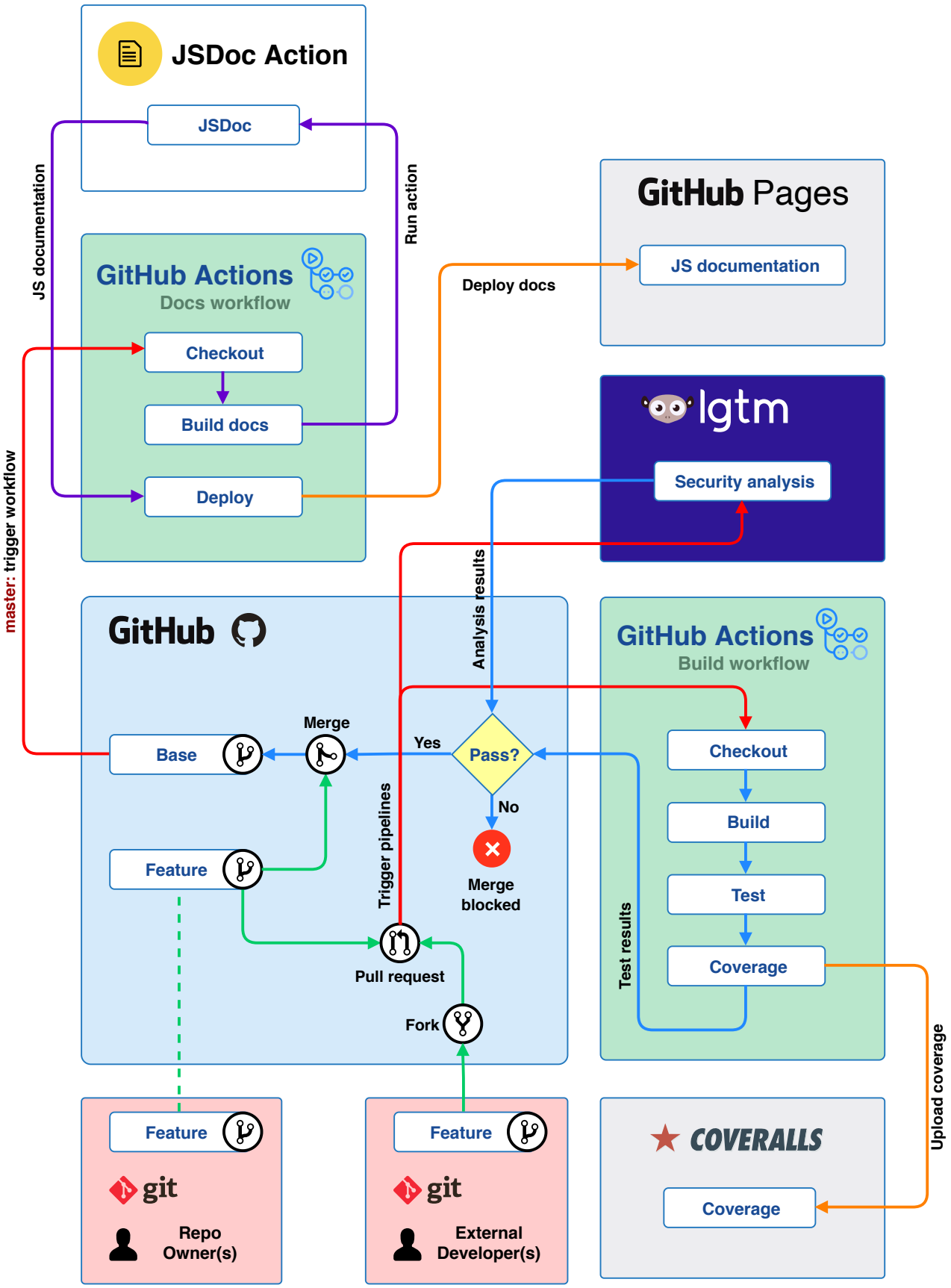


Figure 3.11: CI/CD pipelines

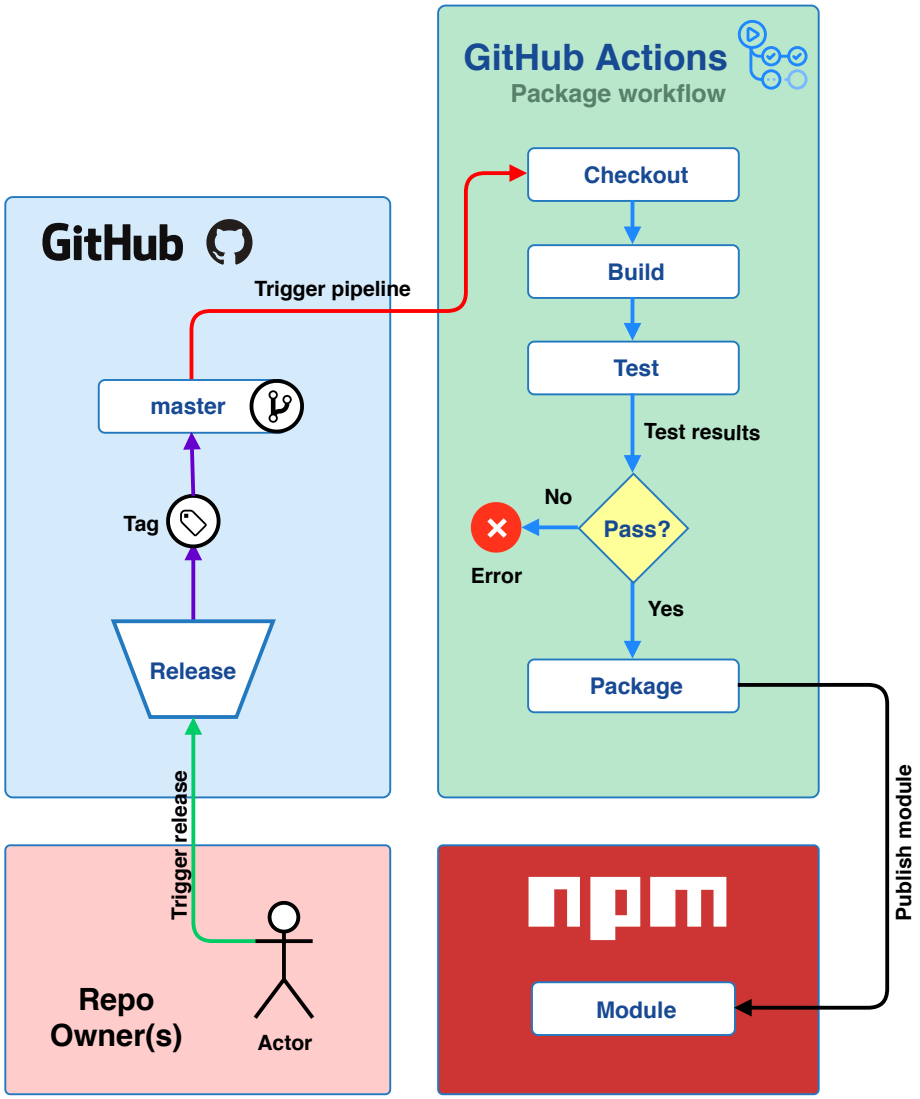


Figure 3.12: Automation of release publishing process.

3.8.2 GitHub Action version tagging

GitHub provides guidelines in terms of versioning and tagging of a GitHub Actions [71]. According to Github, releases should be following Semantic Versioning. When a new release is made, the major tag (v1, v2, etc.) should be moved to point on the Git ref of the current release. Figure 3.13 provides an illustration of the moving of Git tags. This process is tedious. An action named actions-tagger [72] already provides support for this tagging scheme. A workflow based on the actions-tagger action is therefore created, automating this release process.

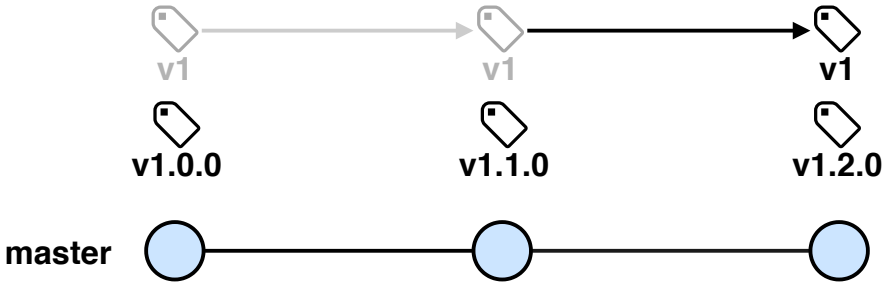


Figure 3.13: Automatic updating of major version tag.

3.9 file-existence-action

In order to be able to produce the general purpose workflows described in Section 3.8.1, it became apparent that it is needed to be able to check if a file existed. Normally, a coverage script is defined in the projects package.json file. This runs all the tests and produces a coverage report file. However, if no such script is provided, the workflow will run the tests manually instead, and no file will be generated. If no such file is generated, this needs to be detected by the workflow, in order to avoid running the upload to Coveralls.io step. This was solved by creating a relatively small GitHub Action, named File Existence. The action is written in TypeScript, and based on a template provided by GitHub. The action is able to check one or more paths for the existence of a file. A boolean result is then available to the subsequent steps in the workflow.

3.10 file-reader-action

Following the problems described in Section 3.9 above, another problem arises if the coverage script is defined, but no tests are created. If this is the case, an empty coverage file is created. Trying to upload this to Coveralls.io results in an error. The need for reading the contents of a file was therefore necessary, in order to check if the coverage file is empty. If it is empty, the Coveralls step should not be executed. The solution was to create a small GitHub Action, named File Reader. The action is written in TypeScript, and based on a template provided by GitHub. The action is able to read the contents of a file at a path supplied by the user. The output is then available to the subsequent workflow steps.

3.11 3D models

For testing the various systems, several 3D models are used. These are created with the open-source 3D modeling software Blender. The torus is a default shape in Blender. So is the monkey face model. The monkey model is named Suzanne, and often used for testing purposes. In addition to these two 3D models, a third one is developed for the purpose of testing the coloring system of the Voxelizer engine. This is a 3D model of an anvil. It is built from the ground up. A procedural generated texture is created for the model. The shader setup can be seen in Figure 3.14 In order to actually apply the generated texture to the 3D model, the model is UV unwrapped. See Section 2.11.1 for more details on texture maps. Lastly, for performance and reproducibility, the texture has been baked onto an image. The final baked image is shown in Figure 3.15.

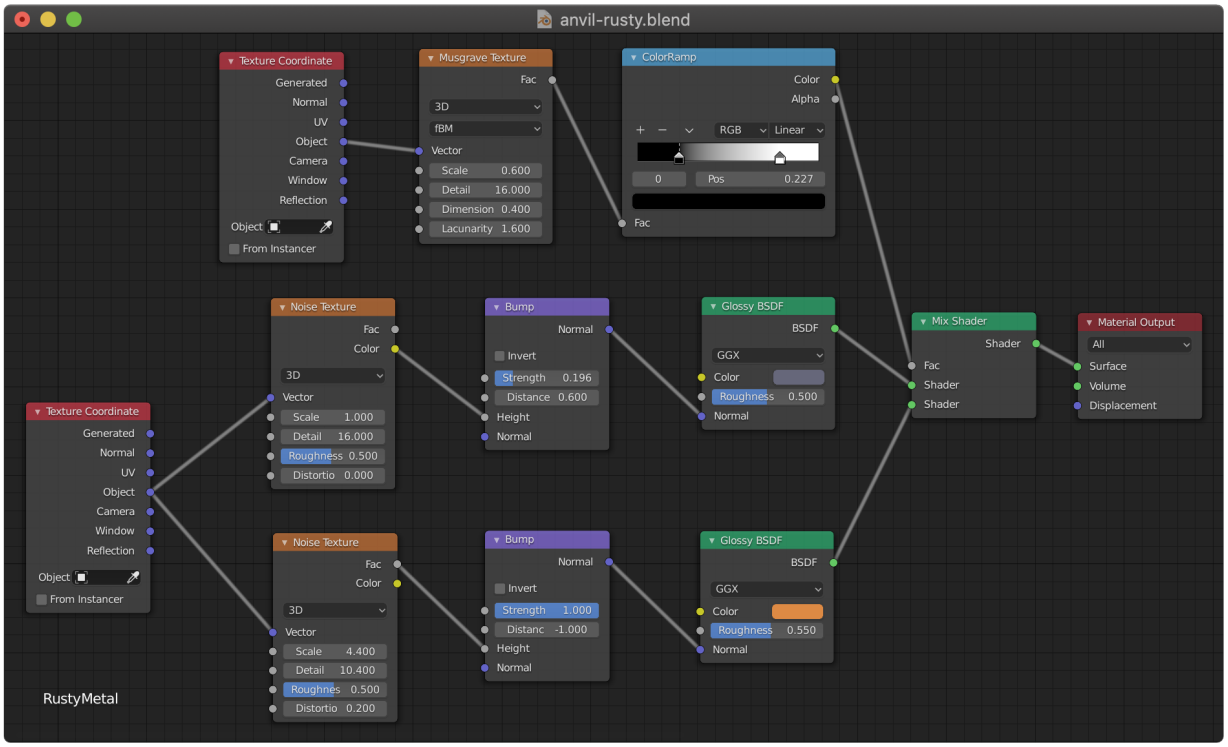


Figure 3.14: Procedurally generation of rusty metal texture.

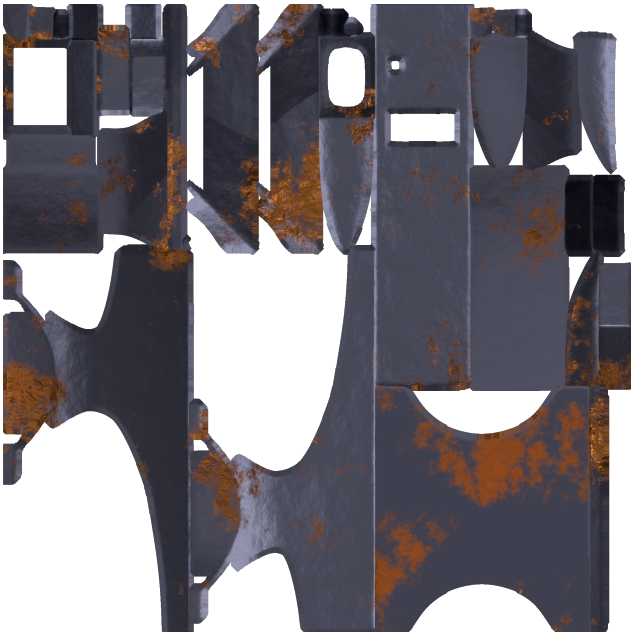


Figure 3.15: Baked rusty metal texture.

Chapter 4

Result

This chapter presents the results of this thesis. The chapter starts with presenting the different JavaScript projects developed and improved in connection with this thesis. First, the new three-voxel-loader three.js plugin, and the improvements of the Voxelizer engine are presented. This is followed by the new BINVOX package, and the desktop application for the Voxelizer engine. Then, the results of the JSDoc GitHub Action is presented. Section 4.8 will present the results in terms of automation of the projects. Finally, Section 4.9 presents some achievements that are accomplished, including some statistics on the popularity of the projects.

4.1 three-voxel-loader

The three-voxel-loader is a plugin for three.js. It is published to the npm registry under the name "three-voxel-loader", and the source code is available at GitHub under "andstor/three-voxel-loader".

The plugin is able to generate a three.js mesh based on voxel data in a variety of data formats. Figure 4.1 shows a screenshot of a VOX model loaded with the three-voxel-loader plugin. It is also possible to control the size of the individual voxels. This is shown in Figure 4.2. The size can be any number between $\langle 0, 1]$.

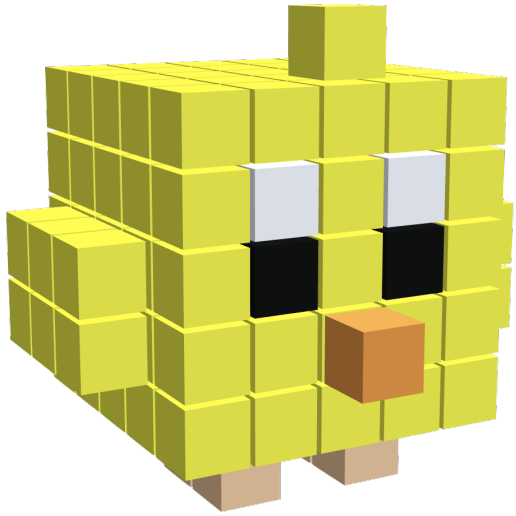
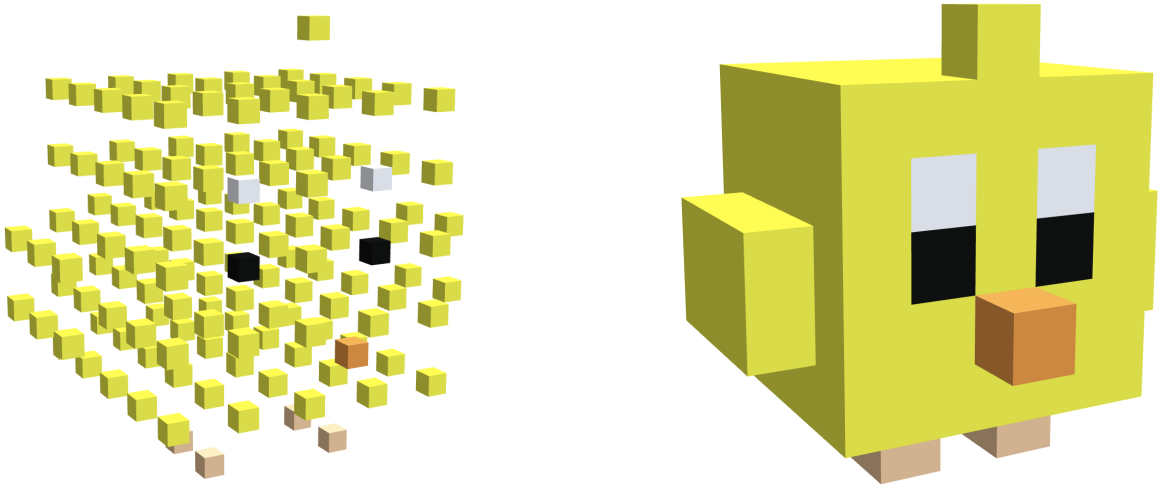


Figure 4.1: Screenshot of Chicken stored in VOX file format loaded with the three-voxel-loader plugin.



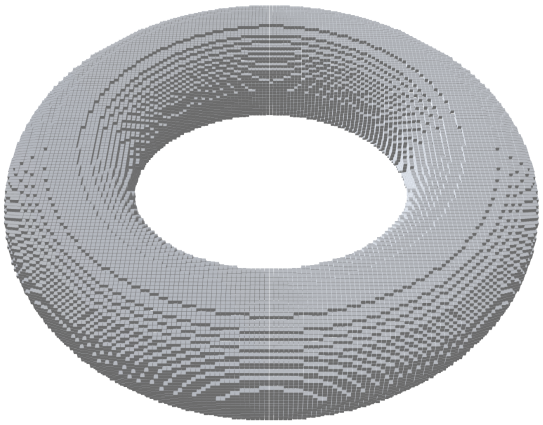
(a) Voxel size set to 0.3.

(b) Voxel size set to 1.

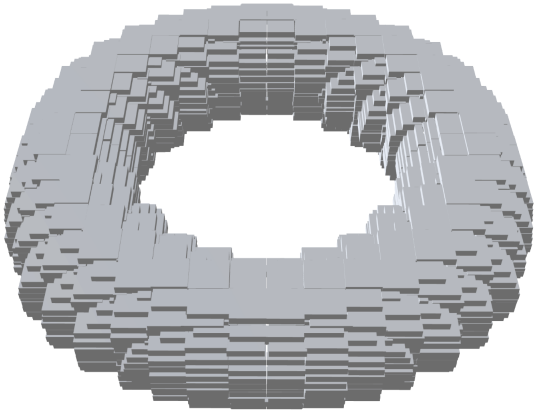
Figure 4.2: Generated meshes with different voxel sizes.

4.1.1 Level Of Detail

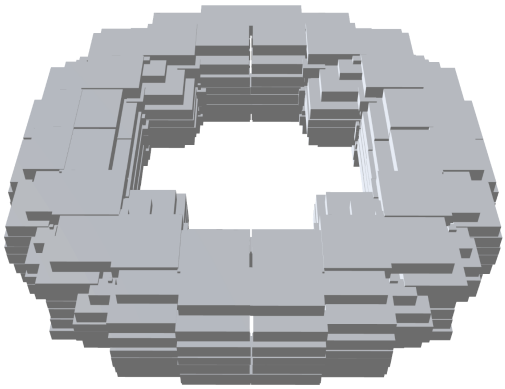
The plugin implements a Level Of Detailing (LOD) system. Figure 4.3 shows four images of a torus loaded with different LODs. Depending of the resolution of the model, a very high number of triangles may be generated. By using the LOD system, this number can be drastically reduced. For example, the mesh in Figure 4.3a consists of 406,068 triangles. By setting the LOD (maxDepth) to 2, the number is reduced to only 768 triangles, still preserving the shape. This can be seen in Figure 4.3d.



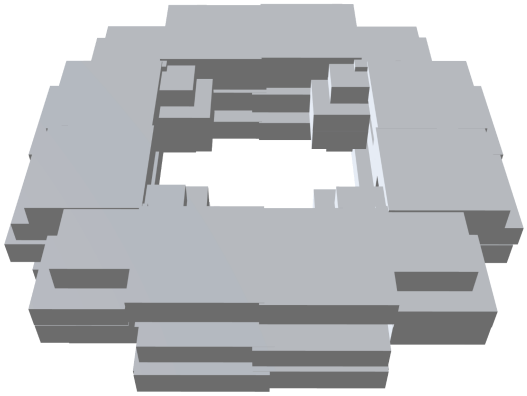
(a) Full resolution torus mesh (406068 triangles).



(b) Simplified torus mesh (19740 triangles) with a LOD (maxDepth) of 4.



(c) Quite simplified torus mesh (4752 triangles) with a LOD (maxDepth) of 3.



(d) Very low detail torus mesh (768 triangles) with a LOD (maxDepth) of 2.

Figure 4.3: Torus meshes with diffrent LOD levels.

4.1.2 Loading support

The plugin is able to load a variety of different voxel data formats. This includes XML, VOX, BINVOX and JavaScript arrays (3D array). In addition to the raw voxel data, the three-voxel-loader plugin also supports color. The data formats that support this is XML, VOX and JavaScript arrays (4D array for color data).

4.1.3 Example

An example of the plugin is deployed to GitHub Pages. Visit `https://andstor.github.io/three-voxel-loader/examples/` to see the example. Figure 4.4 shows a screenshot of the site. The example includes a GUI with controls for changing, among other things, the voxel size and the LOD.

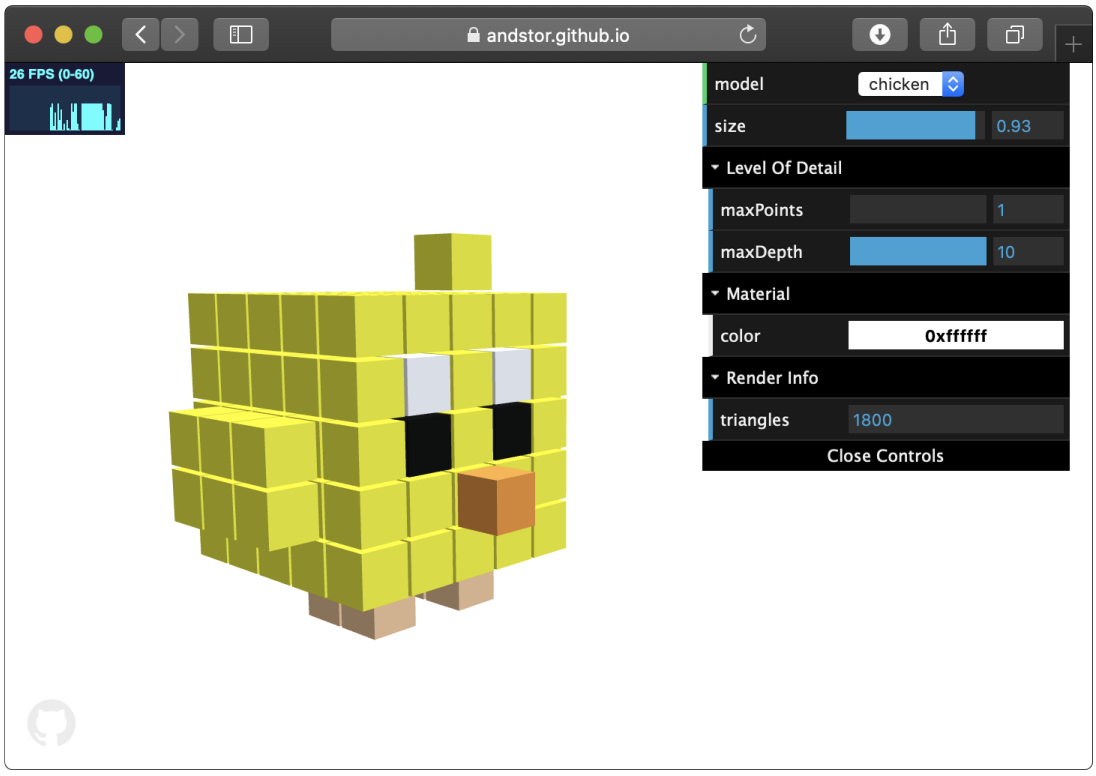


Figure 4.4: Screenshot of the three-voxel-loader example page at GitHub Pages.

4.2 Voxelizer

The new Voxelizer engine version v1.0.0 is greatly improved, compared to the old version v0.1.3. The engine is completly redesigned, and resolves all known problems and bugs the old engine had. Further, several new features are implemented, making the engine even more powerfull. Among the new features are support for coloring and shell voxelization, in addition to several new exporting formats. The performance of the engine is also drastically improved. Other improvements includes both code quality and documentation. In order to provide the engine with an own identity, a new logo for the engine is created, as shown in Figure 4.5.

Figure 4.5: Logo for the Voxelizer engine v1.0.0.

The new version is published to the npm registry, still under the name "voxelizer". The source code is available at GitHub under "andstor/voxelizer". Here, the old version is also available, tagged with the tag "v0.1.3".

4.2.1 Voxelization

The new version of the Vexelizer engine is greatly improved. The engine captures a lot more deatils of the 3D models. It is much more stabile, and produces a lot more consistent results. The engine also provides several new voxelization features. This includes both shell voxelization, and coloring support. Figure 4.6 shows a rendered image of an anvil 3D model developed for testing purposes. The anvil has a blue-ish metallic surface, with several red/gold rust spots. Figure 4.7 shows a colored voxelization of this 3D model at a resolution of 2^7. The individual

rust spots are clearly seen, and the overall result seems to be correlating with the original 3D model. Further, in order to showcase the shell voxelization, the voxelization result is cut in half. This is seen in Figure 4.8.

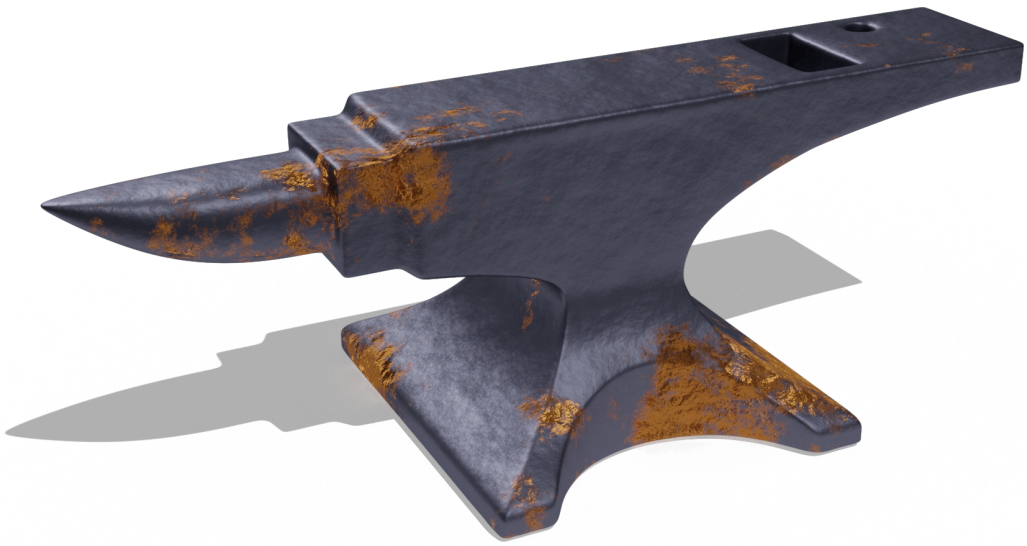


Figure 4.6: Render of textured anvil 3D model.

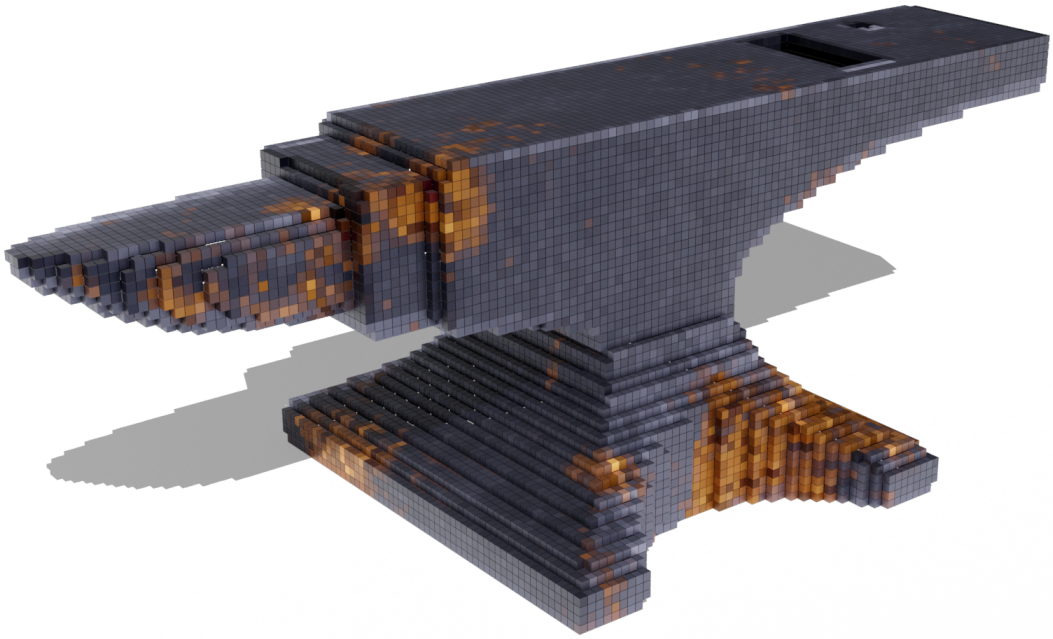


Figure 4.7: Colored voxelization (resolution of 2^7) of anvil 3D model.

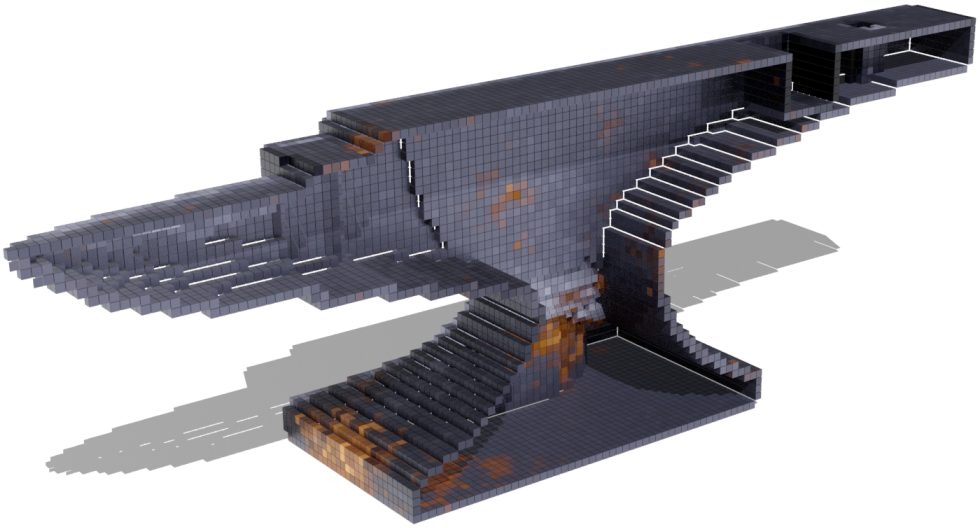
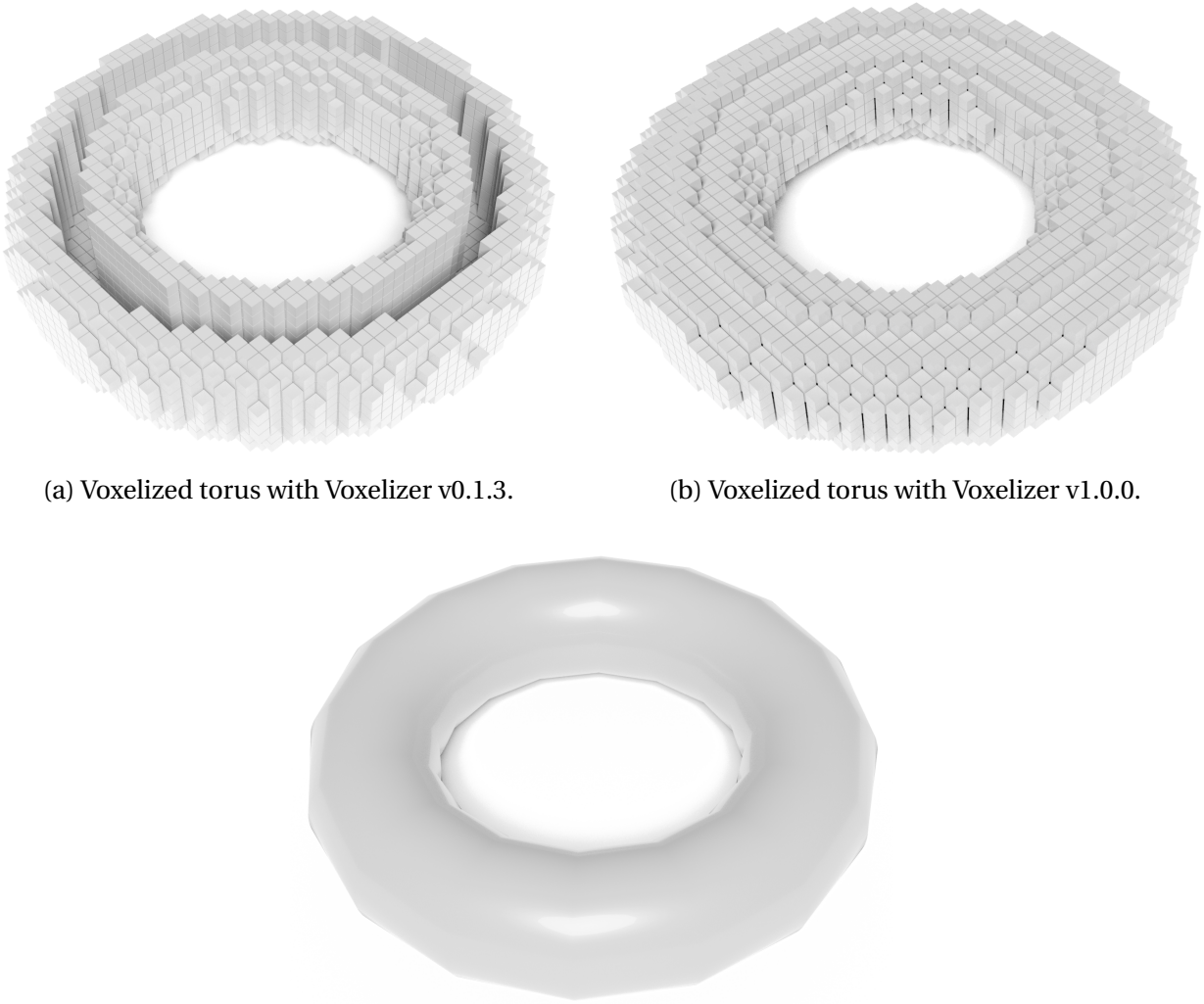


Figure 4.8: Colored voxelization (resolution of 2^7) of anvil 3D model cut in half.

4.2.2 Visual assesment

In this section, the various improvements of the problems the old engine had is presented. Fistly, the old engine produced several holes. The new version fixes this problem. This can be seen in Figure 4.9, which shows the voxelization of a torus 3D model. Figure 4.9a shows the voxelization of a torus with the old version, and Figure 4.9b shows the result with the new version. Comparing the two, it is clear that the holes are no longer present.



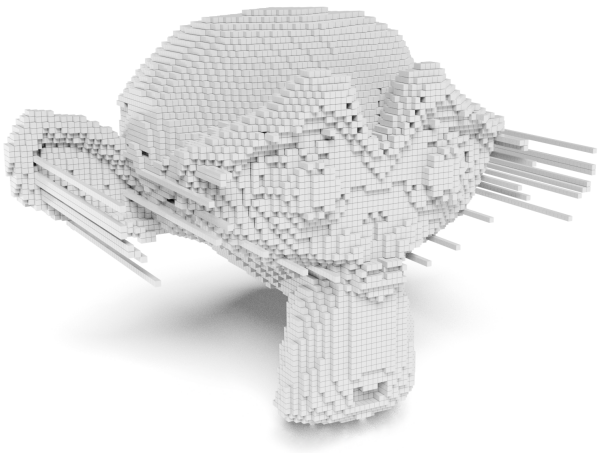
(a) Voxelized torus with Voxelizer v0.1.3.

(b) Voxelized torus with Voxelizer v1.0.0.

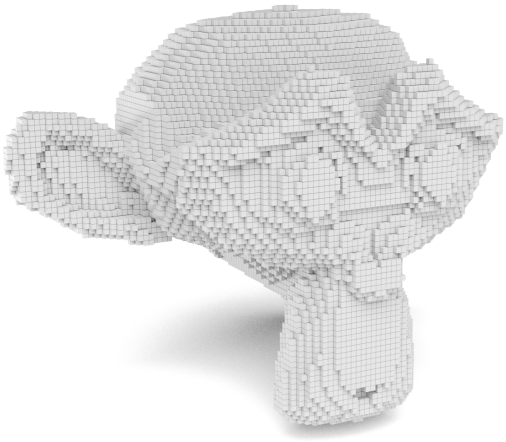
(c) Original torus 3D model.

Figure 4.9: Voxelization of a torus with Voxelizer v0.1.3 and v1.0.0. The voxelization is done with a resolution of 40.

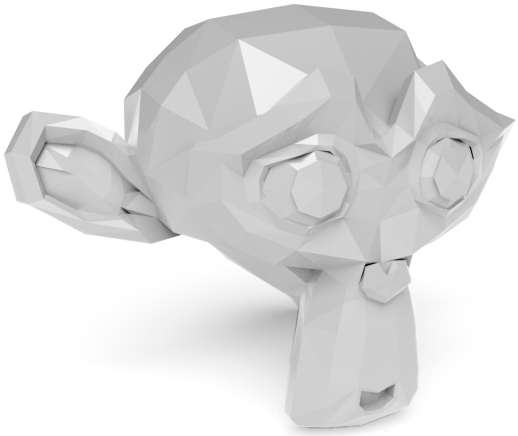
As mentioned in Section 2.14, the old engine produced a lot of artifacts. The new version fixes this problem. This can be seen in Figure 4.10, which shows the voxelization of a monkey 3D model. Figure 4.10a shows the voxelization of a monkey with the old version, and Figure 4.10b shows the result with the new version. Comparing the two, it is clear that the new engine produces voxelizations where the artifacts are more or less completly removed.



(a) Voxelized monkey with Voxelizer v0.1.3.



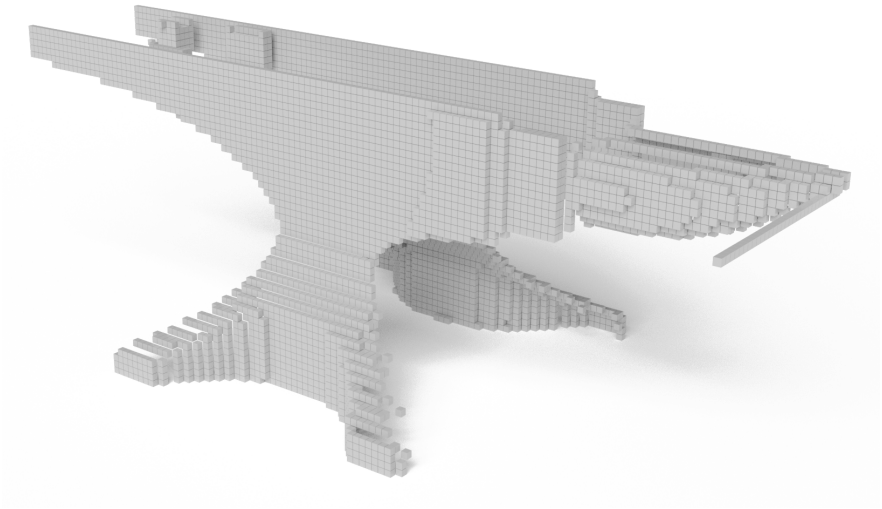
(b) Voxelized monkey with Voxelizer v1.0.0.



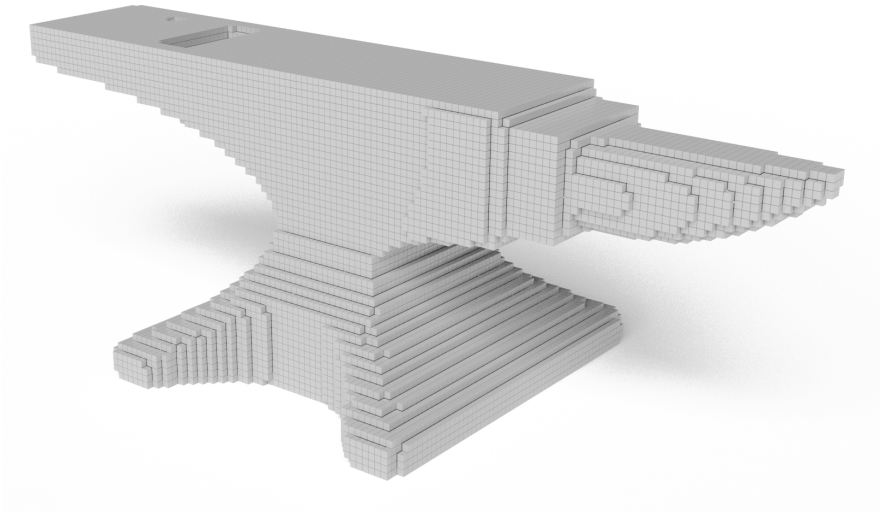
(c) Original monkey 3D model.

Figure 4.10: Voxelization of a monkey with Voxelizer v0.1.3 and v1.0.0. The voxelization is done with a resolution of 100.

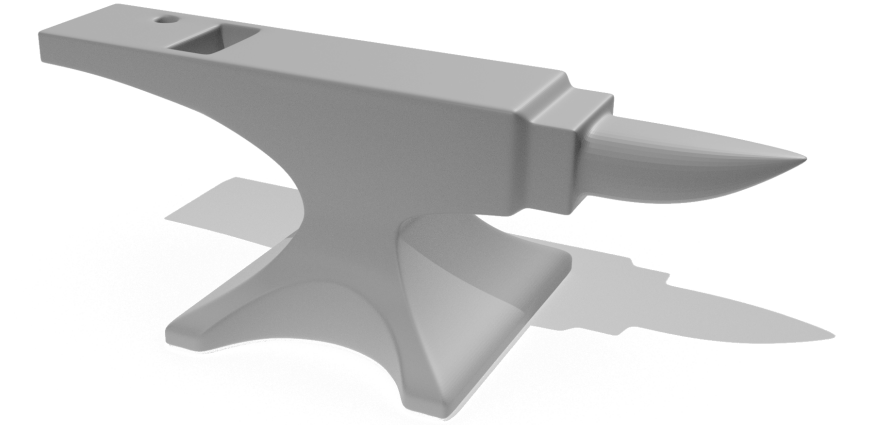
The voxelization with the old engine often fails in filling in the gap between the sampled front and back. This produce completly unusable result. Also, since it is only sampled from the front and back, the details from the other sides of the models are not captured in the voxelization. The new version is much more robust, and captures details from all six sides of the model. This results in a much more accurate representation of the model. This improvement can clearly be seen in Figure 4.11, which shows the voxelization of an anvil 3D model. Figure 4.11a shows a voxelization of an anvil with the old engine. Here, the voxelization has failed in the filling process. Figure 4.11b shows the result with the new version. Comparing the two, it is clear that the new engine produces a lot more accureate voxelizations.



(a) Voxelized anvil with Voxelizer v0.1.3.



(b) Voxelized anvil with Voxelizer v1.0.0.



(c) Original anvil 3D model.

Figure 4.11: Voxelization of a monkey with Voxelizer v0.1.3 and v1.0.0. The voxelization is done with a resolution of 2^7.

4.2.3 Performance

The engine's algorithm is completly overhault. The old raycasting algorithm has a time complexity of:

$$\mathcal{O}(n^3 \times m)$$

where n is the number resolution of the voxelization, and m is the number of triangles in the 3D model. In order to produce and more accurate result, the new algorithm does a lot more sampling. Despite this, the time complexity is redused. With the settings set to colorless and filled, the time complexity of the upgraded raycasting algorithm is:

$$\mathcal{O}(n^3 \times \log(m))$$

where n is the number resolution of the voxelization, and m is the number of triangles in the 3D model.

Following is a speed and memory comparison of the old and new vesion of the Voxelizer engine. The tests are executed in Node.js v12.16.3. The hardware used is a MacBook Pro (13-inch, 2018) with a 2.3GHz quad core processor and 16GB of 2133MHz LPDDR3 RAM, running MacOS Catalina v10.15.4.

First, the versions are tested with a low-poly mesh. Then, a high-poly mesh is used. The engine is tested at various resolutions. Note that a resolution of 2 results in $2 \times 2 \times 2$ voxels. The old engine version is only able to do a colorless, filled voxelization. In order to make a fair comparison, the new Version's RaycastAlgorithm is set to be colorless and filled. As input, the mesh will be programatically generated by the code provided in Listing 4.1. This produces a torus mesh with 3200 triangles.

Listing 4.1: JS code for generating low-detailed torus mesh.

```
1    let geometry = new THREE.TorusBufferGeometry( 10, 3, 16, 100 );
2    let material = new THREE.MeshBasicMaterial();
3    let torus = new THREE.Mesh(geometry, material);
```

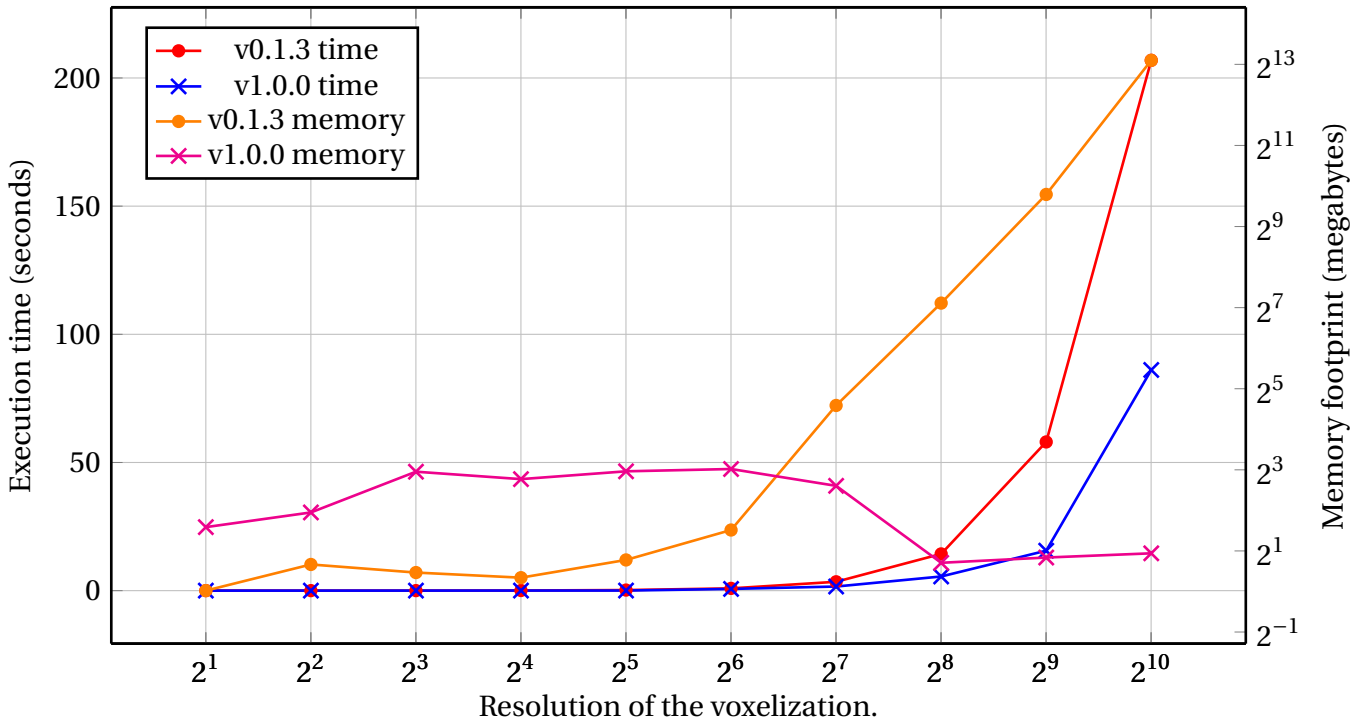


Figure 4.12: Plot over execution time and memory footprint for voxelization of a low-detailed mesh with the old and new Voxelizer engine.

Table 4.1: Execution times and memory footprints for voxelization of low-detailed mesh with Voxelizer v0.1.3

Resolution	**Time (sec)**	**Memory (MB)**
2^1	0.0254	1.0146
2^2	0.0116	1.5868
2^3	0.0195	1.3817
2^4	0.0624	1.2659
2^5	0.2288	1.7142
2^6	0.8879	2.8593
2^7	3.4578	24.0315
2^8	14.3298	138.2336
2^9	58.0136	885.5446
2^{10}	260.8811	8764.2631

Table 4.2: Execution times and memory footprints for voxelization of low-detailed mesh with Voxelizer v1.0.0

Resolution	**Time (sec)**	**Memory (MB)**
2^1	0.0154	3.00344
2^2	0.0172	3.8611
2^3	0.0214	7.7465
2^4	0.0382	6.8227
2^5	0.0403	7.7983
2^6	0.6752	8.1104
2^7	1.6094	6.0841
2^8	5.5269	1.6326
2^9	15.5968	1.7861
2^{10}	86.0982	1.9199

Figure 4.12 shows a graph over the execution time and the memory footprint for a voxelization of a low-detailed mesh with the Voxelizer engine v0.1.3 and v1.0.0. The raw data is available in Table 4.1 and Table 4.2.

The new engine has a relatively low memory footprint. At resolutions from 2^1 to 2^7, the memory consumption stays around 7MB. Then, at a resolution of 2^8, the memory drops to around 2MB. This is most likely due to the JavaScript Garbage Collector (GC) kicking in. At larger resolutions, the memory seems to stay relatively stabile around 2MB. The old engine starts with a very low memory footprint. It stays low, between 1MB and 2MB, from a resolution of 2^1 to 2^6. Then, the memory footprint starts to increase rapidly. A total of 24MB are used for a resolution of 2^7. At 2^{10} the footprint has risen to 8.7GB. This is most likely due to the fact that the old engine used normal JavaScript arrays that dynamically expands. A lot of obsolete data objects will accumulate, and the GC seems to not be able to handle this well.

Regarding execution time, this is also greatly improved in the new engine version. This is especially noticeable at large resolutions. At a resolution of 2^{10}, the old engine use 260 seconds to voxelize the 3D model. The new version only use 86 seconds, a reduction of about 60%.

An new mesh is generated with the code in Listing 4.2. This produces a highly detailed torus mesh with 1,000,000 triangles. The are presented in the figures below.

Listing 4.2: JS code for generating high-detailed torus mesh.

```
1 let geometry = new THREE.TorusBufferGeometry( 10, 3, 1000, 500 );
2 let material = new THREE.MeshBasicMaterial();
3 let torus = new THREE.Mesh(geometry, material);
```

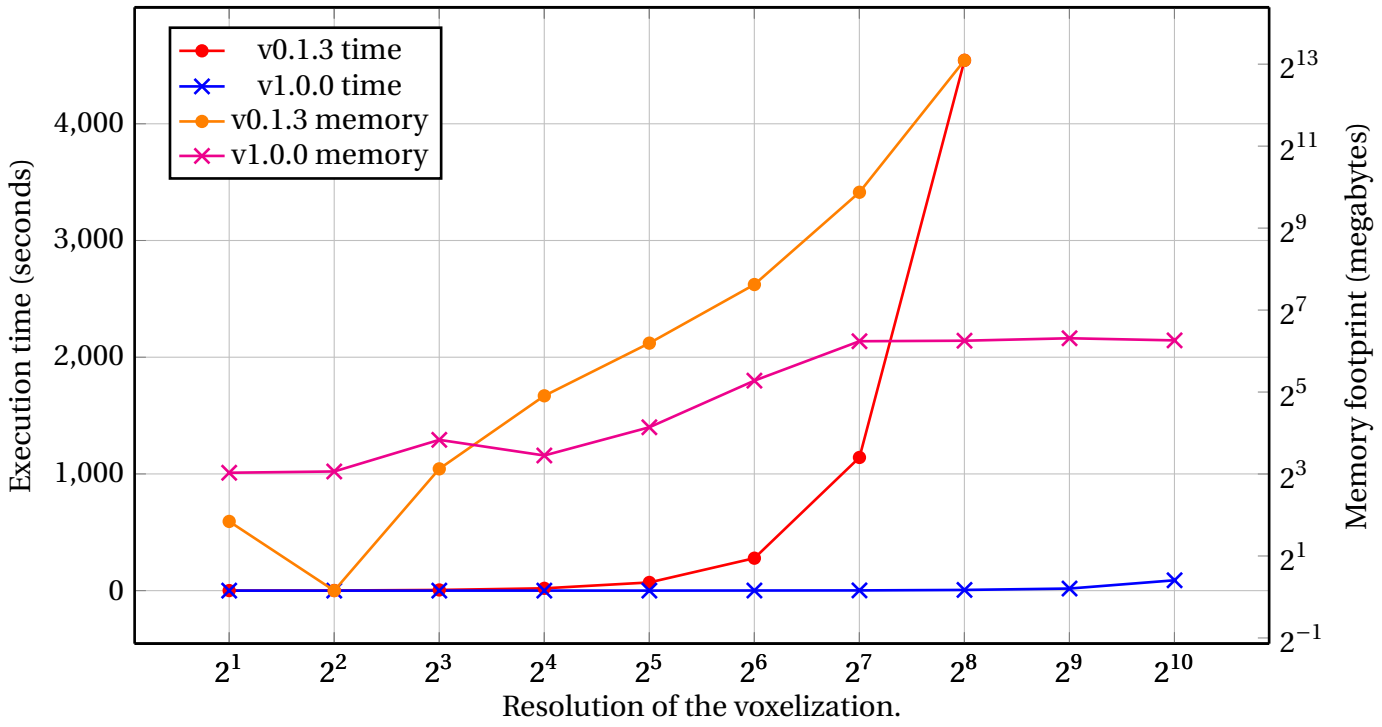


Figure 4.13: Plot over execution time and memory footprint for voxelization of a high-detailed mesh with the old and new Voxelizer engine.

Table 4.3: Execution times and memory footprints for voxelization of high-detailed mesh with Voxelizer v0.1.3

Resolution	**Time (sec)**	**Memory (MB)**
2^1	0.5388	3.5880
2^2	1.2370	1.1134
2^3	5.8674	8.7263
2^4	19.8983	30.0106
2^5	70.2835	73.1078
2^6	278.1215	197.0107
2^7	1141.1212	937.7180
2^8	4543.6064	8726.9914

Table 4.4: Execution times and memory footprints for voxelization of high-detailed mesh with Voxelizer v1.0.0

Resolution	**Time (sec)**	**Memory (MB)**
2^1	0.17265	8.1706
2^2	0.1612	8.3496
2^3	0.2277	14.2518
2^4	0.2755	10.9389
2^5	0.3402	17.6270
2^6	1.0793	38.7798
2^7	1.9604	75.5248
2^8	6.0971	76.1971
2^9	17.9376	79.4146
2^{10}	88.8092	76.6116

Figure 4.13 shows a graph over the execution time and the memory footprint for a voxelization of a high-detailed mesh with the Voxelizer engine v0.1.3 and v1.0.0. The raw data is available in Table 4.3 and Table 4.4.

The results shows similar characteristics as the ones with the low detailed mesh. However, very noticeable is the large difference in execution time. The old version is able to voxelize the mesh with 1,000,000 triangles in 4543 seconds, about 75 minutes. The new version is able to do the same in only about 6.1 second. The new version is also tested at a resolution of 2^{10}. This takes only 88.8 seconds. This massive performance improvement is mainly due to the raycasting optimization discussed in Section 3.4.4.

The memory consumption of the old version is more or less the same as with a low detailed mesh. The new version does consume more memory than in the previous test. This is because the BVH tree generated for the raycasting performance improvement does need a bit of memory. The memory footprint of the new version seems to gradually scale up from 8MB at a resolution of 2^1, to 76MB at resolution 2^7. From here on, the memory footprint stay more or less constant. This is because the BVH implementation gradually builds up the BVH tree, as described in Section 3.4.4.

4.2.4 Exporting

Several new exporting options are added to the upgraded Voxelizer engine. This includes XML, BINVOX and ndarray. The old multidimensional JavaScript array exporting option is also available, ensuring some backwards compatibility. The exporters system is made extensible, making it easy to add new exporting options in the future.

4.2.5 Code quality

This section describes how the code quality is improved. The old codebase had several problems. Firstly, all the code was contained in one large file. This made it hard to navigate and the code was messy. The new version makes use of ES Modules, making it possible to organize the code in different files. Secondly, the engine previously hardcoded the one algorithm used. The new version includes an easily extensible algorithm system. This is ensured through the use of

inheritance and factory patterns, as discussed in Section 3.4.2. Thirdly, the code did barely include tests for the system. The new version now includes unit tests for almost the whole engine. Lastly, the code had no code documentation. This is resolved by writing JSDoc for all classes and functions. This documentation is also made easily accessible at GitHub Pages with the help of the new JSDoc Action, presented in Section 4.5. A screenshot of the public documentation page is shown in Figure 4.14.

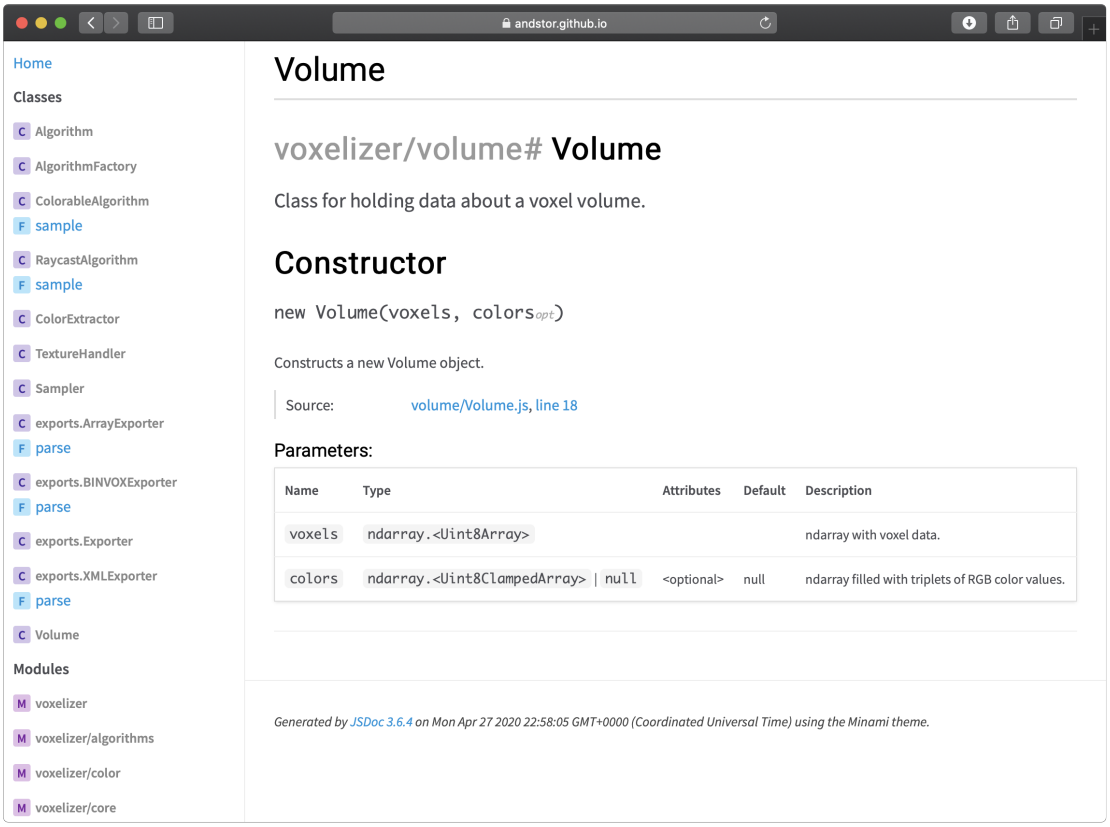


Figure 4.14: Public documentation for the Voxelizer engine.

4.2.6 Example

An example of the engine is deployed to GitHub Pages. Visit `https://andstor.github.io/voxelizer/examples/` to see the example. Figure 4.15 shows a screenshot of the site. The example includes a GUI with controls for controlling, among other things, the resolution, shell or solid voxelization, the LOD, and clipping planes for visually inspecting the results.

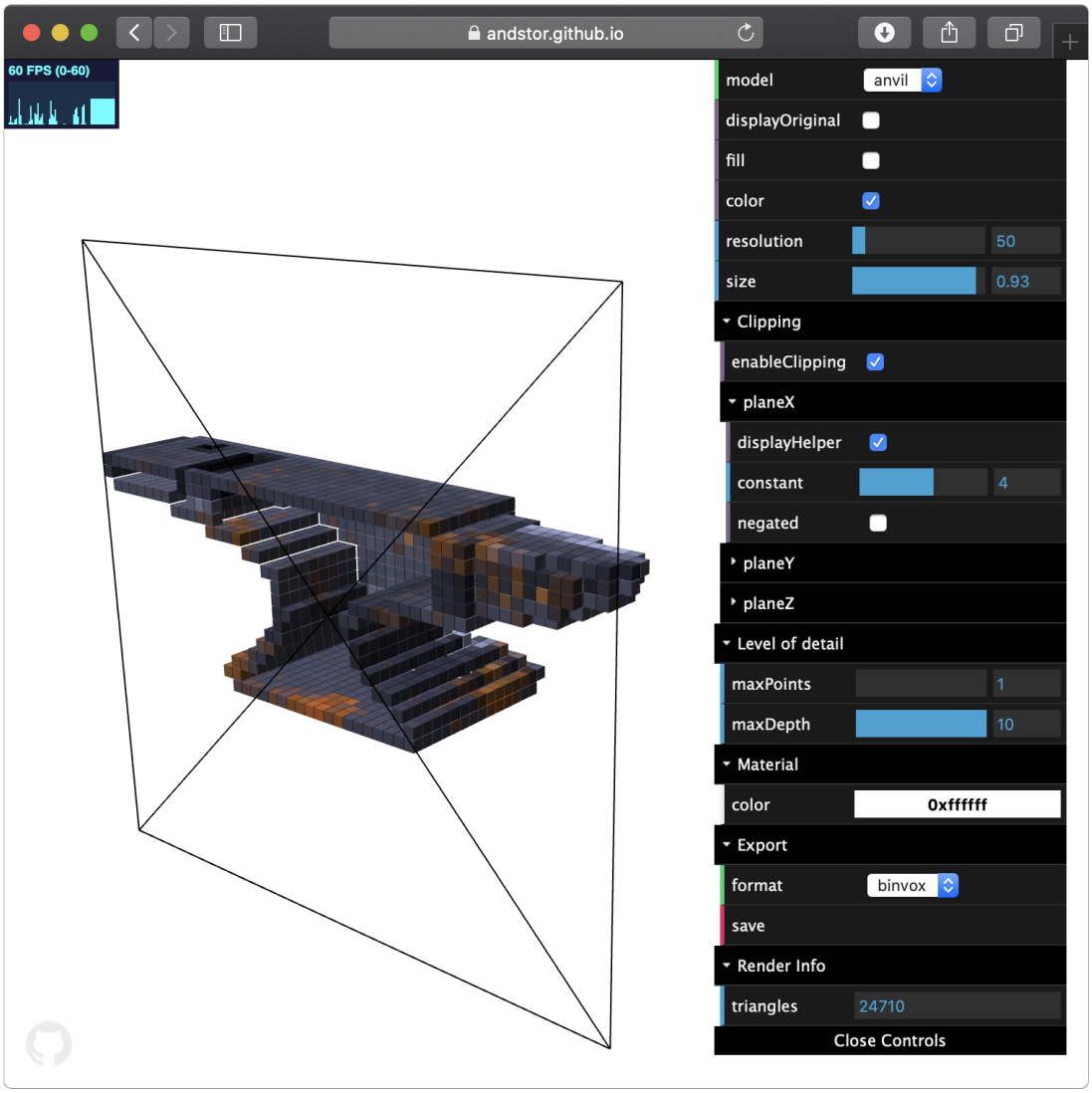


Figure 4.15: Screenshot of the Voxelizer engine example page at GitHub Pages.

4.2.7 Usage example

This section provides an example of how to use the Voxelizer engine v1.0.0. This example assumes that the project is bundled with a tool such as Webpack or Browserify, enabling the use of the ES Modules import syntax. Also, the node package manager (npm) is assumed to be installed.

First, the engine and its peer dependency three.js needs to be installed. This is easily done with npm. To install the Voxelizer engine and the peer dependency three.js, run the following commands:

```
$ npm install voxelizer
$ npm install three
```

Then, Listing 4.3 provides the example code now to be described in steps:

1. First, the Voxelizer engine v1.0.0, and its peer dependency three.js needs to be imported. This is done in lines 2 and 3.

2. Line 4 imports the GLTFLoader. This makes it possible to load a GLTF file.

3. Then something to voxelized is needed. This could be any three.js mesh with BufferGeometry. By using the imported GLTFLoader, a GLTF file can be loaded. The GLTFLoader is instantiated on line 7.

4. To actually load the model, the asynchronous "load" function is used at line8. This function needs a path to the file to load.

5. To set up the Voxelizer sampling class, some options are needed. These are defined on lines 12-15, making the engine produce a colored shell voxelization.

6. The sampler is then instantiated at line 16.

7. Line 19 and 20 actually samples the model with a resolution of 10.

8. In order to inspect the results, the returned volume data is fed to a XMLExporter at lines 23-26. This converts the volume data to a XML resource.

9. Line 25 prints the XML output to the console.

The result of the example code is a 10^3 voxel result of the loaded 3D model.

Listing 4.3: Voxelizer engine v1.0.0 example usage.

```
1  // Import via ES6 modules
2  import * as THREE from 'three';
3  import { Sampler, XMLExporter } from 'voxelizer';
4  import { GLTFLoader } from 'three/examples/jsm/loaders/GLTFLoader';
5
6  // Load 3D model.
7  const gltfloader = new GLTFLoader();
8  gltfloader.load('path/to/file.glb', function (data) {
9    let mesh = data.scene;
10
11   // Setup Voxelizer.
12   let options = {
13     fill: false,
14     color: true
15   };
16   const sampler = new Sampler('raycast', options);
17
18   // Voxelize 3D model.
19   const resolution = 10;
20   let volume = sampler.sample(mesh, resolution);
21
22   // Export result to XML.
23   const exporter = new XMLExporter()
24   exporter.parse(volume, function (xml) {
25     console.log(xml)
26   });
27 });
```

4.3 BINVOX

BINVOX is a package for parsing and building BINVOX files. It is published to the npm registry under the name "binvox", and the source code is available at GitHub under "andstor/binvox".

The package handles BINVOX files according to the BINVOX file format specification [54]. The package able to parse a BINVOX resource, turning it into JSON. Further, it can do this in reverse. Hence, properly formatted JSON data can be turned into a BINVOX file resource. The package provides both an ES Module build and a UMB build. It can therefore be used with both Node.js and in the browser.

4.4 Voxelizer Desktop

The Voxelizer Desktop application allows for easy use of the Voxelizer engine. It is a standalone cross platform desktop application. Ready-made installation files for macOS, Windows and Linux can be found at `https://github.com/andstor/voxelizer-desktop/releases/latest` on GitHub. The source code is also available at GitHub under "andstor/voxelizer-desktop".

4.4.1 Features

The Voxelizer Desktop application includes several features.

- **Importing** - The application supports several 3D model file formats. This includes GLB, GLTF, OBJ and STL. Some data formats need additional resources like texture images or binaries in order to work. This is also supported.
- **Voxelization** - The Voxelizer engine provides the application with voxelization capabilities. Several voxelization options are available, including: resolution, coloring and filling.
- **Visualization** - It is possible to view and inspect both the loaded 3D model and voxelized result through an interactive window.
- **Exporting** - Voxelizer Desktop enables the user to export to a XML file or a BINVOX file, and save this to the file system.

- **Auto updating** - When a new version of the application is released on GitHub, the application will automatically download and install the new version.

4.4.2 GUI

The user is first presented with an elegant and easy drag and drop user interface. Here, the user can just drop the 3D model files. Figure 4.16 shows a screenshot of the start screen, where a file is dropped onto the window.

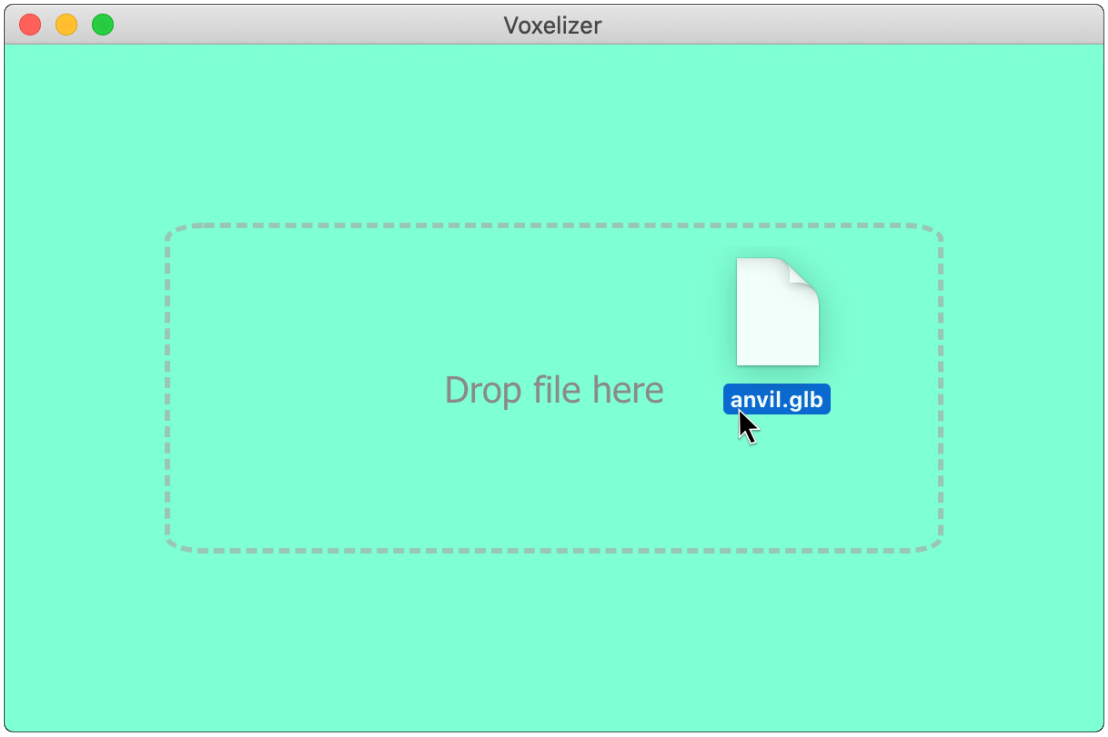


Figure 4.16: Voxelizer Desktop drag and drop start screen.

When a user drops one or more files, the application starts to load them up. If no supported filetype is found, an modal warning shows up like the one seen in Figure 4.17. Otherwise, a spinning loading wheel is presented, as shown in Figure 4.18, to indicate that the 3D model is loading.

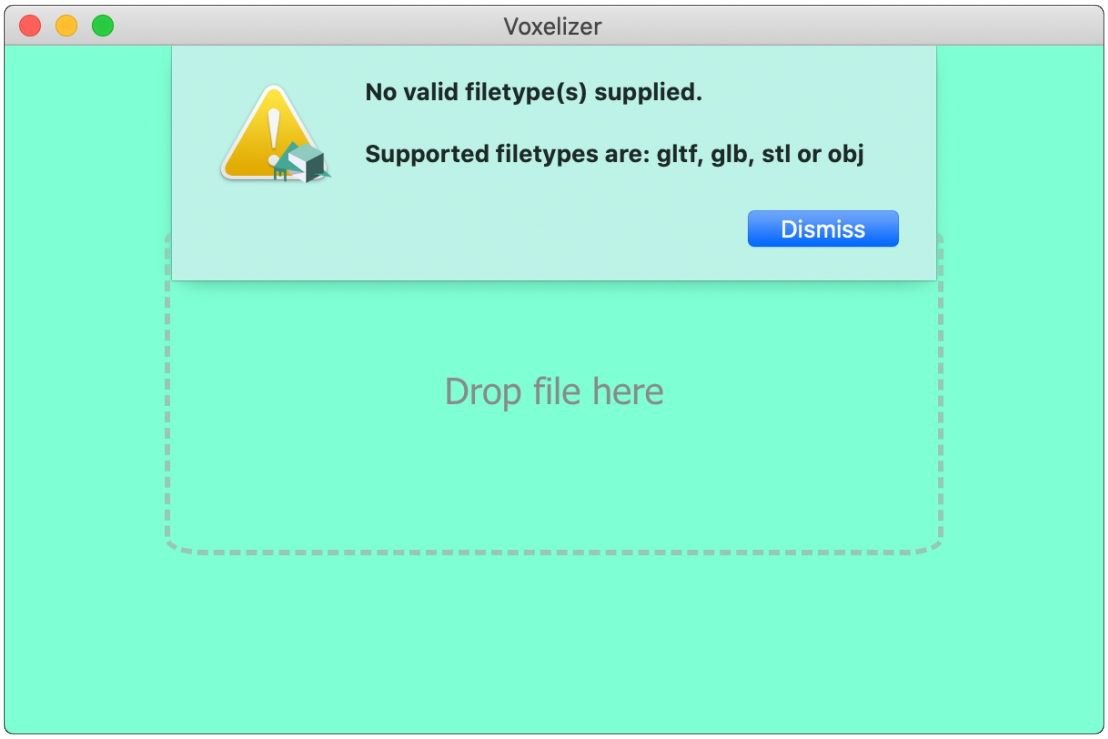


Figure 4.17: Voxelizer Desktop drag and drop start screen filetype error.

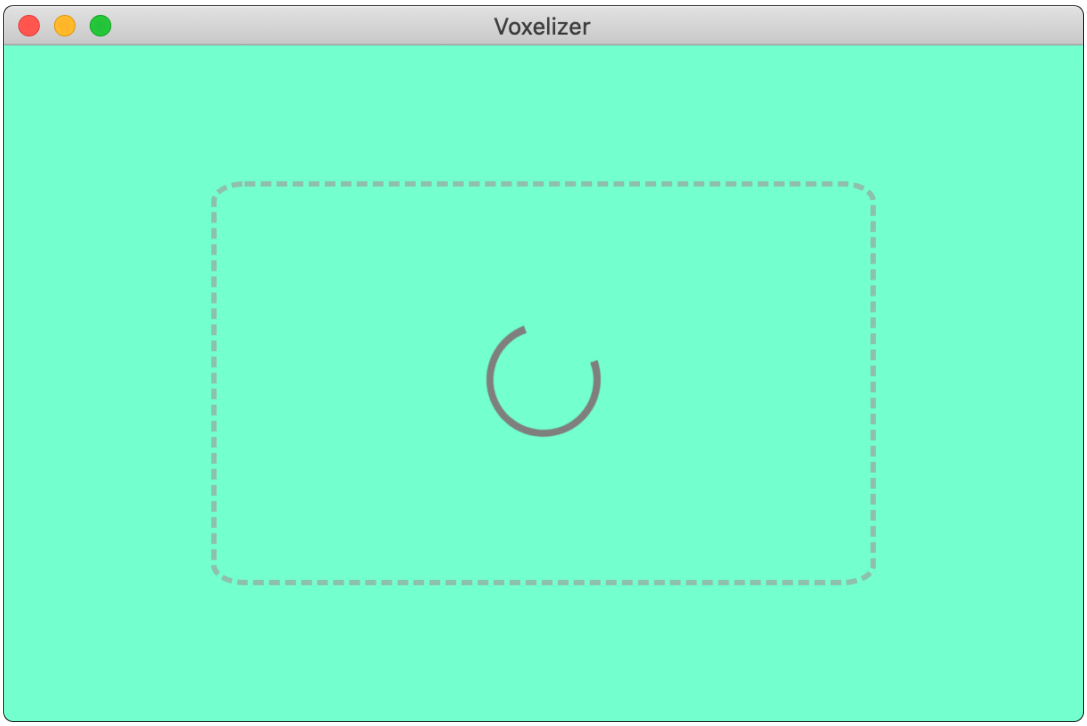


Figure 4.18: Voxelizer Desktop loading 3D model.

When the loading finishes, the user is presented with the main interface. This is shown in Figure 4.19. Here, the loaded 3D model is first presented in an interactive display. Further, at the bottom there are various controls for the voxelization and for exporting.

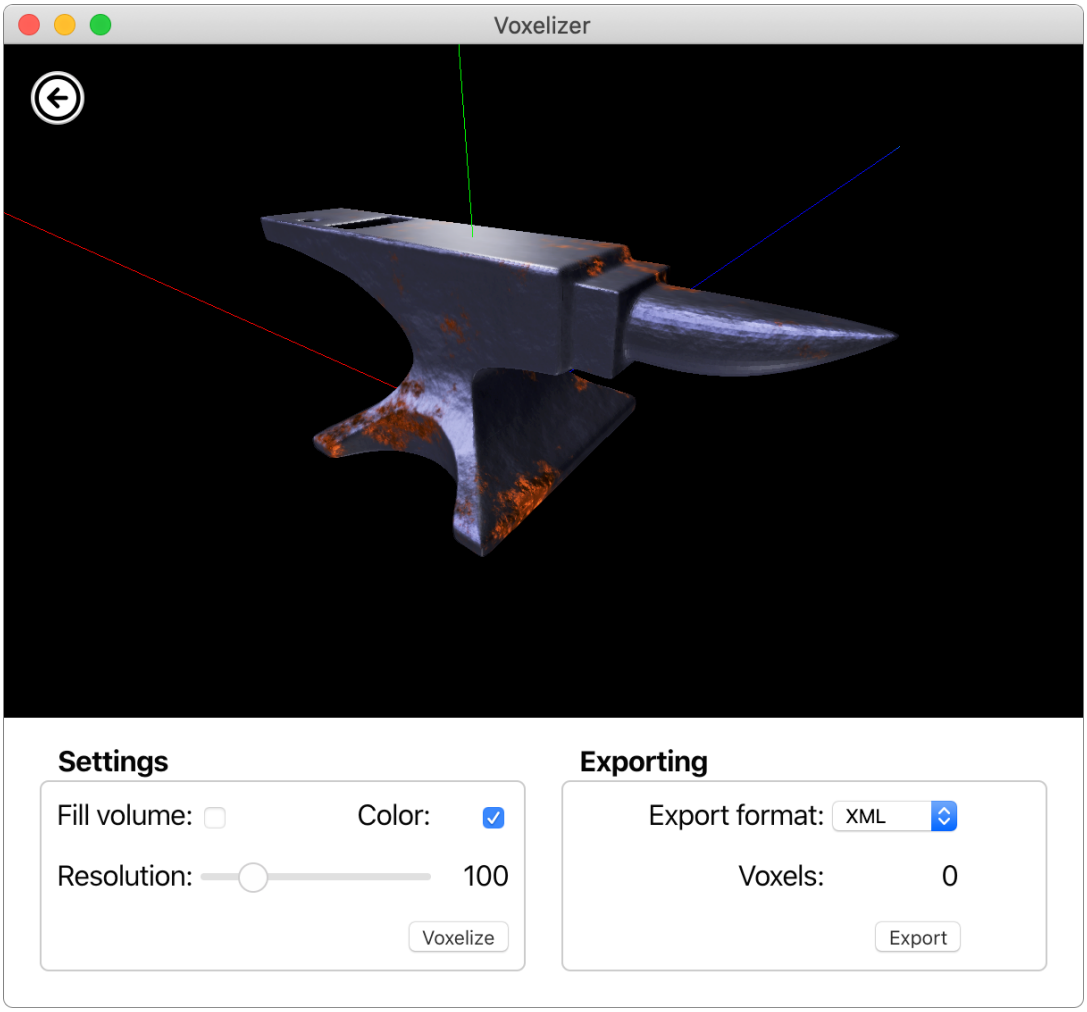


Figure 4.19: Voxelizer Desktop main interface.

The application has support for dark mode. Meaning, depending on the system settings, either a light-theme or a dark-theme is selected. The application also has localization support (language). Currently, English and Norwegian Bokmål is available. The selected language will depend on the system settings. Figure 4.20 shows the main interface in dark mode and with a Norwegian interface.

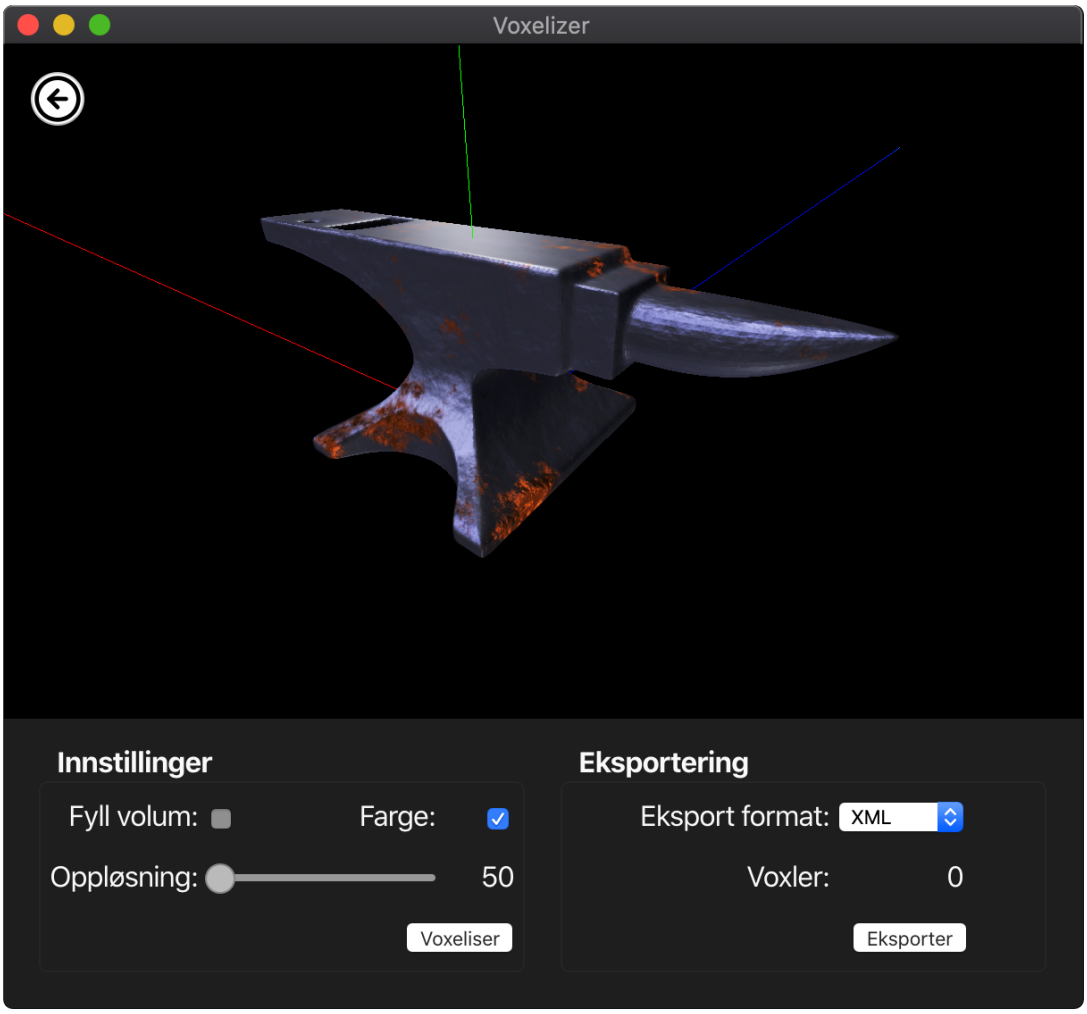


Figure 4.20: Voxelizer Desktop with dark mode and Norwegian language.

If one tries to export before voxelizing the model, a modal will show a warning message. Figure 4.21 shows this error message.

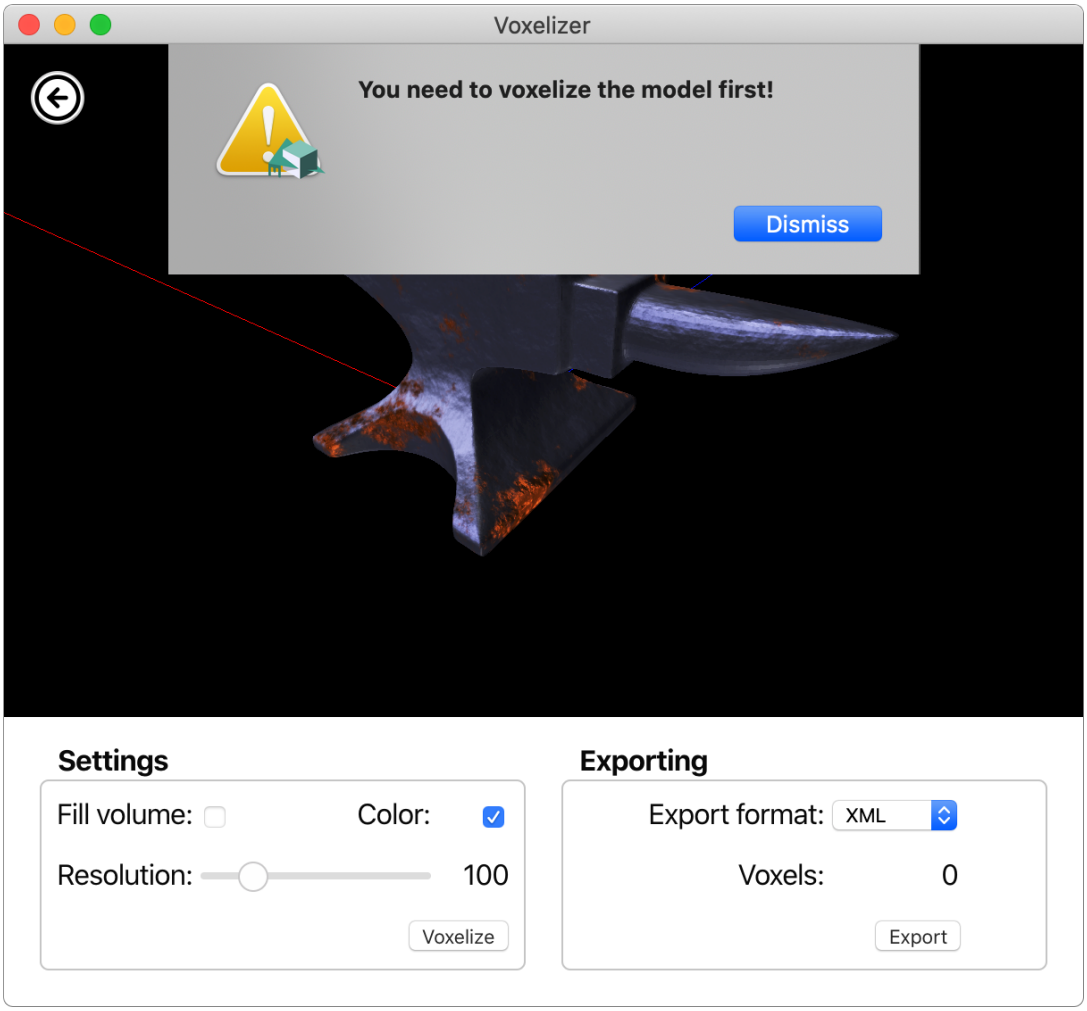


Figure 4.21: Voxelizer Desktop voxel warning.

To actually voxelize the model, the user needs to click on the **Voxelize** button. This starts the voxelization process. When finished, the result is presented in the 3D graphics window. This can be seen in Figure 4.22

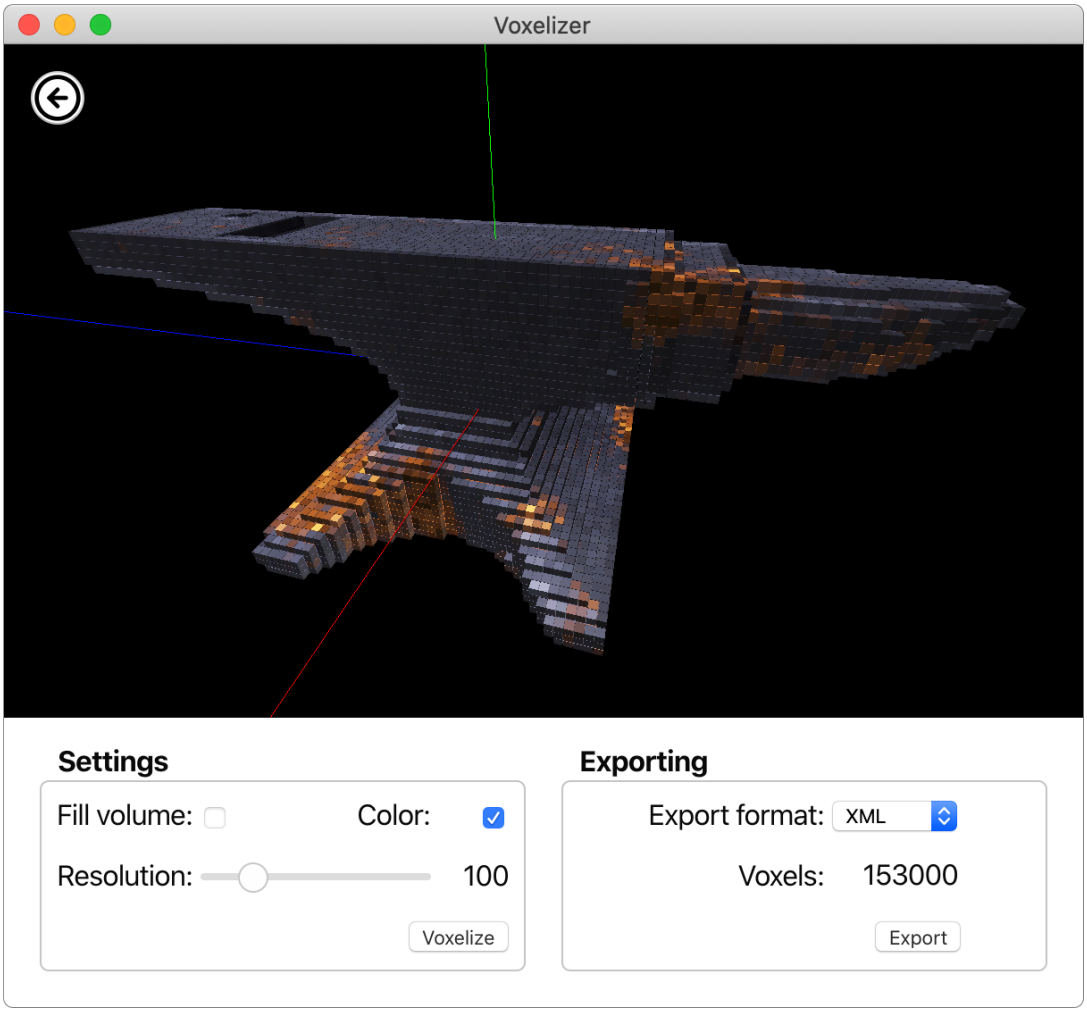


Figure 4.22: Voxelizer Desktop displaying voxelized result.

To export and save the voxel data in the selected file format, the user needs to click on the **Export** button. This opens up the operating system's file system dialog, as can be seen in Figure 4.23. A file name and location then needs to be selected. When the user clicks on the **Save** button, the file is saved.

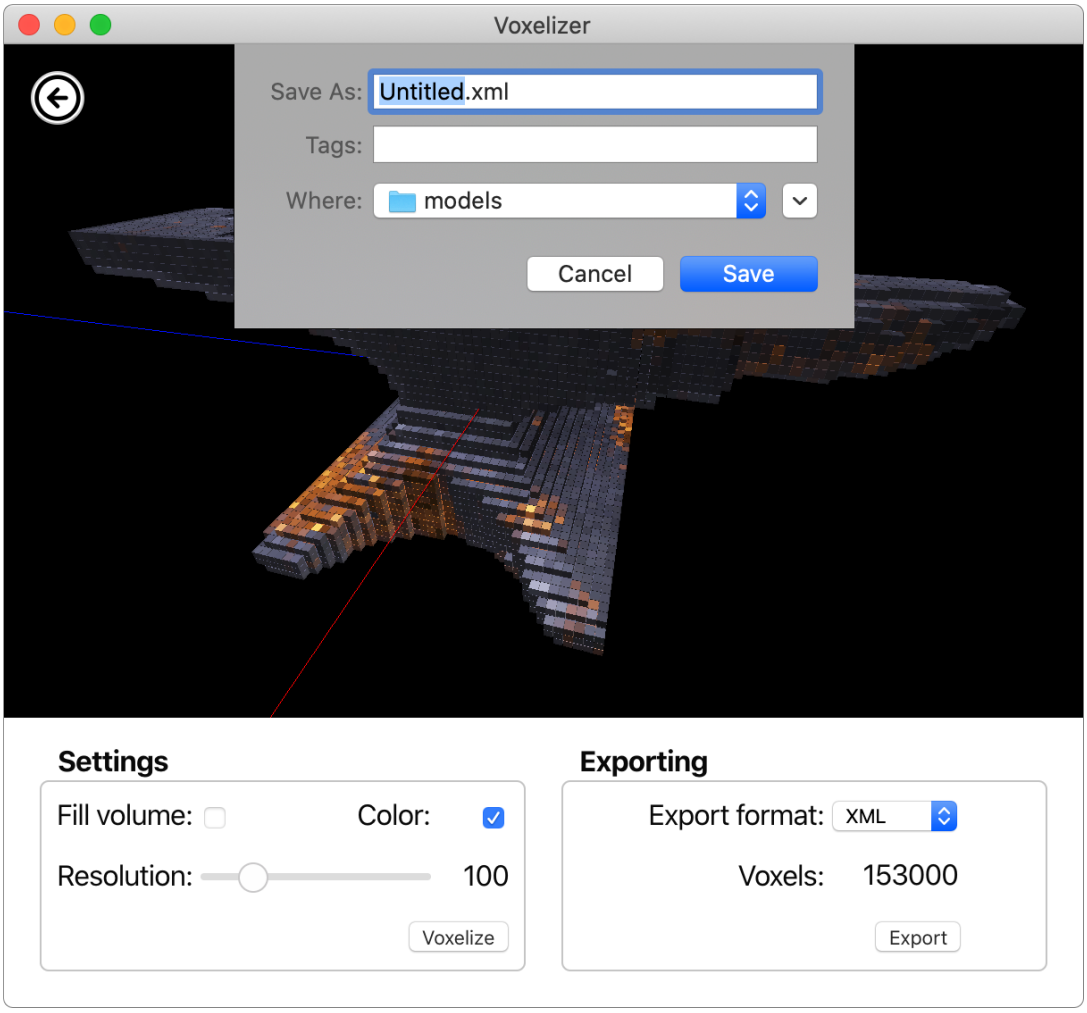


Figure 4.23: Voxelizer Desktop OS file dialog for saving voxel data.

In order to voxelize a different 3D model, the user can click on the arrow icon in the upper left corner. When the user exits the application, the application state is saved. Among other things, this enables the application window to remember its window location and size.

4.5 JSDoc Action

The JSDoc Action makes it easy to automate the process of generating JSDoc documentation. It is a GitHub Action, and is published to the GitHub Marketplace under the name "JSDoc Action". Figure 4.24 shows the graphics generated by the GitHub Marketplace for the JSDoc Action. The source code is available at GitHub under "andstor/jsdoc-action".

Figure 4.24: Graphics from the GitHub Marketplace for the JSDoc Action.

The action is easy to combine with other deployment actions. This makes it very simple to publish the generated documentation to any hosting service, for example GitHub Pages. The JSDoc Action also supports templates. These are installed with npm, so any package can be used. See the npm documentation for more details on the installation command [73].

All that is needed for a minimum base setup is a workflow step like the one defined in Listing 4.4. This will generate documentation for all source files in the "./src" directory, and output the built files to a "./out" directory. Any other GitHub Action might further process this output.

Listing 4.4: Basic JSDoc Action workflow step

```
1 - name: Build docs
2   uses: andstor/jsdoc-action@v1
3   with:
4     source_dir: ./src
5     recurse: true
6     output_dir: ./out
```

4.5.1 Example usage

In the following, a complete example to illustrate how easy it is to automate the JSDoc documentation process with the JSDoc Action is provided. This assumes that GitHub is used for hosting the code to generate documentation for.

First, a workflow yaml file needs to be created. This file needs a name, for example **documentation.yaml**, and has to be uploaded to the default branch in the user's repository, under ".github/workflows/". Then, the actual workflow needs to be defined. Listing 4.5 provides an example workflow. This workflow is set to only run when code is pushed to the master branch. Ubuntu is then chosen as the platform to run the workflow job on. Several workflow steps are then defined:

1. First, the user's code repository is cloned with the help of the Checkout Action by GitHub. This makes the user's source code available to subsequent steps.

2. Second, the JSDoc Action is used for generating the actual JSDoc documentation files. Lines 15-19 defines several input options to the JSDoc Action. The source directory is set to "./src". The output directory is set to "./out". A path to a JSDoc configuration file in the user's repository is then specified. In order to freshen up the plain JSDoc theme, the minami template is used. This is the name of the package on npm. Finally, the README.md file in the user's repository is used as frontpage.

3. Third, the GitHub Pages action is used for deploying the generated documentation files to GitHub Pages. It needs two input configurations. One is a deployment key. See the documentation for the GitHub Pages action for how to set up this. The other is a directory to get the files to publish. This is set to the the JSDoc Action's output directory, "./out".

When the workflow file is finished and saved to the correct place, the repository will feature automatic JSDoc documentation generation.

Listing 4.5: Example documentation workflow file

```yaml
name: Documentation
on:
  push:
    branches:
      - master
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Build
        uses: andstor/jsdoc-action@v1
        with:
          source_dir: ./src
          output_dir: ./out
          config_file: ./conf.json
          template: minami
          front_page: README.md
      - name: Deploy
        uses: peaceiris/actions-gh-pages@v3
        with:
          deploy_key: ${{ secrets.ACTIONS_DEPLOY_KEY }}
          publish_dir: ./out
```

4.6 file-existence-action

The File Existence action is a GitHub Action. The action is published to the GitHub Marketplace by the name "File Existence", and the source code is available at GitHub under "andstor/file-existence-action".

The action is able to check if one or more files exists during a workflow run. The user just supplies the paths as inputs to the action. The action then produces a boolean output variable which is available to the subsequent workflow steps. If any files are missing, the output is set to

false. Otherwise, true. It is also possible to make the action trigger an error if one or more files are missing. This will effectively cancel the entire workflow.

4.7 file-reader-action

The File Reader action is a GitHub Action. The action is published to the GitHub Marketplace by the name "File Reader", and the source code is available at GitHub under "andstor/file-reader-action".

By providing a path as input to the action, the action is simply able to read the contents of a file during a workflow run. The action produces an output variable with the contents of the file. This variable will be available to the subsequent workflow steps.

4.8 Automation

Several automation systems are implemented for the various projects. This makes the maintenance of the projects very easy. All JavaScript package building and publishing is fully automated. Automatic JSDoc generation and publishing is also set up. Further, maintenance tasks for the GitHub actions are automated. See Section 3.8 for a walkthrough of how the automations systems work, and what they do in details. For the GitHub Actions, automatic updating of version tags are implemented, according to the GitHub guidelines. For the JavaScript packages, the following tasks are automated:

- Building
- Testing
- Code coverage generation, and uploading to Coveralls.io
- Security analysis with LGTM by Semmle
- JavaScript JSDoc documentation generation with the JSDoc Action
- Publishing of JavaScript package to the npm registry

Figure 4.25 shows how some of these automated task (defined in Workflows) are run on GitHub's CI/CD system. Here, one can see all the checks are passed. Also, an alert is triggered by the security analysis by LGTM, stating that three new alerts are introduced in the code.

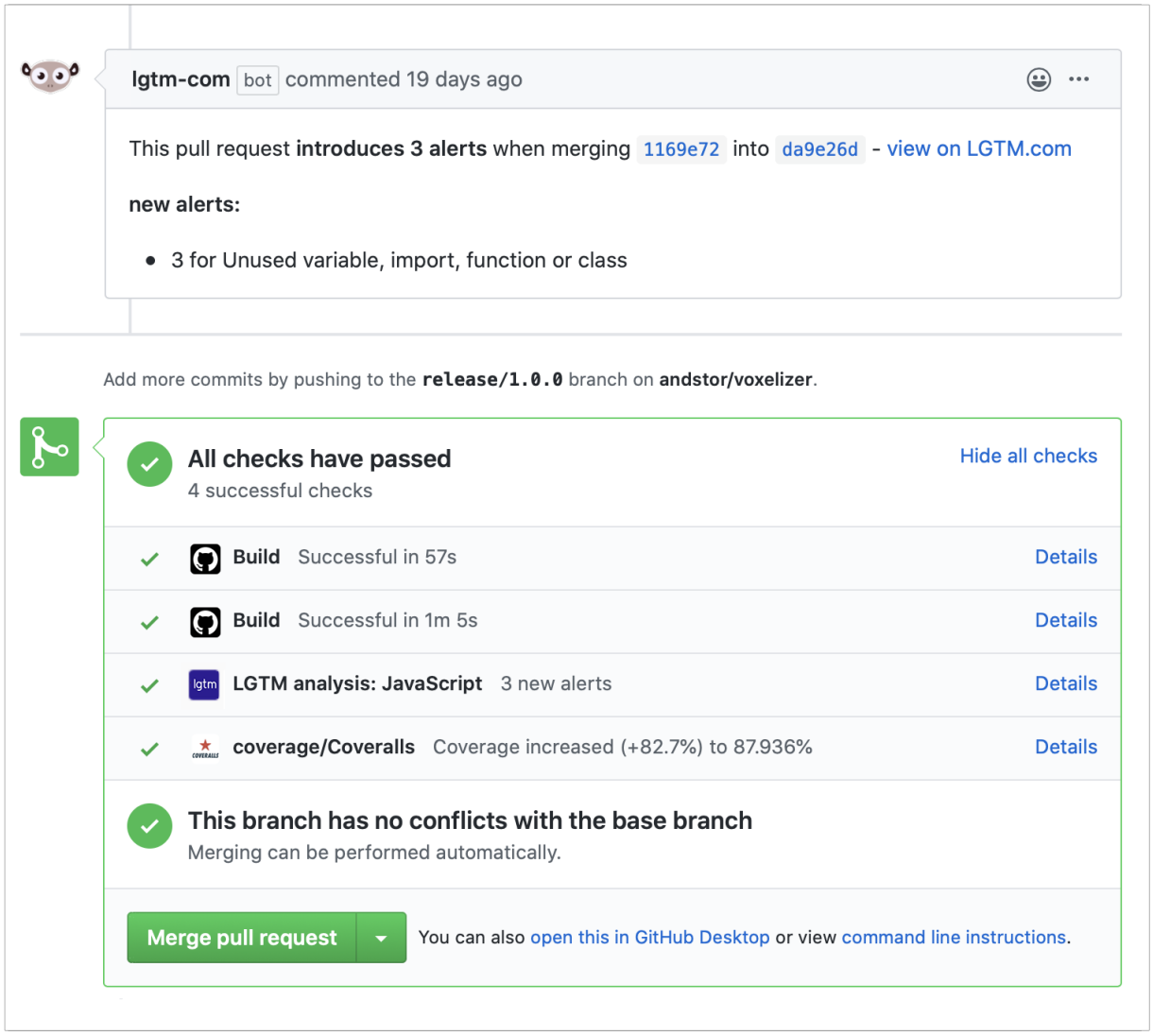


Figure 4.25: Screenshot of automation checks for a pull request on GitHub.

4.9 Popularity and achievements

Several of the projects has already gained interest by the public. GitHub users are able to star a repository. This is one of the main measurements available for how popular a repository is.

Several of the repositories created in connection with this thesis have received numerous of stars already, and is used by several people. Table 4.5 shows how many stars the different repositories have, as of May 17th, 2020 [1].

Table 4.5: Repositories statistics as of May 17th, 2020.

Repository	Stars	Used by
jsdoc-action	13	22
file-existence-action	6	17
voxelizer	4	3
three-voxel-loader	4	1
file-reader-action	1	5
binvox	1	2
voxelizer-desktop	1	0

Another achievement is the inclusion of a link to the JSDoc Action in the README.md file of the original JSDoc repository. As of May 17th, 2020 the JSDoc repository has 10.6 thousand stars and is used in 39.3 repositories. This gives the action a lot of marketing. The approved pull request is number #1745. Figure 4.26 shows a screenshot of the **Templates and tools** section of the README.md file. The link to the JSDoc Action is located at the bottom of the **Build tools** subsection, marked in red. Figure 4.27 shows the response to the approved pull request from the lead maintainer Jeff Williams (hegemonic) of JSDoc.

[1] The statistics also include personal staring and usage.

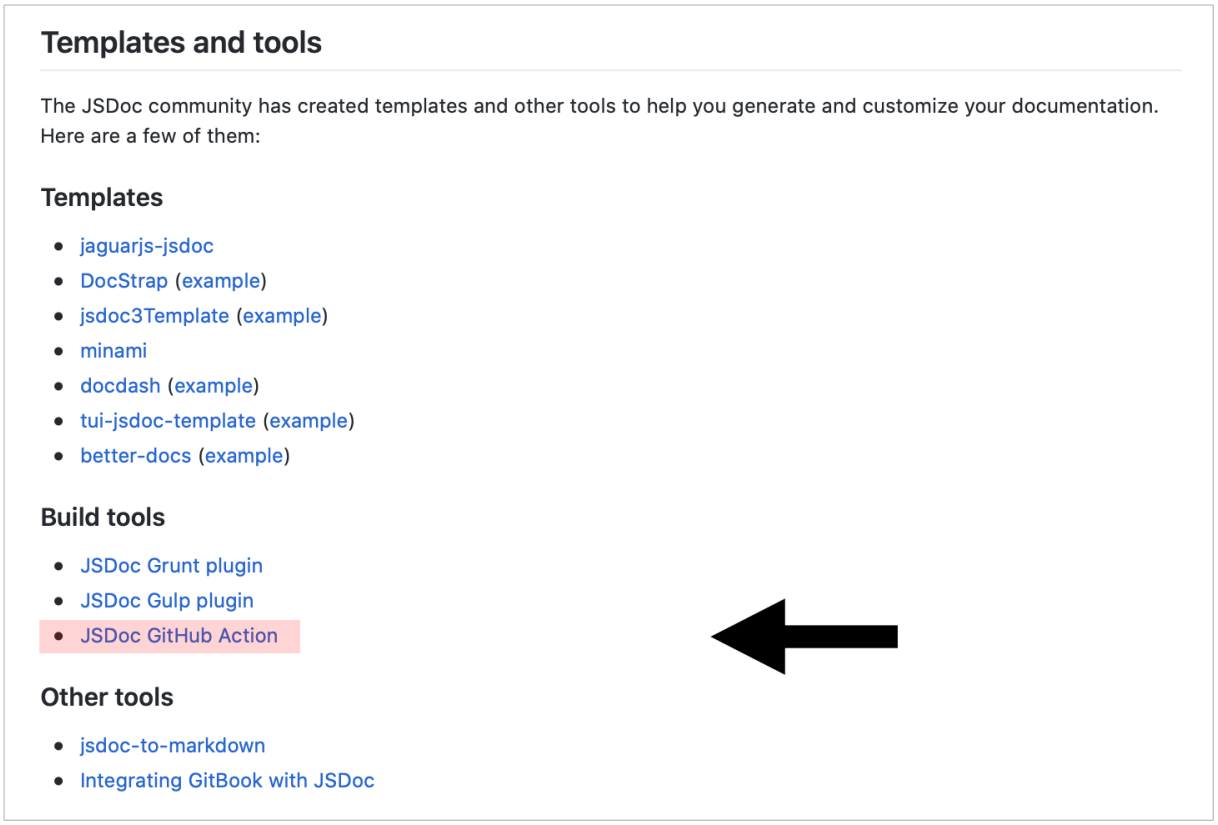


Figure 4.26: Screenshot of JSDoc README.md file.

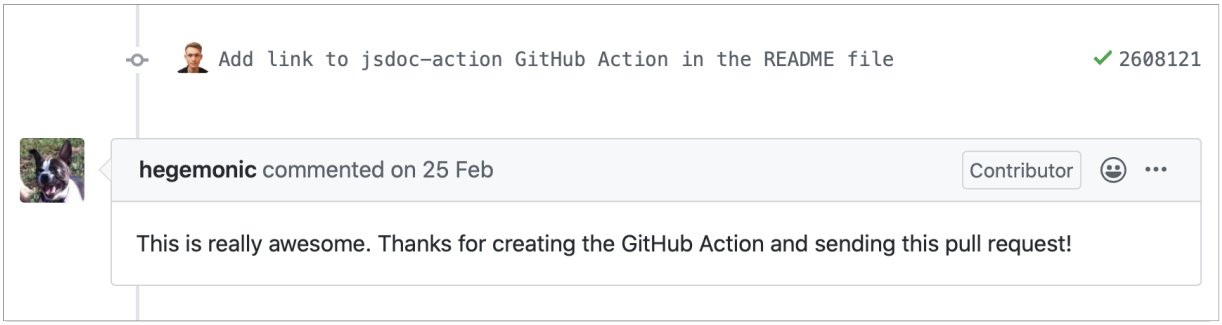


Figure 4.27: Screenshot of response from JSDoc lead maintainer.

Chapter 5

Discussion

This chapter discusses the achieved results, the execution of methodologies, and how the projects could be further improved. The chapter starts with Section 5.1 that discusses the completeness of the entire project, compared to the requirements specification. Section 5.2 describes how the chosen working methodology worked out. The improvements of the Voxelizer engine are discussed in Section 5.3, closely followed by the Voxelizer desktop application in Section 5.4. The level of automation achieved is described in Section 5.5. The rest of the supportive projects are discussed in Section 5.6. Finally, Section 5.7 discusses some features that are left as future work.

5.1 Requirements specification completeness

Most of the primary objectives are completed according to the requirement specification. Also, the majority of the user-stories are addressed, as can be seen from the backlog included in Appendix D. The project is therefore overall considered a success.

Due to various extra work like the binvox package, the file-existence-action and the file-reader-action, some functionality had to give way to others. It was not enough time to make the CLI tool for the Voxelizer engine. However, this could be a valuable companion tool for the Voxelizer engine, and should therefore be considered for future work.

5.2 Working methodology

Scrum was used as the working methodology for this project. However, since Scrum is designed for teams, the method needed some adaptations. The changes described in Section 3.2.1 worked out very well. The method helped in keeping the project on track. Further, the desire to complete the sprints functioned as a strong motivator throughout the entire project. This is backed by the sprint cumulative flow diagram available in Appendix C. The diagram shows a reasonable steady stream of completed issues throughout the entire project.

5.3 Voxelizer

Compared to the old Voxelizer engine, the new version produces voxelization results of a lot higher quality. The improvement can clearly be seen from the results described in Section 4.2.2. The engine is also greatly improved in terms of performance. Although, the level of performance gain shown in Section 4.2.3 was not expected, especially the huge reduction in memory footprint. Further, several new features are introduced. Being able to produce shell and color voxelizations makes the engine much more attractive.

During development, it became clear that extending the importing support for the different formats specified in the requirements specification was not beneficial. three.js already provides support for around 40 file formats. However, at the time of the creation of the old engine version, the loaders were hard to import and use. Since then, these loaders have implemented support for ES Modules. It was therefore decided that the task of loading the models should be left up to the user.

5.4 Voxelizer Desktop

The Voxelizer Desktop ensures easy use of the Voxelizer engine. It fulfills all the main requirements. The application provides an intuitive user interface, and the voxelization process is very easy. By supporting several of the most popular 3D file formats, like OBJ, GLTF and STL, the user is most likely able to voxelize his/her files directly, without additional conversion steps.

5.5 Automation

The different projects are highly automated at an early stage. This made the various maintenance task throughout the project very easy. Future support and maintenance is therefore also expected be easy. Especially valuable is the JSDoc action. This will ensure available and up-to-date high-quality API documentation for the projects.

5.6 Supportive projects

Several tools were main to aid the various projects. This includes the three-voxel-loader plugin, the BINVOX package, the File Existence action and the File Reader action. With the exception of the three-voxel-loader, these tools are mainly the result of refactoring. Keeping the code in smaller modules, or packages, makes it easier to maintain. It also makes it easy for others to use the software for other projects. Further, this often means an increase in both testing and contributions to the project.

The three-voxel-loader three.js plugin proved to be very valuable in terms of testing. Inspecting the raw voxel data is more or less impossible. By visualizing the voxelized results, testing and debugging the Voxelizer engine was very easy.

5.7 Future work

The uncompleted requirements specification and the user-stories provides an excellent basis for future work. In addition to this, the following sections suggests some additional improvements.

5.7.1 three-voxel-loader

The three-voxel-loader plugin generates a cube (CubeBufferGeometry) for every voxel. Even for voxels that are not visible from outside the model. When loading a large and filled voxel model, this results in an enormous number of faces being rendered, putting a heavy load on the hardware. A future improvement could be to only render to extract the surface mesh of the voxels result, when a voxel size of 1 is used. This would dramatically reduce the number of

triangles needed to render the voxel model.

Another interesting feature could be to support isosurface extraction. This is the opposite process of the voxelization process. It tries to approximate a 3D mesh based on voxel data. The most popular isosurface extraction algorithm is the the marching cube algorithm, published in 1987 by Lorensen and Cline [74]. Including this feature into the project could be of great value.

5.7.2 Voxelizer

Due to the fact that JavaScript is single-threaded, no multithreading is implemented in the Voxelizer Engine. However, it is possible to implement a sort of multithreading with Web Workers in the browser, or Worker Threads in Node.js. A future improvement could be to implementing support for this type of multithreading directly into the Voxelizer engine. This way, the heavy voxelization calculations could be split up into chunks and voxelized in parallel.

Chapter 6

Conclusion

The project is overall considered a success. A primary objective of this thesis was to improve the Voxelizer engine. The engine is completely overhauled, addressing all known issues with the old version. Several new features are also added, like coloring and shell voxelization. The performance of the engine is also greatly improved. The speed of the voxelization is significantly increased, and the memory footprint is reduced by several orders of magnitudes. The exporting capabilities are also extended with several file formats and data structures. This includes ndarrays, XML, and BINVOX. Exporting to BINVOX files is made possible with the new BINVOX JavaScript package.

To make voxelization easy and accessible, a complementary cross platform desktop application is developed for the Voxelizer engine. This features a sleek drag and drop interface, along with visualized voxelization results, made able with the new three-voxel-loader three.js plugin.

To ensure the projects keep a high level of quality, and are easy to maintain, a lot of automation is implemented. This has removed much of manual and laborious work. It has also made the software a lot safer, drastically reducing the potential of human errors and bug introductions. To achieve this level of automation, several GitHub Actions are developed. This mainly includes the highly successful documentation tool, the JSDoc Action. It is embraced by the popular JSDoc tool, and is already gaining in popularity. Two more actions are also developed for automation purposes. This is the File Existence action, and the File Reader action.

To summarize, the Voxelizer engine is greatly improved, and a complementary desktop application is developed. Alongside, several automation tools are created and used in the projects.

Bibliography

[1] J. Zadik, B. Kenwright, and K. Mitchell, "Integrating real-time fluid simulation with a voxel engine," *The Computer Games Journal*, vol. 5, no. 1, pp. 55–64, Jul. 2016. DOI: 10.1145/37402.37422. [Online]. Available: https://doi.org/10.1007/S40869-016-0020-5.

[2] P. Min. (Jan. 2004 - 2020). "Binvox," [Online]. Available: http://www.patrickmin.com/binvox (visited on 05/06/2020).

[3] Scrum.org. (2020). "Scrum," [Online]. Available: http://www.scrum.org/ (visited on 05/06/2020).

[4] Atlassian. (2020). "What is kanban?" [Online]. Available: https://www.atlassian.com/agile/kanban (visited on 05/06/2020).

[5] Git. (). "About git," [Online]. Available: https://git-scm.com/about.

[6] V. Driessen. (Jan. 2010). "A successful git branching model," [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/ (visited on 04/29/2020).

[7] GitHub. (2020). "Github actions," [Online]. Available: https://github.com/features/actions (visited on 05/10/2020).

[8] ——, (2020). "Workflow syntax for github actions," [Online]. Available: https://help.github.com/en/actions/reference/workflow-syntax-for-github-actions (visited on 05/07/2020).

[9] ——, (2020). "Github pages," [Online]. Available: https://pages.github.com (visited on 05/02/2020).

[10] M. D. Network. (Mar. 2019). "Prototype-based programming," [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Prototype-based_programming (visited on 05/04/2020).

[11] Netscape. (Dec. 1995). "Netscape and sun announce javascript," [Online]. Available: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (visited on 05/03/2020).

[12] E. International. (2020). "Ecma international," [Online]. Available: http://www.ecma-international.org (visited on 05/03/2020).

[13] TIOBE Software. (Mar. 2020). "TIOBE index for may 2020," [Online]. Available: https://www.tiobe.com/tiobe-index/ (visited on 05/03/2020).

[14] npm. (2020). "Npm," [Online]. Available: https://www.npmjs.com (visited on 05/03/2020).

[15] RequireJS. (). "Requirejs," [Online]. Available: https://requirejs.org (visited on 05/04/2020).

[16] UMD. (Oct. 2017). "UMD," [Online]. Available: https://github.com/umdjs/umd (visited on 05/04/2020).

[17] Mozilla Developer Network. (Apr. 2020). "Browser support," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (visited on 05/04/2020).

[18] Babel. (2020). "Babel is a javascript compiler. babel is a javascript compiler. babel is a javascript compiler," [Online]. Available: https://babeljs.io (visited on 05/04/2020).

[19] Webpack. (2020). "Webpack," [Online]. Available: https://webpack.js.org (visited on 05/04/2020).

[20] Rollup. (2020). "Rollup," [Online]. Available: https://rollupjs.org/guide/en/ (visited on 05/04/2020).

[21] Browserify. (2020). "Browserify," [Online]. Available: http://browserify.org (visited on 05/04/2020).

[22] TypeScript. (2020). "Typescript," [Online]. Available: https://www.typescriptlang.org (visited on 05/04/2020).

[23] Microsoft. (2020). “Microsoft,” [Online]. Available: https://www.microsoft.com (visited on 04/30/2020).

[24] Google. (May 2020). “Google closure compiler,” [Online]. Available: https://github.com/google/closure-compiler (visited on 05/04/2020).

[25] JSDoc. (May 2020). “Jsdoc,” [Online]. Available: https://github.com/jsdoc/jsdoc (visited on 05/04/2020).

[26] GitHub. (May 2020). “Dependency graph,” [Online]. Available: https://github.com/jsdoc/jsdoc/network/dependencies (visited on 05/04/2020).

[27] ——, (May 2020). “Stargazers,” [Online]. Available: https://github.com/jsdoc/jsdoc/stargazers (visited on 05/04/2020).

[28] K. Group. (). “Webgl overview,” [Online]. Available: https://www.khronos.org/webgl/.

[29] ——, (2020). “Opengl es,” [Online]. Available: https://www.khronos.org/opengles/ (visited on 05/06/2020).

[30] three.js. (). “Three.js,” [Online]. Available: https://threejs.org.

[31] M. Lysenko. (Jan. 2020). “Ndarray,” [Online]. Available: https://github.com/scijs/ndarray (visited on 05/04/2020).

[32] Mozilla Developer Network. (Mar. 2020). “Javascript typed arrays,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays (visited on 05/04/2020).

[33] GitHub. (2020). “Electron,” [Online]. Available: https://www.electronjs.org (visited on 05/02/2020).

[34] ——, (2020). “About github,” [Online]. Available: https://github.com/about (visited on 01/25/2020).

[35] Microsoft. (2020). “Visual studio code,” [Online]. Available: https://code.visualstudio.com (visited on 05/02/2020).

[36] Facebook. (2020). “Facebook messenger,” [Online]. Available: https://www.messenger.com/desktop (visited on 05/02/2020).

[37] Microsoft. (2020). "Microsoft teams," [Online]. Available: https://www.messenger.com/desktop (visited on 05/02/2020).

[38] Google. (). "Chromium," [Online]. Available: https://www.chromium.org/Home (visited on 05/11/2020).

[39] Facebook. (2020). "React," [Online]. Available: https://reactjs.org (visited on 05/02/2020).

[40] ——, (). "Facebook," [Online]. Available: https://about.fb.com (visited on 05/02/2020).

[41] Semmle. (2020). "Lgtm," [Online]. Available: https://lgtm.com (visited on 05/04/2020).

[42] ——, (2020). "Security at semmle," [Online]. Available: https://semmle.com/security (visited on 05/04/2020).

[43] Coveralls. (). "Coveralls," [Online]. Available: https://coveralls.io (visited on 05/02/2020).

[44] E. E. Catmull, "A subdivision algorithm for computer display of curved surfaces," Ph.D. dissertation, University of Utah, 1974.

[45] B. Foundation. (). "About blender," [Online]. Available: https://www.blender.org/about.

[46] ——, (2020). "Render baking," [Online]. Available: https://docs.blender.org/manual/en/latest/render/cycles/baking.html (visited on 05/03/2020).

[47] S. D. Roth, "Ray casting for modeling solids," *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109–144, Feb. 1982.

[48] D. MacDonald, "Space subdivision algorithms for ray tracing," M.S. thesis, University of Waterloo, waterloo, Ontario, 1988.

[49] Wikipedia. (2020). "Voxel," [Online]. Available: https://en.wikipedia.org/wiki/Voxel (visited on 05/04/2020).

[50] A. Storhaug. (Apr. 2019). "Voxelizer v0.1.3," [Online]. Available: https://github.com/andstor/voxelizer/tree/v0.1.3 (visited on 05/04/2020).

[51] Travis. (2020). "Travis ci," [Online]. Available: https://travis-ci.org (visited on 05/10/2020).

[52] R. van Rüschen. (Jan. 2020). "Sparse octree," [Online]. Available: https://github.com/vanruesc/sparse-octree (visited on 05/10/2020).

[53] D. Klein. (Aug. 2018). "Format-vox," [Online]. Available: https://github.com/sh-dave/haxe-format-vox (visited on 05/08/2020).

[54] P. Min. (). "Binvox voxel file format specification," [Online]. Available: https://www.patrickmin.com/binvox/binvox.html.

[55] A. Storhaug. (May 2020). "Binvox," [Online]. Available: https://github.com/andstor/binvox (visited on 05/08/2020).

[56] G. D. A. Team. (2020). "Dat.gui," [Online]. Available: https://github.com/dataarts/dat.gui (visited on 05/08/2020).

[57] G. Johnson. (Mar. 2020). "Three-mesh-bvh," [Online]. Available: https://github.com/gkjohnson/three-mesh-bvh (visited on 05/09/2020).

[58] Mozilla Developer Network. (2020). "Uint8clampedarray," [Online]. Available: Uint8ClampedArray (visited on 05/10/2020).

[59] Facebook. (). "Jest," [Online]. Available: https://jestjs.io (visited on 05/10/2020).

[60] T. Preston-Werner. (2020). "Semantic Versioning," [Online]. Available: https://semver.org (visited on 05/10/2020).

[61] K. Basques. (2019). "Get started with analyzing runtime performance," [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/evaluate-performance (visited on 05/10/2020).

[62] electron-builder. (2020). "Electron-builder," [Online]. Available: https://www.electron.build (visited on 05/11/2020).

[63] Facebook. (2020). "Create react app," [Online]. Available: https://github.com/facebookincubator/create-react-app (visited on 05/11/2020).

[64] Format.js. (2020). "React-intl," [Online]. Available: https://www.npmjs.com/package/react-intl (visited on 05/11/2020).

[65] J. 3. (2011-2017). "Configuring jsdoc with a configuration file," [Online]. Available: https://jsdoc.app/about-configuring-jsdoc.html (visited on 05/10/2020).

[66] GitHub. (May 2020). "Checkout v2," [Online]. Available: https://github.com/actions/checkout (visited on 05/10/2020).

[67] S. Ueda. (2020). "Github pages action," [Online]. Available: https://github.com/marketplace/actions/github-pages-action#table-of-contents (visited on 05/10/2020).

[68] A. Storhaug. (2020). "Jsdoc action issues," [Online]. Available: https://github.com/andstor/jsdoc-action/issues?q=is%3Aissue+ (visited on 05/10/2020).

[69] Androz2091. (Apr. 2020). "How to use a github repo as template?" [Online]. Available: https://github.com/andstor/jsdoc-action/issues/20 (visited on 05/10/2020).

[70] npm. (2020). "Npm registry," [Online]. Available: https://www.npmjs.com (visited on 05/10/2020).

[71] GitHub. (2020). "Versioning your action," [Online]. Available: https://help.github.com/en/actions/building-actions/about-actions#versioning-your-action (visited on 05/10/2020).

[72] smac89. (May 2020). "Actions tagger," [Online]. Available: https://github.com/marketplace/actions/actions-tagger (visited on 05/10/2020).

[73] npm. (Mar. 2020). "Npm-install," [Online]. Available: https://docs.npmjs.com/cli/install (visited on 05/13/2020).

[74] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987, ISSN: 0097-8930. DOI: 10.1145/37402.37422. [Online]. Available: https://doi.org/10.1145/37402.37422.

Appendices

Appendix A

Preliminary report

Preliminary report
for bachelor's thesis

NTNU
Kunnskap for en bedre verden

TITLE
Open source JavaScript voxelization library, with complementary applications

CANDIDATE			
André Storhaug			
DATE	SUBJECT CODE	SUBJECT	DOCUMENT ACCESS
31.01.2020	IE303612	Bachelor's thesis	- Open
STUDY		PAGES/ATTACHMENTS	BIBL. NR.
Computer engineering		21/0	Not used

SUPERVISOR(S)
Primary: Ricardo Da Silva Torres
Secondary: Saleh Abdel-Afou Alaliyat

Summary

This preliminary report concerns the plans and preparations for a bachelor's thesis at the Norwegian University of Science and Technology (NTNU).

The purpose of the thesis is to improve and further develop the already existing open source project named Voxelizer. Voxelizer is a JavaScript library for converting 3D models into a volumetric representation, a process known as voxelization. The library needs to be refined, professionalized and extended.

To ease the use of the library, a cross platform desktop application and a command line interface will be developed. In order to make the Voxelizer library and the complementary software easy to maintain, the projects will have a focus on automation. Especially, a GitHub action will be developed for automating the generation of JavaScript documentation.

Postal address
NTNU i Ålesund
N-6025 Ålesund
Norway

Visitation address
Larsgårdsvegen 2

Internet
www.ntnu.no

Phone
70 16 12 00

E-mail
postmottak@ntnu.no

Fax
70 16 13 00

Bank account
7694 05 00636

Foretaksregisteret
NO 947 767 880

This assignment is an exam submission done by a student at NTNU in Ålesund.

Contents

Chapter 1

Terminology

Concepts

Voxel Three-dimensional analogue of a pixel, representing a value on a regular grid in three-dimensional space.

Voxelization The process of converting a 3D model into voxels.

Library A collection of data and programming code that are used to develop software.

Cross-platform Computer software that can be run on multiple computing platforms.

Abbreviations

MDN Mozilla Developer Network

API Application Programming Interface

UML Unified Modeling Language

GUI Graphical User Interface

CLI Command Line Interface

Chapter 2

Introduction

This project aims to improving an already existing open source [27] library named Voxelizer [20]. It is a cross-platform library for conducting voxelization of 3D models, and is written in JavaScript.

The background for its creation was an assignment in a simulation course. The objective was to simulate diffusion using a cellular automaton. I wanted to do the simulation in the shape of a 3D object. Hence, I needed some way of constructing a volume representation out of a 3D model. Further, I also wanted to make the simulation with web technologies by making use of Three.js [21], an abstraction layer over WebGL [11].

To my surprise, I couldn't find any simple open source JavaScript solution for this. I therefore decided to make a solution myself. The result was the open source library "Voxelizer". However, due to time constraints, the current state of the library can only be considered a crude prototype. It has several issues, lacks important features and needs to be professionalized.

Alongside the library, a desktop program and a CLI interface will be developed. These will be making use of the Voxelizer library, and will greatly simplify the use of it.

Chapter 3

Project organization

3.1 Project group

Table 3.1: Students in the group

Name of student
André Storhaug

3.2 Steering group

The steering group will consist of Ricardo Da Silva Torres at NTNU in Ålesund, along with Saleh Abdel-Afou Alaliya at NTNU in Ålesund. Torres will act as supervisor and Alaliyat will be the co-supervisor.

Chapter 4

Attitudes

As a computer engineer, one is expected to behave responsible and professional. One should be curious of new technology and strive to provide the best solutions possible. Further, one should take proud in ones own work and feel a certain responsibility of the work that is done.

In a world that is becoming increasingly smaller, good collaboration is essential. In the role as a computer engineer, it is expected that one will come into contact and collaborate with persons from many different disciplines. It is therefore vital to have open mindsets and welcome new ideas. Discussions are to be expected. However, disagreements should be kept factual and handled respectfully.

The development of software is often a large part of the job as a computer engineer. Software has potential to affect human lives. Either directly or indirectly. The creation of software should therefore be rooted in strong ethics and respect for the users privacy. With this in mind, open source is a great way of achieving both transparency and openness. Today, everything revolves around profit. Companies are doing everything from charging huge amounts for proprietary software, to profiting on your personal information. However, open source has become a popular platform in which people can collaborate on software projects. By join forces and helping one another, one can achieve truly great things.

Chapter 5

Project description

5.1 Thesis problem - goals - purpose

5.1.1 Thesis problem

There exists an open source JavaScript library for conducting voxelization of 3D models. This library is called "Voxelizer". However, the library faces several issues and is lacking important features. It needs to be professionalized and made easy to both use and maintain.

5.1.2 Goals

This project has two main goals. The first goal is to improve and extend the open source Voxelizer library in such a way that it fulfills the requirements specified in the next section. The second goal is to develop a cross platform desktop application and a CLI for easy voxelization of 3D models, based the Voxelizer library.

In order to ensure maintainability of the various software projects, automation is critical. Therefore, a common subgoal will be to develop a GitHub action in order to automate documentation generation.

5.1.3 Purpose

The purpose of this project is to make it easy to conduct high quality voxelization of 3D models.

5.2 Requirements specification

The scope of this project is defined and limited by the requirements specification defined in following sections. In addition to this specification below, a backlog with user stories shall be created.

5.2.1 Voxelizer

Algorithms

The voxelisation algorithm should provide an accurate render of the original 3D model (polygon mesh [25]). The result should be geometrically representative witout distortions. No holes should be presen, unless dictated so by the given 3D model shape. Internal cavities and structures needs to be accuratly preserved. Lastly, there should be an absolute minimum of artifacts.

It should be possible to do two types of voxelization. One that is a shell voxelization, and another that is a filled volume version. The shell-type algorithm should only capture the surface of the 3D model. The filled-type algorithm needs to capture a complete volume representation of the 3D model.

One should be able to set the wanted resolution of the voxelization.

Input

The library should support a large variety of different input types. Both in terms of various file types and data structures. Support for popular file formats such as OBJ, STL and gLTF should be implemented.

Output

A diverse mixture of output types have to be supported. This includes relevant file formats and data structures. Some file formats for saving voxel data are VOX by MagicaVoxel [6], XML, BINVOX [14] and minecraft SCHEMATIC format. Relevant data structure exports include 3D arrays and octrees.

It should also be possible to export the voxelized result as normal 3D models. This could be file formats such as OBJ, STL and gLTF. Each voxel in the model should be represented as a cube.

Lastly, one could also support image export for each layer of the voxelized result. File format could for example be JPEG or PNG.

Coloring

The texture of a 3D model should carry over to surface voxels. This should be in the form of the most representative color.

Optimization

tree.js raycasting should be optimized. three.js raycasting is CPU based. It itterates each face in a 3D model, checking if the ray intersects a face or not. However, one can speed up the raycasting by employing a spatial index, for example with the help of an octree [24] or aabb tree .

5.2.2 three.js voxel loader

The three.js voxel loader module needs to be able to load voxel data into a three.js mesh [23]. The module should manage to load the voxel file formats and data structures that the voxelizer library supports exporting. This is voxel data stored in the form of a 3D array or an octree, or in a file format like VOX, XML BINVOX or SCHEMATIC.

It should be possible to customize the appearance of the loaded voxels. Both in terms of size, material and/or color.

5.2.3 Voxelizer Desktop

The Voxelizer Desktop shall be a corss-plattform [26] desktop application. It should work on both MacOS, Windows and Linux. The application should be able to voxelize a 3D models with the use of the Voxelizer library [20]. Also, it should automatically update itself when a new release of the application is published.

The application should provide intuitive GUI. It should be possible to view both the original

3D model and the voxelized result. Also, it should be possible to generate a 2D view of the cross-sections of the voxelized model.

5.2.4 Voxelizer CLI

The Voxelizer CLI should be a corss-plattform [26] CLI application. It should function on both MacOS, Windows and Linux. The application should be able to voxelize a 3D models with the use of the Voxelizer library [20].

5.2.5 JSDoc Action

The JSDoc GitHub Action should be an installable GitHub Action [9], available from the GitHub marketplace [2]. It should automate the process of generating JavaScript documentation with the help of JSDoc [12].

5.2.6 Automation

Automation should be used to ease maintenance of the various software projects. Firstly, JavaScript projects needs to have the documentation automatically generated with JSDoc [12]. Secondly, the process of publishing new versions should be automated to the greatest extent.

5.3 Methodology

The method that will be utilize in this project is the agile methodology Scrum [19]. Scrum is a very popular working methodology in the software development business. It uses an iterative and incremental approach, where each sprint gives an opportunity to improve the development process. Scrum organizes the work in sprints. This is a predefined period of time that is devoted to a set of very defined goals. The tasks to be done are often defined in a product backlog [18].

Scrum is mainly intended for teams. However, even though this is a one man project. The Scrum methodology will serve as a project framework for keeping up with progress, in addition to being able to adapt the project pace to the available working capacity. By breaking down the

tasks to be done in sprints, this will help with organizing the work and steering the project in the right direction, allowing adjustmens along the way.

For this project, each sprint will be two weeks long. After each spring, a review of the completed sprint will be made. This will be an opportunity to reflect on the process, and see which goals were completed and which wasn't. Further, this review will be highly valuable for determining if adjustments should be made for the next sprint.

Scrum also seems to be a good fit because there will be a meeting with the supervisor every two weeks. By organizing the tasks to be done in two-week sprints, this will make the meetings with the supervisor more effective and relevant. New functionallity can be discussed and reviewed, in addition to planning ahead for the next two weeks.

5.4 Information gathering

The main source of information will come from various web resources. Everything from articles to code documentation will be needed for this project. The MDN Web Docs [15] will be an important source for JavaScript documentation. For Node.js related work, the Node.js API Docs [16] will be put to good use. Also, documentation from the third party library Three.js [22] will be essential. Further, Stack Overflow [17] will be a highly valued source of information due to its wast amount of questions and answers in a lot of topics.

5.5 Risk analysis

A qualitative approach will be used for assessing the risk of this project. A risk can be described as the likelihood of an event times the impact. A **MEDIUM** risk level will be accepted. The table 5.1 will be used in order to define the various risk levels.

Table 5.1: Risk level matrix.

		IMPACT			
		LOW	**MEDIUM**	**HIGH**	**VERY HIGH**
LIKELIHOOD	**VERY HIGH**	MEDIUM	HIGH	VERY HIGH	VERY HIGH
	HIGH	MEDIUM	HIGH	HIGH	VERY HIGH
	MEDIUM	LOW	MEDIUM	HIGH	HIGH
	LOW	LOW	LOW	MEDIUM	MEDIUM

In table 5.2 below, a risk assessment and risk control is conducted. The letter “L" stands for “Likelihood”, “I” for “Impact” and "R" for "Risk".

Table 5.2: Risk assessment table.

						Residual risk		
ID	**Description**	**L**	**I**	**R**	**Risk control**	**L**	**I**	**R**
R1	Services like GitHub, Jira and Confluence may go down, making various resources unavailable.	L	VH	H	Perform regular backups of important data.	L	M	M
R2	Sickness, resulting in inability to work.	M	M	M	Practicing good hygiene.	L	M	M
R3	Damaged equipment used for development.	L	VH	M	Exercise caution when handling important equipment.	L	H	M
R4	Lost or corrupt files due to system crash or failure.	M	VH	H	Perform regular backups of important data.	M	L	L
R5	Incompatibilities between technologies.	M	M	M	Properly assess the technology and plan ahead before starting development.	L	M	L
R6	Security vulnerability in package dependency.	VH	H	VH	Automatic package auditing and fixing provided by GitHub [10].	L	H	M

VH: VERY HIGH risk

H: HIGH risk

M: MEDIUM risk

L: LOW risk

The risk assessment done in table 5.2 shows that with the appropriate counter measures, all risks are reduced to a MEDIUM level. This is an acceptable level.

5.6 Primary activities in further work

Table 5.3: Main activities.

Nr	Main activity	Time/scope
A1	**Writing**	**18 weeks**
A11	Preliminary report	3 weeks
A12	Bachelor's thesis	15 weeks
A2	**voxelizer**	**7 weeks**
A21	Core improvements	1 week
A22	Algorithm improvements	2 weeks
A23	Texture support	1 week
A24	Extending 3D model file loading	1 week
A25	Extending data exporting	1 week
A26	Write tests	3 days
A27	Optimization	2 days
A3	**three-voxel-loader**	**2 weeks**
A4	**voxelizer-desktop**	**3 weeks**
A41	Core	1 week
A42	GUI	2 weeks
A5	**voxelizer-cli**	**1 week**
A6	**jsdoc-action**	**1 week**
A7	**Automation**	**1 week**

5.7 Progress plan

5.7.1 Master plan

Following is a gantt diagram 5.1 for the planned time sceduling. This includes all activities listed in section 5.6. These activities primarly include writing and software development. Activity A1 is concerned about writing the preliminary report and the thesis. Activity A2, A3, A4, A5 and A6 is concerned with the development of various software systems, where each activity is a confined project. A7 is concerned with automation of various tasks in many of the software projects.

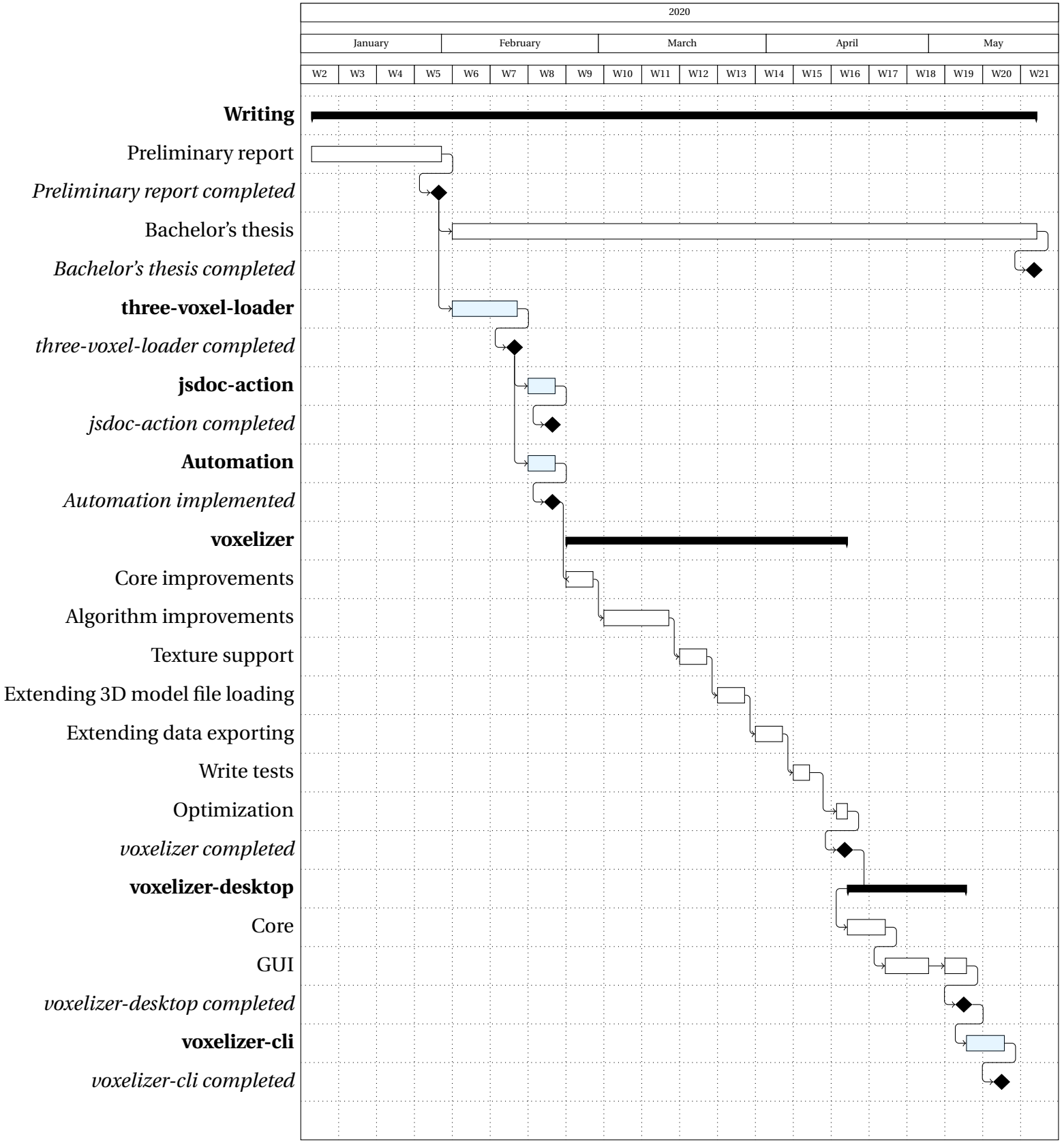


Figure 5.1: Gantt diagram of progress plan.

5.7.2 Project control assets

In order to keep the project on track, Jira [4] will be used. Jira is a project management tool developed by Atlassian, supporting a vast number of features such as issue tracking and project management. The main reason for choosing Jira over for example GitHub's solutions, is that Jira supports the agile methodology Scrum. For managing documents, minutes of meetings, UML diagrams, etc., Confluence [3] will be used.

Since this project revolves around open source projects, Jira and Confluence will only be used for internal related work. For public usage, the GitHub issue tracker and wiki will be used. Any issues, bugs or documentation of public interest shall be be placed on GitHub, instead of Jira and Confluence.

5.7.3 Development assets

For developing the various systems, the development tools listed in table 5.4 will be used.

Table 5.4: Development tools.

Development tool	**Description**
Visual Studio Code [13]	Editor for writing and debugging code.
Blender [7]	3D modeling software.
Git [8]	Version control.
GitHub [1]	Hosting of git repositories.
SourceTree [5]	Git desktop client.

5.7.4 Internal control and evaluation

At the end of each sprint, a review of the completed sprint will be conducted. A burndown chart will be generated for each sprint. This will help identifying if adjustments to the plan is nesseseary.

The requirements specification will serve as the primary criteria in order to decide wether a goal is completed or not. If a system is implemented but contains minor bugs, it will still be considered complete.

Chapter 6

Documentation

6.1 Reports and technical documents

Firstly, a progress report shall be created for every two weeks. Secondly, documentation for the various systems will be produced. In order to make a successful software library, good documentation is imperative. A lot of this documentation will mainly be automatically generated by JSDoc. The automation will be integrated in the workflow of a new version release of a system (GitHub repository). This ensures the validity of the documentation, as well as ensures future maintenance. This integration will be provided by a GitHub action, also to be a part of this project. The generated documentation will be publicly available, hosted at GitHub Pages. Lastly, various UML diagrams will be created. These will serve as illustration for the relationship between the various systems.

As mentioned in section 5.7.2, internal documentation will be kept private in Jira and Confluence. Documents of public interest will be placed publicaly available on GitHub.

Chapter 7

Planned meetings and reports

7.1 Meetings

A meeting with the advisor will be held every two weeks. These meetings will be used for reporting on the current progress. The meetings are an opportunity of gathering constructive feedback from the supervisor. Further, they will serve as documentation for working both professionally and responsible. A minutes of a meeting report will be written for every meeting.

7.2 Progress reports

Progress reports will be developed up-front of each meeting. These will describe what activities were planned, and what activities were actually seen through. If any deviations from the plan occurred during the period, these should also be included in the progress report. Further, the activities planned for the next period should also be outlined. The report will be sent to the supervisor at least a day before the meetings. This will form the basis for the matters to be discussed at the meetings.

Chapter 8

Planned deviation management

In the event of deviations from the current plans, both in terms of progress or content, several measures needs to be taken. If the deviations from the plan are of greater significance, the supervisor should be alerted. If the deviation is of lesser importance, it should be discussed with the supervisor at the regular meeting. One should then consider to change the planned approach.

Many of the planned systems builds upon one another. Therefore, if a task shows to be harder and more time consuming than first anticipated, it should consume time from tasks of lower priority. However, if a task exceeds its planned time scedule because of minor bugs, then these bugs should be properly documented and the task considered finished. These bugs should then be revisited at a later stage if there is time to spare. Since the systems are open source projects, these bugs might also be resolved by volunteers after this project is finished.

Bibliography

[1] About github, . URL https://github.com/about.

[2] Extend github, . URL https://github.com/marketplace.

[3] Atlassian. Confluence, . URL https://www.atlassian.com/software/confluence.

[4] Atlassian. Jira, . URL https://www.atlassian.com/software/jira.

[5] Atlassian. Sourcetree, . URL https://www.sourcetreeapp.com.

[6] ephtracy. Magicavoxel. URL https://ephtracy.github.io/index.html?page=mv_main.

[7] Blender Foundation. About blender. URL https://www.blender.org/about.

[8] Git. About git. URL https://git-scm.com/about.

[9] GitHub. Github actions, . URL https://github.com/features/actions.

[10] GitHub. Managing vulnerabilities in your project's dependencies, . URL https://help.github.com/en/github/managing-security-vulnerabilities/about-security-alerts-for-vulnerable-dependencies#alerts-and-automated-security-updates-for-vulnerable-dependencies.

[11] Khronos Group. Webgl overview. URL https://www.khronos.org/webgl/.

[12] JSDoc. Jsdoc. URL https://github.com/jsdoc/jsdoc.

[13] Microsoft. Visual studio code. URL https://code.visualstudio.com.

[14] Patrick Min. Binvox voxel file format specification. URL https://www.patrickmin.com/binvox/binvox.html.

[15] Mozilla. Mdn web docs. URL https://developer.mozilla.org/en-US/.

[16] Node.js. Node.js documentation. URL https://nodejs.org/api/.

[17] Stack Overflow. Stack overflow. URL https://stackoverflow.com.

[18] Scrum.org. What is a product backlog?, . URL https://www.scrum.org/resources/what-is-a-product-backlog.

[19] Scrum.org. Scrum, . URL http://www.scrum.org/.

[20] André Storhaug. Voxelizer. URL https://github.com/andstor/voxelizer.

[21] three.js. three.js, . URL https://threejs.org.

[22] three.js. three.js docs, . URL http://threejs.org/docs/.

[23] three.js. Mesh, . URL https://threejs.org/docs/#api/en/objects/Mesh.

[24] Wikipedia. Octree, . URL https://en.wikipedia.org/wiki/Octree.

[25] Wikipedia. Polygon mesh, . URL https://en.wikipedia.org/wiki/Polygon_mesh.

[26] Wikipedia. Cross-platform software, 1 2020. URL https://en.wikipedia.org/wiki/Cross-platform_software.

[27] Wikipedia. Open source, 1 2020. URL https://en.wikipedia.org/wiki/Open_source.

Appendix B

Progress reports

Progress report #1 - 13.02.2020

Main purpose / focus

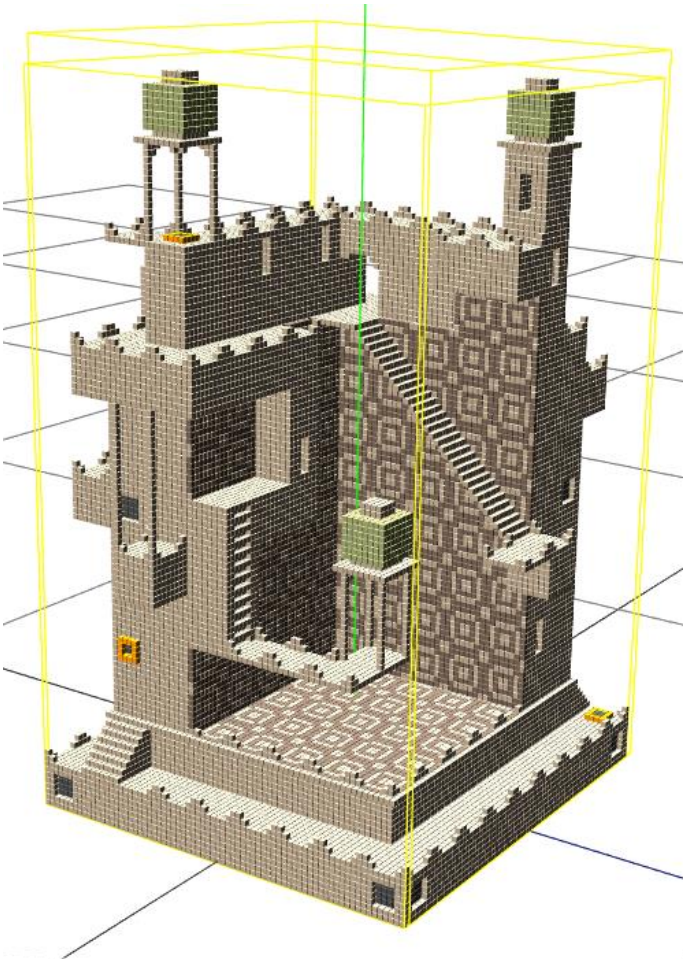
Create a 3D viewer plugin for three.js, in order to visualize voxel data.

Planned activities

1. Generate polygon mesh based on voxel data.
    1. Also read in color of voxels.
2. Implement an octree backed structure.
3. File importing support.
    1. 3D array
    2. VOX (MagicaVoxel editor) file format
    3. XML file format
    4. BINVOX file format
    5. Octree
    6. Minecraft SCHEMATIC file format

Completed work

1. Generate polygon mesh based on voxel data.
    1. Colorizes the voxels.
2. Implement a pointer based sparse octree backed structure.
    1. Added simple Level of Detail (LOD) support.
3. File importing support.
    1. 3D array
    2. VOX (MagicaVoxel editor) file format



VOX file credits:
https://github.com/ephtracy/voxel-model

Progress report #2 - 26.02.2020

Main purpose / focus

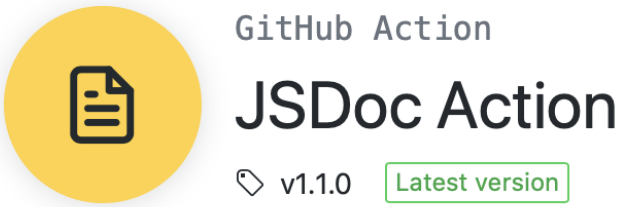
Automation of JSDoc documentation.
Restructure Voxelizer core code base / planning.

Planned activities

1. Create JSDoc GitHub Action.
    a. Publish to GitHub Marketplace.
    b. Upload generated docs to GitHub Pages automatically.
    c. Support JSDoc 3rd party templates.
2. Automatically run tests
3. Automatically publish new JS packages (e.g. on npm)
4. Restructure code base of Voxelizer

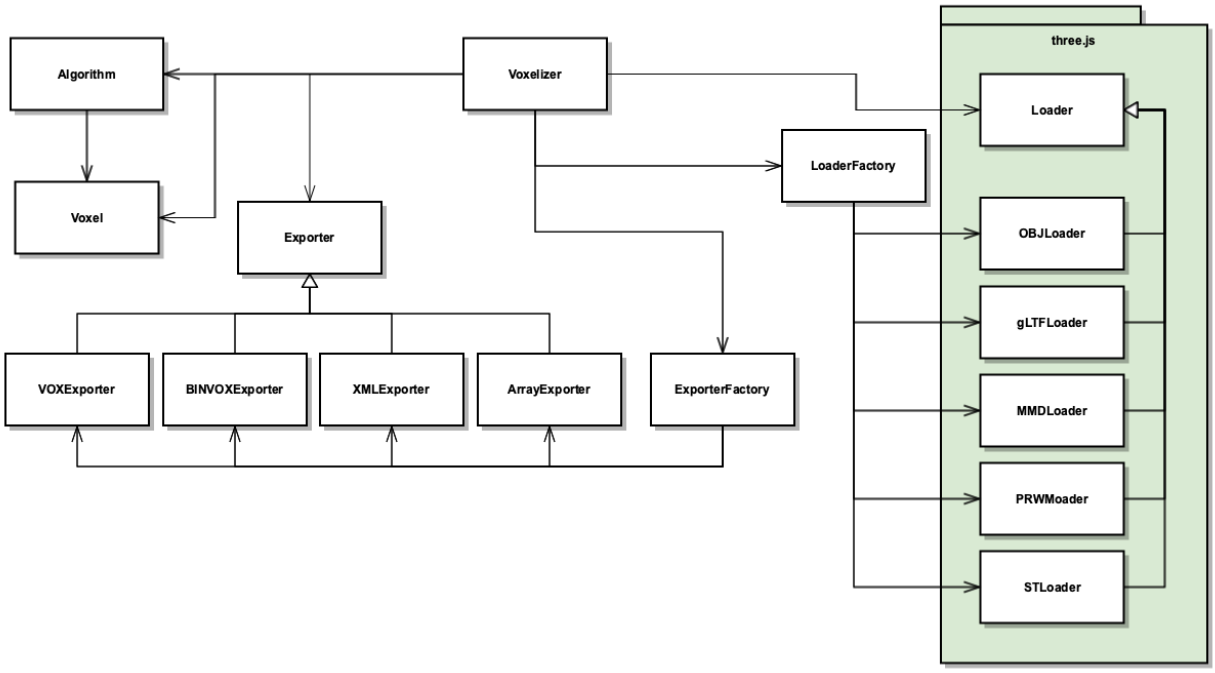
Completed work

1. Created JSDoc GitHub Action.
    a. Supports 3rd party JSDoc templates.
    b. Published to GitHub Marketplace, see here.



    c. Documented simple solution to upload docs to GitHub Pages.
    It can also easily be combined with other deployment actions.
    d. Added link to the jsdoc-action in the main JSDoc repository README file.
    See approved pull request.
2. Restructured Voxelizer codebase / planned ahead.

Voxelizer engine diagram:



Progress report #3 - 19.03.2020

Main purpose / focus

Algorithm improvements

Planned activities

1. Improve algorithm for shell voxelization
2. Improve algorithm for solid voxelization

Completed work

Due to the covid-19 pandemic, I have not been able to complete any of the planned activities these two weeks. The work has been started on, however there is still a lot to do left.

Progress report #4 - 02.04.2020

Main purpose / focus

Finish algorithm improvements and start on surface voxel coloring.

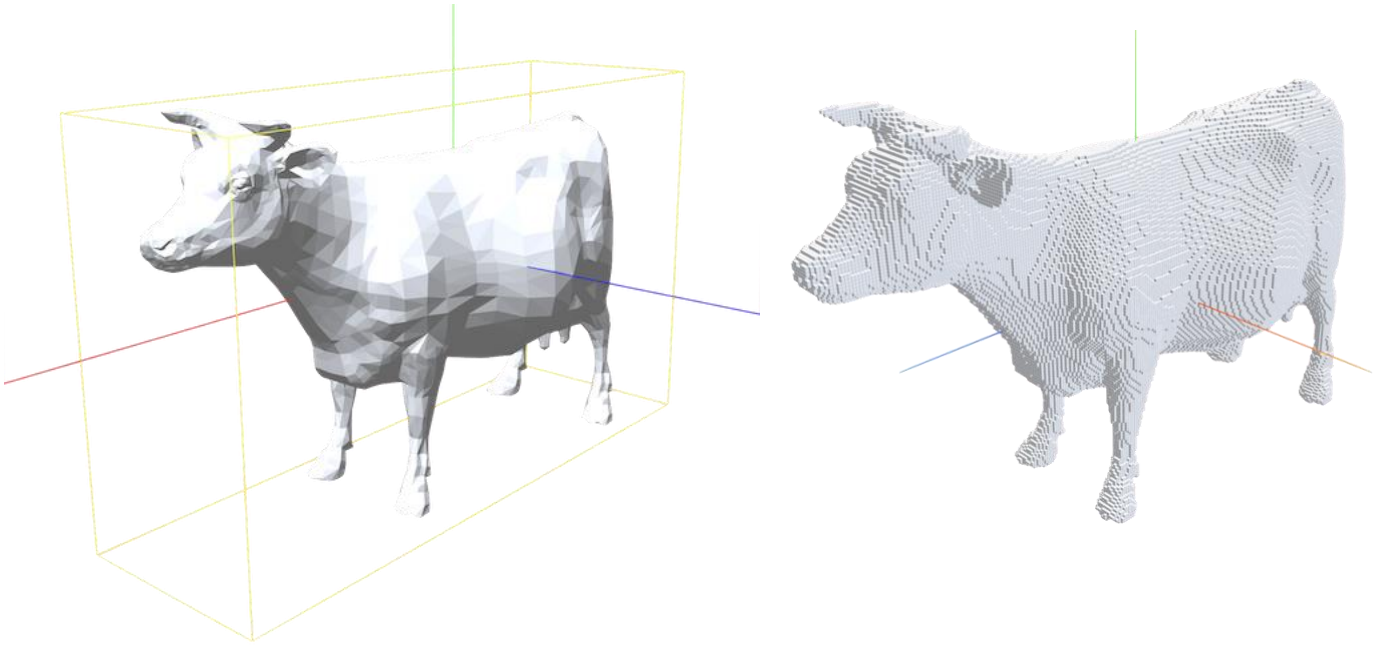
Planned activities

1. Improve algorithm for shell voxelization
2. Improve algorithm for solid voxelization
3. Implement voxel coloring system

Completed work

1. Both the shell and solid voxelization algorithms are completed.
    1. Runtime of O(n^3), excluding time taken for raycasting, where n is the resolution (number of voxels produced is n^3).
    2. Generates geometrically representative result without distortions.
    3. Output is without holes, unless dictated so by the model.
    4. Internal structures and cavities are preserved.

Voxelization example:



The cow model is provided courtesy of the AIM@SHAPE project.

Progress report #5 - 04.15.2020

Main purpose / focus

Support multiple importing formats.
Support multiple exporting formats.
Optimization / improvement of three.js's raycasting.
3D modeling - for testing.

Planned activities

- Implement exporting support
  - BINVOX file format
  - XML file format (coloring)
  - 3D Array (coloring)
  - VOX file format (coloring
  - SCHMATIC file format
- Add importing support of binvox files for three-voxel-loader.
- Implement importing support.
  - STL file format
  - PRWM file format
  - MMD resources
  - glTF file format
  - MTL file format
  - OBJ file format
- Optimize three.js raycasting functionality.
  - Implement a spatial index.
- Create 3D model for testing Voxelizer library.

Deviations

The implementation of the importing support is dropped for the voxelizer library. Three.js provides several "examples" for importing various file formats. Previously, these examples were not included with the base three.js package. The old voxelizer library therefore implemented a wrapper around the OBJ file loader. However, all the loaders are now included in the three.js base package. This makes it much easier to access and use the various loaders. The voxelizer library will therefore leave the process of importing 3D model files into three.js objects up to the user. These loaders produce an Object3D object, which the voxelizer library then can consume.

An AutoLoader was started on, however due to the various loaders requiring a lot of different settings, assets, paths etc. this proved hard to implement properly. Although, this functionality can possibly be used in the voxelizer desktop application.

Completed work

- Several of the exporting formats are implemented.
    - BINVOX file format
    - XML file format (coloring)
    - 3D Array (coloring)
- Importing support of BINVOX files for three-voxel-loader is implemented.

NOTE: The BINVOX functionality has been extracted in a separate repository. See https://github.com/andstor/binvox
It can both parse and build BINVOX files according to the specification.

- The raycasting functionality of three.js is optimized with the three.js plugin https://github.com/gkjohnson/three-mesh-bvh. This utilizes Bounding Volume Hierarchies, effectively reducing the time complexity for raycasting from O(n) to O(log n). It does take some time to generate the tree structure. However, this is negligible for a normal use case of voxelizing a large mesh.
- Created a 3D model of an anvil with Blender.

Anvil 3D model render:

Progress report #6 – 30.04.2020

Main purpose / focus

Voxelizer Desktop application.

Planned activities

- Create Desktop application for using the Voxelizer engine.
    - Voxelize models with the Voxelizer engine.
    - File drop for loading 3D models
        - STL
        - glTF
        - OBJ (and MTL)
    - Internationalization (language translation)
    - React integration for GUI
    - GUI controls options
        - Shell or solid
        - Coloring
        - Clipping
    - Exporting support for the exporters provided by the Voxelizer engine.
    - Logo

Completed work

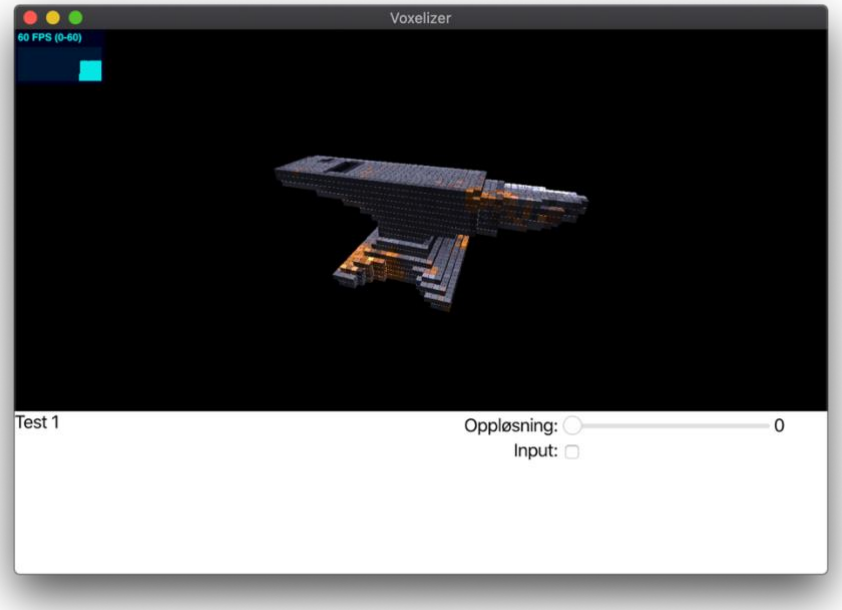
- Electron application core setup
    - React integration
    - Internationalization (language translation)
    - Uses the Voxelizer engine
    - Uses the three-voxel-loader plugin
    - Auto updating
    - File drop for loading 3D model
        - glTF

A lot of time was spent on creating the necessary compilation and development scripts. Security recommendations provided by the Electron documentation has also been studied.

Due to a bug in the bundling process for the three-voxel-loader plugin, the module was not able to be imported by the Electron framework. The bundler had to be changed from Webpack to Rollup. This has also been done for the binvox package. New releases have been published.

Voxelizer Desktop application images
File drop:

Voxelized model result:



Progress report #6 - 14.05.2020

Main purpose / focus

Finish Voxelizer Desktop application.

Planned activities

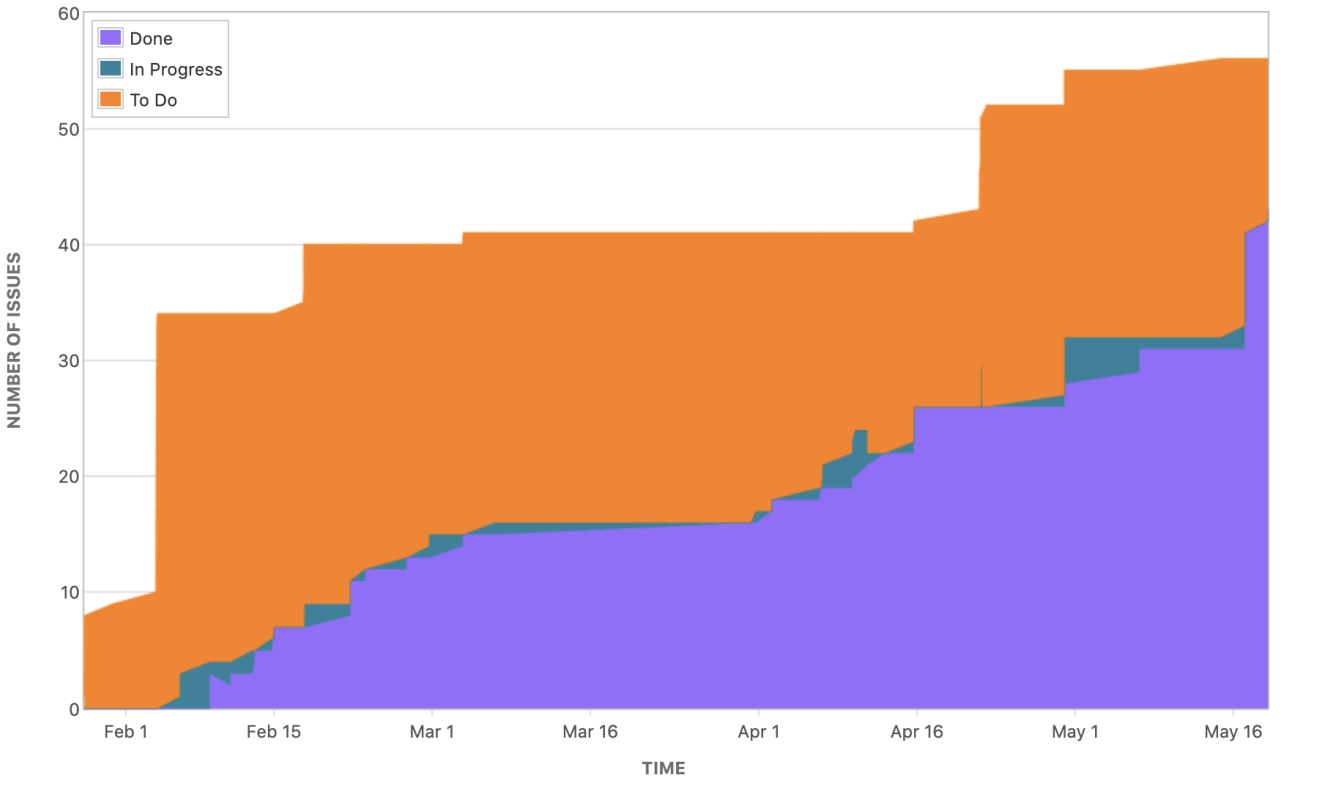
- File drop for loading 3D models
  - STL
  - glTF
  - OBJ
- GUI controls options
  - Shell or solid
  - Coloring
  - Clipping
- Exporting support for the exporters provided by the Voxelizer engine.
- Logo
- Build and publish the application.

Completed work

All the planned activities are completed, except clipping. The application is packaged and uploaded to GitHub. Installation files for both Windows, Linux and macOS can be found at https://github.com/andstor/voxelizer-desktop/releases/latest

Appendix C

Sprint cumulative flow diagram



Appendix D

Backlog

Backlog - voxelizer

Table of Contents

3D Testing Model

DONE

A 3D model should be created to be a able to test the system.

It needs a relatively high level of complexity.

It should be textured in a way to showcase the coloring functionality of the Voxelizer system.

Resolution

DONE

As a user, I want to be able to set the wanted resolution of the voxelized output.

Relates	
relates to	Shell voxelization
relates to	Solid voxelization

Restructure code base

DONE

As a developer, I need the core codebase to have a good structure and be easy to maintain, so that other functionality that builds upon the core is easy to develop with high quality.

three.js raycasting optimization

DONE

three.js itterates all faces during raycasting. This needs to be drastically reduced. It can most likely be implemented with a spatial index.

Surface voxel coloring

DONE

As a user, I want to be able to produce color voxelizations.

Blocks	
is blocked by	Shell voxelization

MTL file format import support

CLOSED

Blocks	
is blocked by	OBJ file format import support

Solid voxelization

DONE

As a user, I want to be able to produce a filled volume voxelization of a 3D model.

Relates	
relates to	Resolution
Blocks	
is blocked by	Shell voxelization

Shell voxelization

DONE

As a user, I want to be able to produce a shell voxelization of a 3D model.

Relates	
relates to	Resolution
Blocks	
blocks	Solid voxelization
blocks	Surface voxel coloring

SCHEMATIC file format export support

TO DO

As a user, I want to be able to export the voxel data to the SCHEMATIC file format, so that i can import it into the game Minecraft.

BINVOX file format export support

DONE

As a user, I want to be able to export the voxel data to the BINVOX file format.

XML file format export support

DONE

As a user, I want to be able to export the voxel data to an XML file.

VOX file format export support

TO DO

As a user, I want to be able to export the voxel data to the VOX file format, so that i can import it into the MagicaVoxel editor.

Octree export support

TO DO

As a user, I want to be able to export the voxel data to an octree data structure.

3D array export support

DONE

As a user, I want to be able to export the voxel data as a normal nested JavaScript array.

STL file format import support

CLOSED

glTF file format import support

CLOSED

OBJ file format import support

CLOSED

Blocks	
blocks	MTL file format import support

Automatic testing

DONE

As a repository maintainer, I want all new code changes to be automatically tested.

Automatic publishing

DONE

As a repository maintainer, I want the publishing of new modules to be automated.

Backlog - three-voxel-loader

Table of Contents

Octree import support

DONE

As a user, I want to be able to load voxel data stored in an octree data structure, so that I can easily view voxel data.

3D array import support

DONE

As a user, I want to be able to load voxel data stored in a 3D array, so that I can easily view voxel data.

Generate three.js mesh

DONE

As a user, I want to be able to generate a three.js mesh from voxel data, so that I can visualize the voxel data.

Relates	
relates to	Customize styling of voxels
Blocks	
is blocked by	Octree backed structure

Octree backed structure

DONE

As a user, I want the voxels to be stored in an octree, so that I can manipulate the voxels easily.

Blocks	
blocks	Generate three.js mesh

Customize styling of voxels

DONE

As a user, I want to be able to customize the styling/appearance of the voxels.

Relates	
relates to	Generate three.js mesh

SCHEMATIC file format import support

TO DO

As a user, I want to be able to load voxel data stored in SCHEMATIC file format, so that I can easily visualize voxel data.

BINVOX file format import support

DONE

As a user, I want to be able to load voxel data stored in BINVOX file format, so that I can easily view voxel data.

XML file format import support

DONE

As a user, I want to be able to load voxel data stored in XML file format, so that I can easily view voxel data.

VOX file format import support

DONE

As a user, I want to be able to load voxel data stored in VOX file format, so that I can easily view voxel data.

Backlog - voxelizer-desktop

Table of Contents

Dark mode

DONE

As a user, I want the application to honor my computer's dark mode setting.

Desktop application

DONE

As a user, I want to be able to be able to run the program as a desktop application, so that it I easy to conduct voxelization of 3D models.

Exporting support

DONE

As a user, I want to be able to export the vowelized result to a file. The file type should be of one of the types supported by the voxelizer library.

Importing support

DONE

As a user, I want to be able to import 3D models of various file formats.

The 3D model file formats that should be supported are the ones which three.js supports.

Acceptance Criteria:

File formats:

- OBJ
- STL
- glTF

Shell or solid option

DONE

As a user, I want to be able to toggle between doing a shell or filled voxelization.

Coloring option

DONE

As a user, I want to be able to toggle between whether or not to do color voxelization.

Clipping

TO DO

As a user, I want to be able to clip off parts of the model, so that it is easy to inspect the internals off the voxelized result.

Resolution

DONE

As a user, I want to be able to specify the resolution of the voxelization.

Logo

DONE

As a user, I want a logo for the application, so that I can easily find it among my other applications.

Automatic updates

DONE

As a user, I want that the application updates automatically when new releases are published. This way, time could be spent actually using the application, not maintaining it.

Voxelization

DONE

As a user, I want to be able to voxelize a 3D model easily with the help of the voxelizer library.

Language

DONE

As a user, I want the application to be displayed in my language, so that I can easily understand the content.

Backlog - jsdoc-action

Table of Contents

Upload to GitHub Pages

DONE

As a user, I want to be able to publish the generated documentation to GitHub Pages.

Templates

DONE

As a user, I want to be able to use a custom template for JSDoc, so that I can customize the style of my documentation.

Publishing

DONE

As a user, I want to be able to install the GitHub action from the official GitHub marketplace, so that it is easy to find and install the GitHub Action.

GitHub Action

DONE

As a user, I want to be able to generate JSDoc through a GitHub Action, so that it is easy to generate JavaScript documentation

JSDoc

DONE

As a user, I want to be able to generate JavaScript documentation with the use of JSDoc, so that I can generate high quality documentation.