Upcast:
Online 3D Model Creator and Sharing Service
Final Project Report

TU857
BSc in Computer Science (Infrastructure)

Eoin McMahon

C19382426

Dr. Mariana Rocha

School of Computer Science

Technological University, Dublin

31/03/2023

Abstract

Tabletop Role Playing Games (TTRPGs), a genre of board game that has its roots in 1970s with the cult classic Dungeons and Dragons and has historically been followed by certain niches, has hit a resurgence in the past decade with their inclusion in recent popular media such as Netflix’s Stranger Things as well as podcasts hosted by established voice actors playing said games such as Critical Role. This extension into popular media, further solidified with a Dungeons and Dragons branded feature film having been announced, has seen a drastic increase in up take in these games by people who otherwise wouldn’t have given them any thought.

Despite this uptake, playing these games can become an expensive hobby, with the necessary rule books and supplements required to play costing upwards of 100 Euro, which can turn people away or limit what else they can spend on the game itself, like on scenery and maps. Many players would want to use a mini 3D environment in which they can play to mimic the podcasts that would have inspired them but would have few facilities to create such things.

This project aims to provide players of these games a free-to-use service to create and share 3D models based on existing 2D ‘maps’ used in playing games such as Dungeons and Dragons, allowing for more accessible 3D environments to improve the atmosphere that playing the game involves while also enabling users to tap into their creativity and share with others on a dedicated platform where they might not have otherwise. The 3D capabilities will be provided in browser through webGL, allowing users to pick it up with little commitment.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

[illegible]

Eoin McMahon

31.03.2023

Acknowledgements

I would like to thank the following people, who paved the way for this project to exist:

- Jonathan McCarthy, for organizing and managing the Final Year Project module as a whole, alongside giving helpful advice
- My supervisor, Dr Mariana Rocha, for keeping progress on track with draft deadlines and words of encouragement
- My peers at TUD for listening to potential ideas and giving their honest feedback
- My friends outside of the university, who are part of the intended audience of this project and thus gave feedback in that regard
- The TUD counselling service, for hearing out and providing support for any emotional or stress related issue I have faced during the course of this project

Table of Contents

Table of Figures

1. Introduction

The following chapter will cover the introduction to my project, giving necessary background information and details on what I want and don't want to implement, what I may need to do to achieve this and outline the rest of report.

1.1. Project Background

Tabletop role playing games (TTRPGs) such as Dungeons and Dragons have seen a surge of popularity in the recent years, with their inclusion in popular media such as the TV show Stranger Things and web shows focused around said games like Critical Role. This recent spotlight had incited many people, previously unfamiliar with such games, to pick up and learn how to play. While the rules to play are the same with whomever you play, how the world in which you would play can be represented in many ways, based on the preference of the players. It can be described in words alone, represented by drawn maps or even in 3D sculptures and 3D printed environments, though the latter options do require monetary investment.

Out of all the media through which the game can be played, 3D maps, either physical pieces or virtual models, can be the most difficult to set up or acquire. They inherently require more effort to create, necessitating skill in crafts or knowledge of modelling software which can take time to learn and effectively use. In spite of the costs, these models can help make players mor invested in the game as well as keep of track of their character's position in a 3D space, which can be difficult when using a 'theatre of the mind' or 2D map approach.

Free online services exist to facilitate the different styles of play, such as token makers to make neat character references for use on 2D maps or free-use art websites to illustrate what the players might be seeing for minimum investment. Seeing these around, it would make sense to offer a simple 3D modelling service in the browser that can easily share its creations for free, lowering a barrier for entry into this format of playing these games.

1.2. Project Description

This project aims to develop a website that can allow users create 3D environments and share them with others for no cost. This service will provide an editor to upload a 2D image of a background or existing 2D 'battle map', allowing one to extrapolate features such as terrain, walls and similar, resulting in a complete 3D object. This object can be saved directly to your device or shared on the storefront, where relevant info will be displayed. This storefront can be accessed by anyone to then download the 3D object for use in 3D display or 3D modelling.

1.3. Project Aims and Objectives

For a successful implementation of this project, the following objectives should be achieved:

- Research history of 3D computer graphics
- Learn about openGL and the browser implementation webGL
- Touch up on web design (html, CSS)
- Look for a language and/or framework that utilises 3D rendering
- Research existing digital storefronts and take inspiration for design

- Create a website to host my main features(frontend)
- Implement an editor which takes a 2D image, allows modifications in 3D and can save as 3D object
- Create database for storefront entries
- Design the storefront, linking it to database and allowing the editor to create entries
- Implement search function, with tags

1.4. Project Scope

This project will focus on the creation of 3D objects from 2D images and the storing of said 3D objects in a digital storefront for users to easily access them. The storefront will have pages for each item, containing information such as the user who uploaded it, date of creation, rating, downloads etc, while also having appropriate tags assigned to allow for filtering of search results, these tags being assigned automatically based on the image used or manually by the creator.

The website will not facilitate the running of tabletop games, as there exists services a plenty for managing the dice rolls, note taking and much more out there, while I plan to facilitate easier access to 3D models for players who want to enhance their games.

1.5. Thesis Roadmap

This is an overview of the following chapters, summarising what they will cover in this project.

1.5.1. Literature Review:

- Investigation of existing solutions to the problem this project aims to solve, while also researching the technologies most suitable for its implementation.

1.5.2. Experiment Design:

- Software methodologies are reviewed, and the overview of the system implementation is given

1.5.3. Prototype Development:

- A proof of concept for the application is designed and implemented, with figures and examples from the implementation

1.5.4. Experiment Development:

- Implementation of the application, using planned technologies and features, building off of the developed prototype.

1.5.5. Testing and Evaluation:

- Methods for effectively testing and evaluating the system at hand are chosen, describing how they will improve the implementation

1.5.6. Conclusion and Future Work:

- Lessons learned and issues encountered during the implementation as well as plans for the future of the project are discussed

2. Literature Review

2.1. Introduction

This chapter will detail the research conducted prior to beginning the implementation of the project, including existing services that may serve similar purposes and the individual functions that will make up the whole end product.

2.2. Alternative Existing Solutions

Before researching the implementation of this project, it is important to look at existing applications or services that may provide a similar product. As this project involves both 3D model editing and model sharing features, it would be fruitful to look at existing implementations in such domains.

2.2.1 3D Modelling

2.2.1.1 Talespire

A 3D virtual tabletop application, focusing on creating immersive 3D environments in which to set a tabletop game. It features map creation using various terrain and colour tools, with additional decorations to be placed. This also allows for playing the game as well, allowing for addition of models representing players and enemies, facilities for rolling dice and other relevant features. These creations, aside from being experienced by the creator, can be shared with others by manually exporting the data to another installation of this application.

The downsides in using this would be that it is in early access; as a crowd funded project, the version available to the public is not feature complete, not yet featuring advertised functionality such as in-app sharing of creations and user submitted materials and models. On top of this, this is a paid service which has to be bought by the person wanting to create a map and by every other user who wants to experience said map. This price of ~20€ can be daunting for users who would not want to spend much on this hobby, considering the other costs of purchasing required rulebooks, and as such, offering a free-to-use service available on the web browser would help to attract this niche.[1]

[1] 'TaleSpire - FAQ'.

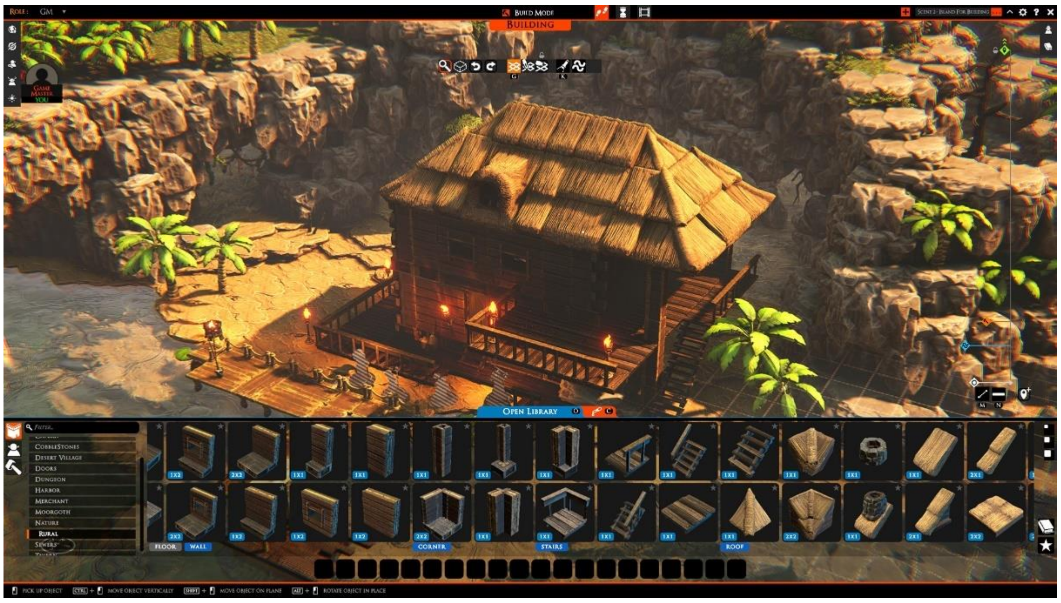


Figure 1: Talespire Editing Suite (Source: Talespire Website)

2.2.1.1 Blender

A free and open-source 3D toolset, featuring modelling, rendering, animations and much more. It is available on all platforms including MacOS and various Linux distributions, which can allow for a wide adoption regardless of device limitations. This software is well documented, allowing for new users to quickly start modelling and exporting their desired creations, being able to use them in a variety of software such as game engines and other animation suites. This is helpful for those who want to quickly create a map for their own purposes with little cost and time investment.

Despite the user-friendliness, the sheer depth of customisation available to users could still scare off new adopters. On top of this, model sharing is done through an external website Blendswap, which is restricted by creative commons licenses. Offering a simplified editor alongside a dedicated storefront for such maps and environments would be helpful for those finding themselves overwhelmed in using Blender.[2]

[2] 'About — Blender.Org'.

Figure 2: Blender 3D (Source: Blender website)

2.2.2 Model Storefronts

2.2.2.1 Sketchfab

Website offering 3D model viewing and sharing, offering exports in a variety of file types. Created in 2012 due to lack of widespread model sharing services at the time, now officially integrated in many main stream modelling software such as Blender and the Autodesk suite of modelling tools, allowing for direct upload and integration of existing models in larger projects. This is due to their official API that allows swift integration of aforementioned services on top of model viewing in app, browser and even augmented/ virtual reality devices.[3]

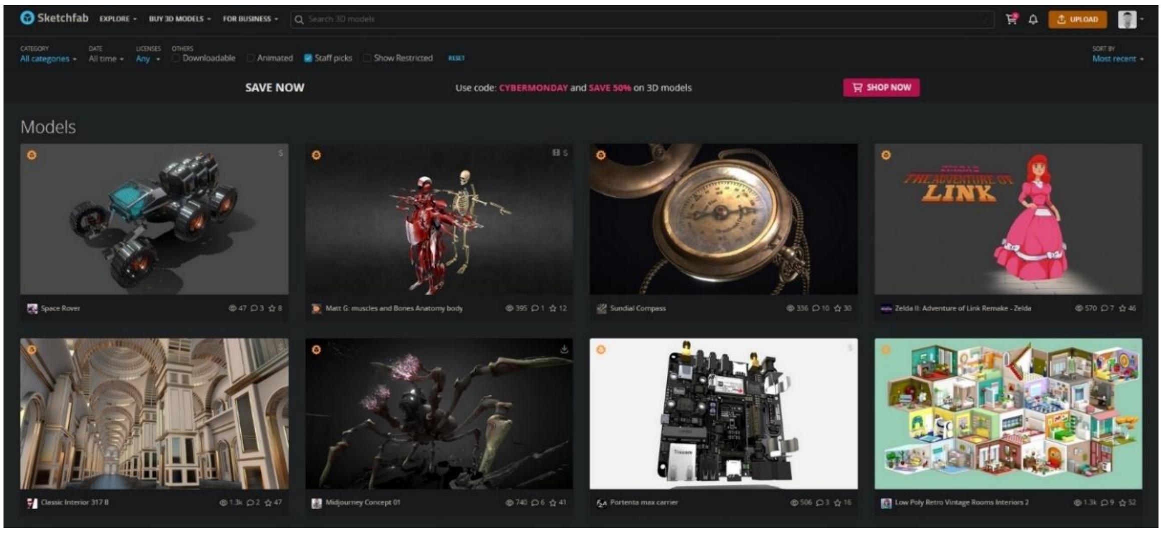


Figure 3: Sketchfab Model Search (Source: Screenshot of Sketchfab Website)

2.2.2.2 Blend Swap

[3] 'Sketchfab - The Best 3D Viewer on the Web'.

Website for sharing models created in blender, dedicated to the blender community. Set up by Blender users who wanted a dedicated place to share models created in said software, allowing them to display the extent and quality once can achieve in using it. Models can be uploaded to different categories depending on what niche they fill, for what they are intended to be used and so on. Uploads are moderated by staff to ensure they fit within the Creative Commons License by not using materials, textures, or models under ownership of an external party.[4]

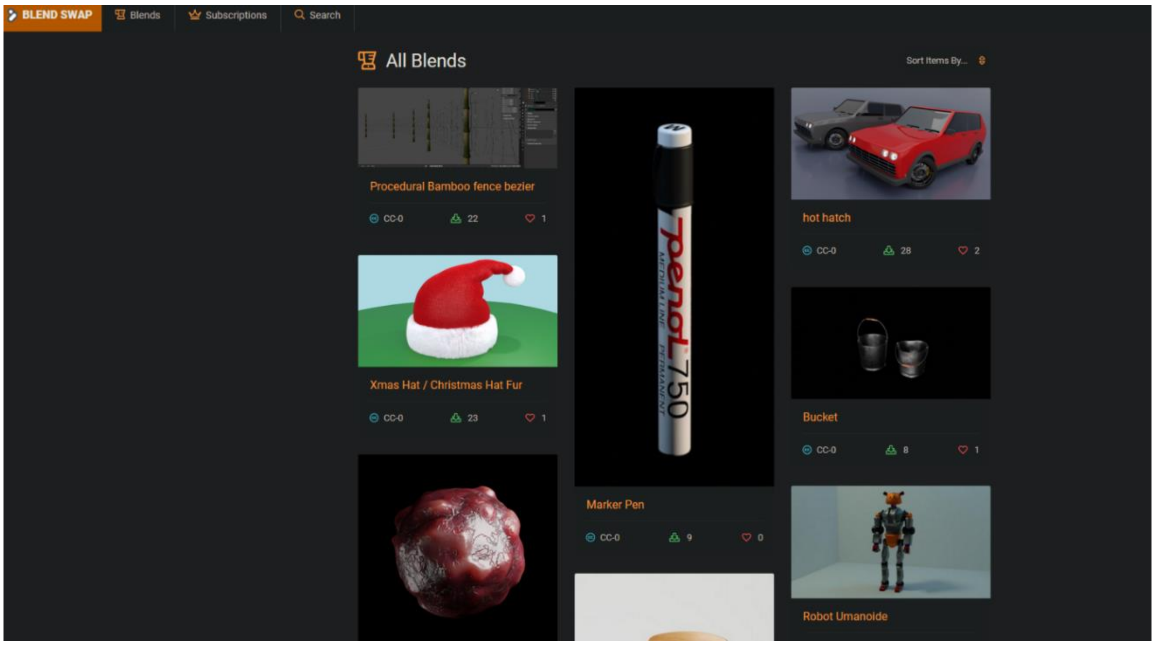


Figure 4: Blend Swap Model Search (Source: Screenshot of Blend Swap Website)

2.2.3 Findings:

Having researched the above services, it is clear that these are alternatives to what could be implemented in this project, as they offer services for easily creating and sharing models for most anyone with access to a computer, allowing for wide accessibility for the intended demographic. While Talespire allowed for creation of models with built in tools and resources, it can be limiting due to a lack of importable objects to vary the output and currently no integrated method of sharing said creations, while costing all who wants to experience it the price of this software. Blender is the closest to the current project intention, as it is free, open source and can natively export to numerous file sharing websites including Sketchfab and Blendswap, both of which offer robust model viewers and search functions to allow for narrowing down what one would like to find when looking for certain models.

Looking at all this, this project would benefit from a model viewer to allow users to preview the model

[4] 'Blend Swap'.

2.3. Technologies Researched

Having looked at existing solutions, it is important to decide on technologies what to use for this project.

2.3.1 Desktop vs Web-based

When first considering this project, it is important to decide on which platform to focus. For what is required, there are two options:

2.3.1.1 Desktop

Creation of a client-side application which runs locally on a user's device. This gives access to local hardware and accompanying languages and libraries that would not be available otherwise on a web browser, such as C++ and DirectX, while also allowing access to some features even without an internet connection. On top of this, less strain is put on a given internet connection than if all rendering and modelling were to be done on this. The model sharing would be handled by external database/ file sharing service and the application would retrieve data stored there for users to browse.

On the flipside, this would mean that the implementation would have to be modified for different operating systems, with executable applications requiring different build methods for windows than it would be for MacOS or Linux distributions, while some specific libraries for 3D modelling or other specific hardware drivers may be unavailable otherwise.

2.3.1.2 Web-based

Website hosting both 3D rendering and model sharing services, the former using libraries for necessary rendering while the latter requiring a combination of front- and back-end communication to facilitate the storefront. The nature of the web application allows for anyone to use the service regardless of hardware or operating system, as these apps can be accessed on a multitude of web browsers with few issues. This requires no commitment to downloading a program which needs to be re acquired when using a different device.

With online applications, an internet connection is required for all features, blocking access for those who may not have WIFI temporarily, while potentially placing strain on said connections if the operations are not optimised, with 3D rendering being more resource intensive than uses for web applications.

2.3.1.3 Decision

In conclusion, this project would benefit from being a web application, as the service would fit a different niche than existing implementations such as Blender, allowing for quick 3D rendering in a modern browser and equally swift sharing of the resulting creation with little hassle. Despite fewer alternatives for 3D graphics APIs on browsers, the existing frameworks are well documented and have supporting libraries for adding more and more features such as Virtual Reality Support and a physics engine. These libraries will ease the implementation of additional features down the road by providing a basis for more functionality.

2.3.3 3D rendering

A major piece of this project includes 3D rendering of models in a browser space. With this in mind, it is important to look at the APIs and libraries available in browsers.

2.3.3.1 WebGL

A 3D rendering API embedded in most modern browsers, released in 2011 by Khronos to allow for wide access to 3D rendering without use of external plugins. Based off an embedded version of OpenGL, a royalty free 3D rendering API that widely used in both personal and commercial projects, from hobby rendering to being used in large scale projects such as video game development. This derivation of an embedded API allows for it to be used across many devices and browsers, including mobile devices, while being light weight for an average internet connection, as it replaces the previously required browser plugins for running 3D content such as Unity Real Time Player and Java Realtime Environment, which plagued users for bloating the browser and taking up space.[5]

2.3.3.2 Three JS

An open-source JavaScript library released shortly after webGL as an addition to the base functionality of the latter, which could only render lines and basic shapes at the most basic level. This library offers a more user-friendly interface for rendering objects while adding features such as textures, lighting, shaders and much more. This higher-level framework allows for more component-based creation of cameras, lighting and scenes to then render to a canvas component on a web page, integrating directly into HTML5.

On top of the base functionality, this library can implement further functionality through additional libraries for animations, physics engine implementations, exporting scenes to common file types and much more. This can allow for simpler implementation of complex functionality that would otherwise take too long to be worth creating. [6]

2.3.3.3 Decision

While handling 3D models and rendering on browsers, webGL is really the only framework one can use due to how integrated it is across devices. With this, it is important to choose the correct library/ additional framework to allow for customisability depending on the intended project. For this, Three.js fits what is needed while also leaving room for possible additional features that may be helpful down the line.

2.3.4 Databases

Considering this project will centre around the creation, manipulation and sharing of 3D object files as well as the management of accounts, it is important to decide on a suitable database and file management system.

[5] 'WebGL'.
[6] 'Fundamentals - Three.Js Manual'.

2.3.4.1 MariaDB

An open-source relation-based database system managed by the organisation of the same name. Created in 2009 following Oracle's acquisition of MySQL as an open-source database compatible with the language to preserve its wide non-commercial usage. It's quite popular as many developers switched to it to maintain their previous MySQL implementations.

Based on Structured Query Language (SQL), it can support many of its features, including JSON API support, data processing and more. It can run on all kinds of systems (Windows, Mac OS, Linux distributions) and can be interfaced by many mainstream languages including C++, Python and JavaScript. On top of this, the database can be deployed to cloud services such as Microsoft Azure.[7]

2.3.4.2 Oracle

A relational database base run by Oracle, one of the largest and most trusted database providers globally. One of the longest running database services in the world, having been developed since the late 1970's Using a relation framework, data can be directly accessed by user's client- and server-side using a proprietary version of SQL.

The architecture of the database is split into logical and physical, allowing for the physical aspect of hardware can be modified without affecting the data already present in logical form. On top of this, failure in such devices would not have too great an impact on the data either as it would then be localised to only a part of either architecture. This prevents the networked storage scheme from crashing due to minor faults. To this end, Oracle is favoured by global enterprises who need stability and security more than anything.

This runs on all major operating systems and can be hosted on the Oracle Database Cloud Service.[8]

2.3.4.3 MongoDB

A free NoSQL, document-oriented database system maintained by the eponymous organisation since 2009, focusing on scalability and flexibility. It stores data in documents similar to JSON files, allowing for customisation between different databases to suit the needs of a user. With the documents being close to JSON format, they can be mapped to objects in code with little hassle, facilitating their use in search and analysis functions. It is also designed as a distributed database which can be highly available and can scale well with additional hardware and documents if the need arises.

It can run locally on all major operating systems while also offering a free cloud hosting service in MongoDB Atlas.[9]

2.3.4.4 Firebase

[7] 'About MariaDB Server'.
[8] 'Cost-Optimized and High-Performance Database'.
[9] 'What Is MongoDB?'

A NoSQL database cloud hosted by Google Firebase alongside its various features. Similar to mongoDB, it uses a document framework, this time using standard JSON format that can be easily accessed and manipulated by almost any code, utilising the freely available APIs provided.

Although cloud hosted, data stored on it is persisted locally when used, allowing for updates and usage even when connection is lost, syncing offline changes to the cloud when re-established. The database also offers security rules to define how data is accessed by users, also providing the option to implement Firebase authentication before querying a document.

Firebase also offers a web storage solution, allowing for storing and access of files by users and admins alike.[10]

2.3.4.5 Selection

Considering the above databases, Firebase was found to have fit the project the most. The real time aspect will be helpful for maintaining data persistence between all users accessing the model sharing service. Additional security rules are nice as it is vital that user info cannot be exposed to unauthorized external users.

The big selling point is the additional file storage functionality hosted on the same cloud as the database, allowing for seamless referencing and storing of the 3D modelling files that will be a main focus for this project.

2.3.5 Front-End Frameworks

In the process of making an interactive web page, the use of front-end frameworks become necessary to ensure that the page is responsive and looks modern. To decide on which front-end to use, the following have been researched.

2.3.5.1 React.js

A JavaScript library which facilitates the rapid development of rich user interfaces for web applications. This is accomplished by allowing developers to make simple views for each state in their application, which will render their functions as components. Having each feature in separate containers makes it easier for one to develop a UI feature quickly in JavaScript then integrate it into the overall project with many other components. As these are written in JavaScript, objects and data can be directly referenced and utilised without necessitating use of Document Object Model (DOM) which can bog down similar UI libraries and frameworks.[11]

2.3.5.2 Svelte

An open-source component framework like React, similarly built to accommodate JavaScript, HTML and CSS to create user interfaces for web applications. Although similar, React continuously runs most of the code in the browser, complete with background tasks such as garbage collection (memory management) that can bog down browser performance. Svelte circumvents this issue by

[10] 'Firebase'.
[11] 'React – A JavaScript Library for Building User Interfaces'.

compiling all components when the application is first run into efficient code, which allows a swift start up and loading between pages.[12]

2.3.5.3 Selection

Despite the speed benefits offered by Svelte, the selection for this project would be React, as it is better supported for longer with an appropriate amount of documentation.

2.3.6 Back-End Frameworks

As this project requires the implementation of databases and file storage, a back-end framework is required to manage communication between elements on the web page and external resources.

2.3.6.1 Django

Free, open-source web framework released in 2005. Written in python, it provides a multitude of code for common operations such as database manipulation, templates for HTML, URL routing, security and much more, allows for swifter development by use of these code snippets. The design of the framework allows for each component to be easily separated for simple editing of the code. This framework is compatible with all major operating systems, while also implementing security features to circumvent potential security attacks across these platforms.[13]

2.3.6.1.1 Express.js

A web application framework based on Node.js, focusing on minimisation of code required to execute standard back-end tasks such as those mentioned above. A benefit of this utilising Node is that it can be used for both front- and back-end, with Node being based off of vanilla JavaScript which has many additional libraries and frameworks to create rich user interfaces that can directly communicate to the back-end through Node. This can simplify the communication between these frameworks and reduce the time spent alternating between languages and documentation to create a functional rich web application. [14]

Express also excels in handling great amounts of user input handling, so much so that companies such as Uber utilise it to manage their back-end.

2.3.6.1.2 Selection

Express.js is the choice for the back-end framework for this project, as it would be convenient to use this in conjunction with the selected front-end of React through the medium of NodeJS, while taking advantage of the code templates/ snippets that Express provides. In particular, user input will be important for both the 3D modelling and sharing if many users were to use it simultaneously, so this would be greatly beneficial.

[12] 'Svelte • Cybernetically Enhanced Web Apps'.
[13] 'Django'.
[14] 'Express - Node.Js Web Application Framework'.

2.4. Other Relevant Research

Aside from the technological requirements for this project, it is also important to look into the background of tabletop role playing games, Dungeons and Dragons in particular, and how they have risen in popularity in the past few years, while also seeing how the COVID19 situation would have affected how people play these games.

2.4.1. Surge in popularity + Accessibility

When it comes to tabletop role playing games, Dungeons and Dragons is by far the most recognisable, having become synonymous due to the relative popularity it had since its inception in 1974 by Tactical Studies Rules. Despite the current phenomenon of popularity it has received recently, it wasn't always in the mainstream of media consumption due to certain stigmas that it had accrued throughout the years.

For one, the nature of the game and its subject material, namely the role playing aspects, the number-crunching of statistics and dice rolls for the main gameplay loop and focus on fantasy settings and stories had tied this past time to a section of nerd-culture, enforced by stereotypical depictions of such culture having characters who enjoy this game and other related media being stigmatised or played off as a joke. This attachment to nerd culture would make it harder for those with passing interest to start playing as they could be self-conscious about being seen playing it and their 'reputation' suffering as a result.

Another unfortunate label assigned to games such as these was the belief that they were satanic in nature, regarding the use of demonic imagery in the stories being told through them. Some countries, particularly in religious parts of the United States, would decry the game as a bad influence on Christian people, leading to local activism against and attempted banning of such materials by local people. This fear would become less prevalent over the years but could still have prevented many in more conservative areas from taking part in this hobby.

These stigmas would generally prevail over the multiple editions of the game released throughout the years until the early to mid-2010s, when more and more pop culture media would start to promote or feature Dungeons and Dragons, a big-name example being Netflix's Stranger Things, depicting an older edition in a show set in the 80s, where the game would have been at its most divisive. This surge of interest coincided with the release of the 5th edition of the game, designed to be newcomer friendly and the system of choice for many of the online tabletop podcasts such as Critical Role, featuring recognisable voice talent which would garner even further attention towards the game.

Within a span of a few years, TTRPGs went from a niche audience to a grander, more general one, leading to many more wanting to try their hand at the game, with most players only recently getting into the hobby when compared to those who had played before the boost in popularity. With this in mind, it would be wise to develop tools and services that cater to a wider demographic of varying technological aptitude, necessitating friendlier user interfaces and general ease of use to capture the widest audience.[15]

[15] Sidhu and Carter, 'The Critical Role of Media Representations, Reduced Stigma and Increased Access in D&D's Resurgence'.

2.4.2. Rise of virtual tabletops during COVID-19 lockdown

A main appeal of tabletop role playing games is the flexibility of the medium in which it is played. Although requiring rulebooks and methods for rolling numbers, such as sets of multi sided dice, the rest of the game is up to the players to customise and how to represent the many facets of the game. Many groups of players would play sitting around a table, the required statistics and representations for their characters being reflected in sheets and figurines on the table respectively. This is what is generally depicted in media and as such is the most popular 'format' in which to play.

Aside from this, it is also possible to play such games over the internet, with players communicating through either voice or text chat, with the numbers being handled physically through dice or managed through services known as virtual tabletops (VTTs), which also offer features for representing maps and storing reference sheets. These tabletop services, such as Roll20, found some success enticing players who wished to play with others across the world through the medium of the internet, though they remained as a smaller proportion of those who played, as most would tend towards in person games with friends and family.

This proportion would shift drastically with the arrival of the COVID-19 outbreak and the subsequent lockdowns and social distancing measures put in place, preventing groups from meeting in person. This would lead to a flood of players to such services to continue games that would have been long running, placing strain on the servers that was designed for a smaller user base and throwing some groups for a loop as they got used to how online games of tabletop role playing games played.

Now that these restrictions are all but gone, many have returned to playing face-to-face in the way they had before but some continued to utilise these services, realising the benefits that improved their way of playing at no extra cost without the need to meet In one place. This time in recent history had drawn attention to the variety of free online services not just for running the game but for acquiring art and maps to set the scene and creating uniform tokens from said art. This project aims to fit into this niche of free online services that can add to the player experience for no additional cost.[16]

2.5. Existing Final Year Projects

Looking through existing final year projects from previous years' graduates, one was found that included 3D rendering.

2.5.1. Project 1: Interdiction

Student: Anthony Hayden

Brief Description:

3D first person shooter game developed in and completely playable on the web browser. Uses HTML5 and JavaScript libraries to manage networking, model rendering, game logic and much more, allowing for an in-depth multiplayer game that only requires a web browser to play.

Takeaways

[16] Scriven, 'From Tabletop to Screen'.

This student used Three JS for the rendering of the 3D assets, environments and characters. This library interfaces with webGL to properly render these models in absence of a local processor, which would be integral to the editor function in this project. Seeing the extent to which that Three can be pushed even on a standard browser with an average internet connection further solidified my choice in selecting it for this project, which, although less demanding, still needs to be efficient to not bloat an average users bandwidth.

2.6. Conclusions

Having researched all these technologies, it is decided that this project will be a web application created in NodeJS, using React and Express in tandem for the front- and back-end, Three.js handling 3D rendering and modelling while Firebase is used for both database and cloud file storage.

3. Experiment Design

3.1 Introduction

This chapter will cover the overall design of the project, encapsulating the software methodology that would be most effective and the overall structure that will be used in development, depicted through diagrams and textual descriptions.

3.2. Software Methodology

In order to find the most relevant methodology, the following models will be compared.

3.2.1. Waterfall

A development methodology that encapsulates the design and subsequent implementation of a project, each of the 5-6 steps feeding into each other akin to a waterfall with multiple dips in its path, with the 'final' step feeding back into the first until the project is satisfactory and is ready to be released. There are usually 6 stages involved in this methodology, though two may be rolled into one in some interpretations:

- **Requirements:** Requirements for a given project are collected and analysed, to be used as a basis for future stages of development.
- **Analysis:** System analysed to generate models for business use, can be rolled into design or dropped when not relevant to project.
- **Design:** Technical requirements of project are decided here, based on functional/ non-functional requirements gathered in first stage.
- **Implementation:** All features are implemented in one fell swoop, finishing when final component is finalised.
- **Testing/Verification:** Product is tested, with issues / errors being documented. This can lead back into the implementation stage or right back to the requirements step if major redesigns are warranted.
- **Acceptance:** After satisfactory rounds of testing, application is deployed to a live environment[17]

[17] 'What Is the Waterfall Model? - Definition and Guide'.

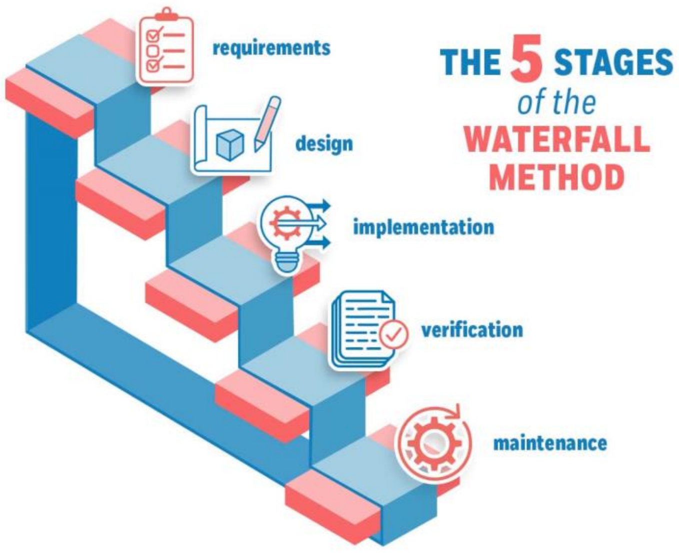


Figure 5: Waterfall Methodology VIsualised[18]

3.2.2. Agile: Feature Driven Development

An Agile-based methodology focused around the client or project stakeholder. As the name implies, a project is split into features, functions that perform an action on an object, which are implemented on top of each other as the project progresses. Divided into 5 stages, with the final two being grouped together:

- **Overall Model Development:** A overall model is developed, determining the technologies and domain used within the project
- **Build Features List:** With this model, features are derived, grouped by the function they possess and how they group to create greater functionality
- **Plan by Feature:** Each feature is planned out before development can start, giving an outline of what to expect
- **Build by Feature/Design by Feature:** This is the development stage, where features are designed, worked on, and displayed to the client on their completion and successful testing. Future features are then factored in the current design and the stage repeats until the entire project is complete. [19]

[18] 'Waterfall Methodology – Ultimate Guide'.
[19] 'What Is FDD in Agile?'

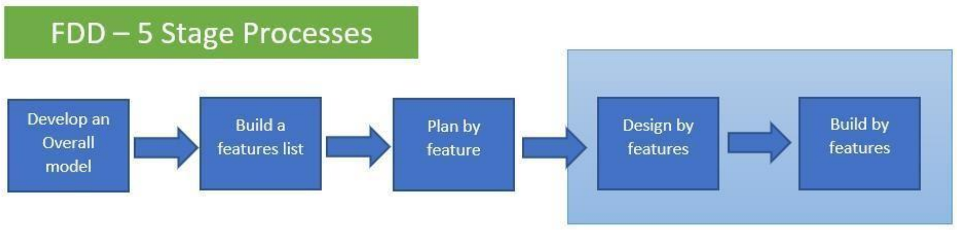


Figure 6: Feature Driven Development Visualised[20]

3.2.3. Kanban

An Agile-based methodology with emphasis on continual delivery which eases the burden on the development team. This is achieved through the use of a 'Kanban Board' (see figure 7), on which features, which are derived from requirements, are placed as 'cards' under at least 4 headings:

- **Backlog:** cards waiting for other cards to be completed to be considered
- **To-Do:** Features ready to enter development, waiting in line for preceding cards to be completed
- **In Progress:** Features currently being worked on
- **Done:** Completed features

Additional columns can be added to suit the needs of the development team, such as features currently in testing or require re-working etc. Cards are moved when circumstances are met, i.e., when a feature leaves development, it moves to testing/ done, while one in the to-do moves to development and perhaps a task is taken from the backlog. This means that Kanban isn't iterated based on time, allowing for progress based on the developers rather than an over seeing force.[21]

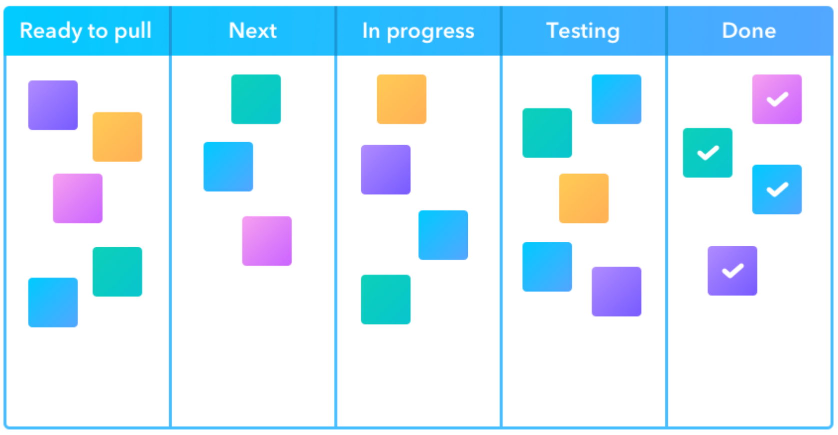


Figure 7: Kanban board example[22]

[20] 'What Is Feature Driven Development (FDD)?'
[21] 'Introduction to Kanban'.
[22] 'Kanban Methodology'.

3.2.4. Decision:

Considering the above methodologies, Feature Driven Development appears to fit the most with the project at hand. As there are two ‘main’ features, the 3D rendering and the model sharing, to be developed, it is clear how to separate them into sub features, creating feature lists for each. This enables design/ build by feature as sub-features are implemented over the time frame available, spawning further sub tasks to inter link the main features when complete. As features get (re)designed throughout the process, they can evolve to suit the current needs of the project and allow for flexibility in design.

3.3. Overview of System

With the methodology chosen, the design of the system and other relevant details can be detailed:

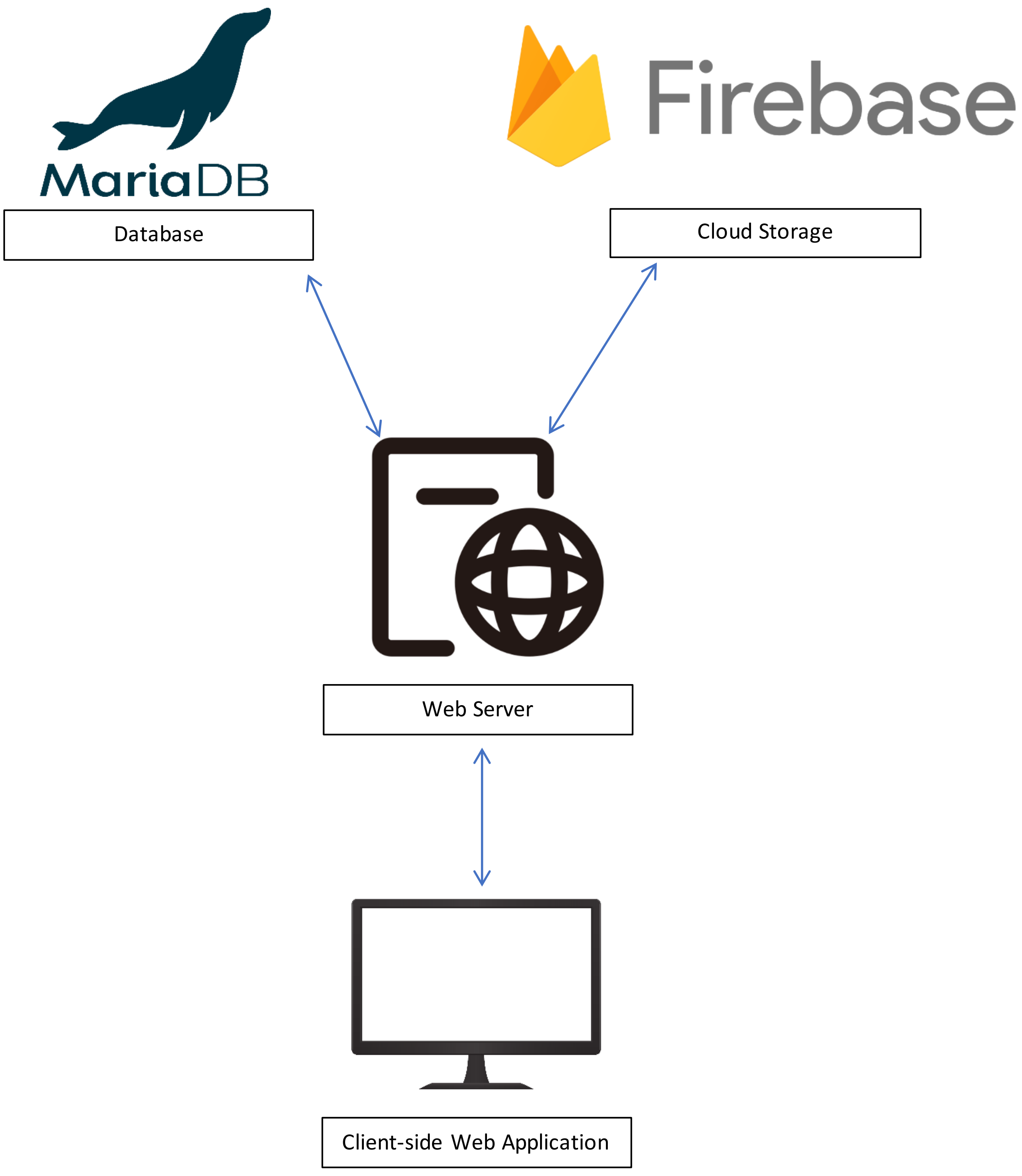


Figure 8: Visualisation of Project Architecture

The system, in its complete form, will encapsulate two main features; a basic, interactive in-browser 3D modeller and a model sharing service, each individually accessible without a login but with functionality being linked when signed into an account. This system will provide an accessible 3D renderer with compatibility with existing 2D images to speed up creation while providing a space to share these creations or find one of another user.

The home page of the application will offer users to start creating models or to browse the available models, with an option to create or log into an account for extended functionality, the front-end built through React while the web server is handled by express, built through a NodeJS app.

When creating a new model, the user will be prompted to upload a 2D image, such as an existing tabletop battle map. They can choose to upload through their computer's file browser or dragging the file directly onto the page using an input/output library. When the image is uploaded, the image size and aspect ratio is determined, being properly applied to a plane object as a texture through the Three JS library, representing the 2D image as though it was painted onto a surface. With this plane object created, the 3D renderer pane will display UI for selecting tools in use for manipulating it. From here, the user can select portions of the image plane, extruding, decreasing or skewing specific parts of the surface to create a simple 3D object with their desired backdrop. Once done editing, the model may be exported in a file type of the user's choice, the default being the widely used GLTF format, then giving them an option to download directly to their device or to share on the model sharing service, the latter option available to a logged-in user or prompting them to create an account.

When sharing the project to the service, the user is taken to the model page creation, prompting for a title, description, tags for use in search functionality, and more miscellaneous, optional parameters. Upon completion of this process, the username and date of creation are added, the details stored in the firebase database under the appropriate table, the created model stored in the Firebase cloud storage and linked to the table in question. This page is added to the model sharing service, open to all users to see and access. Users who have created and posted their creations have the option to edit the page to their liking, along with the option to remove it completely if they so choose.

When accessing the sharing service, users are greeted by a front page, displaying creations that are sorted by date of creation though this can be edited to sort by other metrics. Users can select one of these creations, sort by their preferred tags or use a text search to return creations based on title or tags. When choosing a page to visit, the aforementioned object details are displayed, while offering a 3D renderer canvas to preview the object before accessing. On top of this, users can see how many times this has been downloaded and can rate it out of 5 stars, revealing a score based on the average of previous ratings. Finally, the object can be downloaded, retrieving it from the cloud storage and converting it to the file type of one's choice. As a logged-in user, one may also add a page to their favourites, which can be viewed in their account menu and can get the link to the page which they can share to someone, taking them their directly.

While not required for the functionality of the overall application, implementation of an automatic tag-suggester, taking input from the object created or the initial image uploaded to suggest relevant tags, as to entice users to add descriptors to their creations to diversify search results. This would require image processing and a minor machine learning algorithm, so it is low on the priority of tasks and would be a nice-to-have if time permits.

3.4. Oher Sections

Use Case Diagram

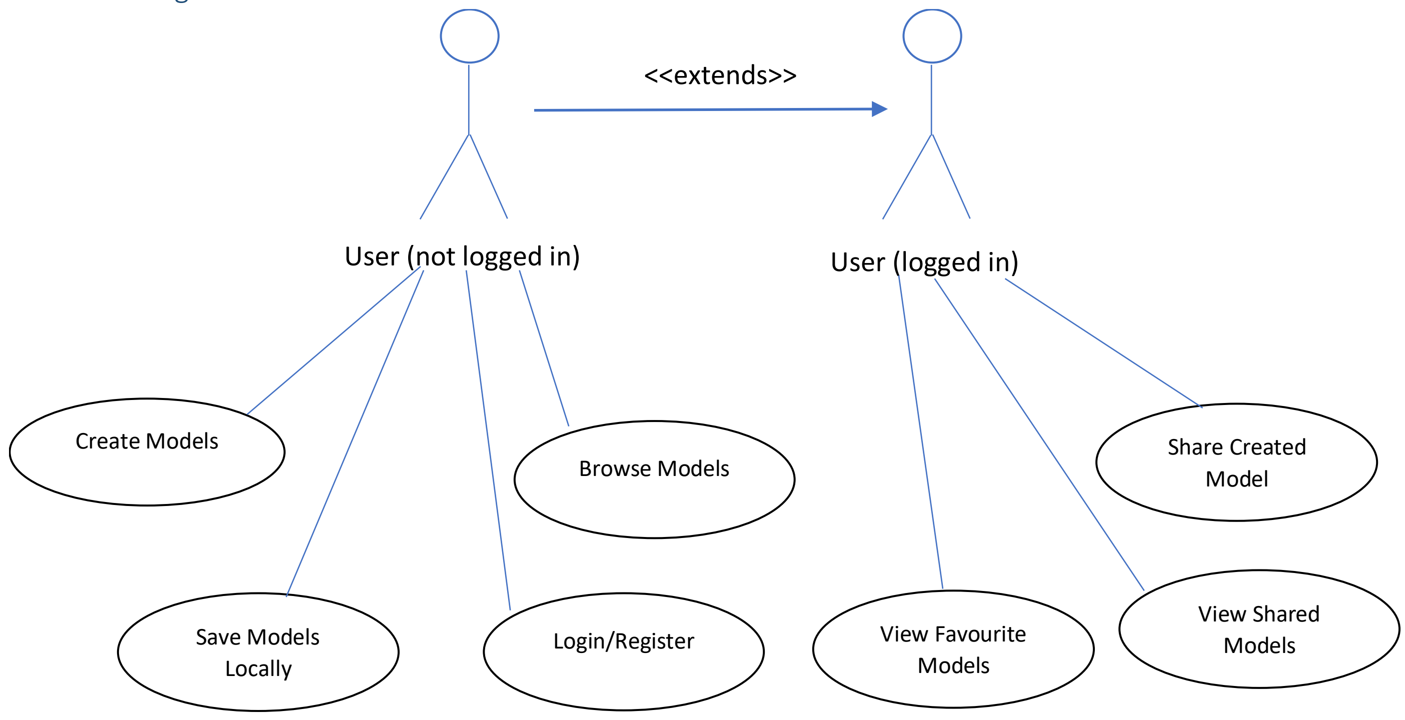


Figure 9: Application use case diagram

3.4. Conclusions

With the methodology chosen and the overview of the system established, work can now commence on considering the testing and evaluation requirements and on a prototype to be shown to the 'stakeholders' of this project

4. Prototype Development

4.1. Introduction

This section will cover the prototype that is developed in tandem with the interim report, as a display of what the system will accomplish when completed.

4.2. Prototype Development

When creating a prototype, it is important to establish what components will be implemented as a demonstration of its functionality and the fidelity of said prototype. A prototype can be considered low, medium, or high fidelity, reflecting the medium through which it is created and subsequently how much time and effort is spent to create a proof of concept. For example, low fidelity may be the user interface quickly sketched on paper, while high-fidelity would involve a form of code implementation or similar prototyping tool to create an interactive and responsive prototype. Meanwhile, the 3D rendering and model sharing service are the core of the project and as such should be displayed in this prototype.

Considering the complexity of the 3D component and the importance placed on the interconnectivity between it and the model share functionality, it is decided that the prototype will be in high-fidelity, implemented as barebones web application, as it can act as a springboard on which further development can commence. It will be locally hosted, as it will not require back-end communication for the moment.

4.2.1. Front-End

The front-end will initially be handled by html, CSS and JavaScript, providing a basic interface through which the outline for the application can be delivered. By keeping the prototype in a basic state, the additional pages required to display the functionality can be easily added and be retrofitted with ReactJS and Nodejs for when project development begins.

These will cover the home, model creator and model sharing pages, the home giving an option between the latter two. These pages will not contain the full responsive functionality of editing models or dynamically adding models to the sharing service, though it is vital for the layout to be created for the proof of concept. (See Appendix A)

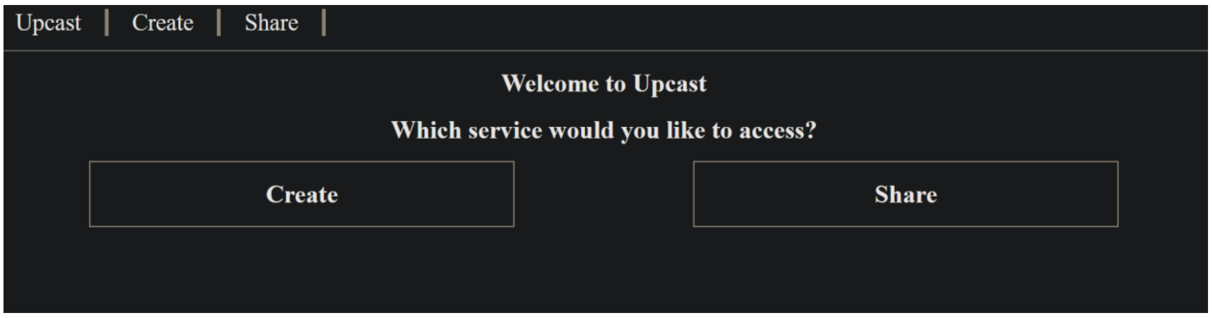

Upcast | Create | Share |
Welcome to Upcast
Which service would you like to access?
Create
Share

Figure 10: Home Page for the website

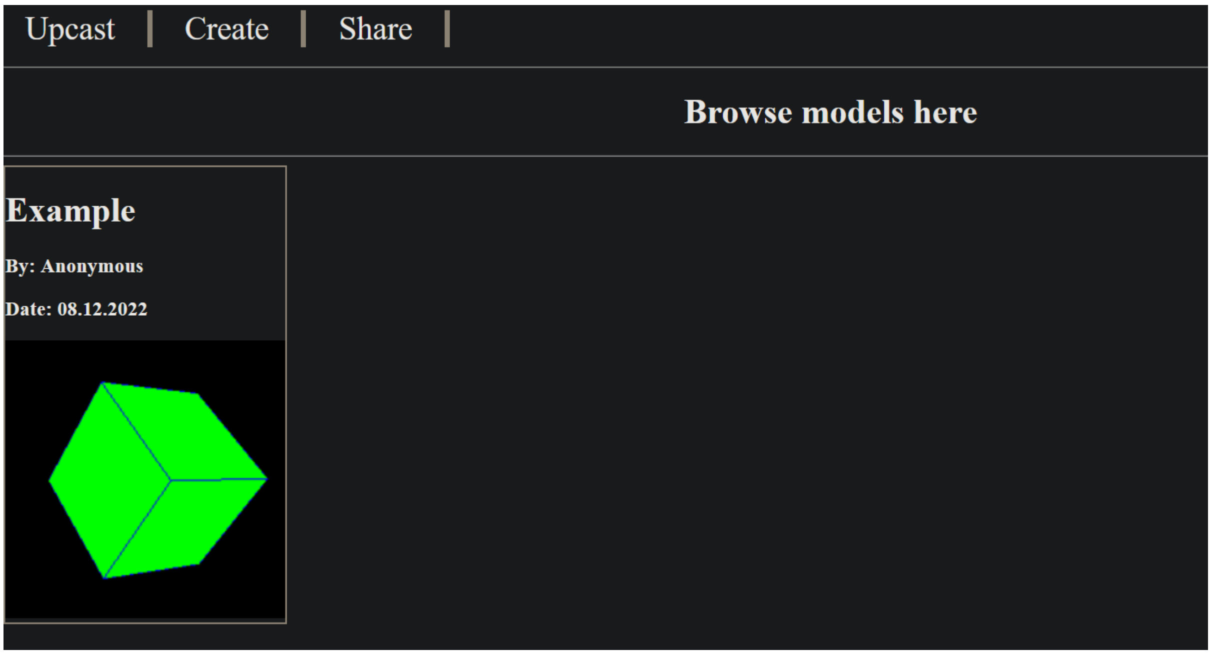


Figure 11: Share page with example object

4.2.2. 3D Renders

On both the model creator and sharing pages, an example of a 3D model, locally stored and referenced in the local host project, will be rendered to view. This is implemented through Three.js, offering a canvas object through html in which the library code can be executed. This will represent the 3D capabilities of the application and, in the case of the creator page, can be edited to implement the editing tools further in development (See Appendix B).

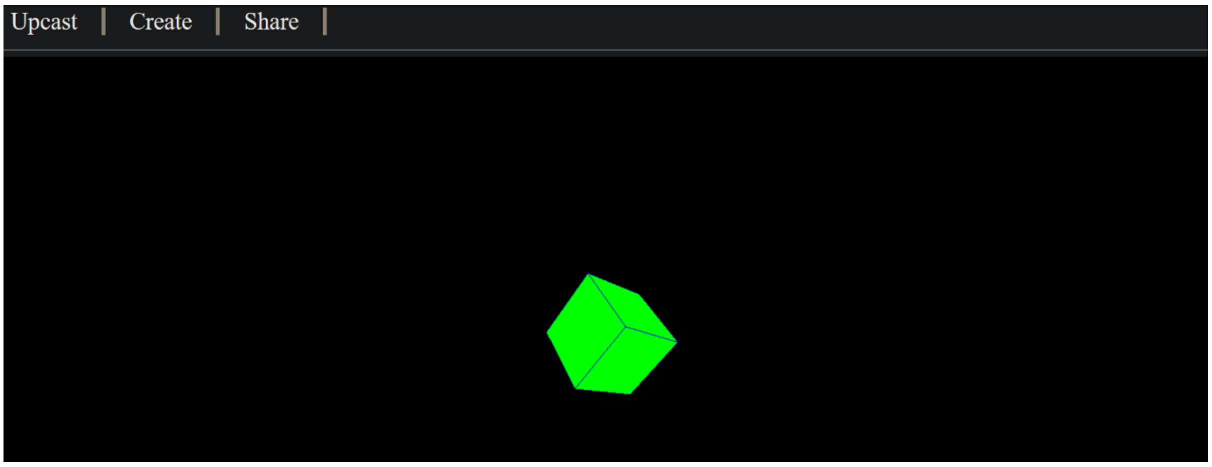


Figure 12: 3D model being rendered in creator page

4.2.3. Login

A basic login will be implemented, available to access on all pages. This will be used to determine if the user can access certain features, in this prototype it will enable a link from the model creator to the sharing service.

4.4. Conclusions

With the development of the prototype, a solid structure for further development is in place, despite not featuring the reactive interface or editing features promised in the final release. This will serve as a guideline onto which such features can be added.

5. Experiment Development

5.1. Introduction

This chapter covers the proper implementation of the application, now that the prototype and interim report has been submitted. This will build on the base set by the prototype, adding the necessary features and technologies outlined in the experiment design.

5.2. Software Development

As detailed previously, this application will feature ReactJS and ExpressJS powering the front-end and Back-end respectively. As the prototype had only featured standard HTML, CSS and JavaScript as a basic front-end, it would be important to implement the front-end to start, on top of which the rest of the functionality can be built.

5.2.1. Front-End

The front-end will cover functionality that is processed on the client machine when the app is being used, displaying the user interface and allowing for reactivity based on the actions of the user. As this would be what the consumer would first notice, it is important to start developing this before touching on back-end development.

5.2.1.1. React set up

Development begins with creating a react app, which is done through NodeJS's node package manager (NPM) command line tool. This is installed through the NodeJS installer, which then allows for creation of a react app in a given directory with the *npm create-react-app* command. It installs the necessary libraries and creates folders in which the files of the app are stored. When completed, the app can be started with *npm start* while in the given directory, automatically opening in the default browser on port 3000.

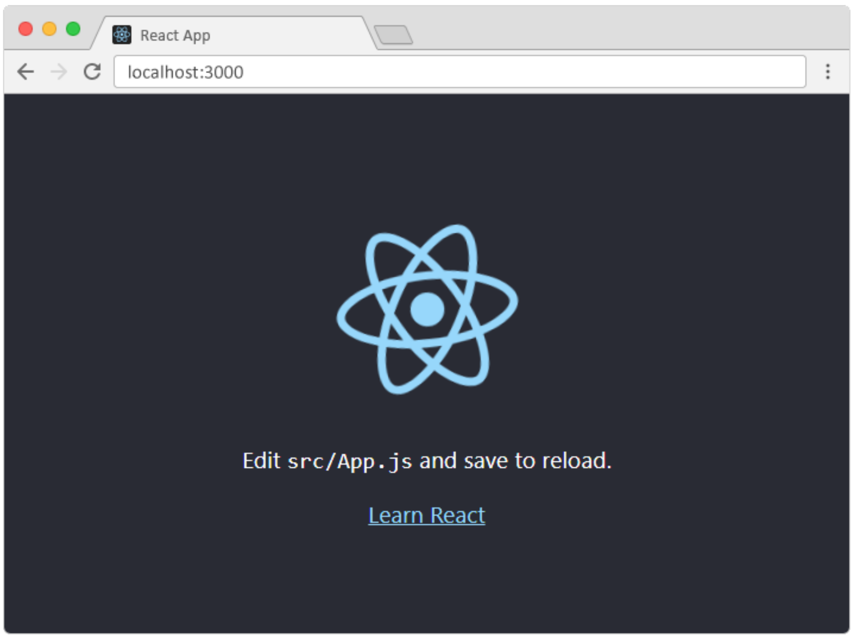


Figure 13: Default React app splash screen

With this, we have a single page application which can be edited with the app.js and index.js files, the latter defining what is rendered to the screen, the former being called by the index to display what is seen in figure 12, adding html, CSS and JavaScript through JSX components with React.

This works fine for one page, but the prototype had multiple pages, with planned functionality for communication between them. In order to facilitate these pages, an additional library *react-router-dom* is imported, installed as all other modules/libraries are with *npm install*. What this library does is tell the index file where the JavaScript files with the other pages are relative to its directory, loading those pages when the URL loads the path of a given app. With this, files for the rest of the pages are made, displayed with the following code.

```
import { createRoot } from 'react-dom/client';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import './index.css';

import Home from './pages/Home.js';
import Create from './pages/Create.js';
import Browse from './pages/Browse.js';
import Bar from './pages/Bar.js';

export default function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route path='/' element ={<Bar />}>
                    <Route index element ={<Home />}></Route>
                    <Route path ="Browse" element ={<Browse />}></Route>
                    <Route path ="Create" element ={<Create />}></Route>
                </Route>
            </Routes>
        </BrowserRouter>
    );
}
```
Figure 14: index.js routing example

The nav bar always being visible as it is rendered as the root element, while home page is loaded by default. The other pages can be accessed by entering them into the URL, but this is awkward to do every time, so links are created on the navbar and home page directing to their respective pages.

```
<div class = "home-div">
      <Link to="/" class = "linkbar">Home</Link>
</div>
<div class = "home-div">
     <Link to="/Create" class = "linkbar">Create</Link>
</div>
<div class = "home-div">
      <Link to="/Browse" class = "linkbar">Browse</Link>
</div>
```
Figure 15: Route linking example

With the router fully set up and working, work proper can begin on implementing the rest of the front-end.

5.2.1.2. ThreeJS implementation

With a solid basis for creating pages in the React app, it's time to implement the 3D rendering libraries from ThreeJS for the create page. This came with initial difficulty, as the standard JavaScript libraries seemed incompatible with the JSX component-based implementation of React, which would have created an awkward method of converting those files to components for use in the apps. Thankfully, there existed an open-source node library that did exactly that, providing most, if not all, of the functionality offered by ThreeJS in React compatible components, through *react-three-fiber.*

Prior to implementing this in the current application, an example app was made (see Appendix C), showcasing features for rendering two cubes on the screen, offering interactivity for changing their size and colour upon mouse interaction, while rotating them every frame. These cubes would be children of the Canvas element, on which all 3D rendering will be done in the project, same as in the prototype, while utilising built in functions for detecting mouse clicks etc. The functionality displayed in this app will help implement future functionality in the project itself.

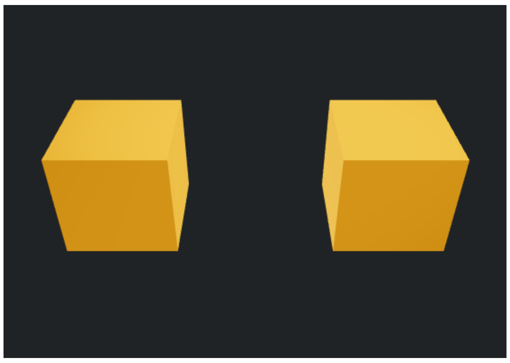


Figure 16: Output of Appendix C

With a better understanding of the library, a canvas element is placed into the create page of the project, using modified code from Appendix C to render a cube. This canvas element is placed inside a div, which is manipulated by CSS to place it near the top centre of the screen. An additional module, *OrbitControls*, is imported from *react-three-drei*, implementing some example code from standard ThreeJS into React, namely controls for moving the camera around a targeted object. This will be used for inspecting the object when model editing is added.

Now that a cube is properly rendering in a separate canvas element, further steps can be taken to modify the cube with user input.

5.2.1.3. Upload of images to model

As per the experiment design, the model being rendered will have an image / texture applied to it, to create the test bed for model manipulation. To accommodate such additions, the scale of the cube is modified to 16 width, 9 height and .1 depth, to simulate a standard 16:9 ratio seen in most displays/ wide images, such as the example image used for testing in figure 16. This 'texture' is added by passing the image to a React *useLoader*, which prepares it for use in JSX with a *TextureLoader*, which is finally mapped to the model itself in the material component in the function which generates the model.

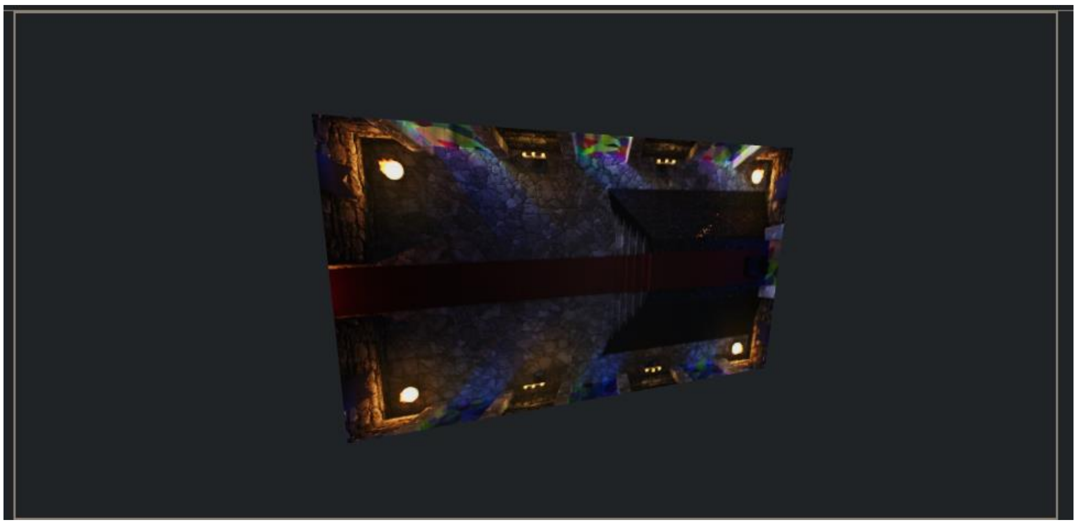


Figure 17: Example image placed on model in canvas element

A texture successfully loads onto the model. This works well, but the image is currently hardcoded and must be changed by altering the page file. It would be prudent to allow for a user to upload their own image to then be placed onto the model.

To do this, an input component can be added to the page, with its type being defined as 'file'. This provides a button for users to browse their local files and to upload one of their choice. This can be limited to certain supported file types, in this case JPG and PNG to prevent errors with text or other unrelated files being chosen. Furthermore, a function can be called whenever something is uploaded, which will be used to convert the files for use in use and texture loaders.

As these images are uploaded, they need to be stored in the file to be accessed anywhere in the page. To accomplish this, a *useState* variable *selImg* is used to store the URL, by passing it into its constructor setSelImg. This variable is declared in the main function, allowing it to then be passed into the constructor of the cube component, which then allows for the image to subsequently be rendered onto the model.

```
<input
      type={'file'}
      onChange= {imgHandle}
      accept={".jpg, .png"}
/>

const imgHandle = (e) => {
      const b = URL.createObjectURL(e.target.files[0])
      setSelImg(b)
      text=b
}
```

Figure 18: Image upload code

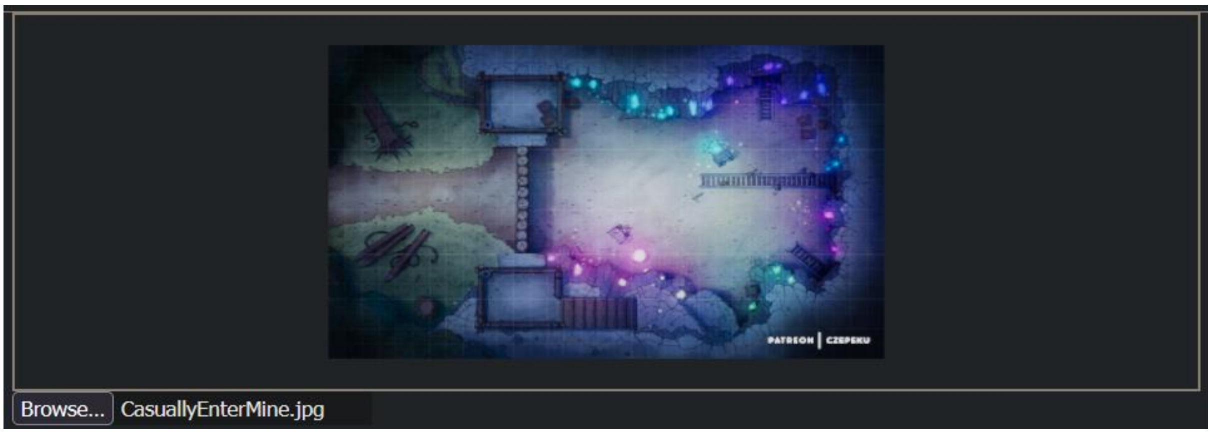


Figure 19: Image successfully uploaded

The model can now be customised with any image that the user uploads. With this level of modification, it would be safe to try to export the model for downloading and sharing purposes.

5.2.1.4. Model Export/ Import

A major feature of the project is the saving and sharing of models after they have been edited. This will involve getting a reference to the model in the canvas, parsing the data with the help of an exporter function, before passing it into a Binary Large Object (blob). With the data in the blob, it can be downloaded for storage or used elsewhere.

To begin, a reference to the model will be made by declaring a *useRef* variable, before assigning it to the reference attribute of the model. This allows access to the model outside of the canvas, so this can be directly passed to the function handling the export. The function in question will differ depending on the filetype chosen for storing these models. The most common extensions for 3D models are obj, fbx and gltf (Graphics language Transmission Format). Out of these, gltf had been chosen, as neither obj or fbx stored textures inherently, so its binary format (glb) was chosen to neatly store the model with added textures.

With the filetype chosen, the *GLTFloader* can be called, taking the model reference with the parse method. Some difficulties were encountered here, as there was little documentation online regarding the exporting of models/ scenes in react-three, as more resources covered the importing of models instead. This, coupled with relative inexperience with JSX formatting, lead to a lot of time being spent on trying to get this to work, eventually learning that the output of the parse must be put into a temporary variable, such as gltf, which then feeds into a function.

In this function, the result is fed into a blob, appended with *application/octet-stream* to define a default binary file. With file in hand, the page creates an invisible link which downloads the model.

```
Const exporter = new GLTFExporter()
exporter.parse(
       canvasRef.current,
       (gltf) => {

       const glbBlob = new Blob([gltf],
       { type: ‘application/octet-stream’})

       const link = document.createElement(‘a’)
       link.href = URL.createObjectURL(glbBlob)
       console.log(link.href)
       link.download = `model.glb`

       document.body.appendChild(link)
       link.click()
       document.body.removeChild(link)
       }
       )
```

Figure 20: File Download Function

With this complete, files can now be downloaded directly to a given device, able to be loaded into model viewing software or imported into other projects, such as game engines etc.

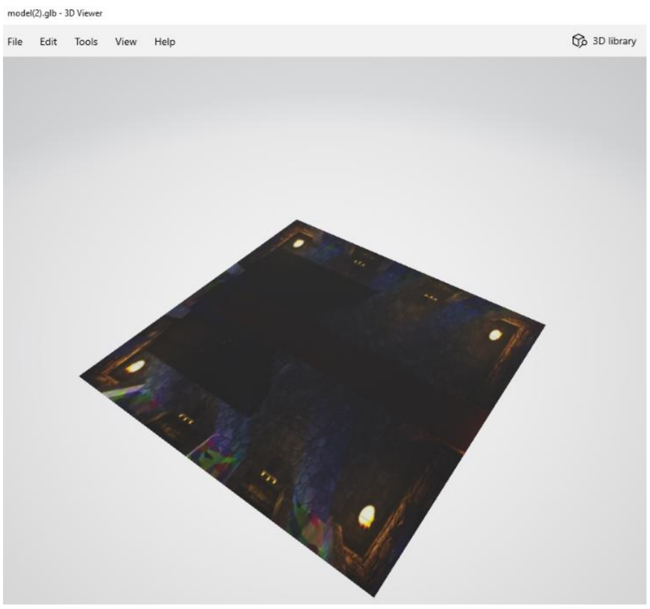
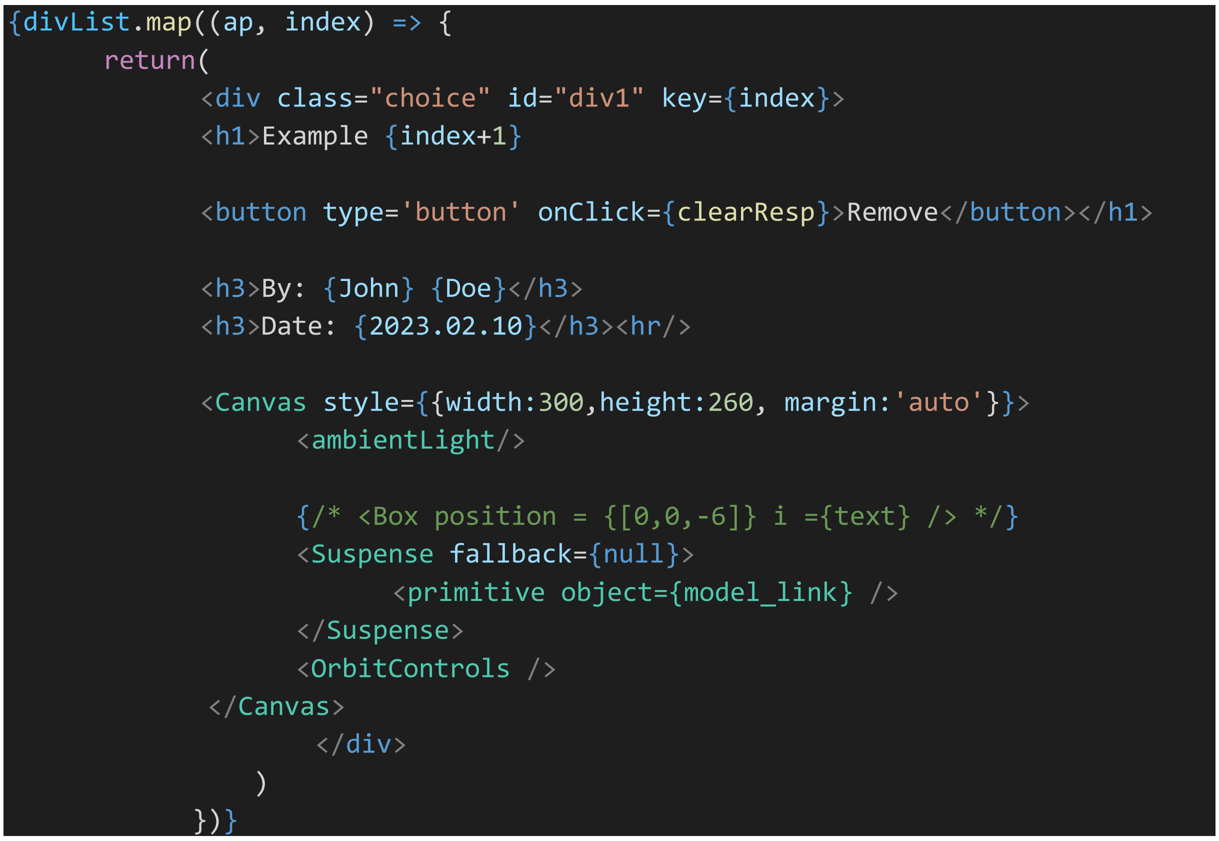


Figure 21: Textured model loaded into external application

5.2.1.5. Dynamic Div Mapping

With the groundwork set for the model creation side of the application, the browse functionality can be worked on. It hasn't been modified much from the prototype (see figure 11), so it still has hard-coded divs depicting what it will display when backend is implemented. As the data it will be displaying is variable in amount, it would be prudent to allow for a dynamic method of adding and/or removing these as data is received.

To facilitate this, a *useState* can be used to hold an array of arbitrary data. No matter what is in the array, the built-In map function of the array can be used to execute code for every element in it. This can be used to generate as many elements as there is data provided by returning JSX elements within this function.



```
{divList.map((ap, index) => {
      return(
            <div class="choice" id="div1" key={index}>
            <h1>Example {index+1}

            <button type='button' onClick={clearResp}>Remove</button></h1>

            <h3>By: {John} {Doe}</h3>
            <h3>Date: {2023.02.10}</h3><hr/>

            <Canvas style={{width:300,height:260, margin:'auto'}}>
                  <ambientLight/>

                  {/* <Box position = {[0,0,-6]} i ={text} /> */}
                  <Suspense fallback={null}>
                        <primitive object={model_link} />
                  </Suspense>
                  <OrbitControls />
             </Canvas>
                    </div>
                 )
            })}
```

Figure 22 Code for propagating divs

This still holds the same data as the static divs had, including a 3d preview of an object by adding a canvas to each. The divs in question are formatted using CSS flexbox, allowing for a certain number of evenly spaced elements before it breaks into a new column, though a grid box may serve better if the elements needed to be spread evenly or keep a tighter formation when the amount of data is too high or too low.

This lays the framework for inserting data retrieved from a database into accessible, readable formats.

5.2.1.6. Model Vertex Editing

The main draw of the project is the ability to edit the geometry of a model through clicking points on it, shaping it to fit the intended vision based on the attached texture. This will allow for the distinction to be made for elevation in an otherwise flat image, which can be helpful if used in a game with strict rules on distance and height requirements.

To begin implementing this, points on the model must be extrapolated from the model being rendered in the canvas. This can be achieved through the previously discovered built-in ray caster in the canvas element. This can be accessed by adding an *onClick* (*onDoubleClick* to not interfere with the orbit controls) method to the model, sending the output *e* to a function *dClickFunction* . In this function, the output *e* returns all intersections made by the ray cast, the first one representing where the mouse is pointing. With this data, the point on this model and the face it makes up can be returned.

Despite having this data, it only references the point, with no way to directly edit the model from this. Before remedying this road bump, a simple proof of concept for model editing can be implemented by rendering smaller scale boxes at these positions, allowing for a temporary example of the final feature to be shown (See appendix D):

An element is added to a *useState* array on double click, mapping a box onto the canvas with its position set with the output. This box is rendered inside the model, as the box 'origin' is set at its centre. This is remedied by adding half the depth of the model to the z component of the point, moving it to be just touching the surface. As these boxes are stored similarly to the dynamic divs as set up in the previous section, they can be added and removed simply through a function tied to a given input or button, either all at once or one at a time, allowing for precise handling of where boxes are placed. For these functions, buttons have been added below the canvas window.

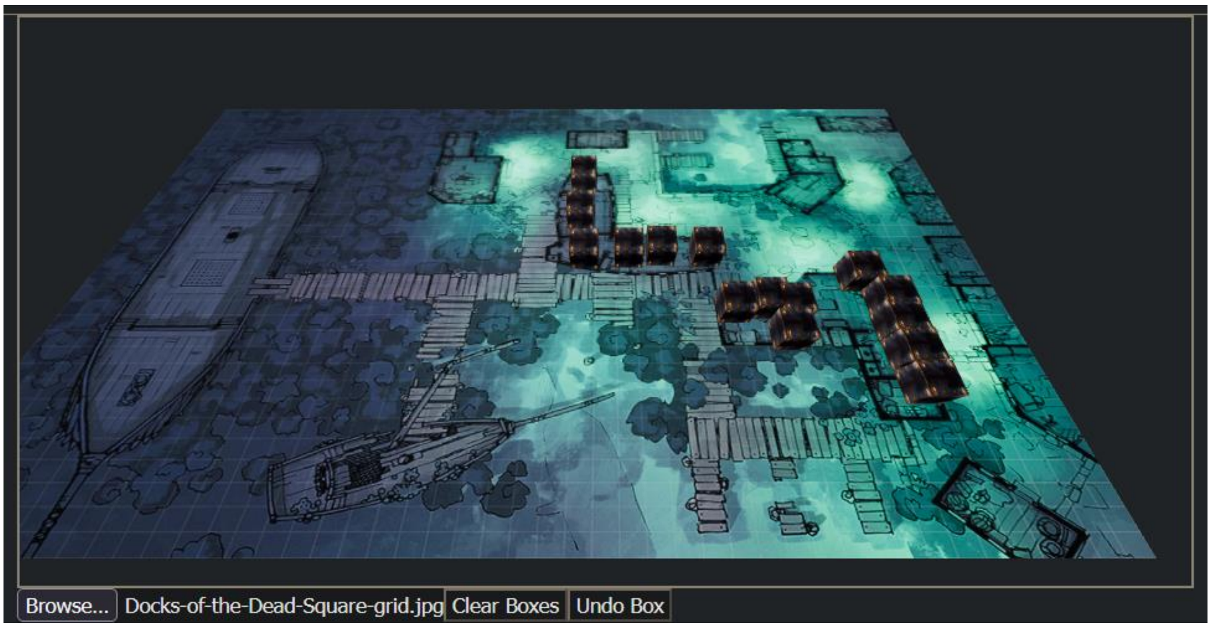


Figure 23: Working example of box placing function

By mapping the boxes inside a group element with the model itself, they can be saved alongside the model in the glb file, offering an alternative to the previously stated goal of editing the geometries if that turns out to be unfeasible.

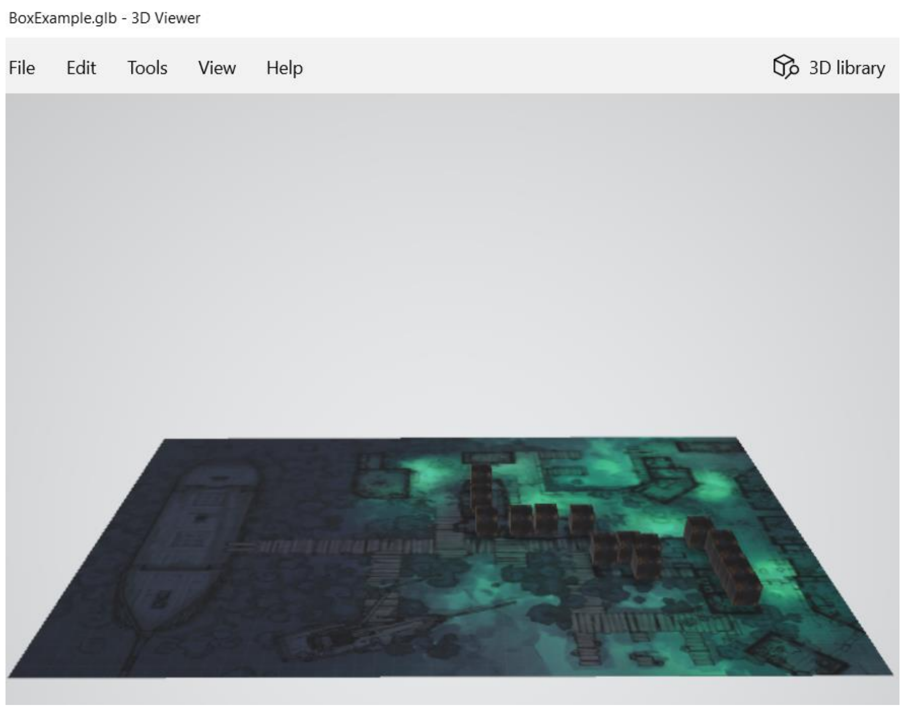
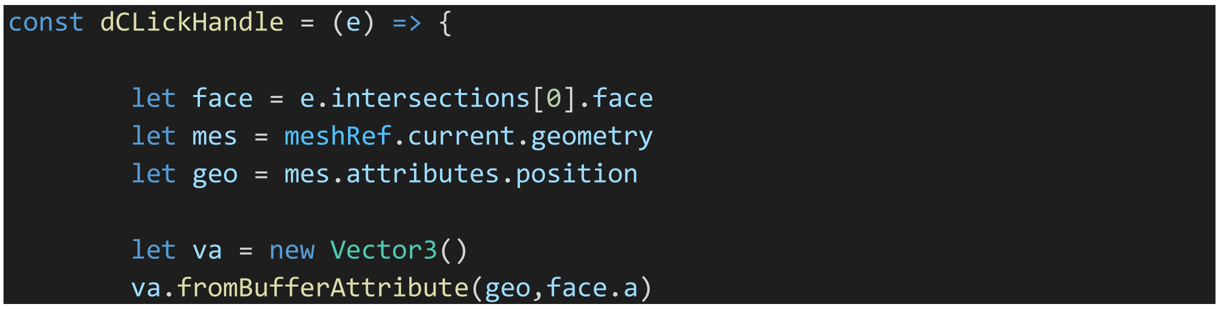


Figure 24: Saved model retaining box placements

With a temporary proof of concept in place, the implementation can now continue by getting a reference to the model. This reference will be different from the one used to export and save the model to a file, as that takes all models rendered in the canvas before parsing it, whereas a direct reference would be preferred for accessing individual attributes such as the position of the vertices. This appeared to be a simple task that could have been resolved with the same *useRef* used for the canvas, but a problem arose when trying this:

When trying to store the reference in such a variable, it would return *undefined*, as though it could not access the original function that returns the model. This took some time to test and research, as most examples found in documentation and other online sources used *useRef* to directly access these models. After sometime researching the matter, the cause of the issue arose; the function that outputs the model was outside the scope of the function that rendered the page, which was the case as it was being called in the browse section to render for every div that gets added. Not wanting to refactor the function at this time for this purpose, a solution came in the form of *forwardRef*, which separated the reference attribute from the other attributes assigned in the canvas, i.e. scale, position etc. This allows the reference to be properly assigned from within the function, proven by printing the value of *meshRef.current.geometry.attributes.positions* to the console, returning an array of all points that make up the model.

Despite having access to all positions in the model and a point on the model from the ray cast, this won't give the necessary information to alter a specific vertex relative to the point returned, as the position returned has no reference for the vertices. Looking through the attributes in the geometry object of the model, an attribute called faces can be returned from both the ray cast and the geometry, holding three points representing the three vertices that make up one of the many triangles forming the shape of the overall model. By referencing face that is intersected by the ray cast, a *vector3* variable can retrieve individual positions that make up a face using its built in *fromBufferAttribute* method, placing the three float values of a faces a, b or c point in it.



```
const dCLickHandle = (e) => {

        let face = e.intersections[0].face
        let mes = meshRef.current.geometry
        let geo = mes.attributes.position

        let va = new Vector3()
        va.fromBufferAttribute(geo,face.a)
```

Figure 25: Retrieving position data from face

With these values isolated, a method of altering these and passing them back to the model to update them in the renderer must be made. This was difficult at first, as there was little documentation centred around manipulation of geometry, and the ones that were found were based on the standard javascript libraries instead of the react-three being used. After some trial and error, a method inside the geometry object was found, setXYZ. Requiring the input of an array that takes three points, it worked by putting the face reference in, followed by vectors containing the values taken from all the points taken from the face. For the purposes of testing, the z value of the face.a vector was increased, as if it were approaching the camera in the scene, but when the code was run and a point double clicked, nothing happened. An attribute was found in the geometry object *needsUpdate* after looking through documentation, as this had shown up there in previous searches for getting the getting the positions earlier. Setting this to true tells the object to re-render all points, moving the material/texture as necessary, which should now display the changes made.

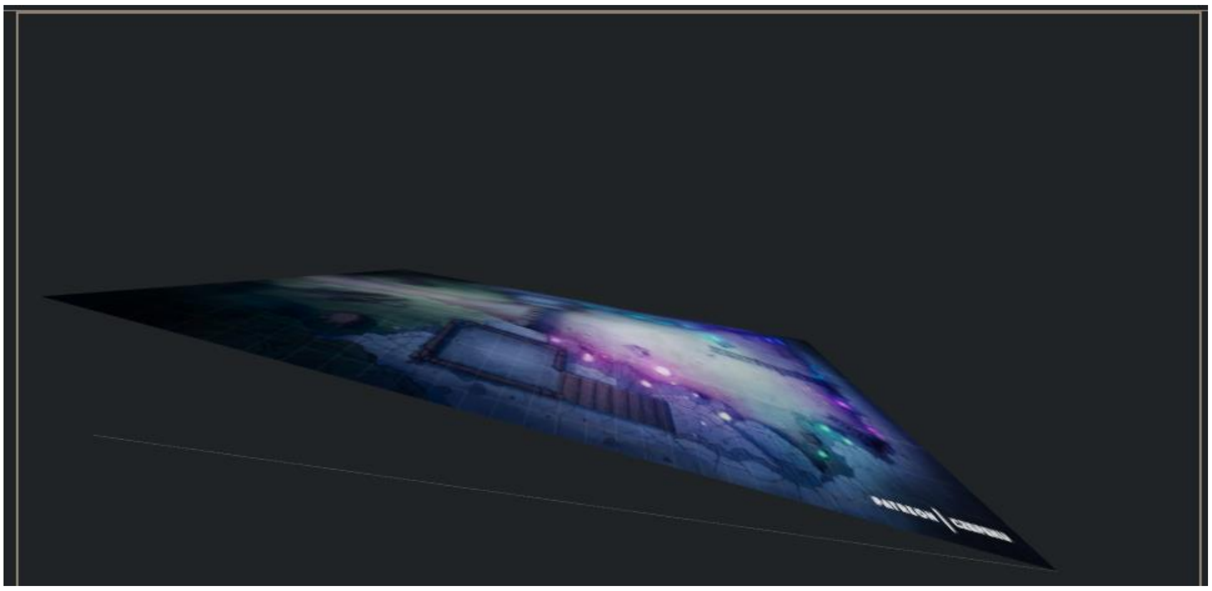


Figure 26: Point a on a face lifted on z axis

The changes are successfully updating, but too much of the model is being lifted give enough granular control over the editing of the model. This is due to the default number of triangles making up each side of this model being two. This, combined with the test function only increasing the z value on point a of any given face mean that only two points can be raised, one of which shown in figure 26. To remedy this, the parameters *widthSegments* and *heightSegments* can edited in the model function to increase the number of available faces to edit. For now, they will be set to 16 and 9 respectively, to match the aspect ratio set for the scale of the model (see Appendix E for the difference these values make). With a greater number of faces, let's see what happens when a point is clicked.

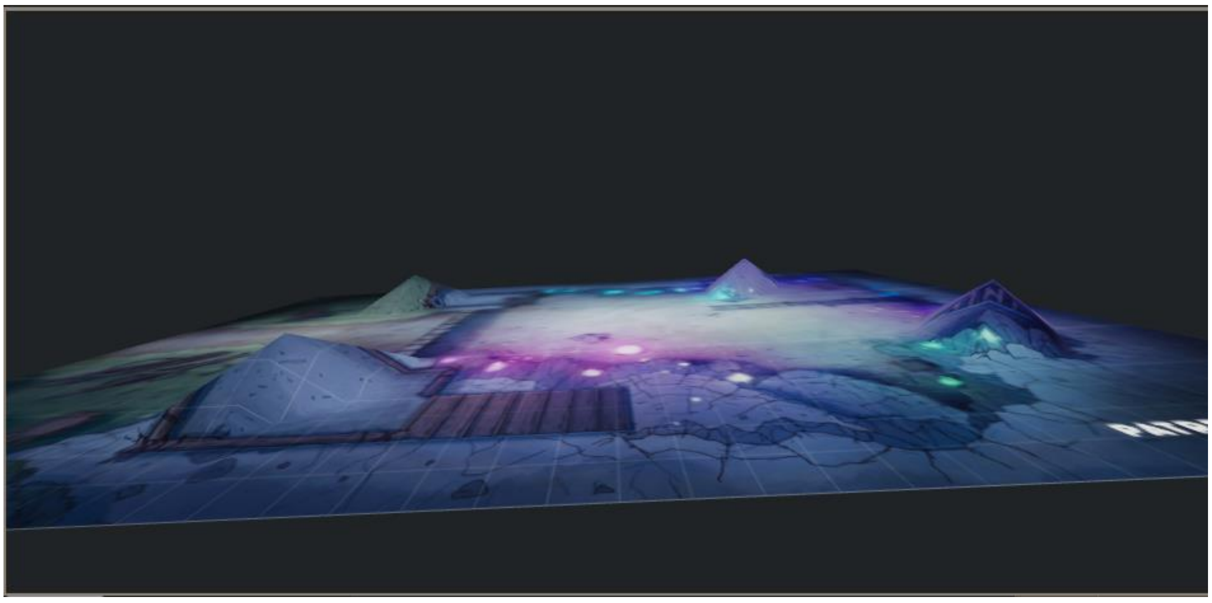


Figure 27: Point extrusion with more width/height segments

With this, precise extrapolation of points is achieved. These points have been offset by constant value of 75 to achieve the height seen in figure 27, though it would be useful to set a limit or ceiling to prevent extruding points too far. A simple if statement checking if the offset value goes over 75 and capping the value to it allows for this consistency, preventing alterations to already extruded points. Furthermore, an option for multiple extrude heights would work for a user who needs to distinguish between different elevations. By setting the extrude height and ceiling to *5 * (scale_value * 5)*, a value can be subbed in to simulate different heights, with 3 returning 75 as in figure 27, while doubling the inputted value gives a linearly scaling height. This would be achieved through a html select input component, assigning the scale value in multiples of 3 to give a consistent increase in height, while also allowing points to be lowered if clicked with a lower value selected, as the ceiling changes with each value. A '0' value was also added to return the points back to their original position if a mistake is made.

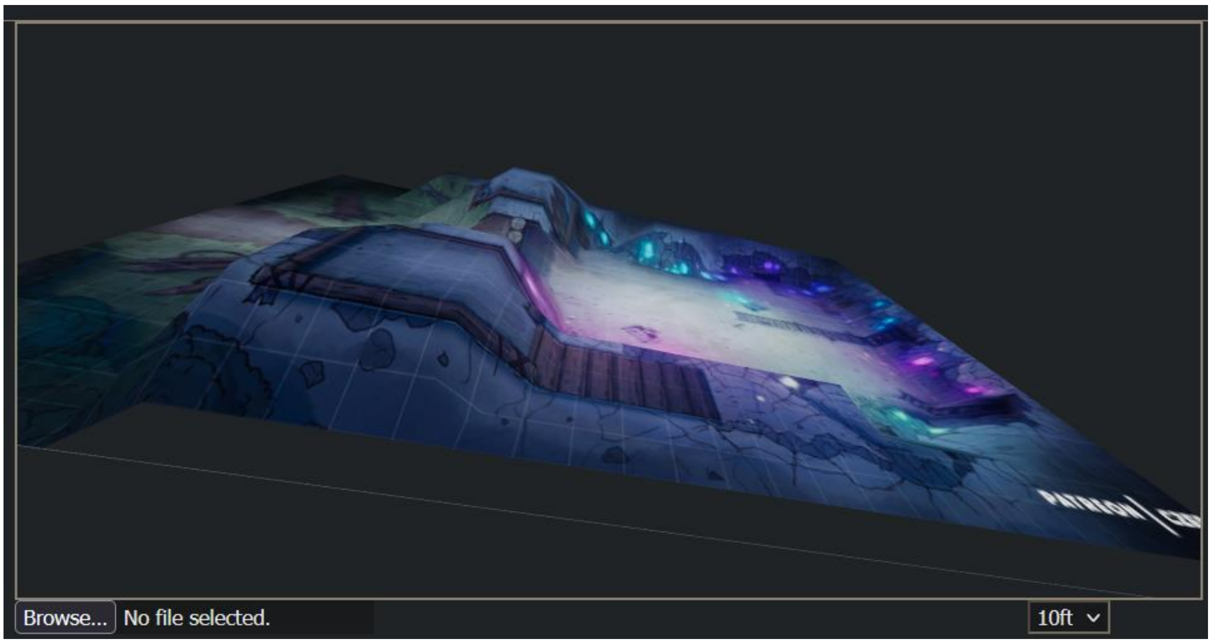


Figure 28: Example of different heights used in one model

Most of the functionality of this feature has been implemented, with modified models successfully saving to glb and options for undoing changes made in editor. The last option to give to users for now would be to change the aspect of the model at will, as the original 16:9 would stretch out for images/textures that were more square or taller than wide. Another selection input can be used to dynamically alter the scale values passed into the model function, allowing for a square or 9:16 aspect to accommodate other such images.

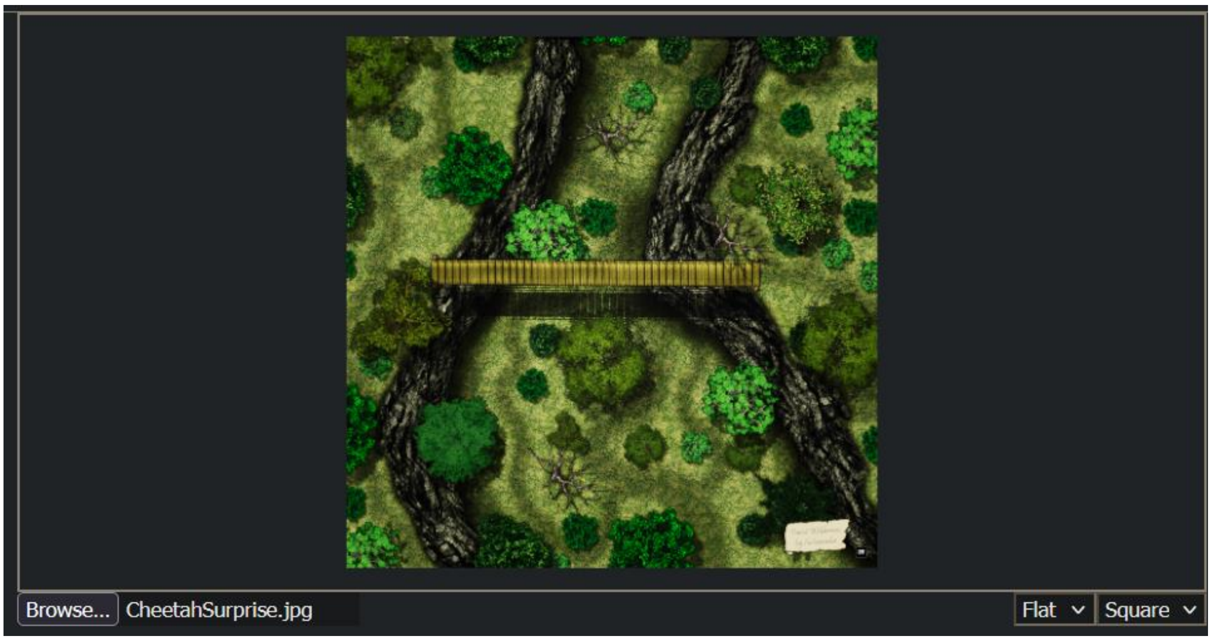


Figure 29: Example of altered aspect ratio to fit image better

5.2.2. Back-End

With the front-end complete, the back-end can now be set up. Using ExpressJS, functionality for accessing cloud storage and databases as well as user registration/logins can implemented, being accessed by the front-end through get, set and post requests.

5.2.2.1. Express setup

To begin, ExpressJS must be installed. This is done in a similar manner to initialising the react app, although the *express-generator* module must first be installed before running it through *npx express-generator* in a given directory. This will generate the necessary modules, files and sub-directories required for running the app, alongside an example page. The app can be started immediately using *npm start*, just as the react app is. It will open in port 3000 on localhost by default, displaying what is shown in figure 30.

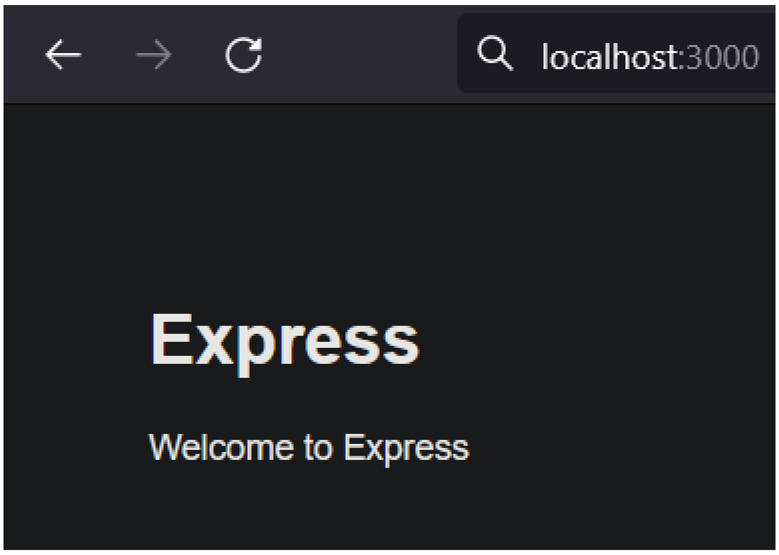


Figure 30: Example express splash page

The server is successfully running, but the port will need to be changed if it is to run in tandem with react. This is changed in the *bin/www.js* file, which defines methods and attributes that determine how it is listening for potential requests. In this file is an attribute *port*, set to a function *normalizePort* which allows overriding of the port or to default to 3000. To ensure it never conflicts with the existing server, the default will be altered to 9000. Both applications can now be run side-by-side, which will be necessary for the back-end to receive requests from and send responses to the front-end.

With the above resolved, it is important to understand how individual APIs are made and referenced. Looking at the *app.js* file, it uses a similar routing system seen in the react app, getting references to existing ‘routes’ for each API, each of which export their functions at the end of those files, then attaching those to the app by calling *app.use()*, allowing them to be accessed through the search bar. Following the same principle, a *testAPI* route is created, which will be used to test both get and post requests in the front-end.

With a testAPI set up, an api call should be made from the front-end to initiate testing. The home page will be chosen, as it isn’t cluttered and can cleanly show off information gleamed from these requests. Using the code defined in the *callAPI* function, a get request is made to the server listening on port 9000, with the react hook *useEffect* ensuring it is performed asynchronously to prevent timing issues, with an empty dependency array to prevent it updating more than once per refresh. This get request should store a string into a variable, which can then be displayed on the screen.

```
useEffect(() => {
        callAPI()
}, [] )

const callAPI = () => {
      fetch("http://localhost:9000/testAPI")
            .then(res => res.text())
            .then(res => setApiResponse(res));
}
```

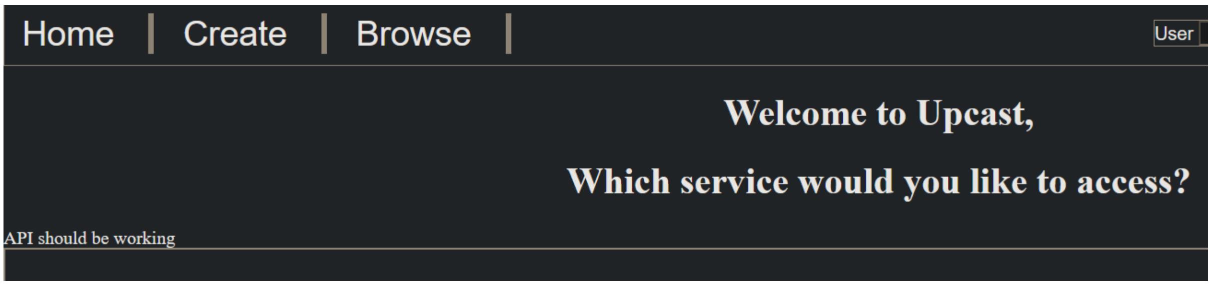
Figure 31: testAPI call code

However, when running both applications, nothing seems to update. Checking the console, a Cross Origin Request Sharing (*CORS*) error is returned, citing that the react app on 3000 is not authorised so data cannot be sent to or received from here. To authorise this api for such calls, return to the express application and install the *cors* module with *npm install*. In the *app.js*, add cors to the application as shown in figure 32, allowing any calls from the react app to reach the server.

```
app.use(cors({credentials: true, origin: "http://localhost:3000"}));
```

Figure 32: CORS configuration

With this set up, return to the home page of react after restarting the express server, and the string "API should be working" should now be on screen, indicating a successful API call.



Welcome to Upcast,

Which service would you like to access?

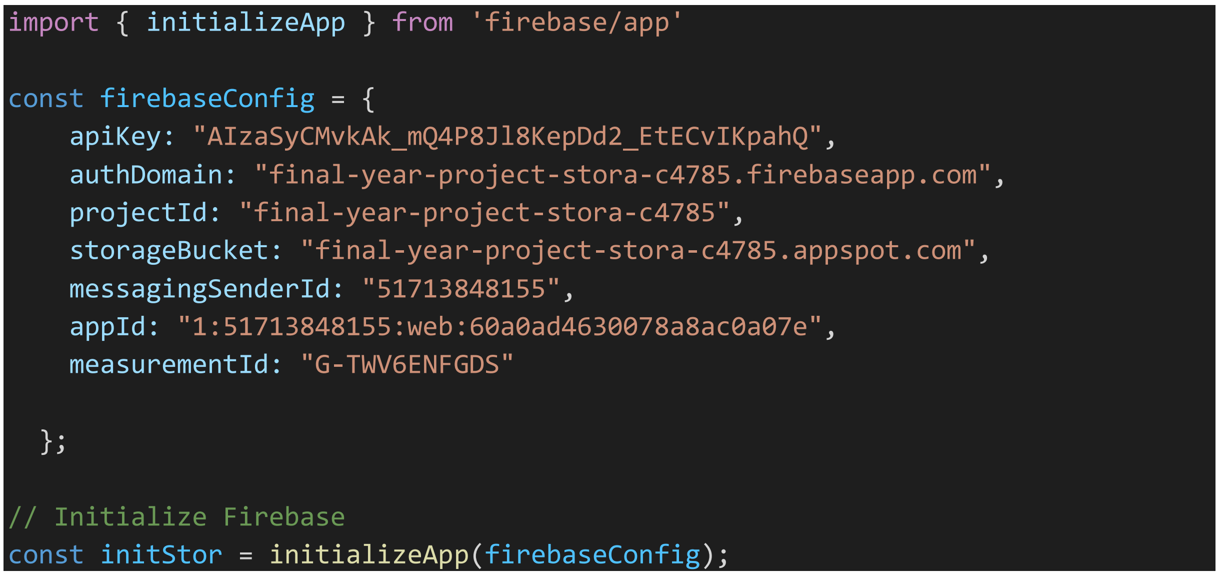
API should be working

Figure 33: Successful API call

5.2.2.2. Firebase

With methods set up to call APIs, it is important to set up the external services that will be referenced by upcoming features. As the storing and referencing of completed models are integral to the browsing feature, the cloud storage will be set up first, using Google's Firebase storage.

To begin, a google account is used to create a storage 'bucket', in which files will be stored. This can be accessed through the code by installing the *firebaseConfig* module and entering values for attributes such as the domain of the bucket, account ID etc, as seen in figure 34. To test for data retrieval, an example image was uploaded directly to the website while a reference is made to the model and bucket. Printing the output of the reference to the console will return JSON with metadata relating to the data, confirming the connection to the bucket.



```
import { initializeApp } from 'firebase/app'

const firebaseConfig = {
    apiKey: "AIzaSyCMvkAk_mQ4P8Jl8KepDd2_EtECvIKpahQ",
    authDomain: "final-year-project-stora-c4785.firebaseapp.com",
    projectId: "final-year-project-stora-c4785",
    storageBucket: "final-year-project-stora-c4785.appspot.com",
    messagingSenderId: "51713848155",
    appId: "1:51713848155:web:60a0ad4630078a8ac0a07e",
    measurementId: "G-TWV6ENFGDS"

  };

// Initialize Firebase
const initStor = initializeApp(firebaseConfig);
```

Figure 34: Firebase configuration file

With a reference to the image, it should be possible to display an image stored in firebase. This can be achieved by using the *useGetDownloadURL* from the firebase module, which returns the URL which downloads the object for use in the application. This URL can be stored in a *useState* and placed in an image component, successfully loading the image from storage.

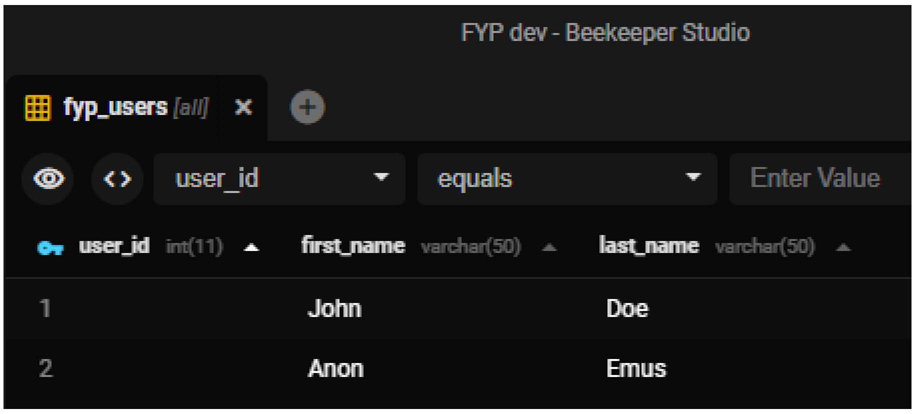
5.2.2.3. File Upload

With direct access to the bucket, the model exporter function can be edited to allow it to upload to storage. After the blob is assigned, a reference is made to the desired location one wants to store the file, as well as the name of the file and its extension. To begin uploading the file, the blob and reference is placed in the *uploadBytesResumable* method, which asynchronously and progressively uploads byte by byte, allowing for displaying of the upload progress through snapshots. When the file finishes uploading, custom code can be executed for debugging the firebase download URL for the file. Finally, checking the directory in the bucket will reveal the file having been successfully uploaded. As it uses the same function for downloading the models, an if statement is added to switch which functionality is needed based on the input attribute, the buttons for calling the function adding true for uploading or false for downloading.

See Appendix F for the completed code for file uploading/ saving.

5.2.2.4. Database setup

With file storage set up and working, a database will be needed to store information for user accounts and for shared models, as there is no simple method of listing files directly from Firebase. To begin, MariaDB is installed locally, running as a background process. To access the database as an admin for creating test tables, a 3rd party database viewing IDE *Beekeeper* is used. Connecting to the database, an example table *fyp_users* is made, containing ID, first name and last name columns. Two sets of test data is inserted, John Doe and Anon Emus.


FYP dev - Beekeeper Studio

fyp_users [all]

user_id equals Enter Value

user_id int(11)	first_name varchar(50)	last_name varchar(50)
1	John	Doe
2	Anon	Emus

Figure 35: Test table format

To then access this data from the back-end itself, a MariaDB module is installed, as was the case for the firebase storage connection. This allows for configuration of the connection to the database, using the username of the admin, the host address, and the database in question as values for attributes.

```
var mdb = require('mariadb');

mdb = mdb.createConnection(
    {
        user:"root",
        host: "localhost",
        database: "fyp_db"
    }
)
```
Figure 36: Config of MariaDB connection

Adding this to a new API *db_select*, a get request can be created, returning the rows returned by a specific query from the *fyp_users* table. Each element returned is added to a *useState* array in the front-end, which is mapped out with the value of each column accessed through JSON notation. This will return two div elements, using the previously defined dynamic div in the browse page, each with a first name 'John' and 'Anon', and last name 'Doe and 'Emus', the title of the divs representing the index of the elements in the array.

```
router.get('/', function(req, res, next) {

    mdb.then(conn => {
        conn.query("SELECT * FROM fyp_users").then(
            (rows => {
                res.send(rows);
            })
        )
    })
});
```
Figure 37: Elements returned to front-end

An additional function to test is a post request, taking a JSON element in the request to filter the query by a given variable, such as by a string in the first name column. This is facilitated by adding a search bar input element, updating the variable to be passed into the post request, alongside a select input element to determine which column is used to filter the result.

```
//Express db_select API
router.post('/', function(req, res, next) {

    b.then(conn => {
        conn.query(`SELECT * FROM fyp_users WHERE ${String(req.body.stype)}
LIKE '${String(req.body.term)}'`).then(
            (rows => {
                if (rows.length == 0) console.log("The search is invalid");
                res.send(rows);

            })
        )
    })

});
//Browse.js
const postSearch = () => {
       post("http://localhost:9000/dbTest", {term: searchTerm, stype:
searchType})
            .then(res => res.data)
            .then((data) => {
                clearResp()
                data.forEach(element => {
                    addResp(element)
                });
            }).catch(e => {
                console.log(`${e} has occured`)
            })
        }
```

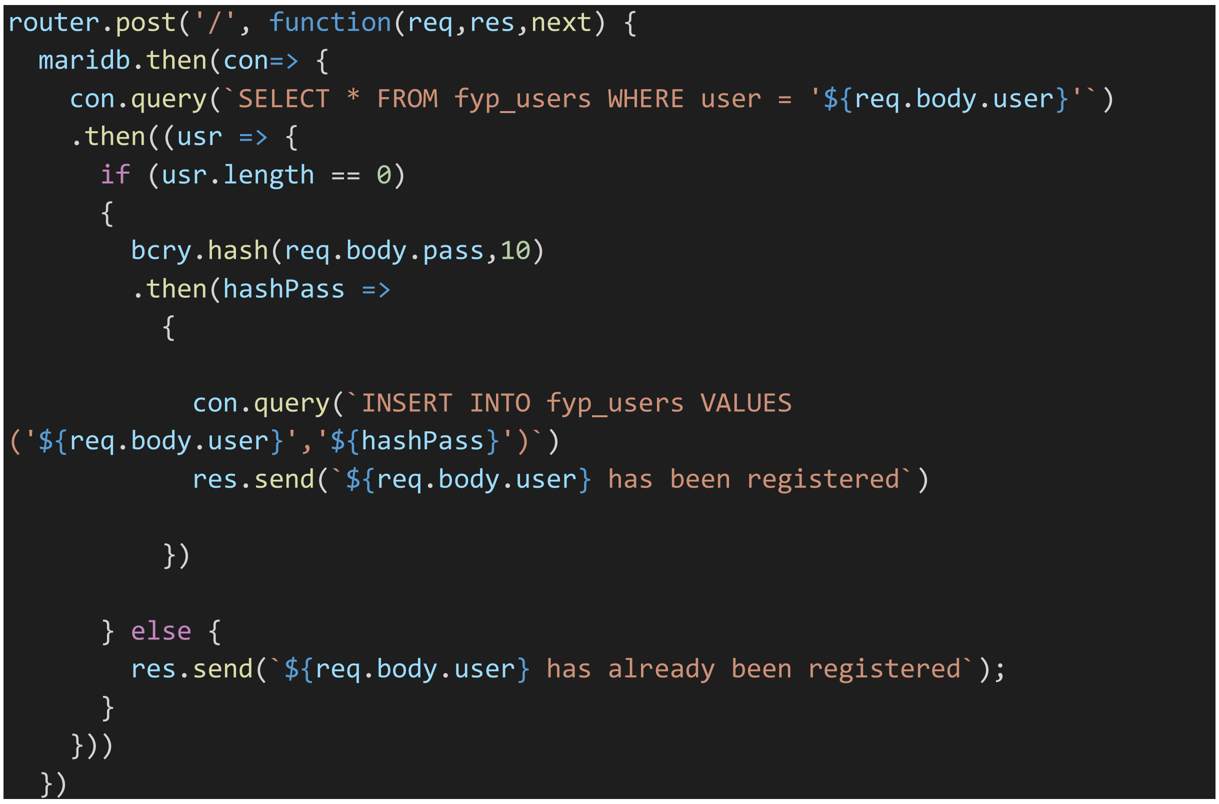
Figure 38: Post request for search filtering

5.2.2.5. Register/Login

With an established framework for accessing a database, a basic user registration and log in system can be implemented. This will involve insert queries into the *fyp_users*, modified to take a password instead of last name, checking if the username already exists before encrypting the password and creating the account for registration, and returning a temporary web token stored in the browser to keep track of who is logged in.

To begin, a simple UI for entering a username and password is created on the nav bar, so that it's always available anywhere on the application. These values are entered into a post request, now handled by the front-end Axios module which offers clean syntax for get, post and set. The API called by the post depends on if the register or login button is clicked, calling the relevant API.

For register, the *fyp_user* is queried to check if the username exists. If it returns no values, then the attached password is encrypted with the *bcrypt* module, hashing it for a set number of rounds to prevent sensitive data from being revealed to 3rd parties. These values are thus entered as a new row, returning a message to the front-end notifying the user of either a successful or unsuccessful registration.



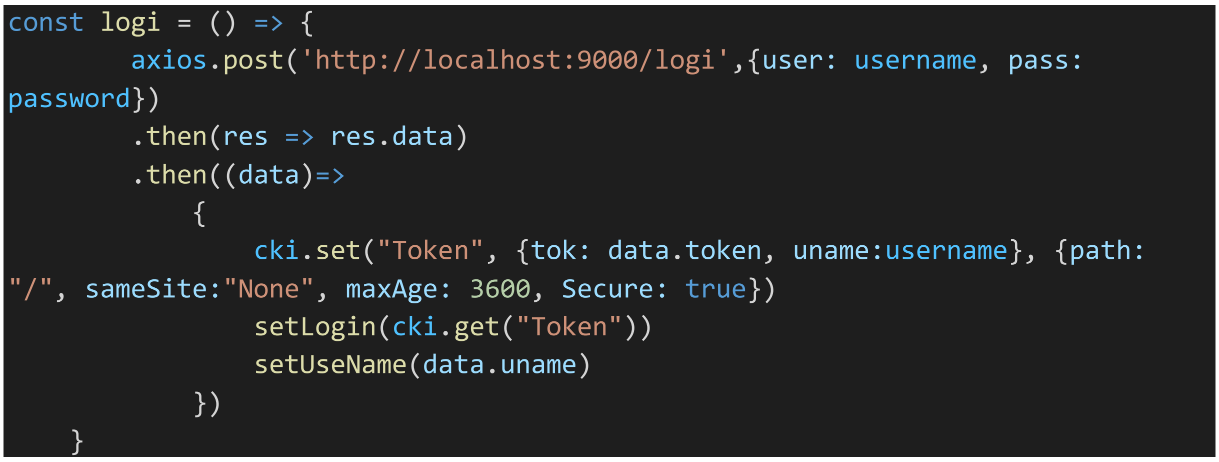
```
router.post('/', function(req,res,next) {
  maridb.then(con=> {
    con.query(`SELECT * FROM fyp_users WHERE user = '${req.body.user}'`)
    .then((usr => {
      if (usr.length == 0)
      {
        bcry.hash(req.body.pass,10)
        .then(hashPass =>
          {

            con.query(`INSERT INTO fyp_users VALUES
('${req.body.user}','${hashPass}')`)
            res.send(`${req.body.user} has been registered`)

          })

      } else {
        res.send(`${req.body.user} has already been registered`);
      }
    }))
  })
```

Figure 39: Register API Code

The log in API, in a similar fashion to registration, checks the database for the provided username. In this case, a null response would provide an error message, citing that the account doesn't exist, whereas a successful return requires comparison of the password provided and the encrypted value in the database, facilitated by bcrypt. If the comparison returns true, a persistent token should be generated to ensure that the login can be tracked through browser sessions. This is achieved through the creation of a JSON web token, storing an identifying string, the username and key for randomizing the token output. This token, along with other necessary data for the application, in this case the username, is returned to the front-end, to be placed in a browser cookie.

Back in the front-end, the status of the log in must be persisted throughout the application, as *useStates* and other methods of storage such as sessions would reset upon page refresh or update, ruining such persistence. This is remedied by browser cookies, storing data in JSON format that can last for a set amount of time, with attributes storing strings, integers etc and can be accessed through get and set requests.

Using the *universal-cookie* module, the data returned by the response from the API is set into a cookie named 'token', holding the value of the web token and the username, the latter of which will be used for determining where models will be stored on Firebase, split by username, and thus show up when such a model is shown in the model browser. This cookie is limited to ~1hr lifetime, expiring and no longer valid to log in after that point.



```
const logi = () => {
        axios.post('http://localhost:9000/logi',{user: username, pass: 
password})
        .then(res => res.data)
        .then((data)=>
            {
                cki.set("Token", {tok: data.token, uname:username}, {path: 
"/", sameSite:"None", maxAge: 3600, Secure: true})
                setLogin(cki.get("Token"))
                setUseName(data.uname)
            })
    }
```

Figure 40: Login post and cookie set

This successfully logs a cookie in the browser, seen in both the Firefox debug tools and printing the cookie from a get request, replacing the data entry in the nav bar with the username. An issue arose, however, with how to check for this log in on every page that isn't the nav bar, as they wouldn't share variables from *useStates* across pages. This had been remedied by checking for the cookie in the *useEffect* hook in the relevant pages, taking Home for example, reflecting the login by addressing the username in the welcome message. The hook in question runs whenever the page is reloaded or updated, but can also updated based on a dependency array, which allows it to run based on if an attribute is updated. This, in tandem with a *setInterval* hook, allows for the necessary *useStates* holding the page-specific data to refresh instantly without refreshing the page, allowing for seamless access to features granted on login with little hassle. This is replicated for the create page, to lock the file upload functionality to only when a valid user is logged in.

With the above login tracking methods complete, a simple logout feature is possible, deleting the token from the browser and forcing an update of the *use*Effect, reverting the UI back to the data entry and for other pages to treat the user as not logged in.

5.2.2.6. Model Browsing

With all the pre-requisite functionality needed for the model sharing page complete, these features should be modified accordingly. The file upload function now sends a post request to a new *addModel* API, attaching the filename, username and the *downloadURL* retrieved from the upload to insert into a new *fyp_models* table, alongside the current date. This table will be used to keep a record of models that have been shared to allow for them to be queried directly from the application and potentially sorted by any of the columns.

Modifying the dynamic divs, the most of returned JSON data from *fyp_models* is parsed as normal, displaying the string values for username and filename. The date value returns a null time value alongside the set date, so it is spliced to remove the unnecessary null value. The only data currently unused is the *downloadURL*, as it must be used in a *GLTFImporter* to then be rendered in the canvas elements in the div. This importer is put into a *useLoader* along with the model link when returned, returning a primitive object in the canvas element. This element is wrapped in S*uspense* elements so that the app waits for the model to fully load from the storage to prevent an error occurring by trying to display a half-loaded element.

This, however, hits a snag when running the application, as another *CORS* error is returned. This indicates that Firebase is denying direct access to the models, despite having displayed images uploaded to the storage previously. This is rectified by editing the sharing rules in Firebase through the *gsutils* command line tool, connecting to the bucket through its unique identifier, uploading a *cors.json* file detailed in the figure below to the sharing rules in the bucket, allowing for get requests to be made from the localhost, altering the rules found on the storage.

```
[
  {
    "origin": ["https://example.com"],
    "method": ["GET"],
    "maxAgeSeconds": 3600
  }
]
```

Figure 41: cors.json file, to allow get requests on Firebase

With this alteration made and propagated through the bucket, the browse page will now successfully render the models in a small canvas element in the div, with added orbit controls to allow for a preview of the model before the user downloads it. The previously implemented search and filter functions can be altered with the *fyp_models* table in mind, allowing for filtering by username, model title or date of creation. Finally, an option to download the model can be easily added, using the same methods as for the create page download feature by creating an invisible link and virtually pressing it to initiate the download from Firebase.

As the above features are properly implemented, the aforementioned tag system can be added. A JSON doc column *tags* is added to the *fyp_models* table, containing comma separated values for each row. This is accessed by the same means as any other columns, though the method for inserting values required splicing of a comma-separated input, mapping this data to remove unnecessary white space for each element. After these modifications, tags are added at file creation alongside the file name, being returned in the browse function and allowing for filtering by tag.

5.6. Conclusions

As of the implementation of the last feature above, the experiment is now feature-complete, with all the necessary functionality for this application in place. With these on track, the system is ready for subsequent testing and evaluation.

6. Testing and Evaluation

6.1. Introduction

With the system feature complete, it is important that it both functions properly and that it is evaluated to determine possible underlying issues, such as with the user experience. To this end, both system testing and user evaluation will be performed. The former will ensure that functionality work as intended, while the latter can capture user feedback on what could be altered or improved.

6.2. System Testing

For testing the functionality and limits of the system, three types of testing were carried out, in order to cover as much of the application as possible.

6.2.1. Manual Testing

The act of trying many alternate inputs into potentially vulnerable input fields, testing for unexpected or incorrect results. In particular, this was performed on the register/ login input fields for username and password, and the ensuing post requests / cookies logged as a result. This was important to test so that the correct data would be inserted and selected from the database, while also being able to extract the correct metadata from the cookies.

As the feature was being developed, many different inputs were tested, some including special characters such as spaces, white space and more. None that were tried seemed to have an effect on the overall functionality and did not alter the data in the database either.

Alternatively, the model editing feature required much manual testing, as it took some experimentation to find the standard extrude height to be used. After toying around with some set values, having the scaling factor of *5*(5*scale)* worked well, as a relatively low *scale* extrudes the geometry enough to be seen, while linearly increasing it created even elevation differences, allowing for consistent extrusion no matter which image is selected.

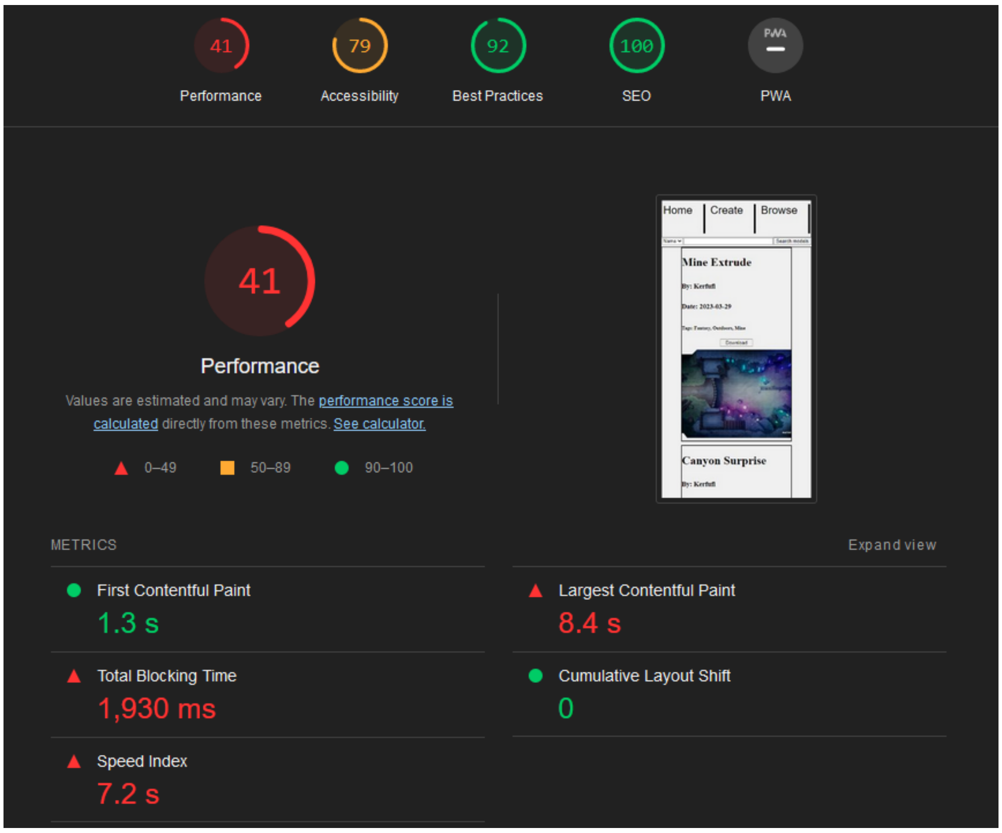
6.2.2. Performance Auditing

Keeping in mind the level of processing required for rendering and manipulating 3D models, it is important to monitor the performance of the application, particularly when multiple models are displaying on screen. As the application is intended for a wide audience for a low level of hassle, work should be done to minimise the burden on the processor.

To do this, a browser extension called *Lighthouse*[23] is added, which can scan current web pages or applications, generating a report on performance usage, accessibility and more. This was done for both create and browse functionality, one to determine single model performance, the latter to see how multiple models affected it. For each page, accessibility scored high, and performance was standard at around 64 % when tested on the relatively light home page. When running those pages, performance dipped to ~55% and 49% respectively, with the latter page having around 2 models

[23] 'Lighthouse'.

loading in from the database. This gives a rough 5% performance drop with every model added on screen, though exact calculations may vary depending on the complexity of the geometry and the fidelity of the texture attached.


41 Performance | 79 Accessibility | 92 Best Practices | 100 SEO | PWA

41

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● First Contentful Paint 1.3 s

▲ Largest Contentful Paint 8.4 s

▲ Total Blocking Time 1,930 ms

● Cumulative Layout Shift 0

▲ Speed Index 7.2 s

Figure 42: Browse page performance audit

With this knowledge, it would be wise to search for ways to minimise the burden placed on the application, so that weaker or metered connections aren't burdened when trying to access this service.

6.2.3. Unit Testing

Creating unit tests for major functionality of an application can be incredibly helpful in the case that an issue is encountered and the specific tests return exactly what is causing it, narrowing down the debugging process. To begin creating these, a *ReactJS* compatible testing framework is chosen, namely *Jest*[24], while some React-specific testing modules can be imported from the current installation. Configuring this framework involves creating a __*tests*__ folder, with the JavaScript files handling the tests having a *.test.js* extension to distinguish them from standard files. As a basic test, the model function will be examined, seeing if it renders with two children objects: The geometry and material. This test will be executed through the command line in the react app directory by running *npm test*, which will search for test files in the __*tests*__ folder.

```
import ReactThreeTestRenderer from '@react-three/test-renderer';
import { Box } from '../pages/Create';
```

[24] 'Jest'.

```
test('mesh to have two children', async () => {
    const renderer = await ReactThreeTestRenderer.create(<Box/>);
    const mesh = renderer.scene.children[0].allChildren;
    expect(mesh.length).toBe(2);
})
```

Figure 43: render.test.js example test

However, when running this test, numerous errors were thrown. Not so much that the mesh is not rendering two children, but that imports at the top of the file is not allowed when files are not specified as modules. This isn't limited to rendering a smaller function, as a second test was written, checking if the home page will render, as detailed in figure 44. This throws similar errors, while introducing some of its own, namely *'hRef cannot be used outside of <Router> context'*, meaning it cannot see the react page routing that was configured in the original set up.

```
test('Render home module', async () => {
    const renderer = await ReactThreeTestRenderer.create(<Home/>);

})
```

Figure 44: Partial render test

Work arounds were researched for the above issues, including marking files as modules in the *package.json* and wrapping functions in router elements to allow for the tests to work as intended, but no headway was made using these methods. Unfortunately, this means that unit tests were unable to be implemented due to unforeseen circumstances.

6.3. System Evaluation

As this application is aimed at a general audience that would want intuitive access to 3D models, it was important to perform user evaluation, so that feedback could be gathered on how to improve the user experience. This evaluation was performed late into the project life cycle, due to the time it took to implement the base functionality to a displayable level, though this can still be taken into account for future work to be done with this project.

Three groups of individuals were asked to evaluate the application: My peers in their final year of computer science, with little or no experience with the subject matter of tabletop roleplaying games. Other computer science students in first and second year, with more personal experience playing roleplaying games and using the maps that act as input for this project and a group of TUD students who play roleplaying games weekly with one of the college's societies, with little or no experience with web development or other computer science topics.

Out of all three groups, none had an understanding of the mathematics behind the manipulation of 3D geometries present in the project, so little was said on this front. The main focus for these groups was the user experience. The home page was considered self-explanatory, with two clear options for whichever feature is required by the user. For create, the requirement to log in to upload models made sense, but some found the interface lacking; there wasn't enough explanation on the page itself to explain how to edit the geometries, as the tool defaulted to such a state that no editing can occur unless an UI setting is altered and that there it was unclear what face of the geometry would be lifted when edited. For browse, little issue was found except for the format in which database entries were returned, as previous experience with digital storefronts such as Amazon would lead them to prefer elements displaying in a column format or a strict n elements per line.

6.4. Conclusions

The results of these tests and evaluations, though unable to be entirely integrated into the final product of the application, can still be taken into account for future projects that involve similar technologies and functionality, as well as any work that is done in relation to this application down the line. The functionality that had been successfully tested and integrated had benefitted from avoiding issues and pitfalls that could have weighed down the project functionality.

7. Conclusions and Future Work

7.1. Introduction

With the project completed, it is important to log the various lessons that were learned and issues that were faced throughout the process of research, development, and testing. On top of this, the result of all this work can still be built upon, refining existing aspects while potentially adding new functionality to expand the core idea for personal use.

7.2. Conclusions

Throughout the project, many different skills with technologies, research and more were required, although some were less developed to begin with. As a result, many useful skills were learned during the course of researching for and creating this implementation.

7.2.1. Lessons Learned

When this project, an application heavily driven by web development and database administration, began, I had the bare minimum experience in both, having not worked with any full stack implementations before, while only utilising databases in isolation. Though having the option to forgo the web application route and to instead build a local application, this opportunity served well in accelerating the speed at which I learned full stack development, quickly picking up how to utilise ReactJS and ExpressJS in tandem in order to implement some of the more complex features found within the project. Though frustrating at first, the learning experience had grown into an enjoyable past time, with hours being spent on implementation resulting in gratification and swiftly moving on to a new, more complex function, while gaining experience that adds nicely to my CV.

7.2.2. Issues Faced

On the flipside of the usually pleasant learning experience, many of the issues faced was a result of learning as the project continued. Namely, the functionality was complex enough that little time was available to enhance the visuals of the application, as base functionality was completed later than anticipated in the project life cycle. The simplicity of the interface had been commented on by the evaluation groups, though they had understood that ensuring functionality came first.

Another major issue found was one of documentation. As *react-three-fiber*, the library used for the rendering and manipulation of 3D models, was ported from the original ThreeJS library for use in ReactJS years following its original release, documentation was sparse when compared to what was available for the vanilla JavaScript library. This became apparent when trying to export 3D models in the front-end development, as very little was found online for the syntax of the exporter, not helped by the fact that it was an example module that was duplicated by *react-three*, with only the syntax for the original JavaScript being available.

This came to a head when trying to implement this at first, as what little was available had resulted in corrupted files when downloaded, or failure to export in the first place. It took some asking around on ThreeJS communities to find out the correct syntax for JSX to correctly export the resulting data, as different *.then()* functions correlated with different outputs, The same issue arose with the *Firebase* upload function, as the first output of the upload turned out to return the status of the upload, such as how many bytes were uploaded, rather than executing code following a successful upload and was solved in a similar manner.

These issues were amplified with documentation regarding the 3D rendering aspect of the library. From what was found online, most resources involving *react-three* involved the importing of models to display them in interactive 3D and, as such, little was found on the manipulation of geometry. In particular, the process of updating the positions of vertices after manipulation was tedious, as the vital *needsUpdate* attribute was moved in a previous update of *ThreeJS*, necessitating a search through sparse recent documents to finally find it in the code.

On top of this, the act of colouring the outline of a specific face of a model when the mouse passed over it, though seemingly simple, became too much to implement. Some methods offered altered the colour of the entire mesh, with others requiring an entire restructuring of how the model is rendered to begin testing it, while also requiring data of the object that isn't available outside of that scope. This feature would have added to user accessibility and would play into a visualisation of how far a face would be extruded before clicking, which also suffered the same fate due to the complexity and potential extra rendering load accrued if implemented.

7.3. Future Work

When first researching this premise, the design document had been made to list the functionality and features that it should have to offer the best user experience and how to achieve this. At the end of this document, it is clear that some design choices did not make the cut, either due to excessive complexity or other reasons. Here, plans can be made for the case that this project will be revisited in the future as a solid base on which accessibility and functionality can be improved, with the potential of releasing it as a publicly available tools that would benefit the niche of Tabletop Roleplaying Games played in a 3D environment.

7.3.1. Front-end features

The aforementioned face highlighting functionality would be a big one to implement, as it would assist the user in identifying exactly which part of the model will be edited when they click, as it can be somewhat unclear now due to the way it had been implemented, requiring a bit of clicking around to raise a given face to make an even plane. This, in tandem with a visual of what the mesh would like raised when mousing over each face would be informative of how the tool works with minimal tool tips required. There are multiple methods to colour the individual faces but would require a refactor of the model rendering by adding definitions for each face that can be referenced, so it would take some time to work.

Continuing with features for the creation aspect, it would work well to be able to determine the aspect ratio of an image before applying the model, as the three options available are catch alls, but not representative of all images that would be uploaded. This could be implemented by reading the metadata of the image when uploaded, extracting the pixel width by height, providing that ratio to the scale settings directly.

Finally, on the models itself, the ability to create primitive 3D objects in the render to simulate objects on the map that would otherwise be awkward to emulate by manipulating the existing geometry, such as the bridges and what's beneath them, would go a long way to improve the feel of some maps. With currently unused box-placing functionality already in place, a similar system could be implemented to add simple bridges, houses, walls and more to prevent awkward stretching of the terrain and to keep the model more natural looking.

7.3.2. Back-end Features

The user log in functionality, used in determining who could upload models to the service, ended up being underutilised, in hindsight. Plans were made to allow for allowing the user to see models that they have made, pin favourites and to press a like button once per model. These are nice to have, but were ultimately not essential to the original idea, but could still be tracked by adding tables in the database for saved content, while adding a 'like' column in the *fyp_models* table to track this statistic.

As seen in the testing and evaluation section, The performance report made by *Lighthouse* had indicated that the rendering of the models, multiple in particular, caused a significant burden on the browser in terms of processing. It would be helpful to explore options for optimising these rendering methods or perhaps the storage of the models themselves. It is unclear whether the performance decrease stems from purely the rendering or from the process of sourcing the models from the cloud storage as well. This may be alleviated through the compression of the model files when stored with subsequent decompression when loading again, while a local storage solution could be utilised whenever the app is hosted by a physical or cloud server, reducing a bottle neck imposed by a third party cloud storage service like *Firebase.*

Overall, hosting the application on a domain or cloud server would be ideal if plans are made to offer this service to the public. With access to online resources, the conversion process for back-end communication would be simple enough, with changing port sources and destinations being simple enough within the settings. An issue with hosting to a wide audience would be security concerns, as the project has not focused on security thus far, instead making sure that the functionality was completed. With the base for the application already set, it would be wise to prevent SQL insertion, ensure the attributes of the database are properly encrypted and that other such counter measures are taken to ensure a safe service that can be used by all.

Bibliography

‘About — Blender.Org’. Accessed 28 November 2022. https://www.blender.org/about/.

MariaDB.org. ‘About MariaDB Server’. Accessed 29 November 2022. https://mariadb.org/about/.

Blend Swap. ‘Blend Swap’. Accessed 28 November 2022. https://www.blendswap.com.

‘Cost-Optimized and High-Performance Database’. Accessed 29 November 2022. https://www.oracle.com/ie/database/.

Django Project. ‘Django’. Accessed 28 November 2022. https://www.djangoproject.com/.

‘Express - Node.Js Web Application Framework’. Accessed 29 November 2022. https://expressjs.com/.

‘Firebase’. Accessed 29 November 2022. https://firebase.google.com/.

‘Fundamentals - Three.Js Manual’. Accessed 28 November 2022. https://threejs.org/manual/#en/fundamentals.

Planview. ‘Introduction to Kanban’. Accessed 3 December 2022. https://www.planview.com/resources/guide/introduction-to-kanban/.

‘Jest’. Accessed 31 March 2023. https://jestjs.io/.

Infinity. ‘Kanban Methodology: All You Should Know’. Accessed 8 December 2022. https://startinfinity.com/project-management-methodologies/kanban.

‘Lighthouse’. Accessed 30 March 2023. https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk.

‘React – A JavaScript Library for Building User Interfaces’. Accessed 29 November 2022. https://reactjs.org/.

Scriven, Paul. ‘From Tabletop to Screen: Playing Dungeons and Dragons during COVID-19’. *Societies* 11, no. 4 (December 2021): 125. https://doi.org/10.3390/soc11040125.

Sidhu, Premeet, and Marcus Carter. ‘The Critical Role of Media Representations, Reduced Stigma and Increased Access in D&D’s Resurgence’, n.d., 20.

‘Sketchfab - The Best 3D Viewer on the Web’. Accessed 28 November 2022. https://sketchfab.com/.

‘Svelte • Cybernetically Enhanced Web Apps’. Accessed 29 November 2022. https://svelte.dev/.

‘TaleSpire - FAQ’. Accessed 28 November 2022. https://talespire.com/faq.

Management Library. ‘Waterfall Methodology – Ultimate Guide’, 21 March 2022. https://managementhelp.org/waterfall-methodology.

The Khronos Group. ‘WebGL’, 19 July 2011. https://www.khronos.org//.

Planview. ‘What Is FDD in Agile?’ Accessed 3 December 2022. https://www.planview.com/resources/articles/fdd-agile/.

‘What Is Feature Driven Development (FDD)? & How It Works?’, 2 September 2021. https://www.digite.com/agile/feature-driven-development-fdd/.

MongoDB. ‘What Is MongoDB?’ Accessed 29 November 2022. https://www.mongodb.com/what-is-mongodb.

‘What Is the Waterfall Model? - Definition and Guide’. Accessed 2 December 2022. https://www.techtarget.com/searchsoftwarequality/definition/waterfall-model.

Appendix:

Appendix A: Code block for Prototype; HTML/CSS

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Upcast: Online 3D modelling and Sharing Service</title>
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <div class="flex-container">
            <div class = "home-div" >
                <a href="Home.html">Upcast</a>
            </div>
            <div class = "home-div">
                <a href="Creator.html">Create</a>
            </div>
            <div class = "home-div">
                <a href="Share.html">Share</a>
            </div>
        </div>
        <hr>
        <h1 style="text-align: center;">Welcome to Upcast</h1>
        <h1 style="text-align: center;">Which service would you like to
access?</h1>

        <div class="flex-container">
            <div class="choice" style="justify-content: center; width:
35%;text-align: center;margin:auto"><h1><a
href="Creator.html">Create</a></h1></div>
            <div class="choice" style="justify-content: center; width:
35%;text-align: center; margin:auto"><h1><a
href="Share.html">Share</a></h1></div>
        </div>
        </div>

    </body>
</html>
```

Appendix B: Code block for Prototype; 3D rendering

```
import * as THREE from "/node_modules/three/build/three.module.js";
import { OrbitControls } from
"/node_modules/three/examples/jsm/controls/OrbitControls.js"

//Creates 'sceene' to hold the 3D object and camera to view
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
window.innerHeight, 0.1, 1000 );


//The renderer is created and added to the page
const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );

const controls = new OrbitControls(camera, renderer.domElement);
//A small green cube and its outline
const geometry = new THREE.BoxGeometry( 1, 1, 1 );
const g = new THREE.EdgesGeometry(geometry);

//Colours for cube and lines are created and subsequently added to overall
mesh
const material = new THREE.MeshBasicMaterial( { color: 0x00ff00});
const linemat = new THREE.LineBasicMaterial({color:0x0000FF});

const lineMesh = new THREE.LineSegments(g,linemat);
const cube = new THREE.Mesh( geometry, material );

//Adds rotation to the cube and its outline
cube.rotateY(120);
cube.rotateX(-90);

lineMesh.rotateY(120);
lineMesh.rotateX(-90);

//Finally adds completed model to screen
scene.add( cube );
scene.add(lineMesh)

//Sets camera distance from object
camera.position.z = 5;

//Animation function to make it spin
function animate() {
    requestAnimationFrame( animate );
    cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;
    lineMesh.rotation.x += 0.01;
    lineMesh.rotation.y += 0.01;

    controls.update();

    renderer.render( scene, camera );
};

//Continuously runs animation when on screen
animate();
```

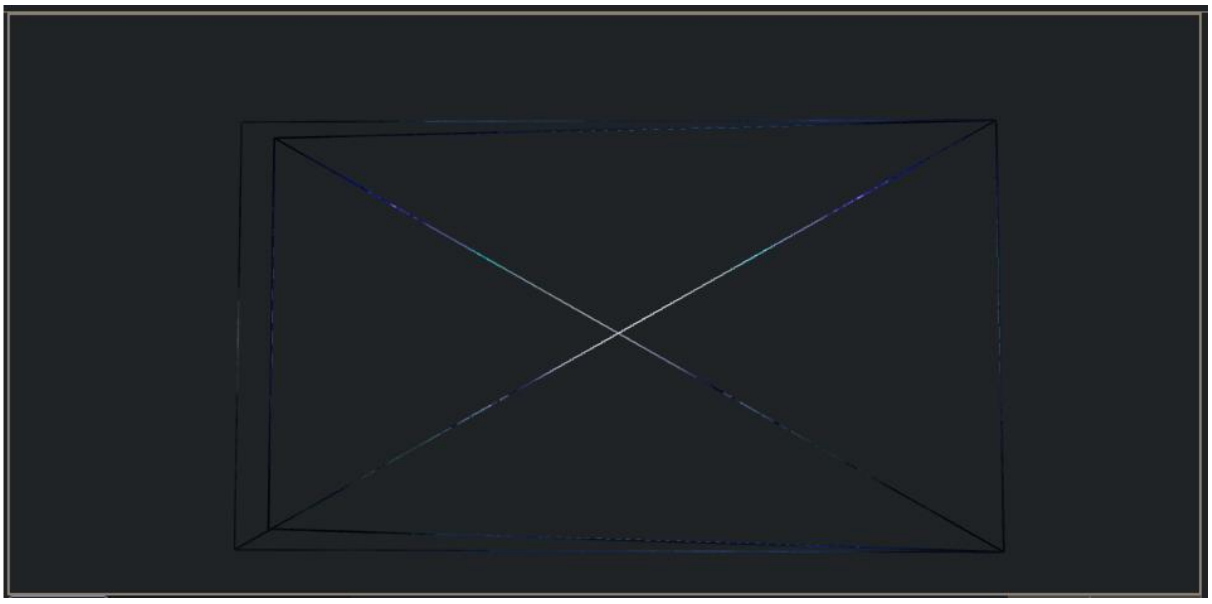
Appendix C: React-Three Test code

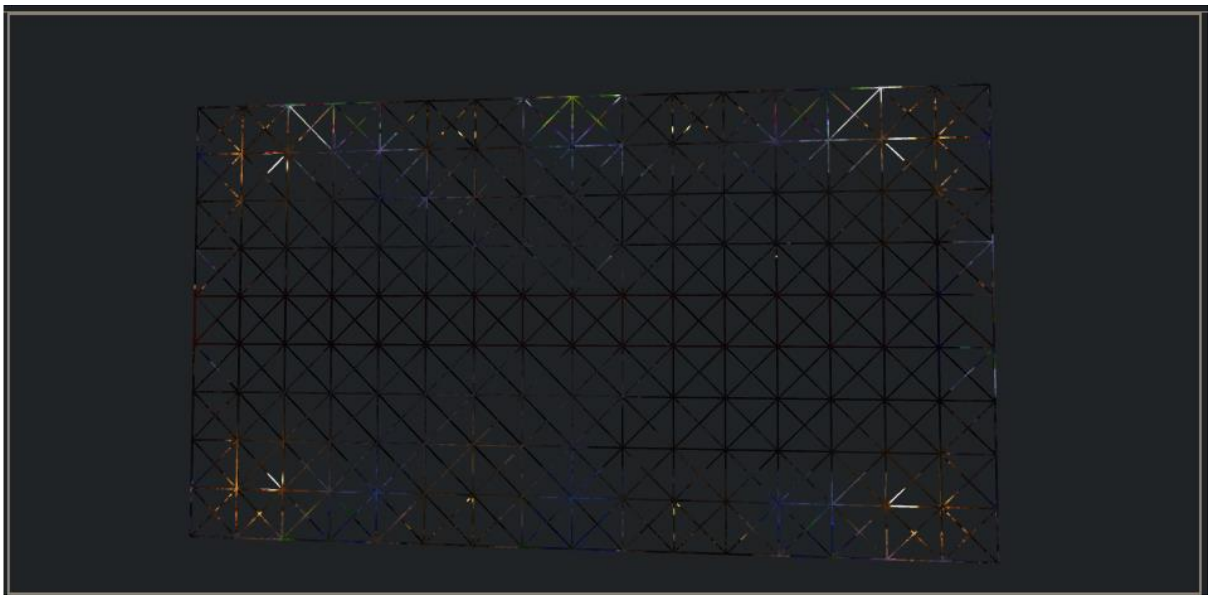
```
import { useRef, useState } from 'react'
import { Canvas, useFrame } from '@react-three/fiber'
import { OrbitControls } from '@react-three/drei'

function Box(props) {
  // This reference gives us direct access to the THREE.Mesh object
  const ref = useRef()
  // Hold state for hovered and clicked events
  const [hovered, hover] = useState(false)
  const [clicked, click] = useState(false)
  // Subscribe this component to the render-loop, rotate the mesh every frame
  useFrame((state, delta) => (ref.current.rotation.x += delta))
  // Return the view, these are regular Threejs elements expressed in JSX
  return (
    <mesh
      {...props}
      ref={ref}
      scale={clicked ? 1.5 : 1}
      onClick={(event) => click(!clicked)}
      onPointerOver={(event) => hover(true)}
      onPointerOut={(event) => hover(false)}>
      <boxGeometry args={[1, 1, 1]} />
      <meshStandardMaterial color={hovered ? 'hotpink' : 'orange'} />
    </mesh>
  )
}

export default function App() {
  return (
    <>
    <Canvas>
      <ambientLight intensity={0.5} />
      <spotLight position={[10, 10, 10]} angle={0.15} penumbra={1} />
      <pointLight position={[-10, -10, -10]} />
      <Box position={[-1.2, 0, 0]} />
      <Box position={[1.2, 0, 0]} />
      <OrbitControls />
    </Canvas>
    </>
  )
}
```

Appendix D: Box render on click functionality

```
const dCLickHandle = (e) => {
      let pos = e.intersections[0].point
      pos.z = pos.z + .25
      setBoxes((boxes) => [...boxes,pos])
      console.log(boxes)
}
...
<group ref={canvasRef}>
      <BoxRef
            ref={meshRef}
            position = {[0,0,-2]}
            i={text}
            scale={[aspect[0],aspect[1],.01]}
            wf={false}
            onDoubleClick={dCLickHandle}
      />

      {boxes.map((position,index) =>
      (
            <Box key={index} position={position} scale={[.5,.5,.5]}/>
      )
     )}
</group>
```

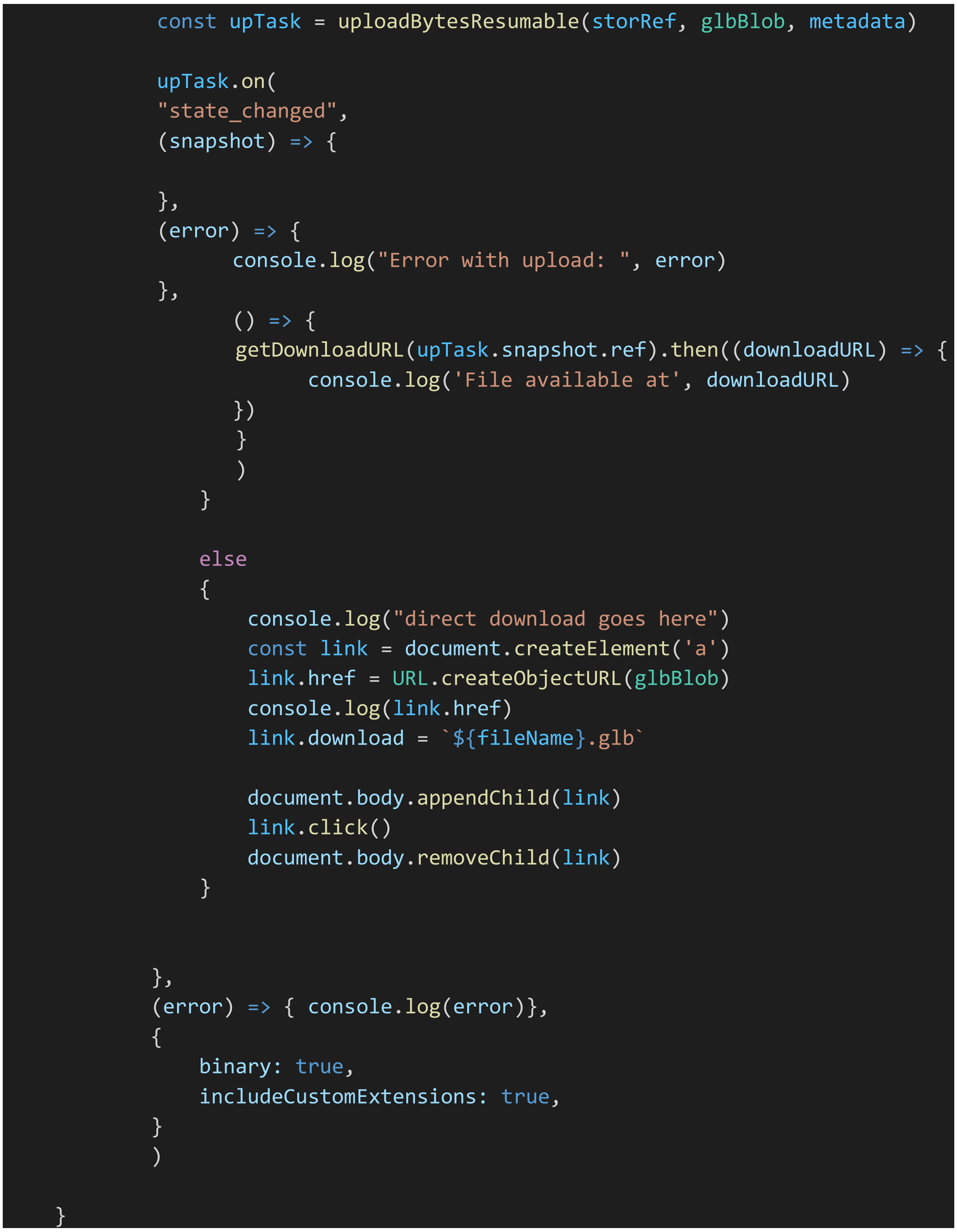
Appendix E: Difference between 1:1 and 16:9 width/height segments





Appendix F: File download/upload combo function

```
exporter.parse(
	canvasRef.current,
	(gltf) => {

		const glbBlob = new Blob([gltf],{ type: 'application/octet-
stream'})

	if (upl)
	{
		const storRef = ref(stor,`dir/${fileName}.glb`)
		const metadata =
		{
		contentType: 'model/glb'
		}
```



```
            const upTask = uploadBytesResumable(storRef, glbBlob, metadata)

            upTask.on(
            "state_changed",
            (snapshot) => {

            },
            (error) => {
                  console.log("Error with upload: ", error)
            },
                  () => {
                  getDownloadURL(upTask.snapshot.ref).then((downloadURL) => {
                        console.log('File available at', downloadURL)
                  })
                  }
                  )
               }

               else
               {
                   console.log("direct download goes here")
                   const link = document.createElement('a')
                   link.href = URL.createObjectURL(glbBlob)
                   console.log(link.href)
                   link.download = `${fileName}.glb`

                   document.body.appendChild(link)
                   link.click()
                   document.body.removeChild(link)
               }


           },
           (error) => { console.log(error)},
           {
               binary: true,
               includeCustomExtensions: true,
           }
           )

    }
```