Polytechnic University of Catalonia

Center of Image and Multimedia Technology CITM

Multimedia Degree, Bachelor’s Thesis

Gallery of interactive applications with 3D components

Case study

Rubén Chiquin

Mentored by Pau Fernandez

Multimedia

Terrassa, 2022-2023

Index

Summary

This thesis has the intention of exploring the potential of cutting-edge web technologies for creating immersive 3D experiences in the browser. This exploration will review different available technologies, and end up taking the form of a gallery of interactive applications, evaluated for user experience and optimized for performance. The results demonstrate the potential of these technologies for web-based 3D graphics and interactive applications.

Key Words

3D, Web, Software

Links

URL on està la maqueta, web, APP, videojoc, vídeo, etc.

Table Index

Figure Index

Glossary

- **JavaScript:** A high-level programming language used to create dynamic and interactive web content. It is supported by all modern web browsers and can also be used for server-side development with Node.js.

- **TypeScript:** An open-source programming language that is a superset of JavaScript, adding optional static typing and other features. It is designed to improve code maintainability and scalability in large applications.

- **React:** A popular JavaScript library for building user interfaces, developed by Facebook. It allows developers to create reusable UI components and efficiently manage the application state.

- **Three.js:** JavaScript library used for creating and displaying 3D computer graphics in a web browser. It is compatible with a variety of web technologies and provides tools for 3D model creation, animation, and interaction.

- **Tech stack:** A tech stack, or technology stack, is a combination of programming languages, software tools, and technologies used to build a web or mobile application.

- **API:** A set of protocols and tools that define how different software components interact with each other.

- **DOM manipulation**: Process of modifying the HTML and CSS elements of a web page dynamically using JavaScript.

- **SPA**: A client-side web application that dynamically updates content on a single web page, without requiring the entire page to be reloaded, resulting in a more fluid user experience.

-

- **React components**: Reusable pieces of UI code that can be composed together to create complex user interfaces.

- **React Hooks**: functions that allow stateful logic to be encapsulated and shared between different components in a more efficient and flexible way.

- **Development dependency**: A software package or library that is necessary during the development phase of a project, but not during production.

- **Shader:** A small program that runs on the GPU and is used to render graphics in 3D applications, games, and simulations. It controls the colors, lighting, and other visual effects of 3D objects.

- **Source code:** refers to the human-readable instructions that a programmer writes to create a computer program.

- **SDK:** Stands for Software Development Kit. It is a set of software tools and resources provided by a company or organization to help developers create applications for a specific platform, operating system, or software framework.

- **Declarative code:** A programming paradigm that focuses on describing "what" needs to be accomplished, rather than explicitly specifying "how" to achieve it.

1. Introduction

1.1. Motivation

The motivation behind this thesis lies in a personal interest in exploring the potential of emerging web technologies for creating engaging and immersive user experiences. The use of 3D graphics and interactive applications has become increasingly prevalent in various industries, and its is worthwhile to investigate the capabilities of React, TypeScript, and Three.js for creating such experiences in the browser.

Additionally, the growing demand for web-based solutions due to the COVID-19 pandemic, which has highlighted the need for innovative and accessible technologies for remote learning, product visualization, and virtual experiences. By exploring the potential of these technologies, this project hopes to contribute to the advancement of the field of web development and provide insights into the design decisions that can impact the overall user experience.

1.2. Formulation of the problem

Despite the increasing use of 3D graphics and interactive applications in various industries, there is still a lack of understanding regarding the design decisions and best practices for creating engaging and immersive user experiences. Additionally, there is a need for accessible and innovative web-based solutions due to the COVID-19 pandemic, which has highlighted the potential of online person-to-person interactions.

While React, TypeScript, and Three.js offer promising capabilities for creating 3D experiences in the browser, there is a need to explore their potential and optimize their performance to ensure smooth rendering and user satisfaction. Therefore, the problem addressed in this thesis is how to leverage React, TypeScript, and Three.js to create a gallery of interactive applications with 3D components that showcase different use cases, evaluate the user experience, and optimize the performance.

1.3. General objectives

- To explore the potential of React, TypeScript, and Three.js for creating immersive and engaging 3D experiences in the browser.

- To develop a gallery of interactive applications with 3D components that showcase different use cases for 3D graphics, such as virtual reality simulations, product visualizations, and educational tools.

- To optimize the performance of these applications to reduce load times and ensure smooth rendering of the 3D graphics.

- To contribute to the field of web development by providing insights into the design decisions and best practices for creating engaging and immersive user experiences using React, TypeScript, and Three.js.

1.4. Specific objectives

- To research the capabilities of the pnmdrs developer collective (explained in detail in this section), and put to the tests it's open-sourced libraries

- To touch different aspects of 3D and interactivity, using different event triggers to perform different 3D actions

- To evaluate the viability of production-ready 3D software on the web, through trial and pragmatic effort.

- To familiarize with the intricacies of digital models at a code level, and to become comfortable manipulating them and using them in applications

- To evaluate, if not in practice at least in principle, the other potential technologies that are arising and the future of the 3D technology for the web.

1.5. Target audience

This thesis is limited in scope to the exploration of the pnmdrs stack. While the findings may not apply to all 3D graphics applications or web development frameworks, the insights and best practices provided can be useful for developers and designers looking to create engaging and immersive user experiences through the web.

The target audience for this thesis includes developers, designers, and researchers interested in web-based 3D graphics and interactive applications. Users potentially benefiting from this work include various industries such as education, entertainment, marketing, and e-commerce, as well as individuals interested in exploring 3D graphics and interactive applications in the browser.

The optimized and user-friendly applications developed through this thesis can offer new ways to visualize and interact with products, educational content, and virtual environments, which can lead to improved learning outcomes, higher user engagement, and better customer experiences.

2. State of the art

Web-based 3D graphics have been rapidly advancing in recent years, driven by the growing demand for immersive and interactive content on the web. Three.js is one of the most popular libraries for creating 3D experiences in the browser, with a wide range of features and plugins that make it suitable for various applications.

Other technologies, such as the ones that will later be mentioned in this section, have also emerged as powerful tools for building 3D experiences on the web. It is this section's intention to lay out the advantages and caveats of each technology and justify selecting one over the rest.

The exploration on optimizing web-based 3D graphics has shown that techniques such as texture compression, model optimization, and lazy loading can significantly improve the performance and load times of 3D applications. These are some of the reasons why choosing the correct tech stack is of most importance.

But before diving into the comparison, let's analyze the hierarchy of abstractions of web-based 3D technologies and compare the repertoire of tech stacks that are available to solve the problem laid out by this thesis.

The hierarchy of abstractions on the tech stack chosen can be conceptualized as in the following figure:

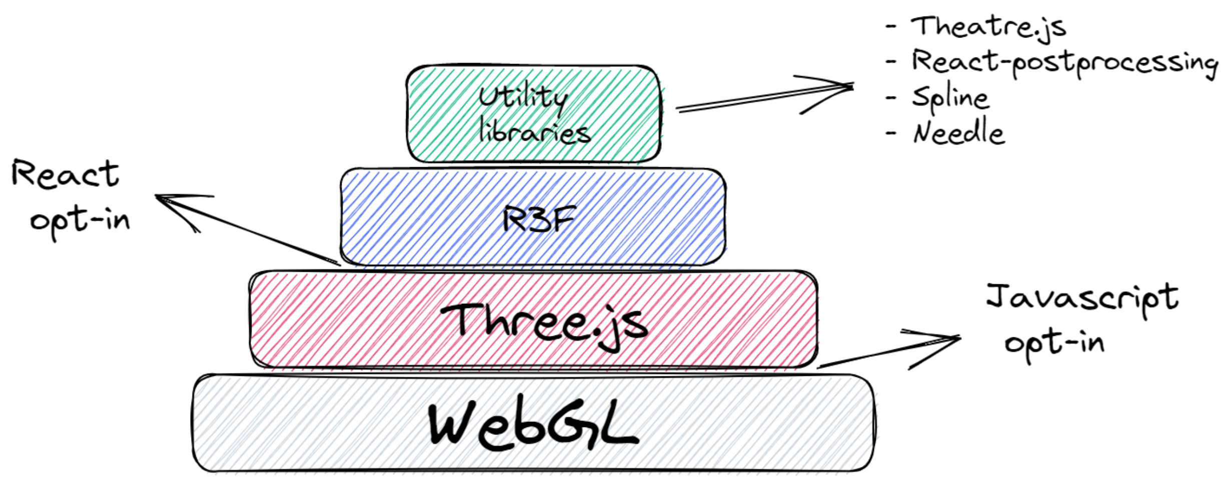


Figure x: Hierarchy of abstractions of the chosen tech stack

It is important to note that there have been layers of abstraction where an opt-in (A conscious and premeditated decision to choose a certain technology over another) has taken place. The first one would be to use technology regarding the web (WebGL). The second, the JavaScript programming language as the tool of choice (More concretely, TypeScript, as a superset of the former language). Finally, a React opt-in, which provides common ground for all the utility libraries that one may use for a specific task.

After careful consideration, these opt-ins were chosen for specific reasons. The personal interest and expertise of JavaScript / TypeScript has been a primary factor, but not the only one. According to multiple sources dated in 2022, such as Stack overflow survey[1] or Github's Octoverse Survey[2] TypeScript has been one of the most loved and wanted programming languages of today's market.

It is also known that React-Three-Fiber is a standard for 3D based solutions, as most utility libraries offer a direct connection to it. Cases such as Shopify[3] use these types of technologies on a daily basis. Given this prominence of usage, it was decided to stick to the React ecosystem, as it is considered the largest and fastest-growing tech stack available on the market.

Having said that and before proceeding to discuss the chosen tech stack, let us note that there will be an honorable mention of promising technologies that were not part of the thesis due to the opt-ins of choice, in section 2.6.

2.1. Three.js[4]

Three.js is a popular JavaScript library for creating 3D graphics and animations in web applications. It provides a comprehensive set of tools for rendering, animating, and manipulating 3D objects in a browser environment.

It maps out the WebGL API, a standard browser plug-in, allowing GPU-accelerated usage of physics and image processing and effects as part of the web page canvas.

It is considered the foundation of using 3D components on the browser. Although it can be used directly in last-generation tech stacks, it is often

[1] (Stack Overflow, n.d.)
[2] (*The Top Programming Languages | The State of the Octoverse*, n.d.)
[3] (*Shopify*, n.d.)
[4] (*Three.js*, n.d.)

layered with other libraries and third-party tools to complete the development tooling.

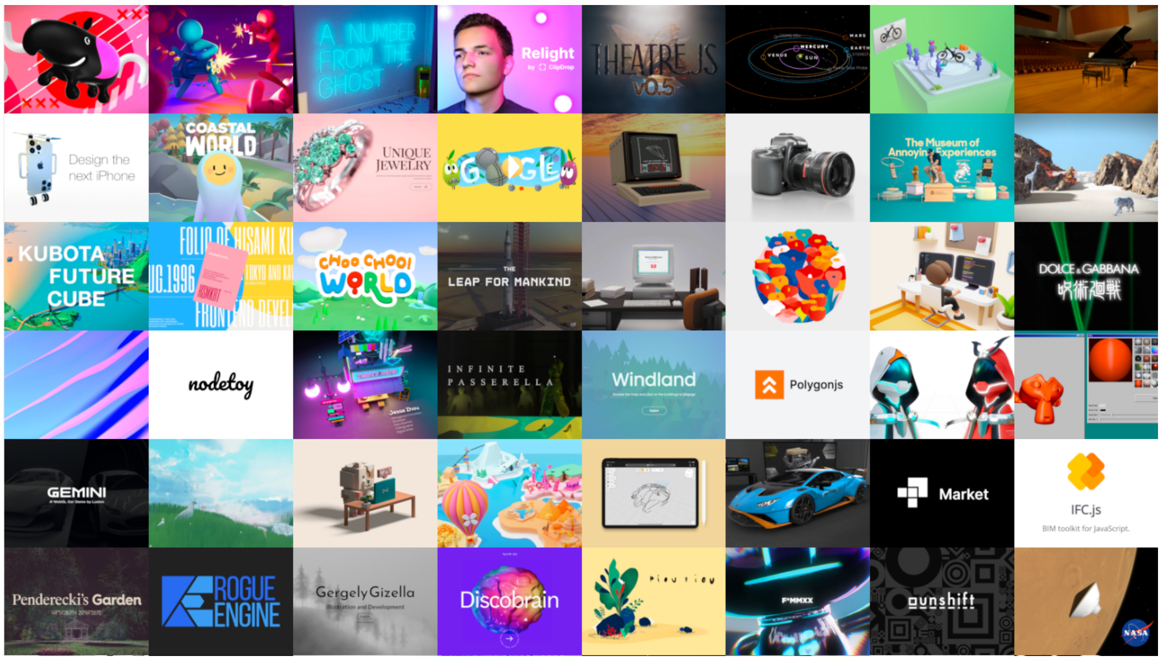


Figure x: Three.js webpage, showcasing star projects

2.2. React Three Fiber (R3F)[5]

In the era of [DOM manipulation]() and [SPA's](), the interactivity of web applications and creation of user interfaces is rarely as easy to implement as it is with frameworks such as Svelte, Astro or React.

React Three Fiber is a popular library that brings together the power of React and Three.js, allowing developers to create immersive and interactive 3D experiences in the browser.

It provides a [declarative]()way of creating and manipulating Three.js scenes and objects using [React components]() and [hooks](), simplifying the development process and making it simpler .

R3F was created and is maintained by Poimandres[6], an open source developer collective for the creative space. This collective has taken over the market niche of 3D web experiences, as they own and maintain not only the R3F library (one of the most popular 3D web libraries), but also a

[5] (*React Three Fiber*, n.d.)
[6] (*Poimandres Collective*, n.d.)

wide variety of complementary libraries that created a large ecosystem and the so-called *s stack*.

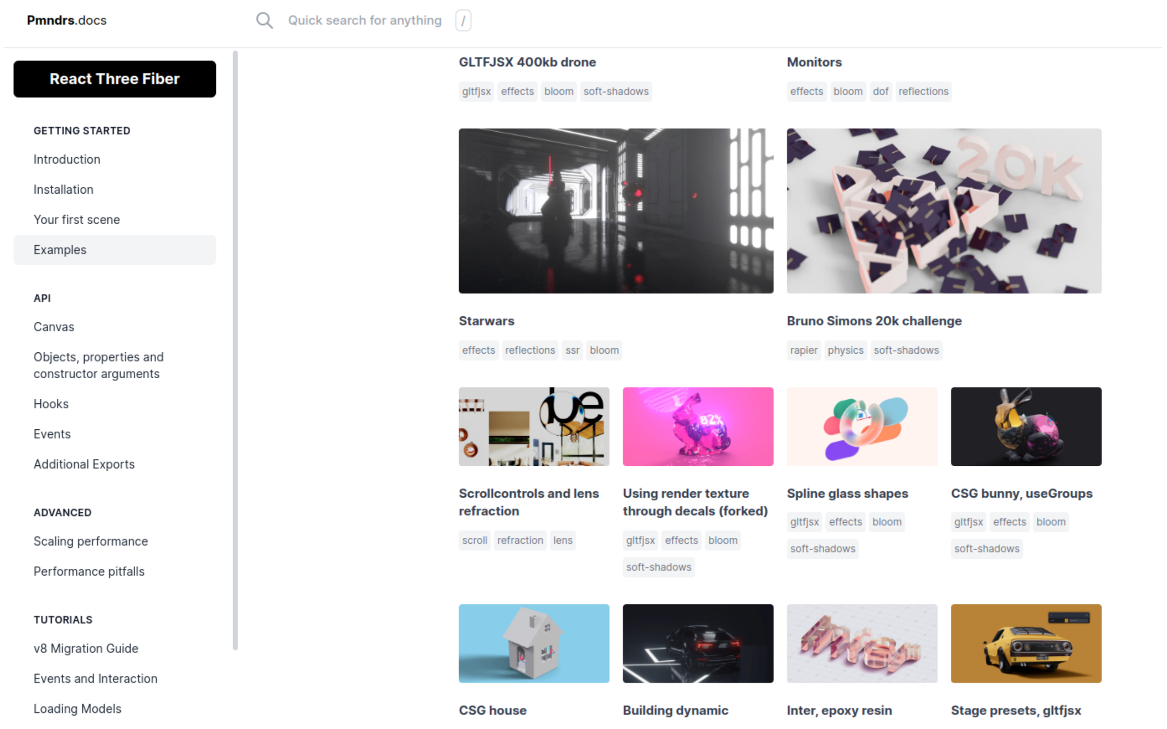


Figure x: Pmndrs documentation

Some of this libraries are[7]:

- **React PostProcessing**[8]: a postprocessing wrapper for react-three-fiber.
- **A11y**[9]: Accessibility to webGL with easy-to-use react-three-fiber components.
- **React Spring**[10]: Spring animation primitives for React, with direct SDK that match the *pmndrs* libraries.
- **Drei**[11]: A growing collection of useful helpers and abstractions for react-three-fiber.

As we will see further, R3F is used as a foundation for other utility libraries, and is a referent for many other 3D libraries.

[7] Full list on (*Poimandres Collective*, n.d.)
[8] (*React Postprocessing Documentation*, n.d.)
[9] (*A11y Documentation*, n.d.)
[10] (*React Spring*, n.d.)
[11] (*Useful Helpers for React-Three-Fiber*, n.d.)

2.3. Spline.js [12]

Spline.js is a free, real-time collaborative 3D design tool to create tridimensional models. It provides a flexible and easy-to-use API for defining and manipulating splines, allowing developers to create complex and dynamic animations with ease.

Spline.js is often used by developers and designers as a studio to construct animations and interactions that are later exported to known formats such as R3F. It is compatible with other popular animation libraries like Greensock and Three.js, making it a versatile and powerful tool for creating engaging and visually appealing web animations.

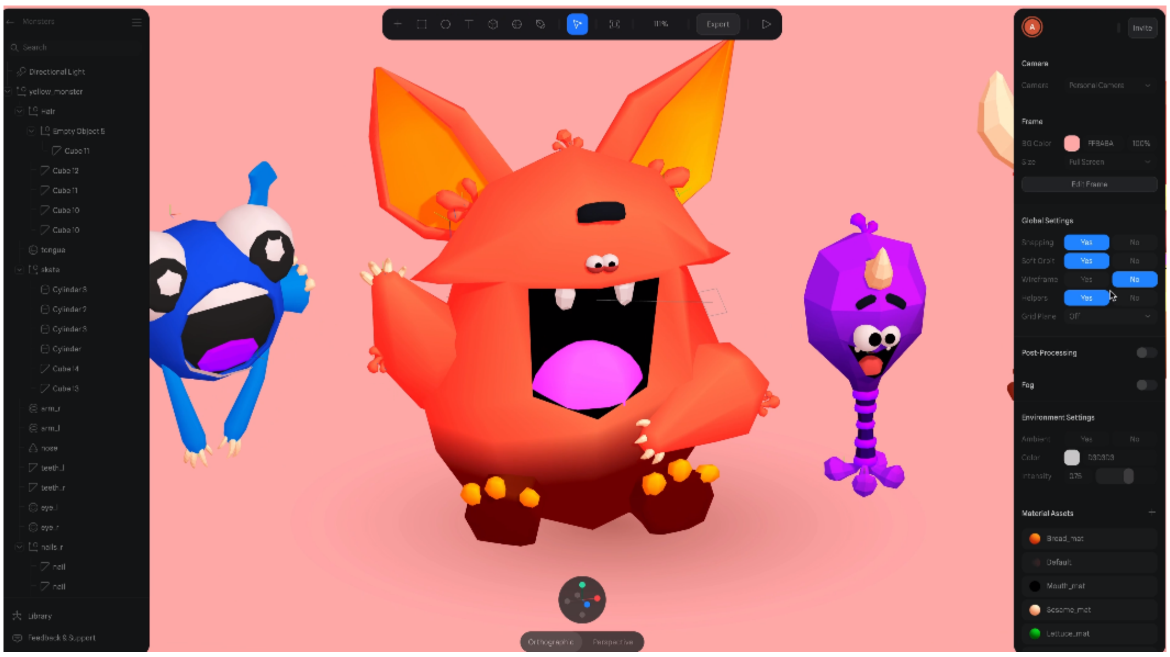


Figure x: Spline studio

[12] (*Spline*, n.d.)

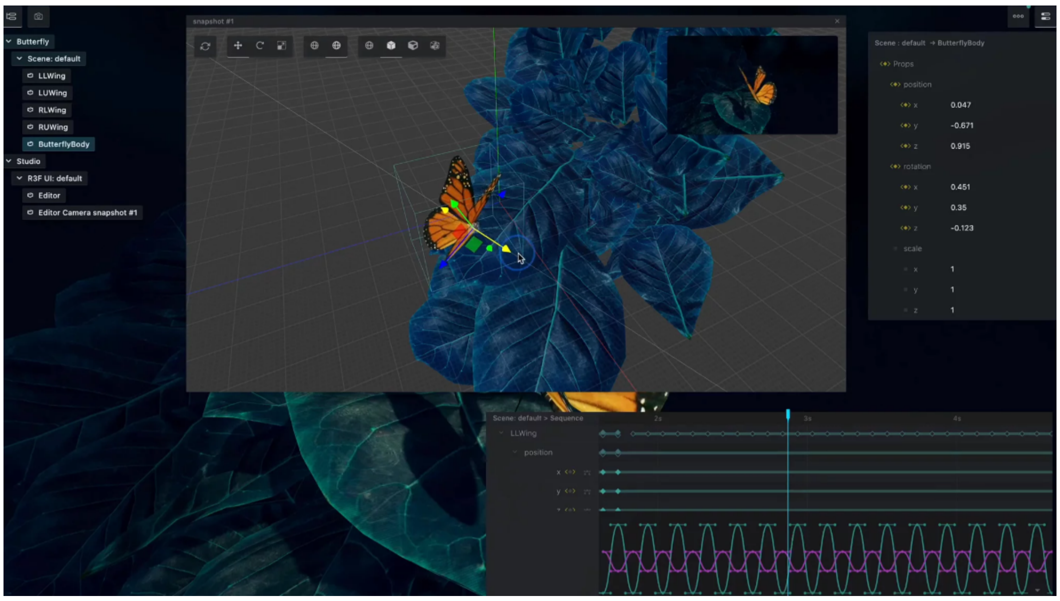
2.4. Theatre.js[13]

Theatre.js is a javascript animation library with a professional motion design toolset. It helps you create any animation, from cinematic scenes in three.js, to complex UI interactions.

It aims to simplify the creation of interactive 3D scenes by providing a higher-level, declarative API. Theatre.js allows developers to create and manage multiple 3D scenes, define animations using timeline-based keyframes, and easily add interactivity and user input handling.

It also includes a powerful layout engine for positioning and animating UI elements, making it an ideal choice for creating rich and engaging 3D user interfaces. Theatre.js is built on top of Three.js and provides a user-friendly interface for building complex 3D scenes, reducing the amount of code needed and speeding up the development process.

It has a direct SDK for three.js and react-three-fiber, and is often meant to be used as a [development dependency]() to test animations, interactions and scene props placement.



position x 0.047
y -0.671
z 0.915
rotation x 0.451
y 0.35
z -0.123
scale x 1
y 1
z 1

Figure 4: Theatre.js studio format example

[13] (*Threate*, n.d.)

2.5. Needle.tools[14]

Needle is a popular open-source testing framework for Node.js applications.

It provides a simple and easy-to-use API for writing automated tests, and offers a number of powerful features, including parallel test execution, test grouping and filtering, and code coverage analysis. One of the key components of Needle is its set of built-in tools, which include a test runner, an assertion library, and a mocking library.

These tools allow developers to write comprehensive, reliable tests that can ensure the correctness and robustness of their Node.js applications. With its intuitive interface and rich set of features, Needle has become a popular choice for Node.js developers looking to improve the quality and stability of their code.

Needle supports the use of popular 3D applications, such as Unity and Blender. It connects these technologies to be used as a 3D studio to test animations, interactions and scene props placement, similar to Theatre.js.

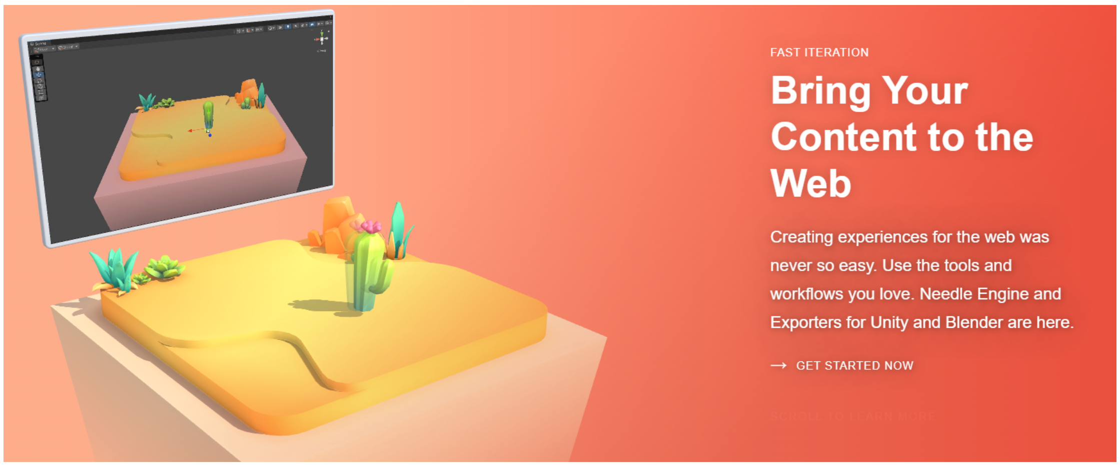

FAST ITERATION
Bring Your Content to the Web
Creating experiences for the web was never so easy. Use the tools and workflows you love. Needle Engine and Exporters for Unity and Blender are here.
→ GET STARTED NOW

Figure x: Needle.js showcase

[14] (*Needle Tools*, n.d.)

2.6. Outside of JavaScript

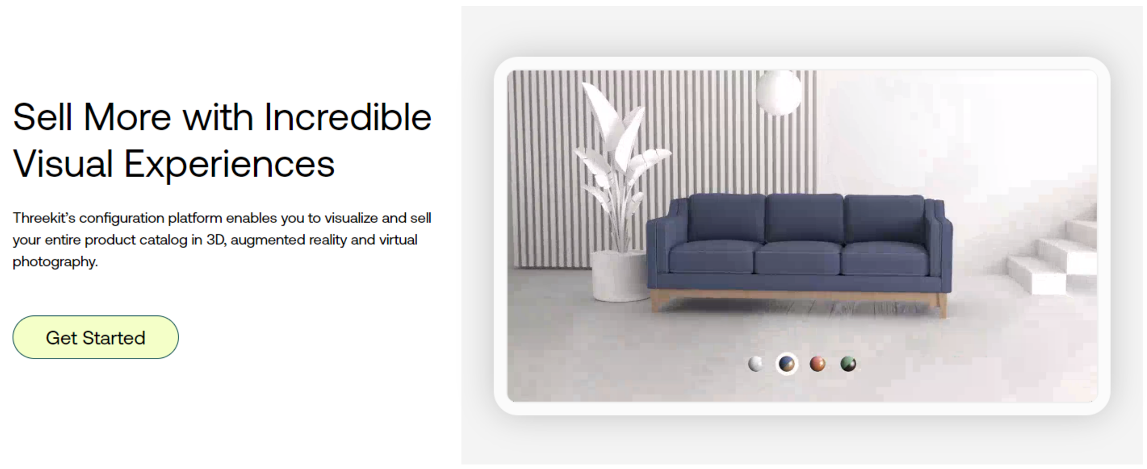
As mentioned before, there were some opt-ins in the tech stack chosen that set aside multiple other options available in the market. And although the opt-ins were previously justified, it wouldn't be fair not to mention other very viable choices with promising features.

2.6.1. Three kit[15]

The Threekit platform includes a variety of tools that make it easy for companies to create and manage 3D product visuals. For example, their configurator tool allows companies to create interactive product configurators that let customers customize product features and see the results in real-time. They also have a visualizer tool that enables companies to create high-quality product images and 360-degree spins.

Another key feature of the Threekit platform is its AR capabilities. Companies can use Threekit to create AR experiences that allow customers to view and interact with 3D product models in real-world environments using their mobile devices.

Unfortunately, Three.kit's software is proprietary and not available for the public, and although they provide a top-class solution to 3D visualization, their platform's technology is far from our reach.


Sell More with Incredible Visual Experiences

Threekit's configuration platform enables you to visualize and sell your entire product catalog in 3D, augmented reality and virtual photography.

Figure x: Three kit's webpage

[15] (*ThreeKit*, n.d.)

2.6.2. Threlte[16]

Threlte is a renderer and component library for Svelte, a popular JavaScript framework for building web applications. Threlte allows developers to use Svelte to create and render 3D graphics and animations using Three.js, a popular JavaScript library for creating 3D graphics.

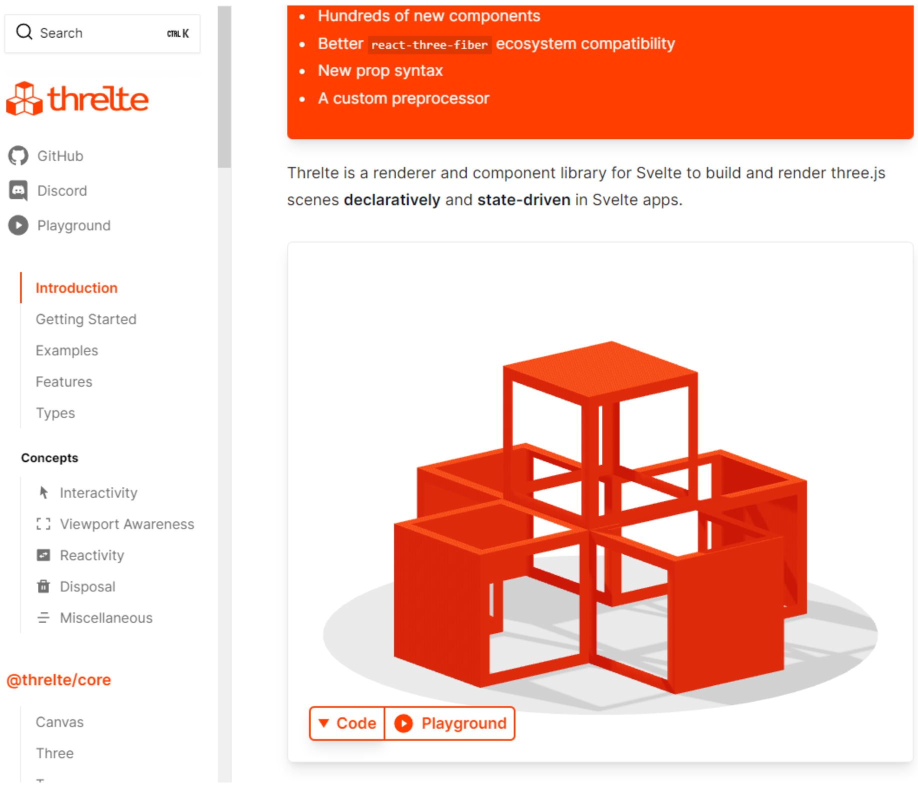
One similarity between threlte and React Three Fiber is that both libraries provide a declarative and reactive way of creating and manipulating 3D objects and scenes using Three.js. This is one of many cases where Three.js is the basis for JavaScript's 3D ecosystem and is mainly powered by Three.js.

Threlte allows developers to write Svelte components that create and render Three.js scenes, while React Three Fiber allows developers to write React components that create and manipulate Three.js objects and scenes.

Another similarity is that both libraries allow developers to manage the state of 3D objects and scenes using the framework's built-in state management tools. Threlte allows developers to use Svelte reactive stores and props to manage the state of Three.js objects, while React Three Fiber provides hooks and context objects for managing state in a React application.

In fact, threlte's philosophy is admittedly based on the sensible defaults of React-Three-Fiber, as the authors have publicly stated in the documentation. Overall, these two technologies are similar in their goals of making it easier for developers to create and manage 3D graphics and animations in web applications using popular JavaScript frameworks.

[16] (*Threlte*, n.d.)



- Hundreds of new components
- Better `react-three-fiber` ecosystem compatibility
- New prop syntax
- A custom preprocessor

Threlte is a renderer and component library for Svelte to build and render three.js scenes **declaratively** and **state-driven** in Svelte apps.

Figure x: Threlte's documentation

2.6.3. Trois[17]

Trois.js is a JavaScript library built on top of Three.js that provides a set of higher-level abstractions and utility functions for building complex 3D scenes and animations. It is designed to make it easier for developers to work with Three.js by providing a simpler and more intuitive API.

One of the key features of Trois.js is its collection of pre-made 3D objects and shapes. These objects can be easily added to a scene and customized with lighting and material properties. Trois.js also provides a number of customizable camera controls, including pan, zoom, and rotate, which can be used to give users an interactive view of a 3D scene.

Another feature of Trois.js is its particle system, which allows developers to create complex particle effects such as smoke, fire, and explosions. The library also provides integration with popular web frameworks such as Vue.js and React.

One of the key benefits of Trois.js is that it abstracts away some of the complexity of working with Three.js, making it easier for developers to

[17] (*TroisJS*, n.d.)

create complex 3D scenes and animations without having to write low-level Three.js code.

In summary, Trois.js is a higher-level library built on top of Three.js that provides a simplified API and collection of pre-made objects and shapes for creating complex 3D scenes and animations in web applications.

2.7. Market study

There is no shortage of examples in the market regarding web applications using 3D technology, and although some used technology far from our reach and not open-sourced, some of them are well acquainted with the tech stack chosen.

The following projects are prime examples of fully performant, professional applications using 3D technology at its best, and were used in this thesis as references, both technically and creatively.

2.7.1. Galleries

During the research phase, there were multiple galleries of reference where one could find recent, innovative applications that used 3D technology. They are not specific examples, but are a good sources of ideas for starting new projects and getting inspired.

Three.js Gallery[18]

The landscape of Three.js, which features a gallery of 3D projects created by the community, can be a valuable reference for anyone creating 3D web applications. The gallery showcases a wide range of projects, from games and virtual reality experiences to product configurators and architectural visualizations.

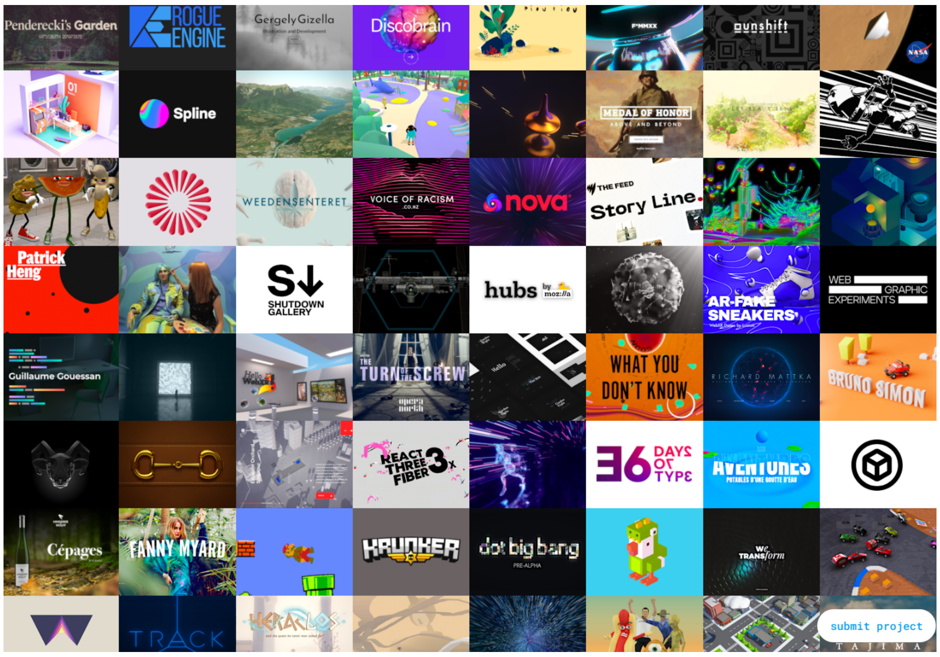
One of the key benefits of using the Three.js gallery as a reference is that it offers a glimpse into the latest trends and innovations in 3D web design. The gallery is constantly updated with new projects created by the community, meaning that developers and designers can stay up-to-date with the latest developments in the field. Additionally, the featured projects are selected based on their quality and creativity, so the gallery offers a curated selection of the best 3D web applications out there.

Another benefit of using the Three.js gallery as a reference is that it provides real-world examples of how Three.js can be used to create different types of 3D web applications. By studying the projects in the gallery, developers can gain a better understanding of how to use the

[18] (*Three.js*, n.d.)

library to create specific types of 3D experiences. They can also get inspiration and ideas for their own projects.

Furthermore, the Three.js gallery is a community-driven project, meaning that it features apps created by developers and designers from all around the world. This can be especially helpful for those looking for different design styles, cultural influences, and regional trends in 3D web design.



Three.js Docs[19]

The Three.js Examples documentation is a collection of practical examples that demonstrate how to use the library to create different types of 3D web applications. Each example is accompanied by code snippets and explanations, making it easy for developers to understand how the code works and adapt it to their own projects.

The Three.js examples documentation can be an incredibly useful open-sourced reference for creating 3D web applications because it offers a wide range of examples that cover various aspects of 3D graphics and animation. Whether you want to create a 3D product configurator, an interactive game, or a virtual reality experience, you can find examples that showcase different techniques and approaches.

[19] (*Three.js Docs*, n.d.)

Furthermore, the examples in the Three.js documentation are created with vanilla three.js, meaning that they use only the core library without any additional plugins or frameworks. This can be especially helpful for developers who are learning Three.js or want to understand the underlying principles of 3D graphics programming. By studying the code and techniques used in the examples, developers can gain an understanding of how to apply certain 3D concepts using JavaScript.

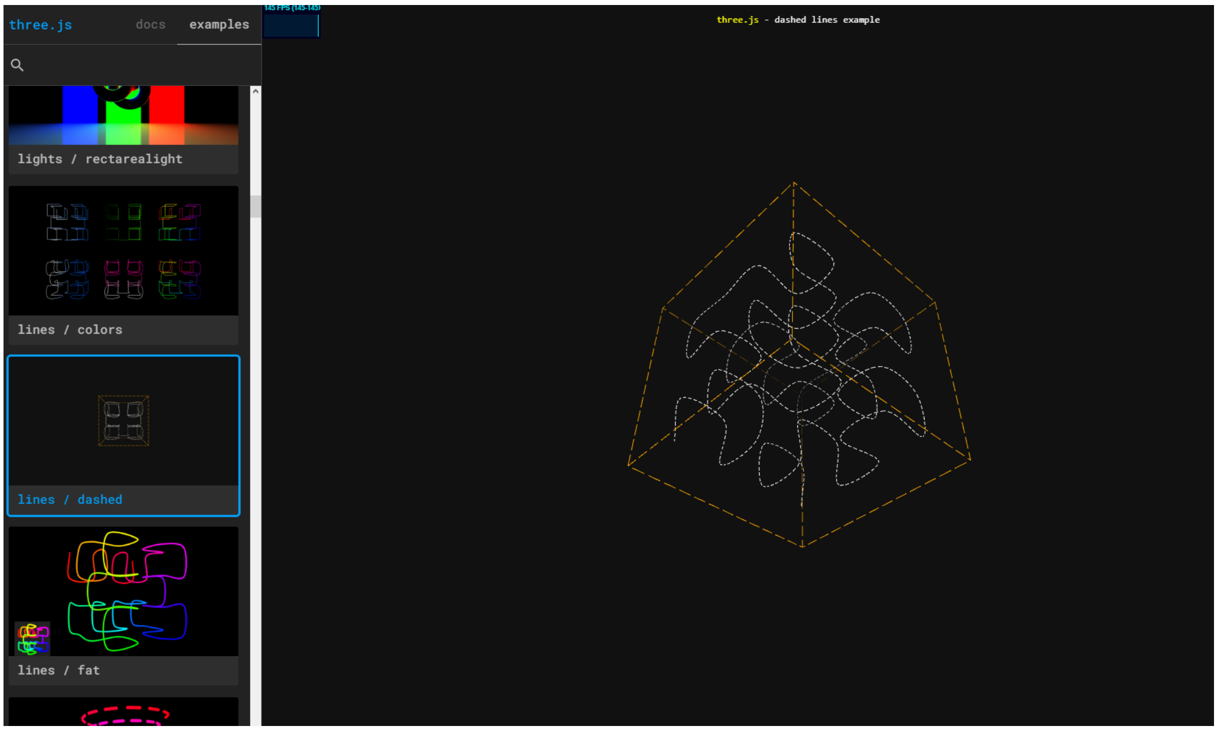


Figure x: Three.js documentation example's section

Awwwards [20]

Awwwards is a website that showcases exceptional design work in the fields of web design, UX/UI design, and interactive design.

It is a platform that recognizes and rewards the best websitesfrom around the world. As a reference for creating 3D web applications, Awwwards.com can be an incredibly useful resource as it features a variety of outstanding examples of 3D web applications that are not only visually stunning but also functional and user-friendly.

[20] (*Awwards - Website Awards*, n.d.)

By exploring and analyzing the different approaches taken by the award-winning 3D web applications featured on Awwwards.com, developers and designers can gain insights into best practices, innovative techniques, and creative solutions that can be adapted and implemented in their own projects.

2.7.2. Specific examples

CineShader[21]

Cineshader is an online platform that provides a library of high-quality cinematic shaders for use in 3D graphics and animation projects. Shaders are computer programs that define the appearance of 3D objects in a scene, controlling how light interacts with the object's surface and creating the final visual output.

Cineshader offers a range of different shaders, each designed to achieve a specific look or effect. For example, there are shaders for creating realistic skin, hair, and fur, as well as shaders for creating stylized or cartoon-like visuals. All of the shaders are created by professional artists and designers, and are optimized for use in a variety of 3D software applications.

One of the main benefits of using Cineshader is that it can save time and effort for 3D artists and designers. Rather than having to create shaders from scratch for every project, they can simply download pre-built shaders from the library and customize them as needed. This can be especially helpful for freelancers or small teams who may not have the resources to create their own shaders from scratch.

Additionally, Cineshader offers a range of resources and tutorials to help users get the most out of the platform. There are video tutorials that walk users through the process of applying shaders to 3D objects, as well as articles and guides on best practices for shader development and optimization.

[21] (*Cine Shader*, n.d.)

Domain Warping
A domain distortion noise technique invented by Inigo Quilez

Figure x: CineShader's display of shaders

Ringba[22]

Ringba is a call tracking and analytics platform that helps businesses track and analyze their phone calls to better understand their marketing campaigns and customer behavior

Ringba.com's use of 3D graphics on their website can serve as a valuable reference for those creating 3D web applications. By examining the design and implementation of Ringba.com's 3D graphics, developers and designers can gain insight into how to effectively use 3D graphics to enhance visual design, improve user experience, and optimize performance.

Although their 3D application does not have interactive functionalities, it provides an insight on how 3D graphics can directly affect the user experience and inferred quality of the product.

[22] (*Inbound Call Tracking for Pay Per Call*, n.d.)

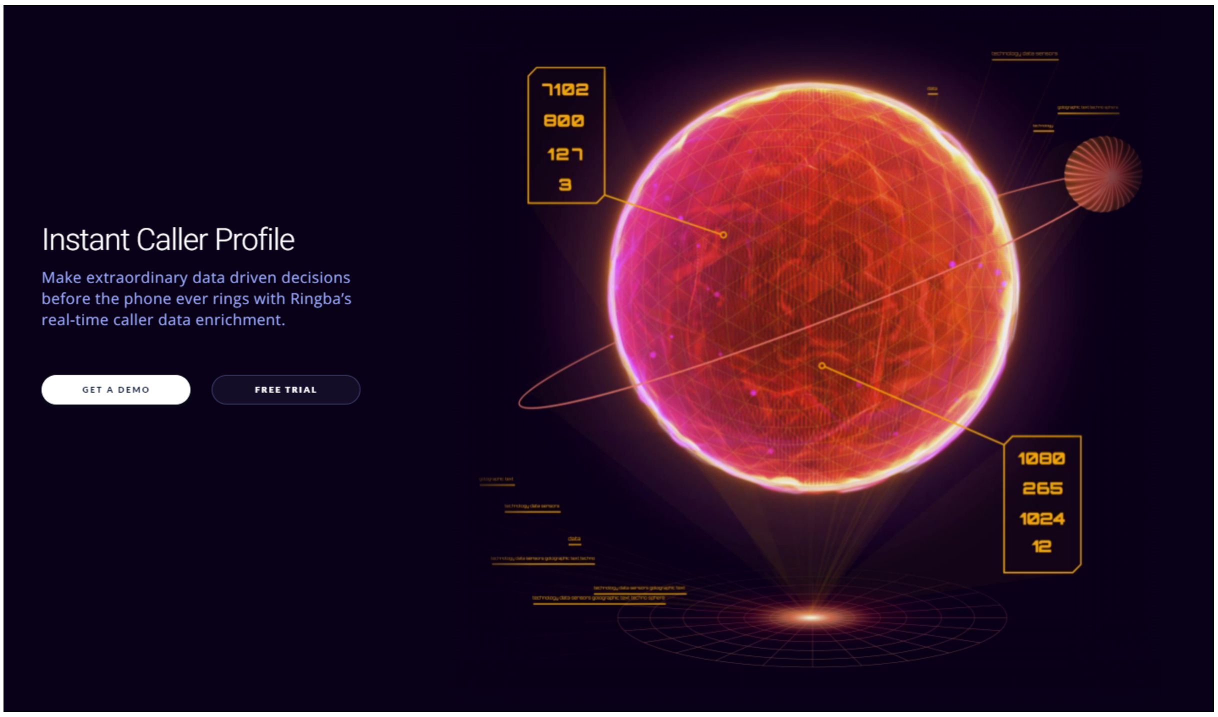

Instant Caller Profile
Make extraordinary data driven decisions before the phone ever rings with Ringba's real-time caller data enrichment.
GET A DEMO
FREE TRIAL
7102
800
127
3
1080
265
1024
12

Figure x: Ringba's use of 3D on their business pitch

Github[23]

GitHub is a web-based platform for version control and collaboration that allows users to store and manage their code and related projects. It provides tools for code review, project management, and continuous integration and deployment. It is considered the biggest platform for code storage, and it is used by millions of developers all around the world.

Github used to have a 3D component in their landing page that aided their explanation on what their market value was, and it proves that 3D applications are used frequently by enterprises and high quality digital agencies.

[23](*How We Built the GitHub Globe*, 2020)

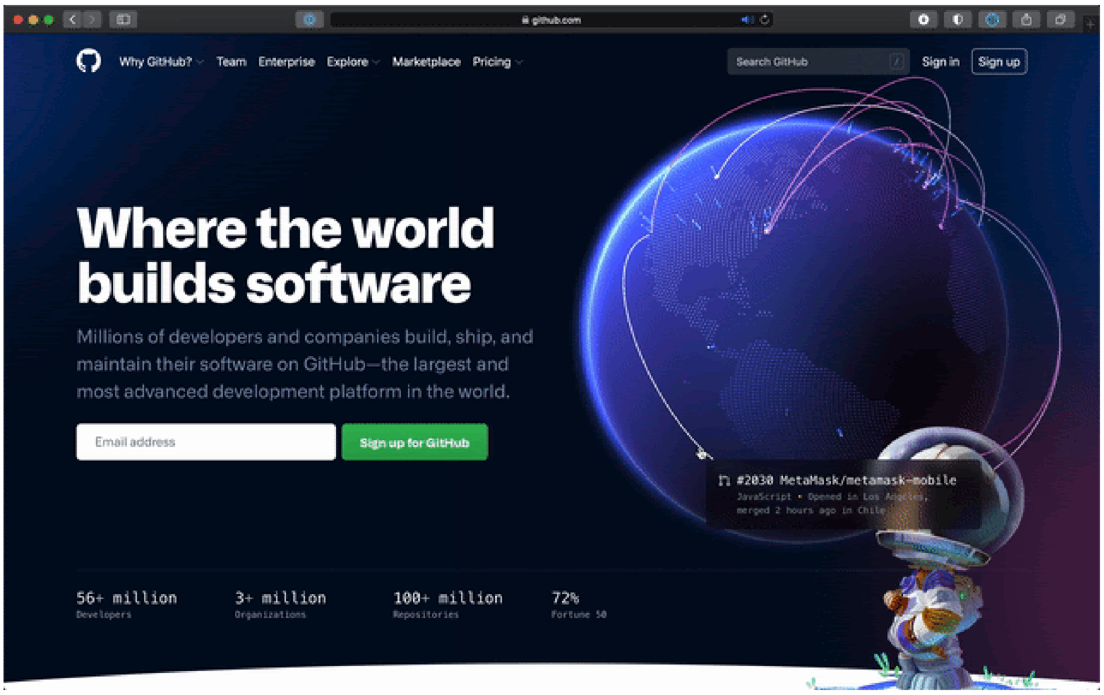

Why GitHub? Team Enterprise Explore Marketplace Pricing
Search GitHub
Sign in
Sign up
Where the world builds software
Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world.
Email address
Sign up for GitHub
#2030 MetaMask/metamask-mobile
56+ million Developers
3+ million Organizations
100+ million Repositories
72% Fortune 50

Figure x: Github's landing page and an interactive globe

3dfy[24]

3dfy is a software company that specializes in converting 2D content into 3D models. Their proprietary technology uses computer vision and machine learning algorithms to automatically create 3D models from 2D images and videos, allowing users to easily create 3D content for use in virtual reality, augmented reality, gaming, and other applications. The company's products include an SDK for developers and a cloud-based service for non-technical users.

The 3D particles on the 3dfy.ai website are a visual representation of the company's core technology, which uses computer vision and machine learning algorithms to convert 2D content into 3D models. The particles move and rotate in a way that simulates the process of analyzing a 2D image or video and extracting depth and spatial information to create a 3D model. This process involves complex calculations and pattern recognition, and the 3D particles help to illustrate how 3dfy's technology works in a simple and intuitive way. Overall, the 3D particles are a creative and engaging way to showcase 3dfy's innovative approach to 3D modeling.

[24] (*3DFY*, n.d.)

Everything 3D, Now.
3·d·fy /'θri:di:faɪ', three-dee-feye/ verb: AI-powered 3D model generation at scale
-3·d·fied adj. -3·d·fier n.

Figure x: 3dfy's landing page, 3D particles

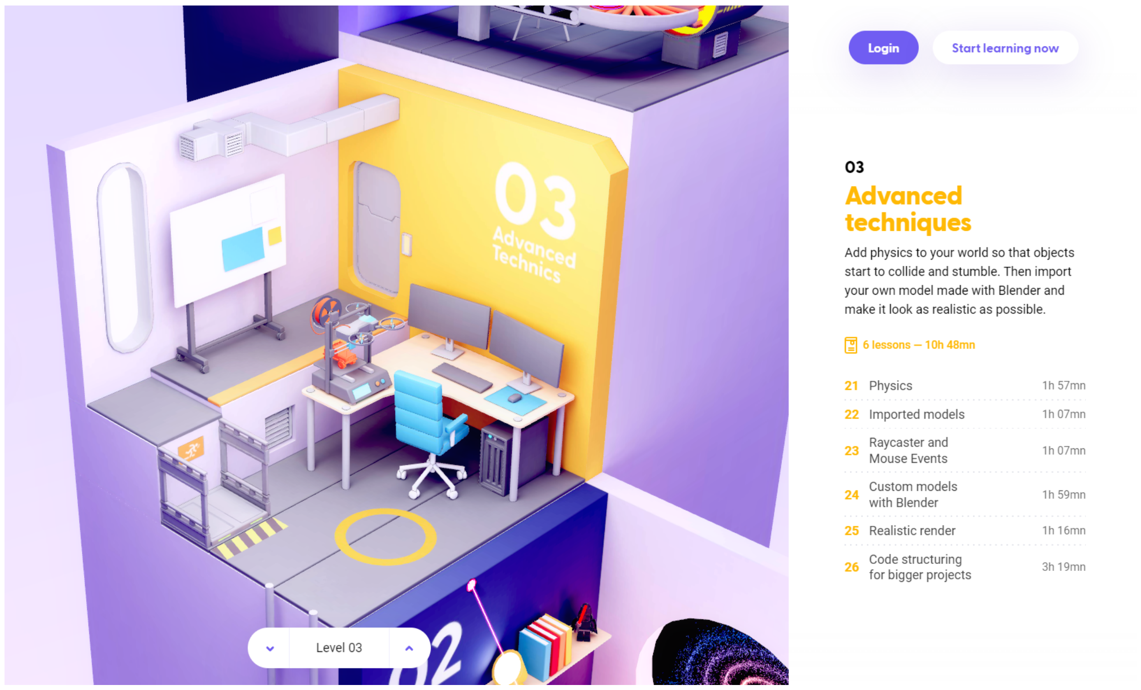
Three journey[25]

Three.js Journey is an online course that teaches web developers how to create 3D graphics and animations using the Three.js library. The course is aimed at both beginners and experienced developers who want to add interactive 3D content to their websites or create 3D games.

The landing page of Three.js Journey features a dynamic and immersive 3D scene that showcases the capabilities of the Three.js library for creating 3D graphics and animations on the web. The scene includes several key components, including a textured and illuminated terrain, a 3D model of a robot, and a particle system that generates small floating orbs.

The landing page itself is a great source of inspiration, as it provides a great example of the capability of Three.js to create amazing digital experiences for the web.

[25] (*Three.js Journey*, n.d.)



03

Advanced techniques

Add physics to your world so that objects start to collide and stumble. Then import your own model made with Blender and make it look as realistic as possible.

6 lessons — 10h 48mn

21	Physics	1h 57mn
22	Imported models	1h 07mn
23	Raycaster and Mouse Events	1h 07mn
24	Custom models with Blender	1h 59mn
25	Realistic render	1h 16mn
26	Code structuring for bigger projects	3h 19mn

Figure x: Three journey's landscape

2.8. Mono Repositories

As the project was being thought of more and more, careful consideration was given to the utilization of monorepos as a viable solution for managing the project's codebase. Monorepos offer advantages such as enhanced organization of multiple related applications or components within a single repository, facilitating code sharing, dependency management, and overall project structure. It seemed a perfect fit for a set of applications with shared dependencies.

An intriguing monorepo option, TurboRepo[26] developed by Vercel, captured attention during the evaluation process. However, due to its beta status, concerns were raised regarding the stability and reliability of TurboRepo, particularly when dealing with a critical research project like the one at hand.

Throughout the exploration of alternative options, two prominent monorepo tools, namely Nx[27] and Lerna[28], were considered. These well-established tools provide advanced features for effectively managing large-scale projects, including streamlined build processes, comprehensive dependency tracking, and robust testing frameworks. Nevertheless, considering the primary focus of this thesis, which lies in exploring and

[26] (Turborepo n.d.)
[27] (Nx n.d)
[28] (Lerna n.d)

experimenting, it became apparent that these monorepo tools were excessively sophisticated and primarily tailored towards enterprise-level development scenarios.

Taking into account the research-oriented nature of the thesis and the specific objectives to be accomplished, the decision was made to forgo the implementation of monorepos. This choice was motivated by several factors:

Firstly, the experimental nature of the web technologies under investigation necessitated a flexible project structure and diverse deployment options. The utilization of a monorepo structure would have introduced constraints that could potentially limit the exploration of various configurations and optimization techniques.

Secondly, while monorepos excel in managing expansive projects comprising multiple interconnected applications or components, the scope of this thesis primarily involved the creation of an interactive application gallery. By adopting a simpler repository structure, the focus could be maintained on the immersive experiences themselves, as well as the evaluation of user experience and performance optimization. The inherent complexities introduced by monorepos would have diverted attention from the core objectives of the thesis.

This claim is backed up by the deprecated attempt to implement turborepo for all the projects, which you can find reference to in the e-graphy and annexes[29].

Lastly, the time constraints associated with the project factored into the decision-making process. The learning curve entailed in implementing and configuring a monorepo tool would have necessitated a substantial investment of time and effort. Given the thesis's emphasis on exploring cutting-edge web technologies, prioritizing the development and evaluation of immersive experiences over establishing and maintaining a monorepo infrastructure was imperative.

In conclusion, while acknowledging the undeniable benefits of monorepos in specific project scenarios, their implementation would have introduced unnecessary complexities and potential limitations for the research objectives at hand. By opting for a more straightforward project structure, the focus remained on the core goals of the thesis, facilitating a more efficient exploration and research.

[29] Monorepo (2020,May 20)

3. Project management

Project management is a critical aspect of any organization or business, with the primary objective of achieving project goals while meeting scope, time, quality, and budget requirements. In today's fast-paced and ever-changing business world, the effective management of projects is essential for the success of organizations. Project management tools play a vital role in facilitating the process of project management by improving project planning, organization, and communication.

The following is a list of all the tools that were used in this thesis for project management.

Github[30]

GitHub is a powerful web-based platform for version control and collaboration. It is particularly useful for software development, as it is commonly used to store source code as is the case of this thesis. By using GitHub's version control, collaboration, issue tracking, documentation, and hosting features, it was much easier to keep track of the work, collaborate with others, and share the results with the world.

Google Drive[31]

Google Drive is a cloud-based storage and collaboration platform that can be used as a management tool for a variety of projects, including research and academic work. By storing files and documents on Google Drive, one can easily organize, access, and collaborate on their work from anywhere with an internet connection.

This was particularly useful to share the writing part of the thesis done, review it and reiterate on corrections done by this thesis's mentor.

3.1. SWOT

This thesis aims to explore the potential of cutting-edge web

[30] (*Github*, n.d.)
[31] (*Personal Cloud Storage & File Sharing Platform*, n.d.)

technologies for creating immersive 3D experiences in the browser. As we assess the potential of these technologies, it is important to take into account their strengths, weaknesses, opportunities, and threats.

The strengths of these technologies include their potential for engaging and immersive user experiences and their ability to provide a new level of interactivity for web applications. However, these technologies also have weaknesses, such as the need for a high level of technical expertise to develop and limited support for older browsers or devices. Opportunities for 3D web graphics and interactive applications are growing, and this field holds promise for new revenue streams and innovative user experiences.

However, there are also potential threats, such as competition from other emerging web technologies, security concerns, and changes to web standards or browser capabilities. This SWOT analysis provides a framework for understanding the potential of cutting-edge web technologies for creating immersive 3D experiences in the browser.

Strengths	Weaknesses
High potential for engaging and immersive user experiences	Requires a high level of technical expertise to develop
Can provide a new level of interactivity for web applications	Limited support for older browsers or devices
Growing demand for 3D content and interactivity on the web	Potential for slower load times and decreased performance
Enables the creation of highly customized and personalized experiences	Limited availability of skilled developers with expertise in 3D web technologies

Opportunities	Threats
Increasing demand for immersive 3D experiences on the web	Potential competition from other emerging web technologies
Growing market for web-based 3D graphics and interactive applications	Security concerns and potential vulnerabilities associated with web-based 3D experiences
Potential for new revenue streams through the development of 3D web applications	Limited market adoption of 3D web technologies
The ability to create new and innovative web-based experiences that stand out from competitors	Changes to web standards or browser capabilities that could affect the viability of 3D web technologies

3.2. Risks and contingencies plan

It is vitally important to detect the risks that could endanger the work and look for solutions so that, if necessary, the project can be resumed.

The possible risks identified for this project, and their corresponding solutions, are as follows, ordered from least to most important:

Risk	**Contingency plan**
Lack of Access to Necessary Technology	Research alternative technologies and tools, or collaborate with other individuals or institutions who have access to the necessary resources.
Technical Issues during Development	Create a comprehensive testing plan and budget extra time for troubleshooting and debugging.
Difficulty in Achieving Desired User Experience	Conduct user testing and seek feedback from experts in the field to identify areas for improvement and refine the design.

Insufficient Time to Complete Project	Prioritize tasks and create a realistic timeline, and be willing to adjust goals and expectations if necessary.
Unexpected Delays or Setbacks	Stay flexible and adaptable, and have contingency plans in place for potential disruptions to the project schedule.
Difficulty in Optimizing Performance	Continuously monitor and test the performance of the applications, and make adjustments as needed to ensure optimal speed and responsiveness.

3.3. Initial cost analysis

The costs of this project are fortunately low budget. As it has no intention to be selled in the market, but rather to investigate over a certain topic, the only costs obtained fall down to academic and utility expenses.

Academic expenses	**Cost**
Three.js Journey course	95$
Utility expenses	**Cost**
Laptop hardware	1000$
8 Months of internet and electricity	800$

4. Methodology

For this thesis, I employed a combination of the iterative design process and the agile methodology to ensure that the final product met the goals that the project was set out to achieve at the beginning.

I began by creating prototypes of the software, which I then tested and gathered feedback on from peer designers and developers, as well as my mentor. Based on their feedback, I made changes and created a new prototype, repeating the process until I had a satisfactory solution.

There were times where certain projects and ideas were promising at first, but resulted in a not so well desired result. These ideas were discontinued and reformed in new ways for other future prototypes, making the creative process much easier.

Throughout the development process, I also utilized the agile methodology, working in short sprints to build small, working pieces of software. After each sprint, I reviewed the work, gathered feedback and using it to refine and improved the product in subsequent sprints.

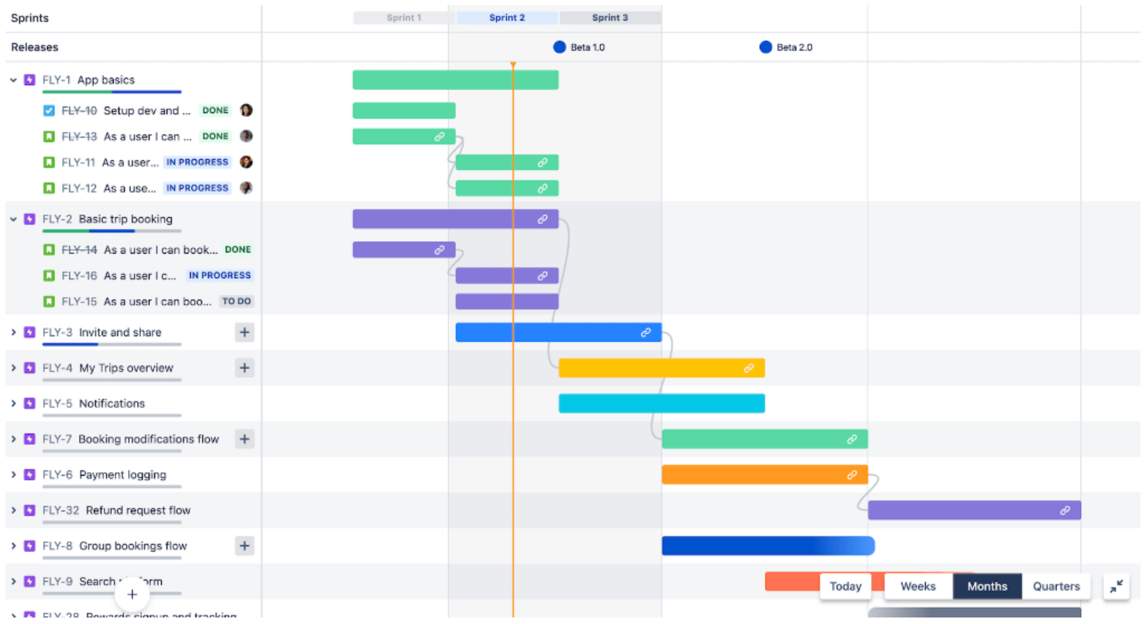

Sprints
Sprint 1
Sprint 2
Sprint 3
Releases
Beta 1.0
Beta 2.0
FLY-1 App basics
~~FLY-10~~ Setup dev and ... DONE
~~FLY-13~~ As a user I can ... DONE
FLY-11 As a user... IN PROGRESS
FLY-12 As a use... IN PROGRESS
FLY-2 Basic trip booking
~~FLY-14~~ As a user I can book... DONE
FLY-16 As a user I c... IN PROGRESS
FLY-15 As a user I can boo... TO DO
FLY-3 Invite and share
FLY-4 My Trips overview
FLY-5 Notifications
FLY-7 Booking modifications flow
FLY-6 Payment logging
FLY-32 Refund request flow
FLY-8 Group bookings flow
FLY-9 Search
Today
Weeks
Months
Quarters

Figure x: An example of Agile methodology on a diagram

Overall, the use of the iterative design process and the agile methodology were critical in the development of my thesis software. By emphasizing flexibility, collaboration, and communication, I was able to deliver a high-quality product that I set out to achieve.

5. Project development

As the nature of this thesis is interactive, it is best explained with interactive documentation. Please refer to the live web for further reading. Additionally, the web version will be included in the annex, without the interactivity.

https://tfg-docs.vercel.app/

6. Conclusions

In conclusion, this thesis has demonstrated the vast potential of cutting-edge web technologies in facilitating the creation of immersive 3D experiences within the browser. By leveraging the power of libraries such as those developed by the pmdnrs collective, the thesis has showcased the practicality and effectiveness of the declarative approach offered by React in enabling the development of sophisticated 3D applications.

The adoption of a declarative coding paradigm, as exemplified by React, has proven to be highly advantageous in the context of 3D web development. By allowing developers to define the desired outcome or state of an application without explicitly specifying the low-level implementation details, declarative code fosters a more intuitive and maintainable development process. This approach enables developers to focus on the higher-level logic and creative aspects of 3D experiences, rather than getting entangled in the intricacies of the GLSL canvas manipulation or imperative code.

Moreover, the strong support and vibrant community surrounding the development of 3D libraries within the React ecosystem have been instrumental in driving advancements in web-based 3D graphics and interactivity. The contributions and collaborative efforts of developers within the community have resulted in the creation of powerful and feature-rich libraries, which simplify the implementation of complex 3D functionalities. These libraries, developed by the pmdnrs collective and other contributors, provide a solid foundation for developers to build upon, accelerating the development process and enhancing the quality of 3D applications.

In conclusion, this thesis has underscored the transformative potential of cutting-edge web technologies, particularly within the React ecosystem, for crafting immersive 3D experiences. The declarative coding paradigm, coupled with the support of robust libraries like those developed by the pmdnrs collective, presents a practical and efficient approach for developers to create remarkable 3D applications. As the developer community continues to expand and collaborate, the future holds exciting possibilities for further advancements in web-based 3D graphics and interactivity.

7.1. Future expansion

The future of 3D graphics and web development is promising, with the recent stable release of WebGPU[32] poised to bring about a transformative shift in the field. This significant development directly relates to how web components will be developed in the future.

With the stable release of WebGPU, a new era of web-based 3D graphics is on the horizon. This low-level graphics and compute API empowers developers to fully leverage the capabilities of modern GPUs, resulting in visually stunning and high-performance 3D graphics within web applications. The arrival of WebGPU opens up exciting possibilities for this thesis research, as it provides a powerful tool for creating interactive applications that can deliver immersive and engaging user experiences.

The stable release of WebGPU also signifies a big shift in the web development landscape. With its standardized API and cross-platform compatibility, WebGPU is set to be widely adopted across major browsers, offering a consistent and efficient way to interact with GPUs. This shift creates a more unified and accessible ecosystem for web-based 3D graphics, enabling developers to reach a broader audience and deliver seamless experiences across different devices and platforms.

To conclude, by embracing this shift towards WebGPU, this thesis could be expanded to use these new technologies, by showcasing the advancements in web development and the potential it holds for the future of 3D graphics.

[32] (*Chrome Ships WebGPU*, 2023)

8. E-graphy

(n.d.). Turborepo. Retrieved June 29, 2023, from https://turbo.build/

(n.d.). Nx: Smart, Fast and Extensible Build System. Retrieved June 29, 2023, from https://nx.dev/

(n.d.). Lerna: Documentation. Retrieved June 29, 2023, from https://lerna.js.org

A11y Documentation. (n.d.). Pmndrs.docs. Retrieved March 24, 2023, from https://docs.pmnd.rs/a11y/introduction

Awwards - Website Awards. (n.d.). Awwwards - Website Awards - Best Web Design Trends. Retrieved March 24, 2023, from https://www.awwwards.com

Chrome ships WebGPU. (2023, April 6). Chrome Developers. Retrieved June 29, 2023, from https://developer.chrome.com/blog/webgpu-release/

Cine Shader. (n.d.). CineShader. Retrieved March 24, 2023, from https://cineshader.com

Github. (n.d.). GitHub: Let's build from here · GitHub. Retrieved March 24, 2023, from https://github.com

How we built the GitHub globe. (2020, December 21). The GitHub Blog. Retrieved March 24, 2023, from https://github.blog/2020-12-21-how-we-built-the-github-globe/

Inbound Call Tracking for Pay Per Call. (n.d.). Ringba. Retrieved March 24, 2023, from https://www.ringba.com/caller-profile/

Monorepo. (2020, May 20). Retrieved June 29, 2023, from https://github.com/randreu28/TFG.monorepo

Needle Tools. (n.d.). Needle Tools. Retrieved March 24, 2023, from https://needle.tools

Personal Cloud Storage & File Sharing Platform. (n.d.). Google. Retrieved March 24, 2023, from https://www.google.com/drive/

Poimandres collective. (n.d.). Poimandres. Retrieved March 24, 2023, from https://pmnd.rs/

Poinmanders list of libraries. (n.d.). pmnd.rs docs. Retrieved March 24, 2023, from https://docs.pmnd.rs/

React Postprocessing Documentation. (n.d.). Pmndrs.docs. Retrieved March 24, 2023, from https://docs.pmnd.rs/react-postprocessing/introduction

React Spring. (n.d.). react-spring. Retrieved March 24, 2023, from https://www.react-spring.dev

React Three Fiber. (n.d.). Pmndrs.docs. Retrieved March 24, 2023, from https://docs.pmnd.rs/react-three-fiber/getting-started/introduction

Shopify. (n.d.). Start and grow your e-commerce business - 3-Day Free Trial. Retrieved March 24, 2023, from https://www.shopify.com

Spline. (n.d.). Spline - Design tool for 3D web browser experiences. Retrieved March 24, 2023, from https://spline.design

Stack Overflow. (n.d.). *Stack Overflow Developer Survey 2022*. Stack Overflow Annual Developer Survey. Retrieved March 12, 2023, from https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-language-love-dread

Threate. (n.d.). Theatre.js - animation toolbox for the web. Retrieved March 24, 2023, from https://www.theatrejs.com

3DFY. (n.d.). 3DFY.ai. Retrieved March 24, 2023, from https://3dfy.ai

Three.js. (n.d.). Three.js – JavaScript 3D Library. Retrieved March 24, 2023, from https://threejs.org

Three.js docs. (n.d.). Three.js. Retrieved March 24, 2023, from https://threejs.org/docs/

Three.js Journey. (n.d.). Three.js Journey — Learn WebGL with Three.js. Retrieved March 24, 2023, from https://threejs-journey.com

ThreeKit. (n.d.). Threekit: 3D Product Configurator & Augmented Reality For Commerce. Retrieved March 24, 2023, from https://www.threekit.com

Threlte. (n.d.). Threlte: Introduction. Retrieved March 24, 2023, from https://threlte.xyz

The top programming languages | The State of the Octoverse. (n.d.). GitHub Octoverse. Retrieved March 24, 2023, from https://octoverse.github.com/2022/top-programming-languages

TroisJS. (n.d.). TroisJS: Home. Retrieved March 24, 2023, from https://troisjs.github.io

Useful helpers for react-three-fiber. (n.d.). GitHub. Retrieved March 24, 2023, from https://github.com/pmndrs/drei#readme

9. Annex

1. An introduction

The purpose of this documentation is to present the findings and outcomes of the author's bachelor thesis. The thesis focuses on the development and implementation of a gallery comprising interactive applications with 3D components. This chapter provides an introduction to the thesis and outlines the structure of the documentation, as well as some key concepts that will later be of utility for the rest of the documentation.

1.1 Scope

This documentation assumes a basic understanding of React.js and TypeScript, while no prior knowledge of 3D engines is required.

1.2 Technology stack

It is essential to understand the layers of abstraction that exist between the code and the visual output on the screen. The following technologies are utilized:

1.2.1 WebGL

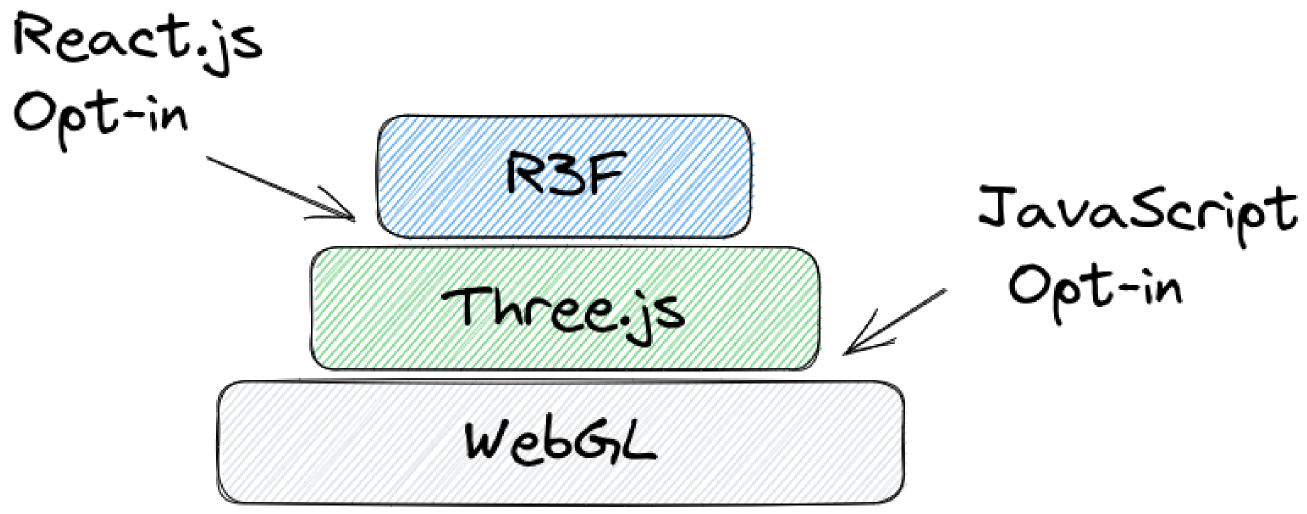
WebGL serves as the primary framework for rendering 3D graphics in web browsers. It offers low-level control over the rendering process and is encapsulated within the HTML `<canvas>` element.

1.2.2 Three.js

Three.js is a JavaScript library that provides a higher level of abstraction for creating 3D graphics. It simplifies the process of generating geometric shapes, such as cubes and spheres, by utilizing JavaScript classes.

1.2.3 React Three Fiber (R3F)

React Three Fiber is a library that further abstracts the development process by enabling the creation of React components for modern and interactive 3D elements. It combines the declarative nature of React with the imperative approach of Three.js.



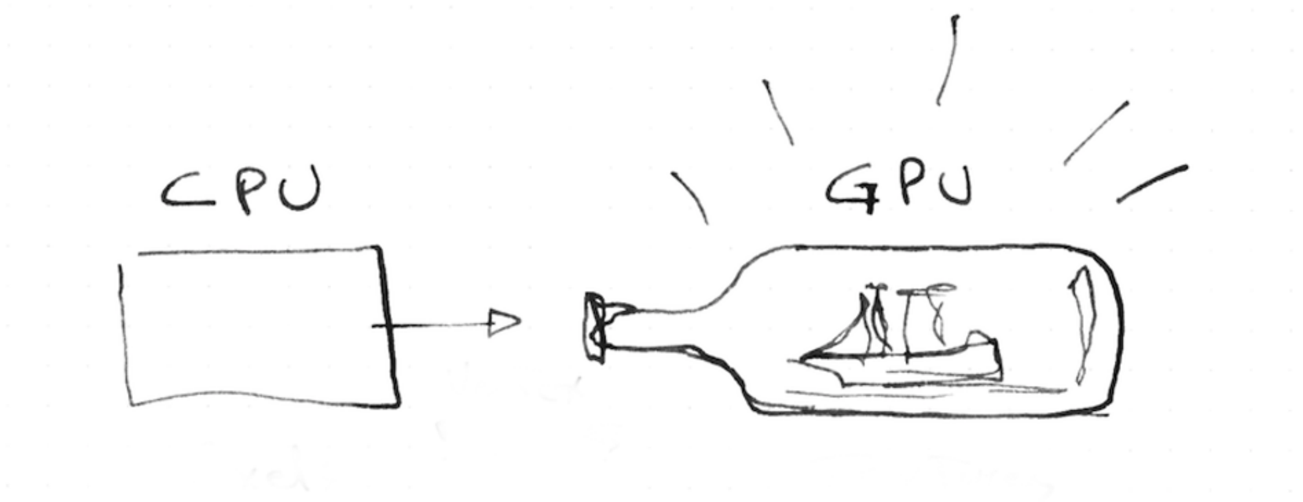
> *It is worth mentioning that the current era of 3D visualization is changing. In this docuemntation's case, it was chosen to work with the currently established WebGL 3D render motor, which has been the standard for the web for many years now.*
>
> *But do note that the upcoming WebGPU technology might replace it in the incoming years.*

1.3 An Overview of Shaders

This chapter provides an overview of shaders, which play a significant role in several projects within the gallery. Shaders enable low-level control over the rendering of specific objects by interacting directly with the WebGL GLSL compiler.

Shaders are primarily used for tasks that require precise control over the rendering process. As expressed by Patricio Gonzalez in his book "The Book of Shaders“:

> *"In shader-land we don't have too many resources for debugging besides assigning strong colors to variables and trying to make sense of them. You will discover that sometimes coding in GLSL is very similar to putting ships inside bottles. Is equally hard, beautiful and gratifying."*



1.4 React 3D Components

Fortunately, not all aspects of the project necessitate handling in GLSL. React Three Fiber provides significant assistance to developers in simplifying the interactivity and rendering process. To illustrate this, let's examine an exemplary code snippet from React Three Fiber. If you are familiar with React, you will find this code structure familiar:

```
import { createRoot } from "react-dom/client";
import React, { useRef, useState } from "react";
import { Canvas, useFrame } from "@react-three/fiber";

function Box(props) {
  // This reference will give us direct access to the mesh
  const mesh = useRef();
  // Set up state for the hovered and active state
  const [hovered, setHover] = useState(false);
  const [active, setActive] = useState(false);
  // Subscribe this component to the render-loop, rotate the mesh every frame
  useFrame((state, delta) => (mesh.current.rotation.x += delta));
  // Return view, these are regular three.js elements expressed in JSX
  return (
    <mesh
      {...props}
      ref={mesh}
      scale={active ? 1.5 : 1}
      onClick={(event) => setActive(!active)}
      onPointerOver={(event) => setHover(true)}
      onPointerOut={(event) => setHover(false)}
    >
      <boxGeometry args={[1, 1, 1]} />
      <meshStandardMaterial color={hovered ? "hotpink" : "orange"} />
    </mesh>
  );
}

createRoot(document.getElementById("root")).render(
  <Canvas>
    <ambientLight />
    <pointLight position={[10, 10, 10]} />
    <Box position={[-1.2, 0, 0]} />
    <Box position={[1.2, 0, 0]} />
  </Canvas>
);
```

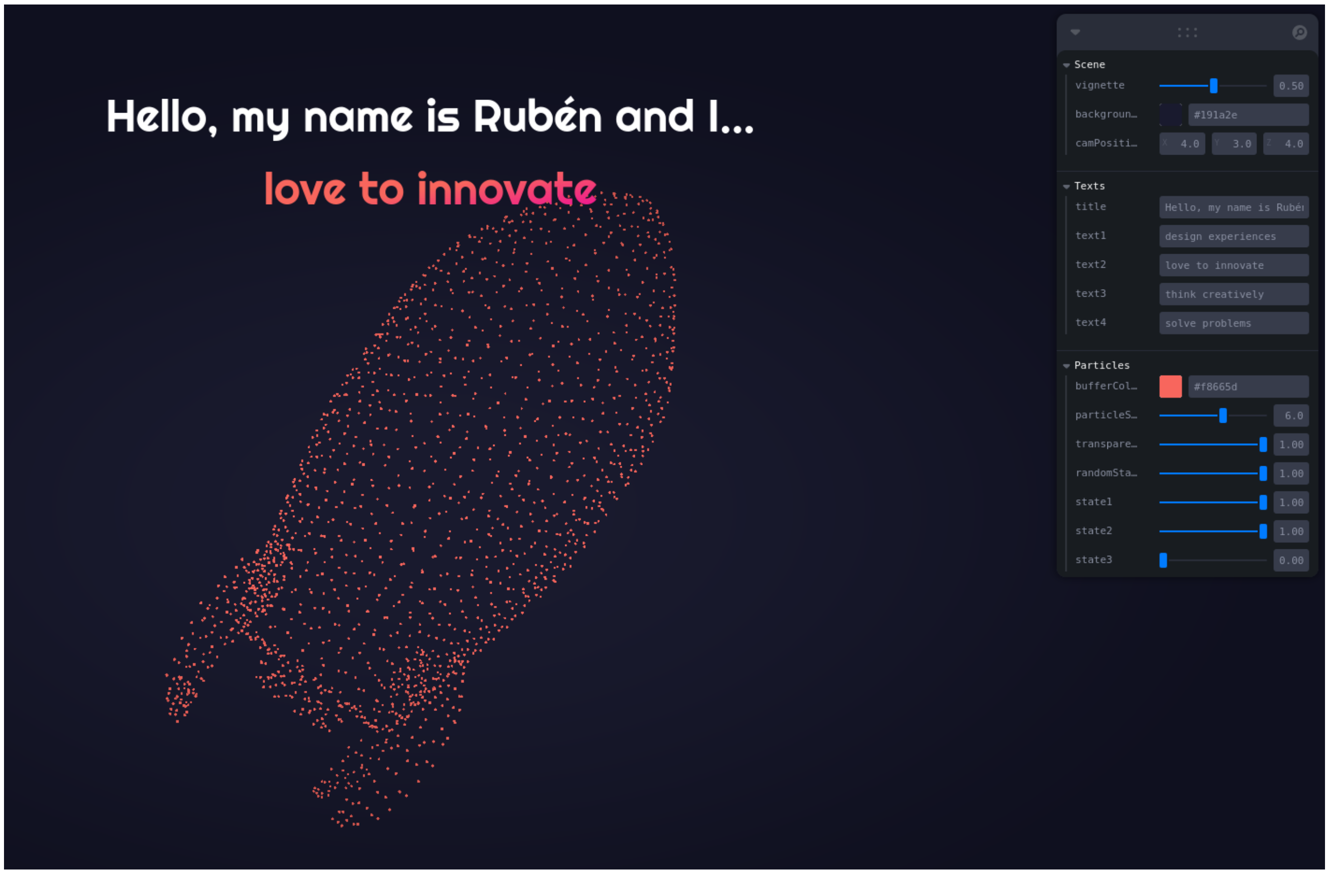
2. Gallery

All the projects are open source and at anyone's disposal. As the projects are very visual, there is also preview links to experience the end result in live.

> *Do take into account that all projects rely heavily on WebGL's render motor, which requieres a computer with decent computing capabilites in order to achieve 60fps.*

Particle showcase

A parametrized particle shader.

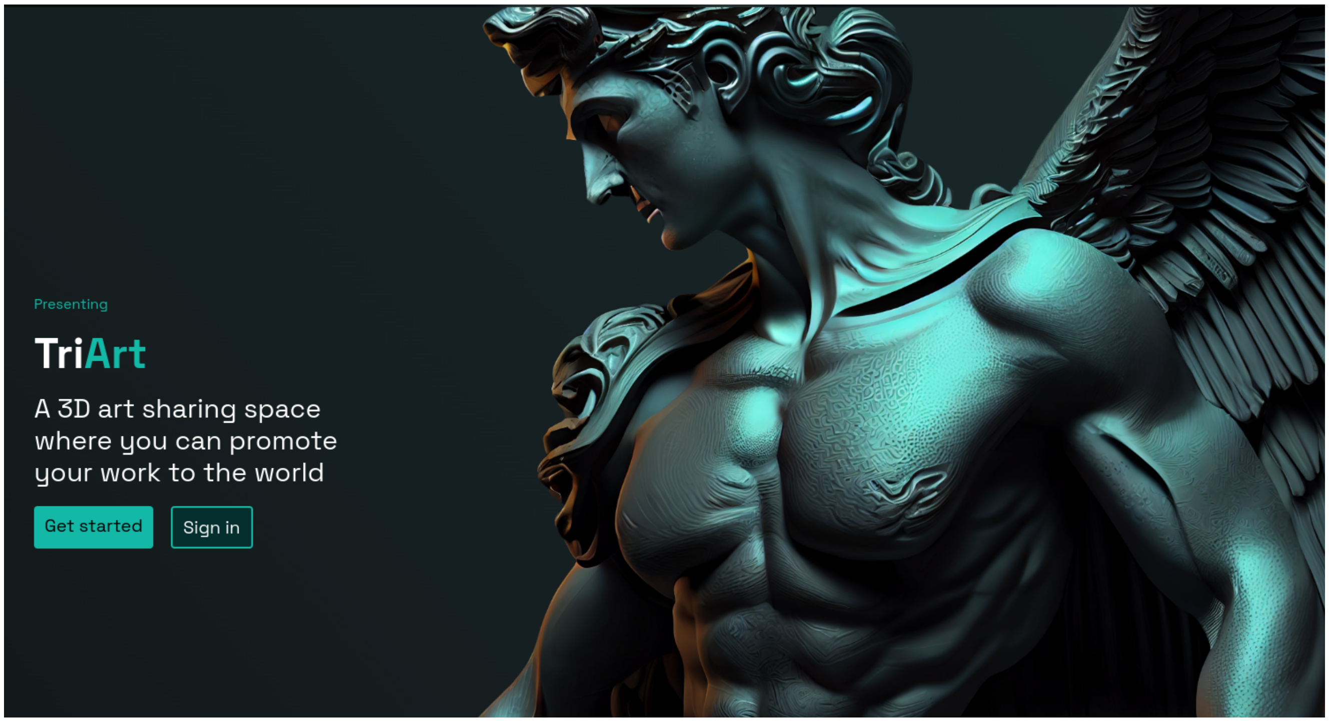


```
Scene
  vignette        0.50
  backgroun…      #191a2e
  camPositi…      X 4.0   Y 3.0   Z 4.0
Texts
  title           Hello, my name is Rubé
  text1           design experiences
  text2           love to innovate
  text3           think creatively
  text4           solve problems
Particles
  bufferCol…      #f8665d
  particleS…      6.0
  transpare…      1.00
  randomSta…      1.00
  state1          1.00
  state2          1.00
  state3          0.00
```

Repository   Live version

TriArt

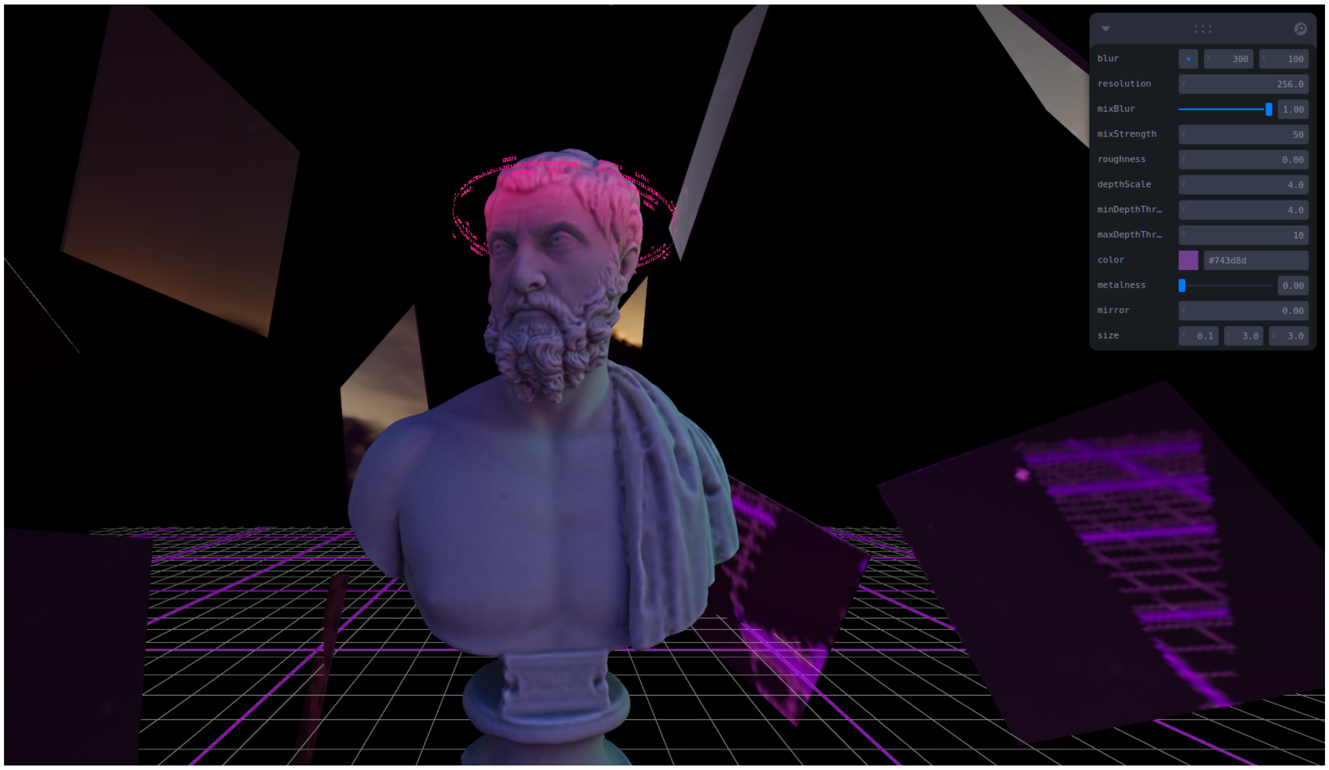
A 3D art-sharing space where you can promote your work to the world.


Presenting
TriArt
A 3D art sharing space where you can promote your work to the world
Get started
Sign in

Repository Live version

Mirror efect

An experimental reflective mirror cloud with post-processing effects.

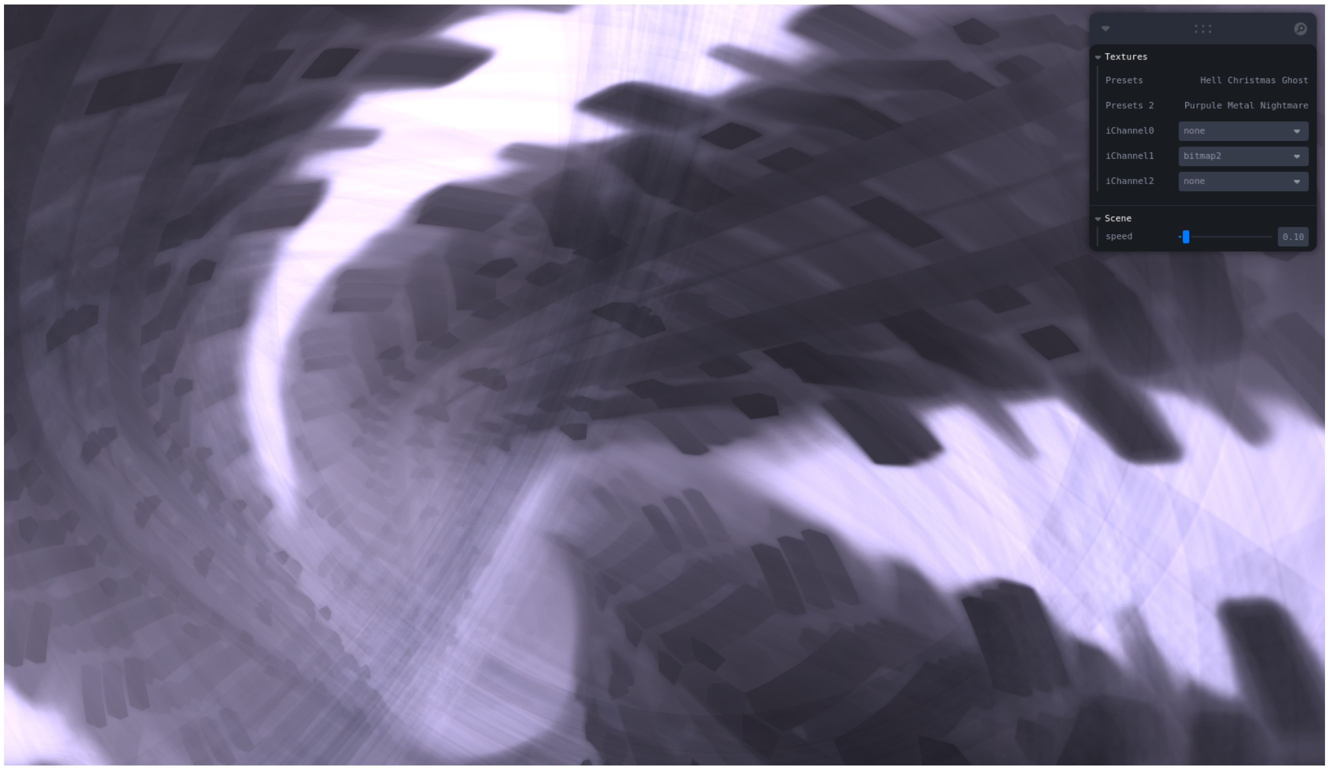


```
blur          X 300   Y 100
resolution    256.0
mixBlur       1.00
mixStrength   50
roughness     0.00
depthScale    4.0
minDepthThr…  4.0
maxDepthThr…  10
color         #743d8d
metalness     0.00
mirror        0.00
size          X 0.1   Y 3.0   Z 3.0
```

[Repository]   [Live version]

Buckle up

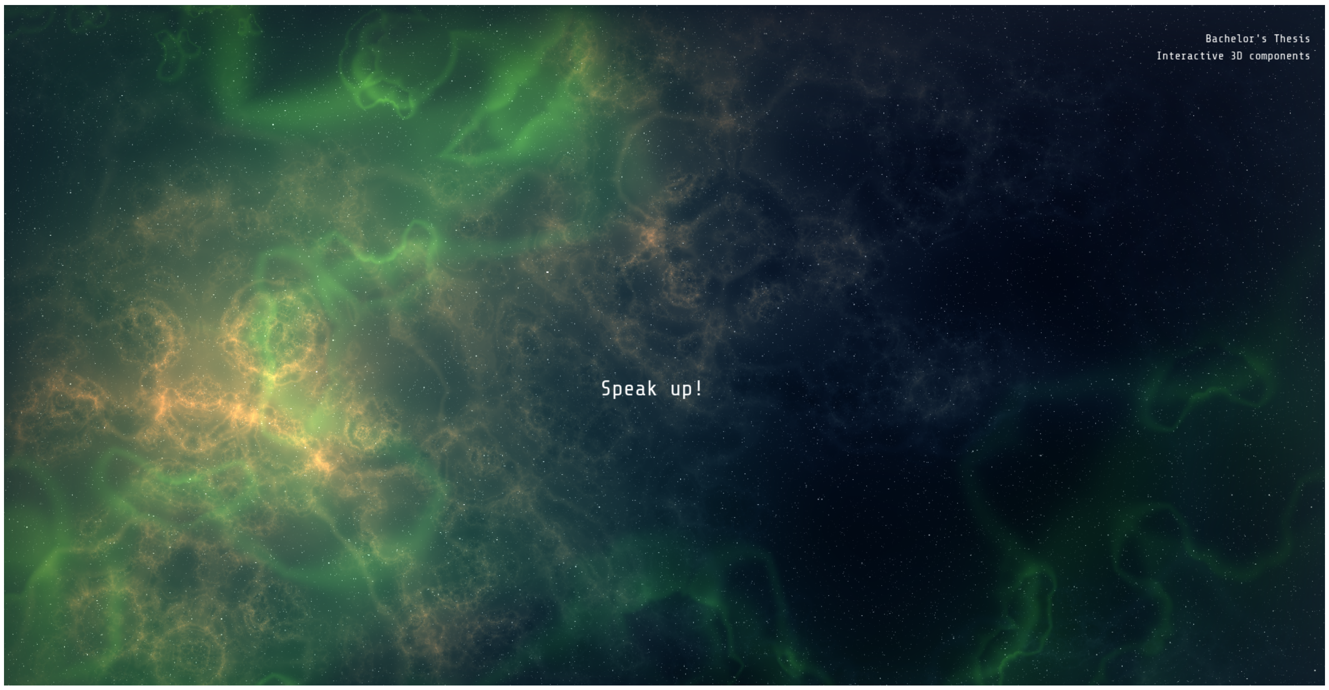
A parametrized shader showcase of an infinite vortex.


Textures
Presets Hell Christmas Ghost
Presets 2 Purpule Metal Nightmare
iChannel0 none
iChannel1 bitmap2
iChannel2 none
Scene
speed 0.10

Repository   Live version

Talking stars

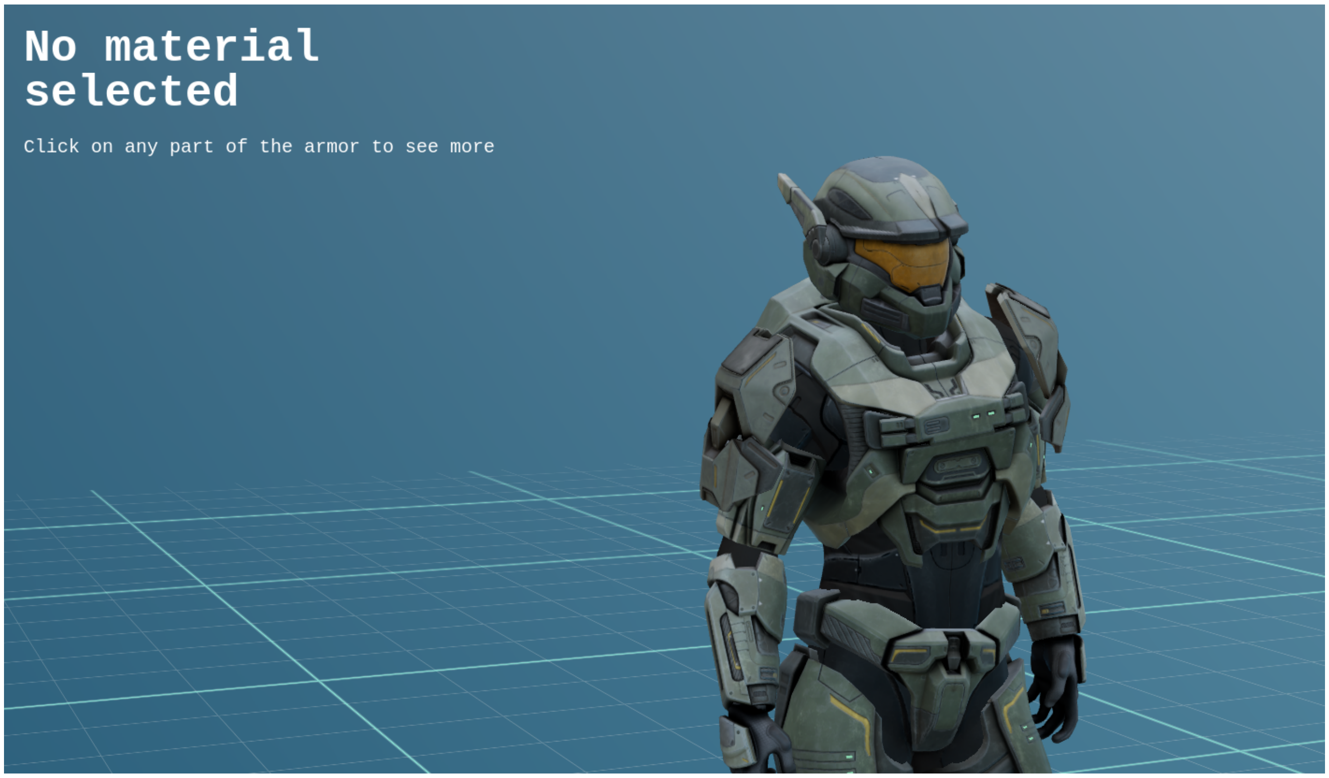
An interactive shader that can hear you speak.



Repository Live version

Halo inspector

An armor inspector of the famous halo charcater.



Repository Live version

3. Common libraries

Throughout all the projects, there are some libraries that were used repeatedly that aren't meant to be the main focus of the thesis. This chapter intends to give a quick summary of how they work.

3.1 Leva Controls

Leva is a GUI library commonly used alongside R3F to debug and have some visual controls over the parameters of your 3D scene.

These controls are commonly employed for debugging purposes, as it may be necessary to experiment with different values to achieve the desired outcome. Using an interactive user interface allows for easier adjustment of these values, eliminating the need to reload the application each time changes are made in order to validate their effects.

> *Leva was created by the Pmndrs collective, the same open-source developer collective who created R3F. This is why there are many examples of 3D scenes with these controls:*

3.2 Zustand / Jotai

Zustand, as well as Jotai are both state management libraries to create stores and share values across a React application. They are often used throughout the projects to avoid Prop drilling. Here's a Zustand example:

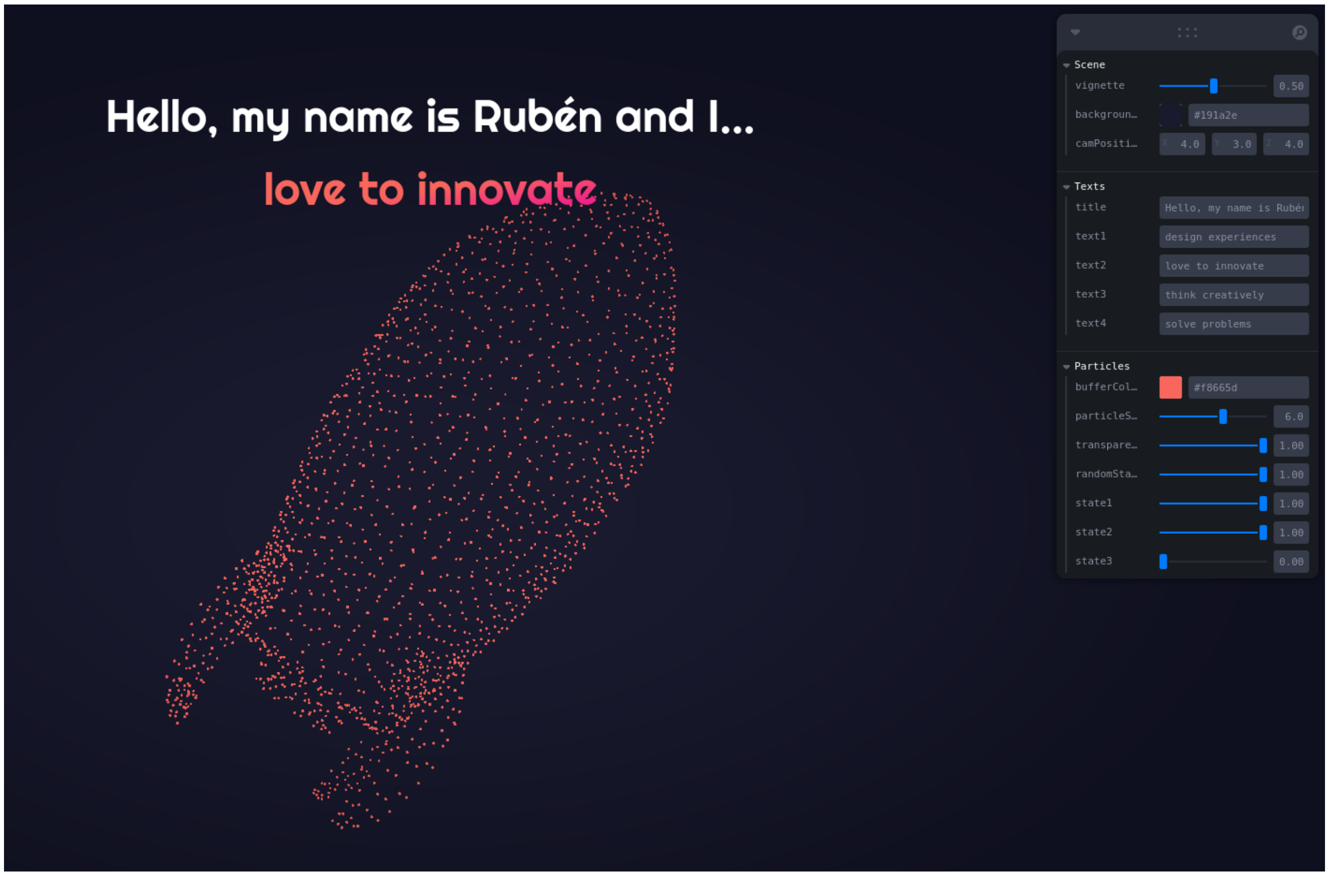
3.3 @a7sc11u/scramble

@a7sc11u/scramble is a library that provides scrambling text animations with a simple React component.

```
<TextScramble className="..." as="span" text={"Hello scramble"} />
```

> *By the time this documentation was written, the the creator of this small library discontinuated it. Now, a new lightweight package (1KB) with the same creator was made, which offers a new recommended way with a custom hook approach, written in TypeScript:*

4 Particle Showcase



```
Scene
vignette        0.50
backgroun…      #191a2e
camPositi…      X 4.0   Y 3.0   Z 4.0
Texts
title           Hello, my name is Rubén
text1           design experiences
text2           love to innovate
text3           think creatively
text4           solve problems
Particles
bufferCol…      #f8665d
particleS…      6.0
transpare…      1.00
randomSta…      1.00
state1          1.00
state2          1.00
state3          0.00
```

Repository   Live version

4.0.1 Installation

To set up the project, follow these installation steps:

```
git clone https://github.com/randreu38/TFG.particle-showcase.git
cd TFG.particle-showcase
yarn install
yarn dev
```

4.0.2 Project Overview

The project consists of a particle system made with a particle shader, that changes its internal state every x seconds, alongside some text. The project incorporates Leva controls for manipulating various aspects of the scene. The following code snippet showcases the structure of the `<Canvas/>` component:

```
<Canvas>
  <PerspectiveCamera makeDefault position={camPosition} />
  <color attach="background" args={[backgroundColor]} /> // A simple solid ba
  color
  <EffectComposer>
    <Vignette eskil={true} opacity={vignette} offset={0.1} darkness={1.5} />
  </EffectComposer>
  <OrbitControls />
  <Buffer />
</Canvas>
```

Furthermore, outside the canvas, the `<DynamicText/>` component is responsible for rendering dynamic text elements:

```
<h1 className="...">
  <span>{title}</span>
  <br />
  <DynamicText />
</h1>
```

4.0.3 The Canvas

Let's delve into the details of the `<Canvas/>` component. It includes a perspective camera with a position managed by the Leva controls. Notably, there is also an `<OrbitControls/>` component present. One might wonder how these two elements coexist without conflicting:

```
<Canvas>
  <PerspectiveCamera makeDefault position={camPosition} />
```

```
    <color attach="background" args={[backgroundColor]} />
    <OrbitControls />
    <Buffer />
    ...
  </Canvas>
```

The reason for their compatibility lies in the fact that the **`<OrbitControls/>`** component merely modifies the transform properties of the default canvas camera. In React Three Fiber, the default camera is implicit. By introducing the **`<PerspectiveCamera/>`** component with specific properties, we can modify the default camera's values.

> *In the context of 3D objects, the term* **`transform`** *encompasses both position and rotation.*

Additionally, the **`<EffectComposer/>`** component from React Three Postprocessing is employed to apply a subtle vignette effect to the scene:

```
  <Canvas>
    {/* ... */}
    <EffectComposer>
        <Vignette eskil={true} opacity={vignette} offset={0.1} darkness={1.5} /
    </EffectComposer>
  <Canvas/>
```

> *In Eskil's vignette technique, the effect originates from the outside and moves inwards, as opposed to the traditional inside-out approach. When the* **`eskil`** *prop is set to true, the offset value should be greater than 1.*

4.0.4 Modeling workflow

In this project, although the primary focus of the thesis was not on modeling, the author took personal responsibility for creating basic models. To achieve this, it was crucial to have real-time feedback on how the models would appear within the particle system.

Fortunately, Blender, the chosen 3D modeling software, offers a command-line interface (CLI) that allows the execution of Python scripts to automate workflows.

To enable constant feedback during the modeling process, the project utilized two scripts defined in the **`package.json`** file: **`yarn model`** and **`yarn watch`**. These scripts allowed the execution of the Python script while maintaining synchronization between the Blender scene and the R3F scene. Here are the scripts declared on the **`package.json`** file:

```
"scripts": {
  "model": "blender modeling/king.blend --background --python modeling/expo
  "watch": "watch 'yarn model' ./modeling"
},
```

The model script executes Blender commands, invoking the Python script for each model, resulting in the export of the models to the GLB format in the specified output directory. Here is the Python script that is executed:

```
import bpy
import sys

print("")
print(" Blender export scene in GLB Format in file "+sys.argv[-1])

# https://docs.blender.org/api/current/bpy.ops.export_scene.html
bpy.ops.export_scene.gltf(
    filepath=sys.argv[-1],
    check_existing=False,
    export_format='GLB',
    ...
    )
```

The **`watch`** script enables the continuous execution of the Python script on different models, ensuring that the R3F scene remains synchronized with the corresponding

`.blend` files. This iterative workflow enables quick iterations and feedback during the modeling process.

4.1 The Shader

> *This project heavily relies on GLSL (OpenGL Shading Language) for its rendering and interactivity. In this documentation, we will cover the essential aspects of the shader code. If you are new to shaders and would like to learn more, I recommend checking out the [Book of shaders] for a comprehensive introduction.*

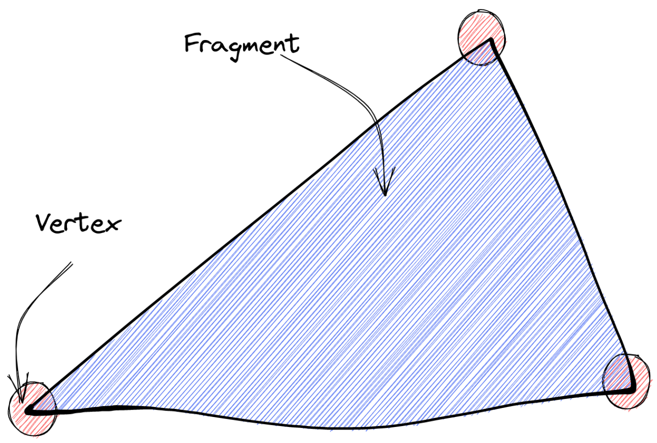
If you have scouted around the `<Buffer/>` component, you may have noticed that there a lot of elements orchestrating it. Let's try to understand it before diving into the code.

The component is named "Buffer" because it serves as a temporary container of information. In the context of 3D objects, each object has various properties such as materials, position, rotation, scale, and opacity. However, in this discussion, we will focus on the object's geometry.

The geometry of a 3D object consists of two key elements: vertices and fragments.

- **Vertices** are 3D points in space. They do not occupy physical space themselves but serve as connectors for the fragments.
- **Fragments** connect the vertices and are responsible for painting pixels on the screen. In this component, the emphasis is on the vertices, which are represented as particles or small dots in space. These particles are not directly connected to each other like traditional geometry; instead, they exist independently.

Now let's explore the code and dive deeper into the implementation of the shader.



The vertices are saved in a Float32Array, which saves thousands of numbers that are later decoded into what you see, in an attribute called `position`.

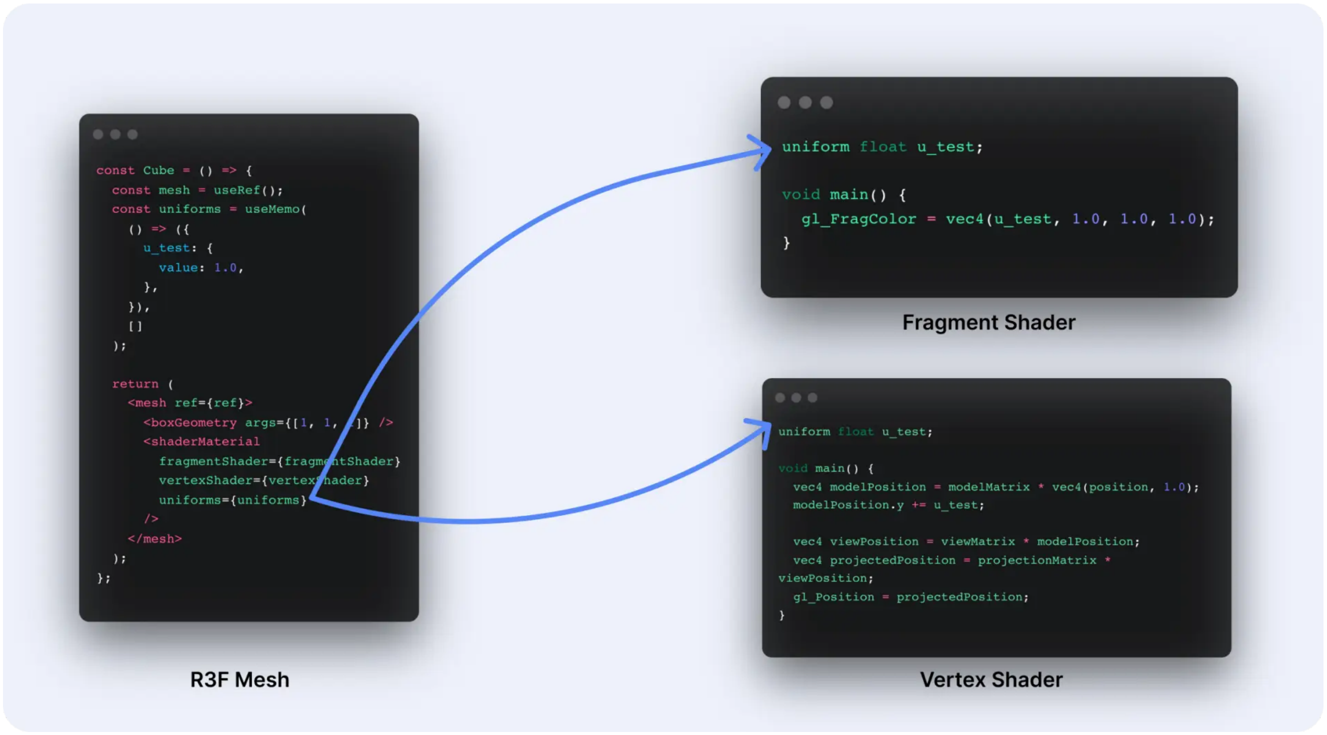
In our implementation, the `<Buffer/>` component represents not just a single object, but rather a collection of objects that change over time. To achieve this, we utilize a box-like structure where we define different positions for various objects such as a lightbulb, a chess piece, a box, etc. These positions are then interchanged dynamically.

Within the shader code, we declare attributes to represent these different geometries, like so:

```
attribute vec3 position; // Box geometry
attribute vec3 position2; // Random cloud geometry
attribute vec3 position3; // Rocket geometry
attribute vec3 position4; // Lightbulb geometry
attribute vec3 position5; // Chess piece
...
```

The purpose of these attributes is to allow us to change the geometries of the objects dynamically. Additionally, we also want to modify other aspects of the shader, such as particle size, color, transparency, and speed. To achieve this, we use uniforms, which can be modified from our TSX code as usual.

> *It's important to note that a `uniform` serves as a bridge between the GLSL code and the TSX code. They act as parameters that are passed into the shader and computed to create the visual effects we observe on the screen.*



```
const Cube = () => {
  const mesh = useRef();
  const uniforms = useMemo(
    () => ({
      u_test: {
        value: 1.0,
      },
    }),
    []
  );

  return (
    <mesh ref={ref}>
      <boxGeometry args={[1, 1, 1]} />
      <shaderMaterial
        fragmentShader={fragmentShader}
        vertexShader={vertexShader}
        uniforms={uniforms}
      />
    </mesh>
  );
};
```

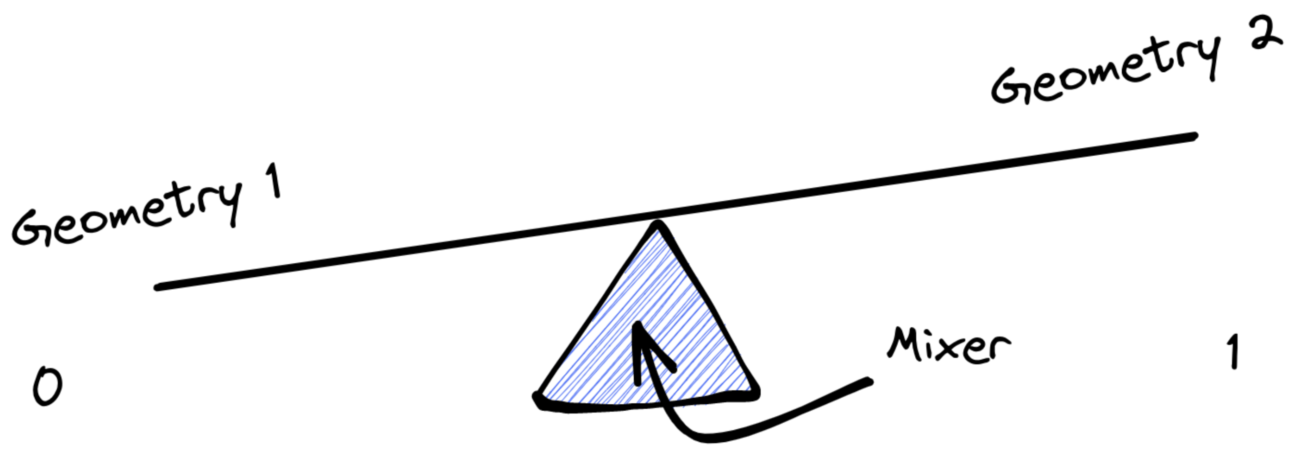
```
uniform float u_test;

void main() {
  gl_FragColor = vec4(u_test, 1.0, 1.0, 1.0);
}
```

```
uniform float u_test;

void main() {
  vec4 modelPosition = modelMatrix * vec4(position, 1.0);
  modelPosition.y += u_test;

  vec4 viewPosition = viewMatrix * modelPosition;
  vec4 projectedPosition = projectionMatrix *
viewPosition;
  gl_Position = projectedPosition;
}
```

These uniforms are then used to interpolate between the different geometries using the **`mix`** GLSL function:

```
//Cycles between geometries
vec3 switchGeometry3 = mix(position5, position, state3);
vec3 switchGeometry2 = mix(position4, switchGeometry3, state2);
vec3 switchGeometry1 = mix(position3, switchGeometry2, state1);
vec3 switchGeometry = mix(position2, switchGeometry1, randomState);
```

Think of the mix function as a scale. The first two parameters are both sides of the scale. The third value (a number between 0 and 1) is what determines how left or right will the balance end up going.



To add some idle animations to the geometries and make them look organic and alive, we utilize a simple sinus animation and incorporate Perlin noise using Stefan Gustavson's Perlin noise function.

The sinus animation creates an up-and-down movement effect by applying the sin function to the x coordinate of the model position, multiplied by a factor of 5, and the current time multiplied by 2.5. This value is then multiplied by 0.02 to control the magnitude of the animation.

The Perlin noise animation is achieved by using the cnoise function with the **`modelPosition.yz`** (excluding the x coordinate) and the current time multiplied by 0.5. The resulting noise is later toned down by bein multiplied by 0.05.

To combine these two animations, we use the mix function, which blends the sinus animation and the Perlin noise animation together with a balance of 0.5. The resulting value is stored in the idleAnimation variable.

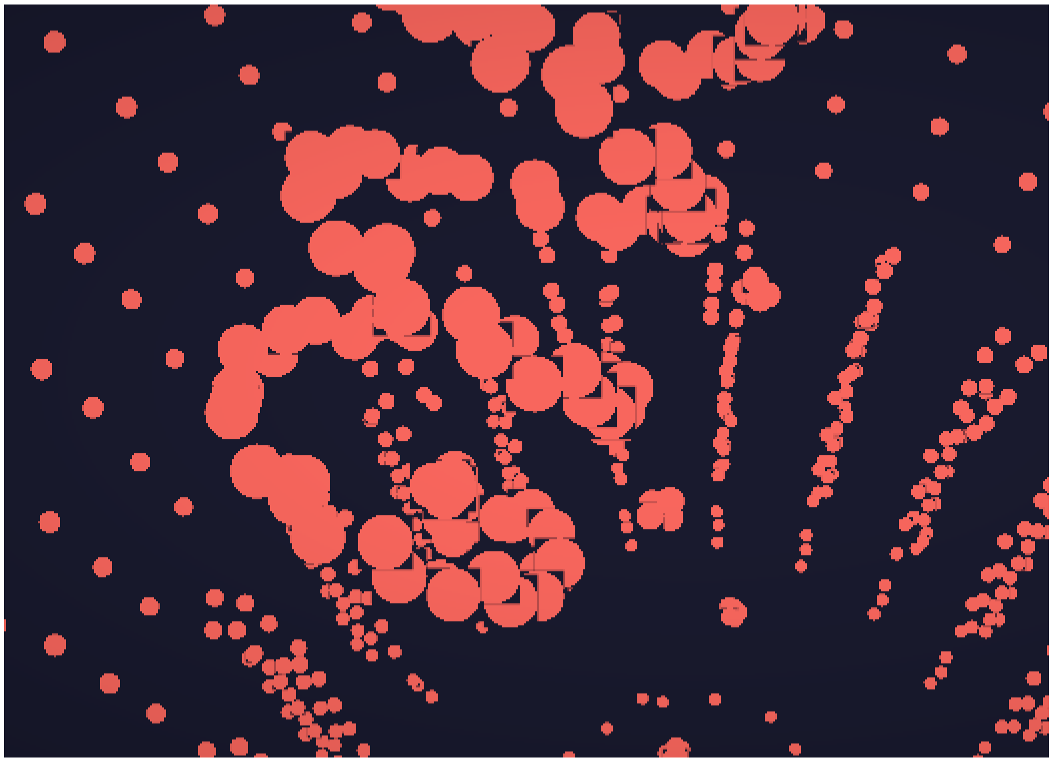
```
//IdleAnimation
float sinusAnimation = sin(modelPosition.x * 5.0 + time*2.5) * 0.02;
float perlinNoiseAnimation = cnoise(vec3(modelPosition.yz, time*0.5))*0.05;
float idleAnimation = mix(sinusAnimation,perlinNoiseAnimation, 0.5);
```

In the fragment shader, the main goal is to determine the color and transparency of the pixels. The **`bufferColor`** uniform represents the color of the particles, and the **`transparencyState`** uniform controls the transparency.

The particles are made to appear round by creating circles in the middle of each particle. This is achieved by calculating the distance of the current fragment coordinate from the center of the particle and comparing it to a threshold value of 0.5. If the distance is less than 0.5, the fragment is inside the circle and is assigned a value of 1; otherwise, it is outside the circle and is assigned a value of 0. This value is then multiplied by the **`transparencyState`** to control the overall transparency of the particle.

```
uniform vec3 bufferColor;
uniform float transparencyState;

void main()
{
    //make particles round
    vec2 xy = gl_PointCoord.xy - vec2(0.5);
    float ll = length(xy);
    float round = step(ll, 0.5);
    float finalAlpha = round;
    finalAlpha *= transparencyState;

    gl_FragColor = vec4(bufferColor, finalAlpha);
}
```

> *It’s important to note that transparency in the GLSL world is not truly achievable due to the limitations of rendering techniques. In this implementation, each particle is treated as a single vertex, resulting in square particles with circles in the middle to create the impression of transparency:*



4.2 Dynamic text

The dynamic text component in this project utilizes the scrambleUI library.for the text animations. To implement this, we cycle between different texts based on the tick state, which determines the speed at which the buffer changes its geometry.

To access the tick state, a Zustand store (similar to Redux) is used. The tick value is obtained from the store using the useStore hook. This tick value is then used to retrieve the corresponding text from an object of text values.

The **`useControls`** hook from the leva library is used to define the texts and their initial values in a control panel.

As seen beolow, the **`<TextScramble/>`** component receives the className prop for styling purposes, the as prop to specify the HTML element to be rendered, and the text prop, which is set to the current text based on the tick value:

```
import { TextScramble } from "@a7sc11u/scramble";
import useStore from "./store/store";
import { useControls } from "leva";

export default function DynamicText() {
  let tick = useStore((state) => state.tick);

  const [params] = useControls("Texts", () => ({
    text1: "design experiences",
    text2: "love to innovate",
    text3: "think creatively",
    text4: "solve problems",
  }));

  const text: string[] = Object.values(params);

  return <TextScramble className="..." as="span" text={text[tick]} />;
}
```

4.3 Buffer

In the **`<Buffer/>`** component, there are two main aspects: the geometry and the material. These are defined within a **`<points>`** element, like so:

```
<points ref={ref}>
  <bufferGeometry>...</bufferGeometry>
  <shaderMaterial>...</shaderMaterial>
</points>
```

4.3.1 The shader material

The material of this object is nothing like any other. Luckily, R3F allows us to create materials of our own, by creating a **`shaderMaterial`** and passing our own fragment and vertex shaders, as well as any uniforms we might need:

```
import fragShader from "./shaders/fragment.glsl";
import vertShader from "./shaders/vertex.glsl";
import { shaderMaterial } from "@react-three/drei";

 const [params, setParams] = useControls("Particles",()=>...)

 const ShaderMaterial = shaderMaterial(
    {
      particleSize: params.particleSize,
      bufferColor: new THREE.Color(params.bufferColor),
      time: 0,
      transparencyState: params.transparencyState,
      randomState: params.randomState,
      state1: params.state1,
      state2: params.state2,
      state3: params.state3,
    },
    vertShader,
    fragShader
```

```
  );
```

> *When importing GLSL files, Typescript doesn't know what to make of them. To tell typescript to import them as strings, you can create a declaration file **`glsl.d.ts`**:*

```
declare module "*.glsl" {
  const value: string;
  export default value;
}
```

One might have noticed that **`time`** is also uniform. This is to keep an internal state of how fast animations go internally, as GLSL does not have access to the frames per second of our **`<Canvas/>`**. To keep track of this, we use the **`useFrame`** hook of R3F:

```
const ref = useRef<myPoints>(null!); // Will later be referenced on the JSX
useFrame((state) => {
  ref.current.material.uniforms.time.value = state.clock.elapsedTime;
});
```

4.3.2 The geometries

Following up on what was discussed in the chapter 2.1, the Buffer contains different **`position`** values, that store the different vertices that compose the geometry. For declaring these values, R3F offers us **Computed Attributes**, which look like this:

```
<ComputedAttribute
  name="position"
  compute={() => {
    const geometry1 = new THREE.BoxGeometry(1, 1, 1, 16, 16, 16);
    const geometry1Attribute = new THREE.BufferAttribute(
      geometry1.attributes.position.array,
      3
    );
    return geometry1Attribute;
  }}
  usage={THREE.StaticReadUsage}
/>
```

This previous example stores in the **`position`** variable a simple Box Geometry divided in 16 for each axis (x,y and z). Now, in order to create the initial effect of assemblin the geometry, we need a cloud of points.

In order to achieve this effect,we need to create another box geometry and randomize the vertice's position and return it as a **`THREE.BufferAtribute`**:

```
<ComputedAttribute
  name="position2"
  compute={() => {
    const geometry1 = new THREE.BoxGeometry(1, 1, 1, 16, 16, 16);
    const geometry2 = new Float32Array(geometry1.attributes.position.count *

    for (let i = 0; i < geometry1.attributes.position.count * 3; i++) {
      geometry2[i] = (Math.random() - 0.5) * 10;
    }
    const geometry2Attribute = new THREE.BufferAttribute(geometry2, 3);
    return geometry2Attribute;
  }}
  usage={THREE.StaticReadUsage}
/>
```

> *The **`THREE.BufferAttribute`** takes two arguments, the Float32 array, and the item size to decode it. As we're working with 3 dimensions (a.k.a **`Vector3`**'s), we declare as second parameter a **`3`***

The rest of the models simply come from our **`.glb`** files and we compute them the same way:

```
import { GLTFLoader } from "three";

const king = useLoader(GLTFLoader, "models/king.glb");
const lightBulb = useLoader(GLTFLoader, "models/lightbulb.glb");
const rocket = useLoader(GLTFLoader, "models/rocket.glb");

const models = [king, lightBulb, rocket];

...

{models.map((model, index) => {
          return (
            <ComputedAttribute /
              name={`position${index + 3}`}
              compute={() => {
                const geometryAttribute = new THREE.BufferAttribute(
                  model.nodes.targetModel.geometry.attributes.position.array,
                  3
                );
                return geometryAttribute;
              }}
              usage={THREE.StaticReadUsage}
            />
          );
        })}
```

> *The **`targetModel`** node was specifically called like so in our 3D object inside the .**`blender`** files. Otherwise, the iteration would have to access different names.*

4.3.3 Animations

For the animations, we use the **`useEffect`** hook to manage our shader uniforms and change the position with the **`tic`** variable:

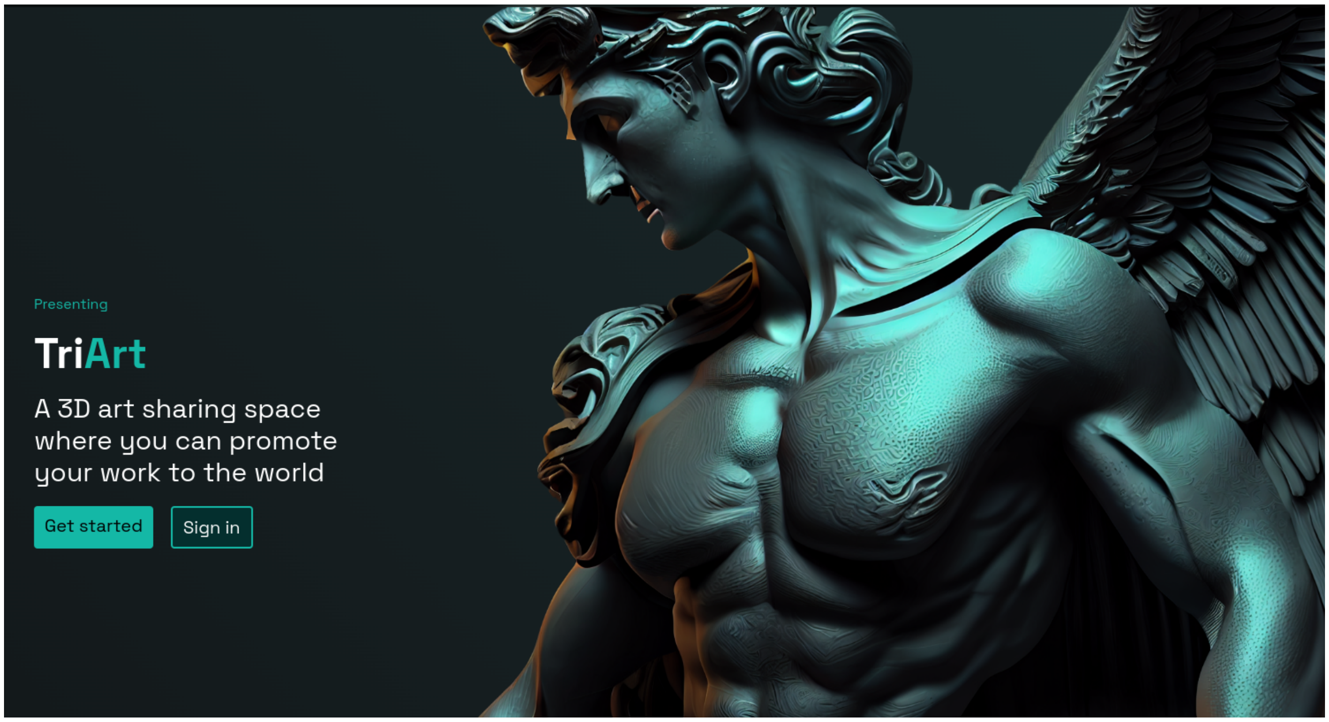
```
 let interval = setInterval(() => {
      if (tick == 3) {
        tick = 0;
        resetTick();
      } else {
        ++tick;
        incTick();
      }
```

Then we use Gsap to interpolate the values and create smooth animations based on the `tick` value.

```
switch (tick) {
   case 0:
     gsap.to(params, {
       state1: 1.0,
       state2: 1.0,
       state3: 1.0,
       duration: 1.25,
       ease: "circ.out",
       onUpdate: () => {
         setParams({
           state1: params.state1,
           state2: params.state2,
           state3: params.state3,
         });
       },
     });
    break;
    ...
```

> *Gsap is an animation library. Their API can be quickly explained with this graph:*

5. TriArt


Presenting
TriArt
A 3D art sharing space where you can promote your work to the world
Get started
Sign in

Repository Live version

> *There is a test user set up in place in case you might want to check out the project without registering:*

```
email: user@example.com
password: secret
```

5.0.1 Installation

To set up the project, follow these installation steps:

```
git clone https://github.com/randreu28/TFG.triart
cd TFG.triart
yarn install
yarn dev
```

> *Caution! This is a Full-Stack application. For security reasons, the environment variables needed to access the database are not public access.*
>
> *To run this locally, either create your supabase database instance or ask for the private keys at the author's contact page.*

5.0.2 Overview

TriArt is a platform where you can upload your 3D artwork and share them with the world. It creates unique links for each model, and the viewer can interact with them with a given set of options. The authors can monitor the model's views and visibility, should they prefer to keep some of their models private.

This project was made with Next13, with the app directory. The styling was done with TailwindCSS and HeadlessUI, and the database with Supabase. All of these pieces work together to make the project possible, but the aspect we will focus on will be the 3D side of it.

> *The reader needs not prior knowledeg of these technologies to follow along. Whenever there are things specific to these techonologies, the documentation will go over them. This documentation's goal is on explaining the decision-making of it all, not the intricacies of the chosen stack.*

5.1 Loading 3D models from the cloud

5.1.1 The usual approach

Loading 3D models has been well-considered by the 3D libraries utilized in this thesis, as evidenced by their implementation in other projects. This process is generally straightforward, thanks to the built-in loaders or helper functions provided by these libraries.

What is usually done is to have the model in a public route, such as https://yoursite.com/model.gltf. This model serves a specific function, and it is known at build time know what shape it has: The materials it uses, the nodes, the animations, etc. This allows us to write the models in JSX declaratively with the gltfJSX library:

But one might wonder, what if the model's information is asynchronous and one doesn't know it's shape beforehand? What if wemust **evaluate the model at run-time**? This is the case when loading models from the cloud.

TriArt revolves around cloud integration, which enables users to store and retrieve their 3D models securely. However, a key challenge arises when interacting with these models without prior knowledge of their specific details. Let's delve into the process of retrieving and interacting with 3D files in such scenarios.

5.1.2 Database structure

In this documentation, we will briefly discuss the structure of the Supabase instance that has been set up for TriArt. While we won't delve into excessive detail, it is important to understand the key components. The Supabase instance utilizes PostgreSQL and includes a table named "ArtWorks," which resembles the following structure:

id	user_id	visibility	url
1	292e4ad19628-9513	public	https://supabase.com/storage/model.glb

The `id` column serves as the unique identifier for each row, and the `user_id` column acts as a foreign key referencing the corresponding user. The `visibility` column determines whether the URL generated by TriArt has an authentication bypass or not. Lastly, the url column represents a straightforward URL generated by Supabase's storagefeature.

5.1.3 Server Side loading and validation

If you visit https://tfg-triart.vercel.app/artwork/32, for example, you'll notice that the URL structure takes the parameter `id` (in this case, 32) to select and display the chosen artwork. This, in combination with React Server Components of Next13, allows us to load the chosen artwork and handle the visibility status validation, without any line running on the client:

```
type Props = {
  params: { id: string };
};

export default async function Artwork({ params: { id } }: Props) {
  const {
    data: { session },
  } = await supabase.auth.getSession();

  const { data, error } = await supabase.from("artwork").select().eq("id", ic

  if (data[0].visiblity === "private" && data[0].user_id !== session?.user.ic
    throw Error("You don't have permission to see this artwork");
  }

  return <Scene url={data[0].url} />;
}
```

5.1.4 Loading unknown models

As previously discussed, knowing what model will you be working on has a lot of benefits. But what if the model comes from a third-party API? R3F has a way of loading models just by the use of a link (either local or public), with the **`useLoader`** hook:

```
  import { useLoader } from "@react-three/fiber";
  import { GLTFLoader } from "three/examples/jsm/loaders/GLTFLoader";

  export default function Scene() {
    const gltf = useLoader(GLTFLoader, "/Poimandres.gltf");
    return <primitive object={gltf.scene} />;
  }
```

5.2 Model Visualization

5.2.1 Animations

Loading unknown models can present challenges because the structure of the GLTF scene and its animations may vary. However, TriArt addresses this issue with its **`<Model/>`** component, which provides a solution for accessing animations:

```
function Model({ url }: Props) {
  const { scene, animations } = useGLTF(url);

  // Animations
  const { animationClips, defaultAnimationsControls, mixer } = useMemo(() =>
    const mixer = new THREE.AnimationMixer(scene);
    const animationClips: any = [];
    let defaultAnimationsControls: any = {};

    for (let a of animations) {
      let action = mixer.clipAction(a);
      animationClips[a.name] = action;
      defaultAnimationsControls[a.name] = false;
    }

    return { defaultAnimationsControls, animationClips, mixer };
  }, [animations, scene]);
}
```

When working with unknown models, maintaining type-safety can become challenging because you lack information about the specific model's structure. In such cases, it becomes necessary to use the **`any`** type, which allows for flexibility in handling dynamic and unknown data.

By using the **`any`** type, you can bypass strict type-checking and handle the model data in a more generic and adaptable manner. This enables you to access properties and perform operations on the model without explicitly defining their types, accommodating the unknown nature of the data.

While relying on the **any** type sacrifices some level of type-safety, it becomes justified in situations where the model's structure is unknown or variable. It allows you to work with the data without imposing strict type constraints, ensuring compatibility with different model formats and variations.

The **useMemo** hooks from the previous code snippet gives us 3 variables: the **defaultAnimationsControls**, which allow us to create the Leva controls later, the list of **animationClips** available, and the **mixer**, which takes care of handling the animations.

Now let us not forget that the **THREE.AnimationMixer** needs to be updated with the frame rate of our **<Canvas/>**, so we use the **useFrame** hook from R3F to keep it updated:

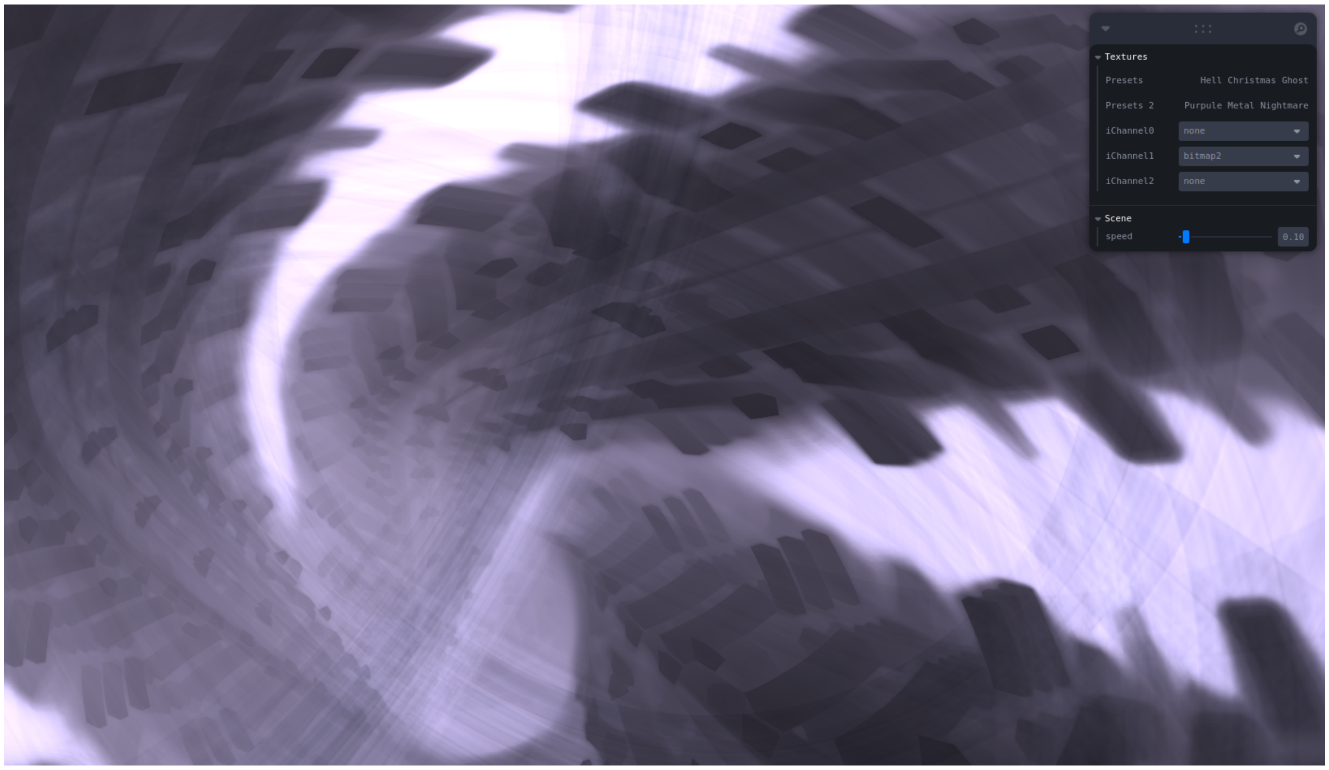
```
useFrame((_, delta) => {
  mixer.update(delta);
});
```

5.2.2 Scale normalization

Lastly, we need to take care of the size of the model. We don't know how big or small the model might be, but we want the user to be able to see it in a proportion that fits on the canvas. To do so, we check the size of it with **useEffect**:

```
useEffect(() => {
  const sceneSize = new THREE.Box3()
    .setFromObject(scene)
    .getSize(new THREE.Vector3()); //Measures the scenesice with a box
  const maxExtent = Math.max(sceneSize.x, sceneSize.y, sceneSize.z);
  const scale = (1 / maxExtent) * 5; //Sets the max scale to one standard `<C
  scene.scale.set(scale, scale, scale); //Sets the computed scale into the lo
}, [scene]);
```

6. Buckle up



Textures
Presets: Hell Christmas Ghost
Presets 2: Purpule Metal Nightmare
iChannel0: none
iChannel1: bitmap2
iChannel2: none
Scene
speed: 0.10

Repository   Live version

6.1 Installation

To set up the project, follow these installation steps:

```
git clone https://github.com/randreu28/TFG.buckle-up
cd TFG.buckle-up
yarn install
yarn dev
```

6.2 Overview

> *This project wouldn't be possible without the help of dila, the creator of the shader on which this project relies.*

This project was meant to explore shaders. To be more concrete, the R3F approach to using Shadertoy's. Shadertoy is a library of shaders created by the community, and it offers a lot of different options with shaders.

> *As the era of WebGPU approaches, the community for shaders is too. If you're interested in the next generation of shaders, you may refer to Compute Toys, a library made only for WebGPU shaders.*

The main objective of this project was to find a way to integrate pure GLSL shaders into a React application and explore their potential for creating interesting visual effects.

Now, let's examine the structure of the application:

```
<Suspense fallback={<Loading />}>
  <div className="...">
    <Canvas>
      <Shader />
    </Canvas>
  </div>
</Suspense>
```

6.3 Shader uniforms

> *In case you are not familiar with shaders, it is recommended that you've read the Particle showcase project, as it gives the base understanding*

The **`<Shader/>`** component is a simple **`<Plane/>`** geometry that occupies the whole viewport, and the custom shader material.

```
<Plane
  ref={ref}
  args={[
    document.documentElement.clientWidth,
    document.documentElement.clientHeight,
  ]}
>
  <shaderMaterial key={ShaderMaterial.key} />
</Plane>
```

In this code snippet, the **`<Plane/>`** component is utilized with specific arguments to define its dimensions based on the client's viewport size. Inside the **`<Plane/>`**, a **`<shaderMaterial/>`** component is added with a unique key to ensure proper updates and rendering of the custom shader material.

To ensure that the shader material receives the correct uniforms, we can define the appropriate types by extending the base types provided by R3F:

```
  interface myMaterial extends THREE.Material {
    uniforms: {
      iTime: { value: number }; //Intenal time state of the shader
      iResolution: { value: THREE.Vector3 }; // Resolution of the shader (viewp
      iChannel0: { value: THREE.Texture }; // Textures for the shader renderinc
      iChannel1: { value: THREE.Texture };
      iChannel2: { value: THREE.Texture };
    };
  }

  interface myMesh extends THREE.Mesh {
    material: myMaterial;
  }
```

In this code snippet, we define the **`myMaterial`** interface by extending the **`THREE.Material`** type. It includes uniforms as a property, which specifies the various uniform values required by the shader. The **`iTime`** uniform represents the internal time state of the shader, iResolution represents the resolution of the shader (viewport dimensions), and **`iChannel0`**, **`iChannel1`**, and **`iChannel2`** represent the textures used for shader rendering.

Similarly, we define the myMesh interface by extending the **`THREE.Mesh`** type. It includes a material property of type **`myMaterial`**, ensuring that the custom shader material is correctly assigned to the mesh.

> *The **`iVariableName`** naming convention comes from shadertoy, and it is being respected to communicate with the shader the same way.*

Next, the **`iTime`** and **`iResolution`** need to be updated every frame, so we can use the **`useFrame`** custom hook from R3F:

```
useFrame((state) => {
  ref.current.material.uniforms.iTime.value = state.clock.elapsedTime * speec
  ref.current.material.uniforms.iResolution.value = new THREE.Vector3(
    document.documentElement.clientWidth,
    document.documentElement.clientHeight
  );
});
```

6.4 Shader material

To declare the initial shader material, we import the fragment and vertex shaders from the respective files:

```
import fragment from "../shaders/fragment.glsl"; // From shaderToy
import vertex from "../shaders/vertex.glsl"; // From shaderToy
```

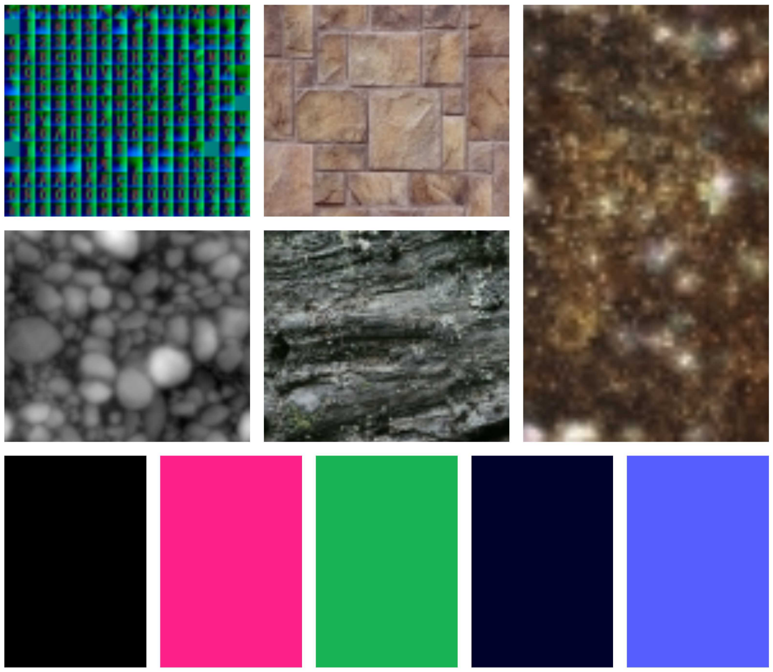
> *To import GLSL files as strings in TypeScript, you can create a declaration file named glsl.d.ts. In this file, you declare a module for *.glsl files and specify that they should be treated as strings:*

```
declare module "*.glsl" {
  const value: string;
  export default value;
}
```

Next, in the Model component, we create the **`ShaderMaterial`** using the **`shaderMaterial`** function provided by react-three-fiber. We pass the necessary uniforms and shader sources to the function:

```
export default function Model() {
  //...
  const ShaderMaterial = shaderMaterial(
    {
      iTime: 0,
      iResolution: new THREE.Vector3(
        document.documentElement.clientWidth,
        document.documentElement.clientHeight
      ),
      iChannel0: textures[activeTextures.iChannel0], // From the leva control
      iChannel1: textures[activeTextures.iChannel1],
      iChannel2: textures[activeTextures.iChannel2],
    },
    vertex,
    fragment
  );
  //...
}
```

Notice that the textures from the **`iChannels`** come from **`activeTextures`**. This comes from the leva controls, which are set up in such a way that they can choose from 10 different textures. The textures chosen provide a wide range of colors and combinations that allow the user to explore the different ways the shader relies on the materials:



6.5 Presets

The leva controls presets offer the user the possibility to interchange between the textures, but there are some presets that the user could select.

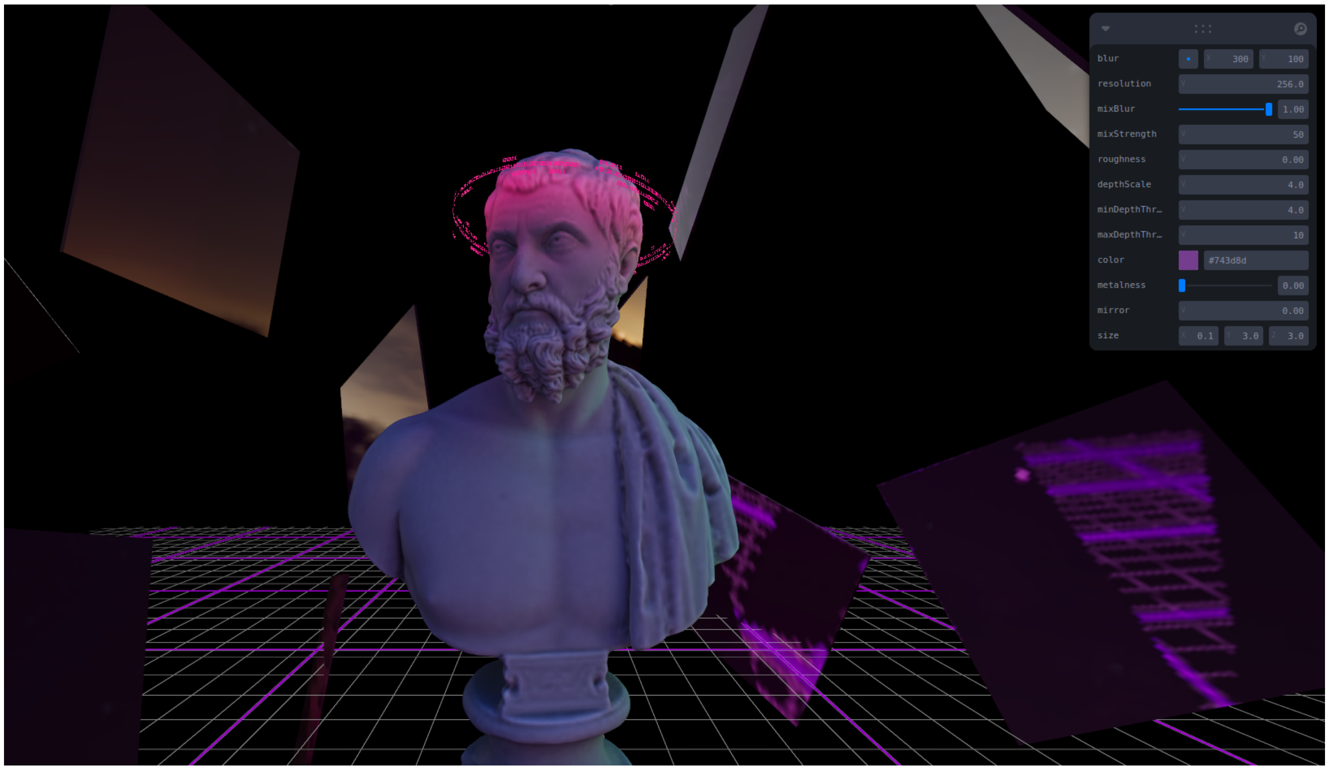
```
const [activeTextures, setActiveTextures] = useControls("Textures", () => ({
  iChannel0: {
    value: 0,
    options: textureControlOptions, //The list of materials
  },
  iChannel1: {
    value: 2,
    options: textureControlOptions,
  },
  iChannel2: {
    value: 0,
    options: textureControlOptions,
  },
}));
```

This already gives the user the ability to get all the combinations possible, but we'd like to create specific combination presets for the users to see:

```
const [activeTextures, setActiveTextures] = useControls("Textures", () => ({
  iChannel0: {
    value: 0,
    options: textureControlOptions,
  },
  iChannel1: {
    value: 2,
    options: textureControlOptions,
  },
  iChannel2: {
    value: 0,
    options: textureControlOptions,
  },

  1: buttonGroup({
    label: "Presets",
    opts: {
      Hell: () => {
        setActiveTextures({ iChannel0: 0, iChannel1: 6, iChannel2: 0 });
      },
      Christmas: () => {
        setActiveTextures({ iChannel0: 6, iChannel1: 1, iChannel2: 0 });
      },
      Ghost: () => {
        setActiveTextures({ iChannel0: 0, iChannel1: 2, iChannel2: 0 });
      },
    },
  }),

  2: buttonGroup({
    label: "Presets 2",
    opts: {
      Purpule: () => {
        setActiveTextures({ iChannel0: 7, iChannel1: 10, iChannel2: 5 });
      },
      Metal: () => {
        setActiveTextures({ iChannel0: 0, iChannel1: 4, iChannel2: 9 });
      },
      Nightmare: () => {
        setActiveTextures({ iChannel0: 0, iChannel1: 0, iChannel2: 4 });
      },
    },
  }),
}));
```

This way, the user can have a set of predefined texture combinations easily with the click of a button.

7. Mirror effect



```
blur            X 300   Y 100
resolution      V 256.0
mixBlur         1.00
mixStrength     V 50
roughness       V 0.00
depthScale      V 4.0
minDepthThr…    V 4.0
maxDepthThr…    V 10
color           #743d8d
metalness       0.00
mirror          V 0.00
size            X 0.1   Y 3.0   Z 3.0
```

Repository   Live version

7.1 Installation

To set up the project, follow these installation steps:

```
git clone https://github.com/randreu28/TFG.mirror-effect
cd TFG.mirror-effect
yarn install
yarn dev
```

7.2 Overview

This project was meant to be an exploratory approach to reflections. The idea behind it was to play around with some mirrors and try to get an interesting effect on them. I used a Roman statue from the artist engine9 that helped me get the style I aimed for.

7.3 Consent bypass

As the project ended up becoming very *artistic* iIt was decided to add an intro message before the actual scene. For that the app had a consent bypass that unclocked the main App once the user interacted with the **`<Intro/>`** component:

```
export default function App() {
  ...

  const [consent, setConsent] = useState<boolean>(false);

  if (!consent) {
    return <Intro setConsent={setConsent} />;
  }


  return (
    <>
      <div className="w-screen h-screen absolute left-0 top-0 z-10">
        <Suspense fallback={<Spinner />}>
          <Canvas>
            <MyScene />
          </Canvas>
        </Suspense>
      </div>
    </>
  );
}
```

7.4 Mirror material

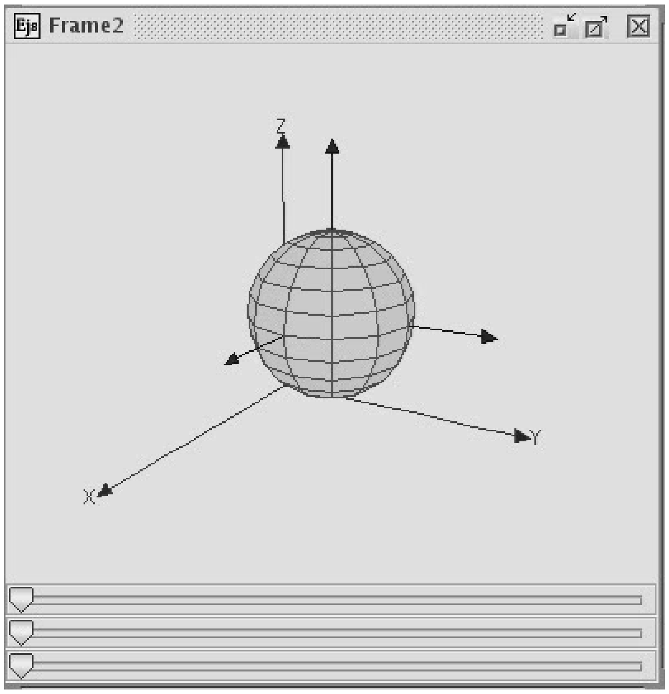
Before discussing the **`<MyScene/>`** component, we'll explore how the mirror material's implementation was achieved. Luckily, R3F offers us a reflector material with some props to configure to get the desired effect. To do so, the use of Leva controls was paramount. In matters like this, it is all about quick iterations and trial and error, and Leva excels at that:

```
import { MeshReflectorMaterial } from "@react-three/drei";
import { useControls } from "leva";

export default function Mirror(props: JSX.IntrinsicElements["mesh"]) {
  const config = useControls({
    blur: [300, 100], // Blur ground reflections (width, heigt), 0 skips blur
    resolution: 256, // Off-buffer resolution, lower=faster, higher=better qu
    mixBlur: { value: 1, min: 0, max: 1 }, // How much blur mixes with surfac
    mixStrength: 50, // Strength of the reflections
    roughness: 0,
    depthScale: 4, // Scale the depth factor (0 = no depth, default = 0)
    minDepthThreshold: 4, // Lower edge for the depthTexture interpolation (c
    maxDepthThreshold: 10, // Upper edge for the depthTexture interpolation (
    color: "#743d8d",
    metalness: { value: 0, min: 0, max: 1 },
    mirror: 0,
  });

  const { size } = useControls({ size: [0.05, 3, 3] });

  return (
    <mesh {...props}>
      <boxGeometry args={size} />
      <MeshReflectorMaterial {...config} />
    </mesh>
  );
}
```

7.5 Mirror generation

For the positioning of the mirror, the mirrors were initially placed according to the vertices of an Icosaedron geometry. Each vertex was the center of the geometry, alongside their Euler angle. Each mirror *looked at* the center of the Icoshaderon. Take, for example, the Three.js example of the Icosahedron and try to imagine the coordinates of each vertex:

> *The **Euler angles**, in contrast to the common radiant angles, describe a rotational transformation by rotating an object on its various axes in specified amounts per axis, and a specified axis order.*



Let us now examine how this process is accomplished within the React ecosystem. First, we need a function to generate the mirror cloud based on a float32Array:

```
/**
 * Generates a cloud of points based on the data array of an object.
 *
 * @param data - The raw data array of an object
 * @param length - The length on which the data array must be subarrayed
 * @returns an array of transform elements
 */
function generateMirrorCloud(data: any, length: number, scale: number) {
  let mirrors: mirror[] = [];
  for (let i = 0; i < data.length; i += length) {
    const dataArray = data.subarray(i, i + length);
    const newPosition = new THREE.Vector3(
      dataArray[0] * scale, // x
      dataArray[1] * scale, // y
      dataArray[2] * scale // z
    );
    const newRotation = new THREE.Euler().setFromVector3(newPosition);
    mirrors.push({ position: newPosition, rotation: newRotation });
  }
  return mirrors;
}
```

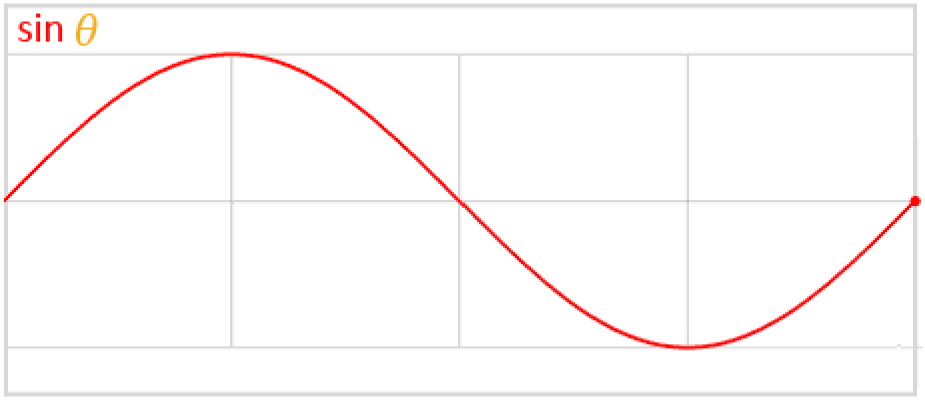
This, in combination with the **`THREE.Icosahedron`** class and the **`useMemo`** hook for performance purposes, we generate the cloud of mirrors:

```
function MyScene(){
const mirrors = useMemo(() => {
  const _Icoshaderon = new THREE.IcosahedronGeometry().attributes.normal;
  let Icosahedron: ArrayLike<number> | undefined;

  if (_Icoshaderon instanceof THREE.Float32BufferAttribute) {
    Icosahedron = _Icoshaderon.array;
  } else {
    throw Error("Type error");
  }

  return generateMirrorCloud(Icosahedron, 3, 6);
}, []);

...

return (
  ...
  <group ref={mirrorGroup}>
    {mirrors.map((mirror, key) => {
      return (
        <Mirror
          position={mirror.position}
          rotation={mirror.rotation}
          key={key}
        />
      );
    })}
  </group>
);
}
```

7.6 Animations

The animations of the mirror cloud are relatively straightforward. We utilize the **`Math.sin()`** function to generate a waving effect along the y-axis of the **`Vector3`** for the entire group, as well as their **`Euler`** rotation.

> *The sinus animation is ideal for producing straightforward "floating" animations, as they are infinite in nature and require minimal effort to implement using the algorithm.*



```
const mirrorGroup = useRef<THREE.Group>(null!);

useFrame((state) => {
  const t = state.clock.elapsedTime;
  const currentPosition = mirrorGroup.current.position;
  const currentRotation = mirrorGroup.current.rotation;

  currentPosition.set(
    currentPosition.x,
    currentPosition.y + Math.sin(t) * 0.005,
    currentPosition.z
  );
  currentRotation.set(t * 0.025, t * 0.025, t * 0.025);
});
```

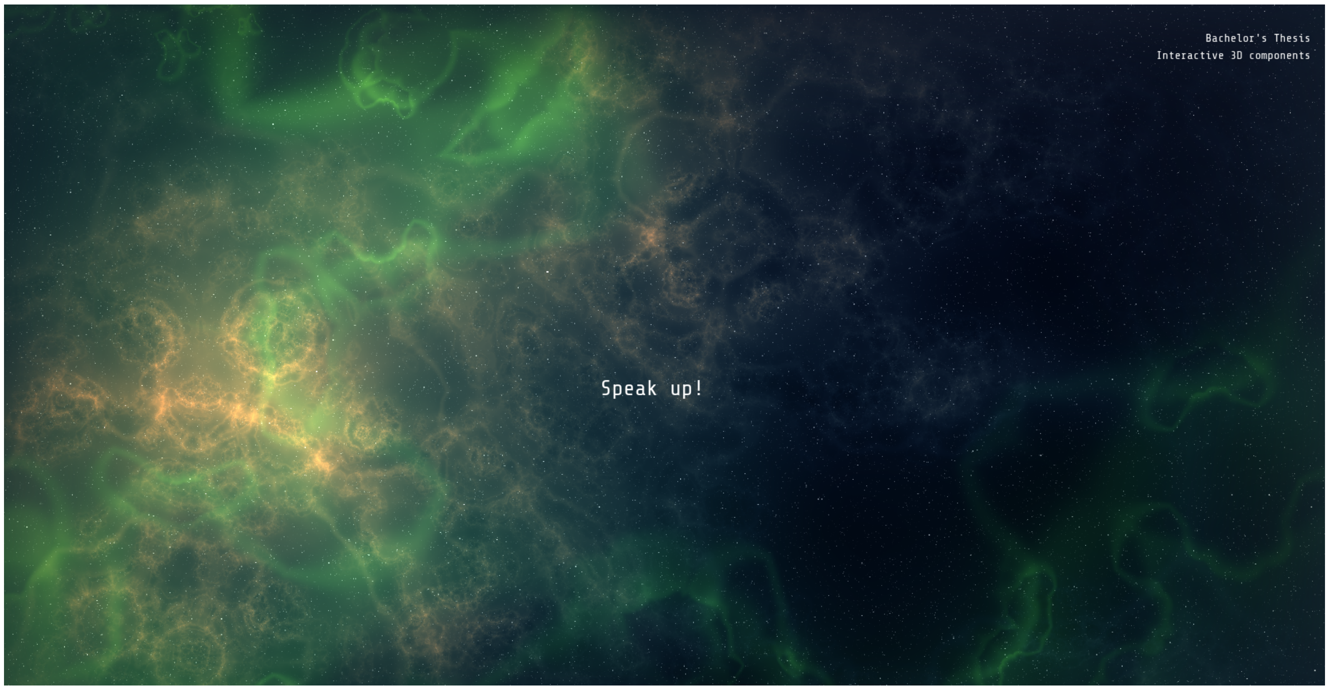
7.7 Post-processing

Lastly, with the help of React-Postprocessing, we will include some glitch effects:

```
<EffectComposer>
  <Glitch // Vector2 as they indicate min amd max values
    strength={new Vector2(1, 1)}
    duration={new Vector2(0.25, 0.25)}
    delay={new Vector2(5, 5)}
  />
</EffectComposer>
```

You may experiment with the **`<Glitch/>`** props to see how they modify the effect on this playground:

8. Talking stars



Repository   Live version

8.1 Installation

To set up the project, follow these installation steps:

```
git clone https://github.com/randreu28/TFG.talking-stars
cd TFG.talking-stars
yarn install
yarn dev
```

8.2 Overview

> *In case you are not familiar with shaders, it is recommended that you've read the Particle showcase project, as it gives the base understanding of them.*

This project was similar to the buckle up project, as both's objectives aimed to play around with shaders with the help of the community of shadertoy. This shader is special from the rest, as it relies on the user's microphone for its rendering process.

> *This project wouldn't be possible without the help of CBS, the author of the Simplicity Galaxy shader.*

8.3 Media Stream

For the use of the user's microphone, the user must give permission access to the microphone. This comes in the form of a Media Stream and is accessible through the **`navigator`** API:

```
navigator.mediaDevices
  .getUserMedia({ audio: true })
  .then((stream) => {
    setStream(stream);
  })
  .catch(() => {
    //...
  });
```

Once the user has granted access to the microphone, we save it in state. This way, we make sure the **`<Shader/>`** component will always have the stream prop:

```
export default function App() {
  const [stream, setStream] = useState<null | MediaStream>(null);

  if (stream) {
    return (
      <>
        <Signature />
        <p className="absolute inset-0 z-10 flex items-center justify-center
          Speak up!
        </p>
        <div className="fixed h-screen w-screen bg-gray-900">
```

```
          <Canvas>
            <Shader stream={stream} />
          </Canvas>
        </div>
      </>
    );
  } else {
    return <PermGranter setStream={setStream} />;
  }
}
```

8.4 The shader's uniforms

Similar to other projects, it is necessary to feed the shader with their necessary **`uniforms`**, which are required for their rendering process. In this case, we only need three:

- The **`iTime`**, which controls the internal clock of the shader
- The **`iResolution`**, which controls the size of the rendering canvas
- The **`iChannel0`**, which in this case, is a **`THREE.DataTexture`** that we will explain later

Let's first implement the types by extending them from the THREE classes:

```
interface myMaterial extends THREE.Material {
  uniforms: {
    iTime: { value: number };
    iResolution: { value: THREE.Vector3 };
    iChannel0: { value: THREE.DataTexture };
  };
}

interface myMesh extends THREE.Mesh {
  material: myMaterial;
}
```

> *The `iVariableName` naming convention comes from shadertoy, and it is being respected to communicate with the shader the same way.*

8.5 Creating the shader's material

Once we have a clear view of what the material needs, let us declare its initial state:

```
import frag from "../shader/fragment.glsl";
import vert from "../shader/vertex.glsl";
//...
const ShaderMaterial = shaderMaterial(
  {
    iTime: 0,
    iResolution: new THREE.Vector3( //ViewPort's resolution
      document.documentElement.clientWidth,
      document.documentElement.clientHeight
    ),
    iChannel0: 1, //Temporaray, we will pass in a real value later
  },
  vert,
  frag
);
```

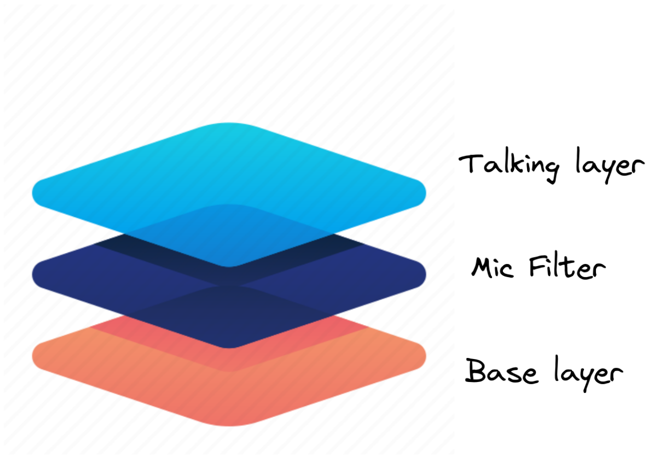
> *When importing GLSL files, Typescript doesn't know what to make of them. To tell TypeScript to import them as strings, you can create a declaration file `glsl.d.ts`:*

```
declare module "*.glsl" {
  const value: string;
  export default value;
}
```

8.6 How the audio affects the shader

The shader is tied to the microphone input through the **`uniforms`**, as it requires a **`THREE.DataTexture`** to pass in to the **`iChannel0`** uniform. This is because the shader is composed of three layers:

- The **base layers**, which are the green galaxy that we see on the screen when we don't talk
- The ***talking*** **layer**, which is what gets lighted up in blue when we do talk
- The **filter**. It only affects the *talking* layer, which isn't a value from 0 to 1 as one would expect, but a material, which either can be completely black, or completely white (Or shades of gray in different areas!). If it's black, the filter would not let any of the *talking* layer pass, and if it's white, it would let it pass completely.

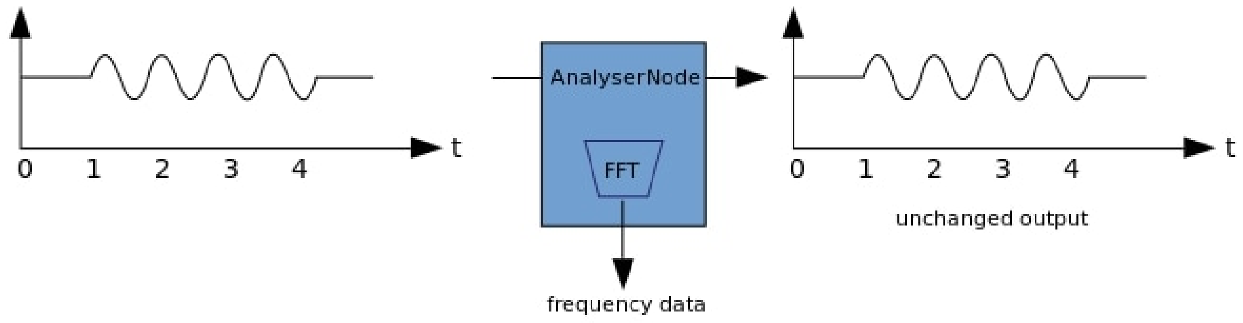


One may think of it as a camera filter. The darker the filter, the less you can see through.

8.7 Audio to Texture interpolation

To create a texture based on the material's input, we need an Audio Context. This will be in preparation for extracting the Hertz frequency of the media stream. You may think of it as a DOM EventListener, we only connect the **`stream`** Media Source to an analyzer that can calculate the Heartz without modifying the input.

```
export default function Shader({ stream }: Props) {
  //Creates an analyser for the media stream
  const audioCtx = new AudioContext();
  const mic = audioCtx.createMediaStreamSource(stream);
  const analyser = audioCtx.createAnalyser();
  const FFTData = new Uint8Array(analyser.frequencyBinCount);
  analyser.fftSize = analyser.fftSize / Math.pow(2, 3); // default is 2048
  mic.connect(analyser);
  //...
}
```



Then, we make use of the custom R3F hook **`useFrame`** to update the uniforms in every frame. That is when the audio-to-texture interpolation occurs:

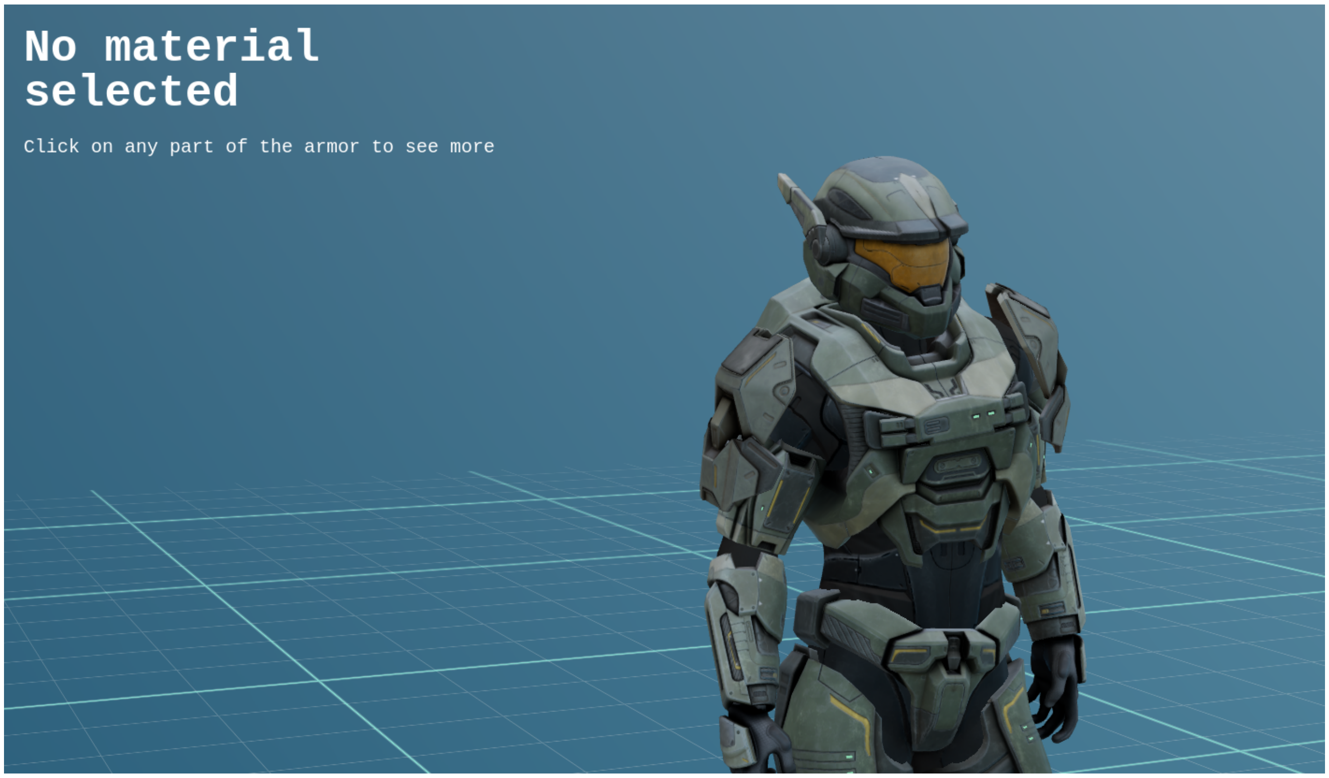
```
const ref = useRef<myMesh>(null!);

useFrame((state) => {
  //updates time and resolution uniforms
  ref.current.material.uniforms.iTime.value = state.clock.elapsedTime;
  ref.current.material.uniforms.iResolution.value = new THREE.Vector3(
    document.documentElement.clientWidth,
    document.documentElement.clientHeight
  );
```

```
    //Gets average mic hz
    analyser.getByteFrequencyData(FFTData);
    const avg = FFTData.reduce((prev, cur) => prev + cur / FFTData.length, 0);

    //Generates a gray scale image based on mic hz
    let amount = Math.pow(32, 2);
    let data = new Uint8Array(Math.pow(32, 2));
    for (let i = 0; i < amount; i++) {
      data[i] = avg * 20;
    }
    const audioTexture = new THREE.DataTexture(data, 12, 12);
    audioTexture.needsUpdate = true;

    //passes it as uniform
    ref.current.material.uniforms.iChannel0.value = audioTexture;
  });
```

Lastly, to assure that the plane occupies the user's viewport we create a simple plane that occupies the **`document`** client dimensions:

```
  <>
    <Plane
      ref={ref}
      args={[
        document.documentElement.clientWidth,
        document.documentElement.clientHeight,
      ]}
    >
      <shaderMaterial key={ShaderMaterial.key} />
    </Plane>
  </>
```

9. Halo inspector



Repository Live version

9.1 Installation

To set up the project, follow these installation steps:

```
git clone https://github.com/randreu28/TFG.halo-inspector
cd TFG.halo-inspector
yarn install
yarn dev
```

9.2 Overview

This project was designed to be an inspector of a given object, in this case, a halo model. The idea is to have each piece of the model be clickable and inspectable, with a complementary text to accompany it.

> *The project couldn't have been possible without the aid of, McCarthy3D the creator of the 3D model.*

Now, let's examine the structure of the application. Note that we are using Jotai as state management, as well as @a7sc11u/scramble

```
import { TextScramble } from "@a7sc11u/scramble";
import { useAtomValue } from "jotai";
import { Suspense } from "react";
import Loading from "./components/Loading";
import Scene from "./components/Scene";
import Signature from "./components/Signature";
import { matAtom } from "./lib/store";
import { getInfo } from "./lib/utils";

export default function App() {
  const material = useAtomValue(matAtom); // Gettting the selected material
  const info = getInfo(material); // Computing the complementary text based c

  return (
    <div className="h-screen w-screen flex justify-center items-center">
      <Suspense fallback={<Loading />}>
        <div className="absolute left-10 top-10 z-10 max-w-xl space-y-5">
          <TextScramble
            className="text-5xl font-bold"
            as="h2"
            speed={1}
            text={info.title}
          />
          <TextScramble
            className=" text-xl"
            as="p"
            speed={5}
            text={info.description}
          />
        </div>
        <Scene /> {/* Where the magic happens  */}
      </Suspense>
    </div>
```

9.3 The model

The model `.glb` file was first imported as a React component using the gltf-to-jsx CLI. As discussed in detail in the TriArt project. The generated result gives us a type-safe JSX through the `useGLTF` hook from R3F. Here's the generated result:

```
/*
Auto-generated by: https://github.com/pmndrs/gltfjsx
author: McCarthy3D (https://sketchfab.com/joshuawatt811)
license: CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)
source: https://sketchfab.com/3d-models/spartan-armour-mkv-halo-reach-57070b2
title: Spartan Armour MKV - Halo Reach
*/

import * as THREE from "three";
import { useEffect, useRef } from "react";
import { useGLTF, useAnimations } from "@react-three/drei";
import { GLTF } from "three-stdlib";

export function Model(props: JSX.IntrinsicElements["group"]) {
  const group = useRef<THREE.Group>(null!);
  const { nodes, materials, animations } = useGLTF("/halo.glb") as GLTFResult
  const { actions } = useAnimations<Animations>(animations as any, group);

  return (
    <group ref={group} {...props} dispose={null}>
      <group name="Sketchfab_Scene">
        <group
          name="Sketchfab_model"
          rotation={[-Math.PI / 2, 0, 0]}
          scale={0.02}
        >
          <group
            name="4757fffebe2a4d47b38143266af5f1a9fbx"
            rotation={[Math.PI / 2, 0, 0]}
          >
            <group name="Object_2">
              <group name="RootNode">
                <group name="Floor">
                  <mesh
                    name="Floor_lambert2_0"
                    castShadow
                    receiveShadow
                    geometry={nodes.Floor_lambert2_0.geometry}
                    material={materials.lambert2}
                  />
                </group>
                <group name="group">
                  <group name="Object_7">
                    <primitive object={nodes._rootJoint} />
                    <group name="Object_9" />
                    <group name="Object_11" />
                    <group name="Object_19" />
                    <group name="polySurface436" />
                    <group name="Helmet" />
                    <group name="Armour" />
                    <group name="Armour_LP" />
                    <skinnedMesh
                      name="Object_18"
                      geometry={nodes.Object_18.geometry}
                      material={materials.lambert1}
                      skeleton={nodes.Object_18.skeleton}
                    />
                    <skinnedMesh
                      name="Object_10"
                      geometry={nodes.Object_10.geometry}
                      material={materials.Spartan_Ear_Mat}
                      skeleton={nodes.Object_10.skeleton}
                    />
                    <skinnedMesh
                      name="Object_13"
                      geometry={nodes.Object_13.geometry}
                      material={materials.Spartan_Ear_Mat}
                      skeleton={nodes.Object_13.skeleton}
                    />
                    <skinnedMesh
                      name="Object_17"
                      geometry={nodes.Object_17.geometry}
                      material={materials.Spartan_Shoulders_Mat}
                      skeleton={nodes.Object_17.skeleton}
                    />
                    <skinnedMesh
                      name="Object_12"
                      geometry={nodes.Object_12.geometry}
                      material={materials.Spartan_Helmet_Mat}
                      skeleton={nodes.Object_12.skeleton}
                    />
                    <skinnedMesh
                      name="Object_16"
                      geometry={nodes.Object_16.geometry}
                      material={materials.Spartan_Legs_Mat}
                      skeleton={nodes.Object_16.skeleton}
                    />
                    <skinnedMesh
                      name="Object_20"
                      geometry={nodes.Object_20.geometry}
                      material={materials.Spartan_Undersuit_Mat}
                      skeleton={nodes.Object_20.skeleton}
                    />
                    <skinnedMesh
                      name="Object_15"
                      geometry={nodes.Object_15.geometry}
                      material={materials.Spartan_Arms_Mat}
                      skeleton={nodes.Object_15.skeleton}
                    />
                    <skinnedMesh
                      name="Object_14"
                      geometry={nodes.Object_14.geometry}
                      material={materials.Spartan_Chest_Mat}
                      skeleton={nodes.Object_14.skeleton}
                    />
                  </group>
                </group>
              </group>
```

```
          </group>
        </group>
      </group>
    </group>
  </group>
 );
}

useGLTF.preload("/halo.glb");
```

Although the output of the shader material might be verbose, it provides granular control over each mesh, which is essential for the required tasks. Additionally, by using a **`useEffect`** hook, we can initialize the breathing animation of the model on the initial load, and thanks to TypeScript, we have full type safety when accessing it.

```
useEffect(() => {
  actions["Take 001"]?.play();
}, []);
```

9.4 Material selection

For the material selection, we relied on the React-Spring library, an animation library similar to gsap, but with a modern react-based approach. The animations (besides the camera movements) are on the material's opacity, as they get more opaque or less depending on whether they are selected or not.

> *The React-Spring library was made by the pnmdrs collective, the same collective that created R3F, among many other libraries that we have used for this thesis. The synergy with our tools is so much so that React-Spring even has a chapter in its documentation dedicated to R3F integrations.*

As we need to control multiple materials simultaneously, we utilized the **`useSprings`** custom hook from React-Spring. This hook enabled us to create seamless opacity interpolations with a user-friendly API.

```
/* Makes the materials transparent, so we can play with its opacity later */
for (let _mat in materials) {
  let mat = materials[_mat as keyof typeof materials];
  mat.transparent = true;
}

/* Creates a state using springs to interpolate the opacities */
const [opacities, opacitiesOptions] = useSprings(
  Object.keys(materials).length,
  () => ({
    opacity: 1,
  })
);

/* Links the spring state to the material's value each frame */
useFrame(() => {
  let i = 0;
  for (let _key in materials) {
    const key = _key as NonNullable<MatName>;
    materials[key].opacity = opacities[i].opacity.get();
    ++i;
  }
});
```

Once we have set up the **`useSprings`** hook, we can proceed to create the onClick function handler.

```
/**
 * Reduces the opacity of every material except the one clicked
 */
function handleClick(e: ThreeEvent<MouseEvent>) {
  e.stopPropagation();
  let matName: MatName; // An array of strings representing all the possible

  if (e.object instanceof THREE.Mesh) {
    matName = e.object.material.name;
  } else {
    console.error("You didn't click on a Mesh");
```

```
      return;
    }

    const arrayMats = Object.values(materials);

    opacitiesOptions.update((i) => ({
      opacity: matName === arrayMats[i].name ? 1 : 0.1,
    }));
    opacitiesOptions.start();
    setMat(matName); //Saves it on Jotai, our state management library
  }
```

In addition to the **`onClick`** function handler for zooming in, we can create another function to handle the *zoom out* functionality when the user clicks on any other part of the screen.

```
  function handlePointerMissed() {
    opacitiesOptions.update(() => ({
      opacity: 1,
    }));
    opacitiesOptions.start();

    setMat(undefined);
  }
```

And we simply attach it to the parent group:

```
  <group
    onClick={handleClick}
    onPointerMissed={handlePointerMissed}
    ref={group}
    {...props}
    dispose={null}
  >
    {/* .... */}
  </group>
```

9.5 Camera movements

Every time a material is selected, there is an interpolation of the camera's position (a **`Vector3`**, in Three.js lingo), and a position of the material in close-up. For that, it was implemented a **`<CustomCamera/>`** component that handles all of these interpolations.

Before going into the **`<CustomCamera/>`** component, there is a small detail we might want to add to the model, to appropriately reference the meshes later. On each skinned mesh, it is pertinent to set the name attribute to the material's name, like so:

```
<skinnedMesh
  name={materials.lambert1.name}
  geometry={nodes.Object_18.geometry}
  material={materials.lambert1}
  skeleton={nodes.Object_18.skeleton}
/>
```

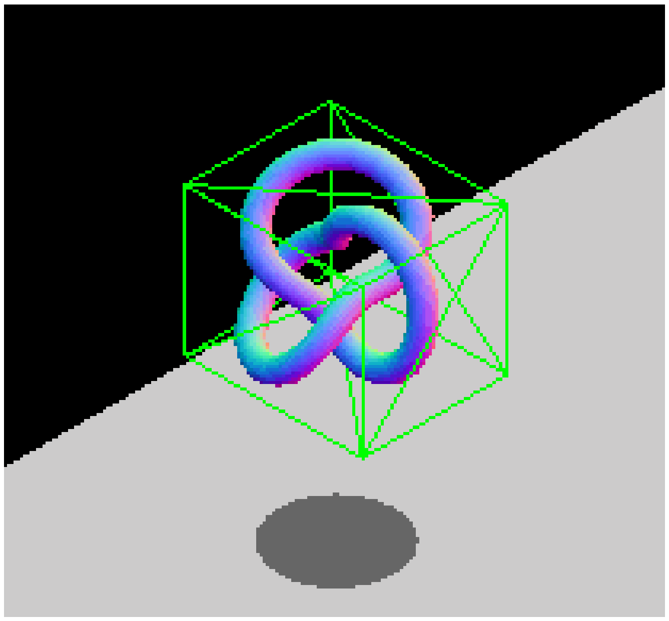
This will allow us to find each particular mesh by its name on the **`<CustomCamera/>`** component:

```
import { CameraControls } from "@react-three/drei";
import { useThree } from "@react-three/fiber";
import { useAtomValue } from "jotai";
import { useRef } from "react";
import { matAtom } from "../lib/store";

export default function CustomCamera() {
  const ref = useRef<CameraControls>(null!);
  const material = useAtomValue(matAtom); //From our store

  //Let's us get any value of the raw THREE.scene
  const get = useThree((state) => state.get);

  if (!material) {
    // When the user click's anywhere but the model
    ref.current?.setLookAt(3, 2, 6, 0, -0.5, 0, true);
    return <CameraControls ref={ref} distance={5} />;
  }

  //We're able to do this due to the previous step
  const selectedNode = get().scene.getObjectByName(material);

  if (!selectedNode) {
    //On loading stages there isn't any node and it returns undefined
    return <CameraControls ref={ref} distance={5} />;
  }

  ref.current?.fitToBox(selectedNode, true); // The native THREE.js way to ir

  return <CameraControls ref={ref} distance={5} />;
}
```

Fortunately, R3F has a wide range of controllers for the camera. This time we're using the R3F adaptation of the [camera controls library](), which can be interpreted as an inhereted extension of the more common **`<OrbitControls/>`**, with the addition of extra features, such as methods for tridimensional interpolations:

At first, it was considered interpolating using the **`setPosition`** method, which already out of the box supports a smooth camera transition, but had the inconvenience of finding the specific **`Vector3`** for each material by hand. This approach assumed that the object may never move from it's original postion, and it was not the most elegant of the approaches.

Luckily, the camera controls library also comes with the **`fitToBox`** method, which creates a bounding box around the specified object and makes it so that the camera zoom's in into said bounding box with a specified padding from the viewport's dimension.



This allows us to delegate the finding of the appropriate framing of the selected object, no matter from which starting position the camera finds itself, or if the object is moving or has moved since we first declared the **`Vector3`** for each material.