TDE
DIGITAL CREATIVES IN SPORTS

Feasibility Study

Improving the Grand Prix experience for F1 viewers at home

S8 Graduation FHICT

4 Sept 2023 - 16 Jan 2024

By Jordi Franssen

Introduction

With this feasibility study, I want to validate if the concept can actually be realized. I established that there are a few techniques that need to be tested in order to be certain that the product can be realized. I need to know if YukaJS is capable of simulating an Formula 1 race based solely on timing data. (Edit: YukaJS won't be used anymore.) Next ThreeJS's performance on mobile devices could be a thing. I have to be sure that low-end mobile devices can render an entire simulation. Also, data and network performance might be an issue. Large amounts of data need to be sent to the client, is this actually possible, and what does the network architecture look like?

YukaJS POC

Approach

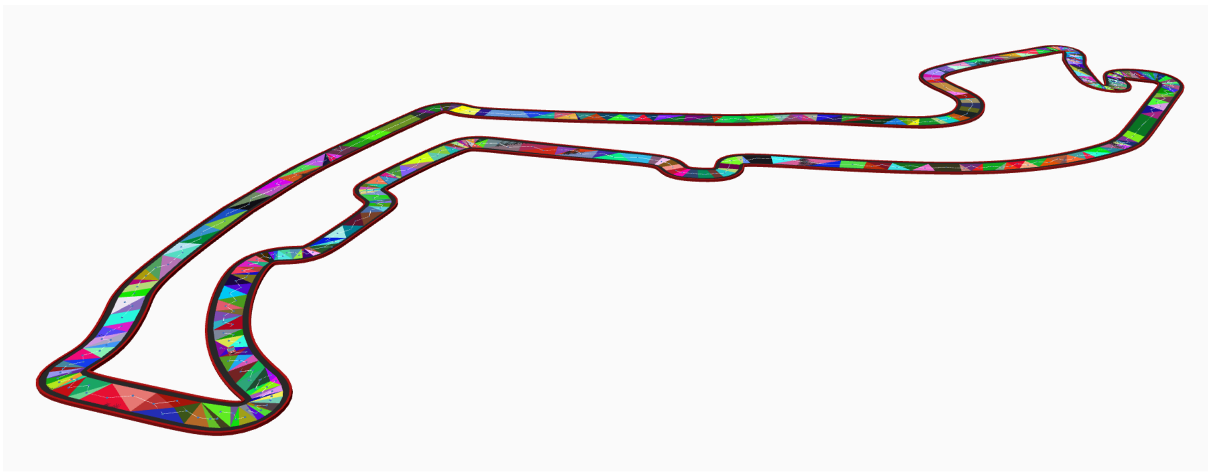
The goal of this feasibility study is to find out if it's possible to simulate the behavior of multiple cars on a racetrack and to check if the simulations are accurate enough. Since I'm still new to ThreeJS and YukaJS, I'll approach this POC in a way that it's easily scalable. Therefore, I'll start with one car on a track and the simple instruction to drive to a specific point. Later, I'll add multiple cars to see if it's possible to prevent them from colliding with each other. Next, I could try to make the cars drive complete laps instead of driving to a specific point and play around with timing and speeds to see if it's possible to have the behavior of the cars affected by timing data.

Make Obstacles And Restricted Areas In A Map Using Navigation Meshes - Three...

I found this tutorial that covers a lot of what I need for this POC, so this seems like a good place to start.

Change in approach

I conducted research on the different data sources available for my tool. In this research document it was concluded that there isn't any exact location data of Formula 1 cars available that can directly be used by ThreeJS. Therefore, Yuka.JS would have to simulate the car behavior based on timing data like lap times, sector times and velocities. However, this conclusion turned out to be false. During an update with John, a colleague was able to tell that there is in fact exact location data available from the FIA API, it just wasn’t present in the data samples I received. This alters the course of the development phase massively, as it will now be much easier to position the cars with ThreeJS.



I learned a lot by making this POC. In the ThreeJS-Journey course, I learned that React-Tree-Fibre is a front-end framework based on React that allows developers to develop ThreeJS applications more easily. Later I found out that Threlte is a similar front-end framework that's built around SvelteKit.

Since I'm probably going to build my MVP in SvelteKit. I decided to try and build this iteration in Threlte. However, I found out that this makes it way more difficult to implement YuksJS. Therefore, I changed my approach and decided to build this POC in SvelteKit instead.

Acquiring the 3D model of the Monaco GP track was easy as I could download it for free on Sketchfab. Next I had to generate a so-called navmesh based on the downloaded model that the car uses to know where it can drive and not. This had to be done in an old version of Blender as the newer versions don't support this feature anymore. This should also be possible with Unity3D. I gave this a shot as well because I think it's better to use up to date software instead of deprecated software. However, Unity3D is maybe even more complex than Blender so following the tutorial turned out to work best this time. The triangle-shaped colors are showing this navmesh in the illustration above.

The tutorial made the car move on a click event and the car would drive to the position of the click. I didn’t manage to recreate this as it was too difficult to detect whether the click happened on the track or not, so I decided to make the car drive to a predetermined point on the track instead. This works, YukaJS generates a path that’s most efficient to reach the predetermined location. Therefore, it already simulates a most efficient driving line. However, the car understeers a lot and therefore has a difficulty following the line and staying on the track. I know there are multiple functions and variables that can be changed in YukaJS that’ll have an effect on the line-following behavior of the car.

As I discovered that there's exact location data available to position the race cars in ThreeJS, YukaJS won't need to be used anymore. Therefore, there's no point in further elaborating this POC.

ThreeJS performance

From the target audience analysis it is concluded that RN365 is mostly visited on mobile devices, which, sometimes, have much less performance than laptops and PCs. Therefore, it's important to determine if mobile phones are powerful enough to run the simulation. This won't be super important for the first version, as I will build just a simple version of the race track and use dots to represent the cars. However, this can be elaborated into a much more detailed version with models of racecars. The question is whether mobile phones are powerful enough to visualize this.

The primary thing to keep an eye on that's important for performance is the refresh rate in frames per seconds (FPS). Online there is not much information found about actual requirements that define whether a 3D app is working smoothly or not. Therefore, I tried a performance benchmark example of ThreeJS.

https://threejs.org/examples/webgl_instancing_performance.html

By adding objects, the scene becomes more difficult to render. My phone, a Google Pixel 6 Pro, tends to struggle at around 7500 objects. With this number of objects, ThreeSJ outputs around 30 FPS. I noticed that this is somewhat the limit of what I would call a smooth experience. Anything below 30 FPS looks laggy and makes the phone respond slow to user inputs. I can use this simple test as a requirement while testing during the development phase.

ThreeJS optimizations

Actual requirements, like a maximum number of objects or model sizes are not defined for ThreeJS. There's enough to be found on the topic of optimizations however. Online there are lots of practices found on how to optimize a ThreeJS scene so it runs smoother.

Network and server performance

Besides the render performance of ThreeJS, it's also good practice to look into the network performance and requirements. The FIA API updates about every 230 milliseconds, which is quite fast. It is definitely possible to send the same amount of data to the client, but is this actually necessary or is it maybe possible to save bandwidth by sending a minimum amount of data, and how much impact would this have on server overhead?

To determine the absolute minimum amount of data, I made a POC that visualizes the data in a very simple and minified way. I made a simple front-end that places a dot in a square, based on the car’s position, every time a message is received from the server. This way, the shape of the map is visualized as the driver completes a lap around the track. The server sends a message about every second. However, every message contains between three to five position updates. I started by placing the dot on every first position update per message. This resulted in a laggy experience. An update every second is already slow, but it also feels laggy as the position updates aren’t received with a consistent interval.

Fortunately, each message contains a timestamp for each position update. Therefore, I was able to calculate the delays between the position updates per message. This way I was able to accurately display the position updates between the messages.

Therefore, all positioning data is needed to accurately display a car’s position on the track. Each message contains about 4kb of data. Each race contains about 10.000 messages. This means that 10.000 * 4kb = 40.000kb / 40mb of data is sent to each client, if the client is connected during the entire race. The live blog page of RN365 has between 200.000 and 400.000 visitors during the race according to Google Analytics. In the worst case scenario 400.000 clients will use the tool during the entire race. So, 40mb * 400.000 = 16.000.000 mb, or 16 terabytes worth of data that needs to be sent by the server.

According to colleague Jordi, who is an expert on server architecture and deployments, it should be possible to make this work. However, some extra investigation is necessary to be sure the server won't crash during heavy use.

Conclusion

The YukaJS POC wasn't finished, as YukaJS probably won't be used anymore. This doesn't mean that building this POC was a waste of effort. Since the tool uses an undocumented data source. There's no guarantee that the datasource with position data will work forever. Therefore YukaJS could potentially be used as a backup. I now know the capabilities of YukaJS and I know it is possible to simulate an entire Formula 1 race with it. The risk analysis will conclude if YukaJS is a viable option to use as a backup in the case the FIA API cannot be used anymore.

The ThreeJS performance test, didn't only conclude that ThreeJS is up for the job to simulate an entire F1 race. But I also established criteria for a smooth experience. I now know that at least 30 FPS are needed for a smooth experience, and that a medium spec smartphone can handle around 7500 moving objects before the framerate drops below 30 FPS. This is way more than the 20 moving cars driving around a Formula 1 track.

I also calculated a worst case amount of data that needs to be processed by the server. Jordi, an expert on software architecture and hosting confirmed that it's possible, but some additional research might be necessary. How the software architecture will look will be discussed in an interview with Jordi.

Summary

There are two important things that need to be tested in order to be sure that the end product can be realized. We know that most users will use the tool on their phone, so first I need to be sure that mobile devices are powerful enough to handle a ThreeJS scene.

Next, I must confirm that network and server performance won't be an issue. Therefore, I must calculate how much bandwidth the server needs to be able to handle.

The ThreeJS performance test, didn't only conclude that ThreeJS is up for the job to simulate an entire F1 race. But I also established criteria for a smooth experience. I now know that at least 30 FPS are needed for a smooth experience, and that a medium spec smartphone can handle around 7500 moving objects before the framerate drops below 30 FPS. This is way more than the 20 moving cars driving around a Formula 1 track.

I also calculated a worst-case amount of data that needs to be processed by the server. Jordi, an expert on software architecture and hosting confirmed that it's possible, but some additional research might be necessary. How the software architecture will look will be discussed in an interview with Jordi.

Learning Outcome Clarification

- Learning Outcome 1: Professional Duties
- Learning Outcome 2: Situation-Orientation
- Learning Outcome 3: Future-Oriented Organisation
- Learning Outcome 4: Investigative Problem Solving

This deliverable is a professional duty on a bachelor level in the activities of Analysis and Advise as I analysed the performance of ThreeJS and advised for the use of a criteria for establishing a smooth experience with ThreeJS. I also analysed if the large amounts of data that need to be handled by the server could be a problem and advised that this won't be a problem, but further research is necessary. This is in line with IT-area User Interaction. Therefore, Learning Outcome 1: Professional Duties applies.

This deliverable is relevant and valuable as it plays a role in the concepting phase of the project. Therefore, Learning Outcome 2: Situation-Orientation applies.

This deliverable is an effective approach to validate if the concept can actually be realised. Therefore, Learning Outcome 4: Investigative Problem Solving applies.

I identified potential problems and found an effective approach to find an answer to my question. Therefore, Learning Outcome 4: Investigative Problem Solving applies.