TTK4550 Specialization Project

Tobias Bergkvist

December 17th, 2019

Contents

Introduction

Background

The Heave Problem

When drilling from a floating rig or drilling ship, the heaving motion of the floater causes major pressure fluctuations in the well when the drill pipe is in slips during connections. Pressure fluctuations in the order of 10-20 bars have been observed in practice, sometimes giving an unacceptable risk of mud loss or kick. The only remedy for the problem is to wait for wind and waves to subside. There is a potential for saving time and cost by obtaining accurate information about downhole conditions on which to base the decision to wait or drill forward. HeaveLock has developed a simulator that predicts downhole pressure fluctuations based on weather information, response amplitude operator of the rig, well geometry, fluid properties etc. The simulator produces a large amount of data and the data needs to be visualized to the user in an easy way. In this project work, the objective is to visualize simulator data for a chosen well by developing a web-based interface.

HeaveSIM functionality (Heavelock's simulator)

TODO

HeaveSIM Output

`./HeaveSim simulations/Well_2/Connection_4739mMD/`

- `pipestress.csv`
- `heavevelocity.txt`
- `well_path.csv`
- `geometrydef.txt`
- `pipepressure.csv`
- `chokeopening.txt`
- `trends.csv`
- `fluiddef.txt`
- `annuluspressure.csv`
- `controldef.txt`
- `chokedef.txt`
- `pipeflow.csv`
- `pipevelocity.csv`
- `bitdef.txt`
- `annulusflow.csv`
- `mainpumprate.txt`
- `drillstringcontrol.txt`

For this task, the files of interest are: `well_path.csv`, `geometrydef.txt`, `pipepressure.csv`, `fluiddef.txt`

Task description

The following tasks should be performed by the student:

1. Background: Brief general description of the heave problem in offshore drilling from a floating rig and the idea behind the simulation tool
2. Background: Brief description of HeaveSIM functionality (Heavelock's simulator).
3. Make a systematic overview of the output that is produced from a HeaveSIM simulation
4. Interface:
    - Static 3D visualization of well trajectory including information on:

    - Riser and seabed, casings and liners, open hole section
    - Drill string and Bottom Hole Assembly
    - Simulation points with click-on functionality to retrieve additional information
  - Animated 3D visualization or colorized visualization of a chosen simulation
    - Visualize movement of rig, drill string and bit
    - Visualization of pressure along well
    - Visualization of stress along drill string
    - Visualization of contact friction along drill string
5. The user interface shall be integrated into Heavelock's dashboard solution
6. Implement and test the user interface according to the specified requirements.
7. Write report.

Concretization of problem

PROBLEM CONCRETIZATION OF GOAL AND METHOD START OF PROBLEM

- Looking at the file output directly conveys little/no knowledge
- 2D-visualizations are generally good, but can be abstract and hard to relate to.
- A 3D-model/visualization is very intuitive and easy to relate to and understand - even for non-technical people.
- In the oil industry, employees often have to use browsers like Internet Explorer. (Alternatively: Mobile browsers)
- Installing a program takes more work, and is less protable than using a web application.

Before we start, it might be useful to define some terms and concepts:

- If the pressure drops too low or rises too high, this can cause problems

Looking directly at the output from HeaveSIM, it is hard to get an idea of what the oil string geometry looks like, and exactly how the pressure is fluctuating (and where there are potential problems). If the pressure fluctuations along with the three dimensional path of the string could be seen at the same time - it could prove very useful for quickly localizing problems on the oil string. It could also serve as a tool for conveying information to people that have less technical knowledge - due to how relatable and intuitive a 3D-model is.

We want to create an animated and interactive 3D-visualization in a web client. In the recent years, a technology called "WebGL", based on "OpenGL" has been implemented and standardized for Web Browsers. The main purpose of WebGL is to allow web browsers to utilize the graphic processing units of computers and mobile phones for 2d and 3d-rendering. This is perfect for this use case - since the goal is rendering and animating something in 3D.

Based on the layout of the `well_path.csv`-file, one could create a 3d-cylinder with a specific color for each pipe segment in the well_path definition. Since the well_path defintion does not give us the spatial centers of these cylinders directly, we would need to compute this ourselves (Based on measure depth and angles).

WebGL only concerns itself with points and triangles. Creating a cylinder from triangles would require quite a bit of code. It would be useful if there was some kind of abstraction that would allow for creating more complex geometric objects (such as a cylinder), and controlling the colors of these objects. We are in luck, as a library known as THREE.js will make our life quite a bit easier.

When it comes to using libraries, and writing modular code that works for multiple browsers, it is common to use a bundler for this purpose. A bundler will combine modules with dependencies into static assets. It will also be able to minify your code to reduce bundle size (remove whitespace, and rename variables/identifiers to be as short as possible) and perform tree shaking (remove code that will never be executed from the bundle) along with other optimizations and useful features.

To start with, the well_path.csv-file is loaded using the library d3.js (Data-Driven-Documents). d3.js is commonly used for interactive 2D-visualzations, but it has a csv-parser built in, which is the only reason we are using d3 (for now). Since we don't have the actual positions pf

Why a Web Application?

The main advantage of creating a web application is that it allows the application to run on multiple platforms, including desktop (Windows, MacOS, Linux) and mobile (iOS, Android) and many others. Creating native applications for all of these platforms would take a lot more work. Note that browsers are not all the same - and it does take some work to make an application usable across multiple browsers, with both mouse and touch gestures - and a wide variety of display sizes.

Implementation

TypeScript vs JavaScript

Most of this project will be written in TypeScript. TypeScript is a superset of JavaScript that compiles to regular JavaScript and runs type checks against the code. It also works very well together with Visual Studio Code (as both are created by Microsoft), which massively improves autocomplete and code hints.

Parcel vs FuseBox (vs WebPack)

FuseBox has been built with TypeScript support being one of the main goals. Parcel, in contrast uses something called Babel (explain) to transpile code. - FuseBox has some configuration, but a lot less than Webpack. Parcel is known for being "zero-config". - Parcel had trouble transpiling node_modules (which are the project dependencies). This is something FuseBox does by default. - FuseBox v4 has just been released, but it has very little documentation

Algorithms

spatial centers of cylinders from `well_path.csv`

assigning radiuses to the each of the cylinders with `geometrydef.txt`

deciding placements of casing shoes

Choice of technologies?

Optimizations

1. Load everything from csv files directly into the client. Takes a long time to draw. Algorithm to calculate positions of all pipe segments.
   - The real length and radii of the pipe make the 3d-structure hard to see. To fix this, we will use LENGTH_SCALING and RADIUS_SCALING with some empirical values based on the data.
2. Change the color of the styles each timestep. It takes a long time to load pipepressure.csv. It also takes a lot of time to calculate a new color for every pipe segment.
3. Attempt precomputing all of the colors. This uses a lot of RAM, and makes the application crash on mobile. It takes around 30 seconds before the application becomes responsive. The frame rate doubles compared to before on computers. We want this to work on mobile as well, so this is not acceptable. There has to be a better solution.
4. Using a custom shader material, along with a data-texture. We are able to use the WebGL-function texture2D on a "uniform sampler2D dataTexture", along with "texture2D(dataTexture, vec2(col, row))" to select the color. Each pipe segment will be given a row (based on measuredepth) to fetch its color from. A time uniform will be used to pick the column of the texture/image and set the color. This image is pregenerated. Loading pipepressure.csv to perform the thing is still really slow.

    - Images are also compressed based on the format, allowing for faster load times in the client. Comparatively, the image is (150kB?) instead of the (8MB?). This offers a massive load time improvement.
5. Since only the first (with headers) is needed from pipepressure.csv when the images are pregenerated, we can also pregenerate a json file with these values in a list. This significantly improves loading time. It still takes around 6-7 seconds for the 3d-view to load on mobile. This is in part to the cpu-intensive work of calculating the positions of all the pipe segments. We also seem to need some specific pregenerated files - and it is not easy to switch to a different scenario. How could this be solved?
6. We will now create an API in Python (since this is what we used to pregenerate the images, and pandas is good for working efficiently with large datasets). Using FastAPI, we can expose Python as a REST-api, where the client can access simulation data. Computing all the positions, along with the simulation images is now the job of the API. Since pandas is written in C, and this is utilized, we get a massive performance boost when precomputing values. We can now easily extend the client to simply get all the information it needs from the API, and responsible for as few precalculations as possible, improving the time it takes for the application to become interactive.
    - We now only load the pipe segments that are actually going to be drawn (radius > 0). The data from the different files is merged in the api before it is being sent to the client.

NOTE: Should absolutely make measurements from the different stages to compare performance! This is critical.

Guiding optimization principles

TODO: Add references for each of these statements

- Focus on making something work before improving performance. Premature optimization is the root of all evil.
- Always measure performance and find the bottlenecks before deciding what to optimize. Don't spend time optimizing something that isn't a bottleneck.
- Bandwidth is usually the bottleneck of any web application. We generally want to minimize the amount of data sent over the internet.
- Computations on a server is generally faster than in light clients. Especially when it comes to mobile clients.
- Moving a lot of data between the CPU and the GPU can be expensive. Minimize the amount of data moved in the main render loop.
- The GPU is a lot better at parallelization than the CPU. If you want to perform a lot of operations for every cycle in the render loop, it might be cheaper to do it on the GPU than on the CPU. (especially if it doesn't entail moving a lot of data to and from the GPU in the main render loop)

User guide

Desktop/Laptop (Chrome, FireFox, Edge, Internet Explorer)

You have access to more features on desktop (through use of keyboard)

control	description
+ *(keypress)*	Increase playback speed by 1x
- *(keypress)*	Decrease playback speed by 1x
Q *(keypress)*	Set pipe material to "Basic Material" (no light-model: default)
W *(keypress)*	Set pipe material to "Standard Material" (realistic lightning)
Scroll up	Zoom in

control	description
Scroll down	Zoom out
Hold left mouse down/drag	Rotate scene
Hold right mouse down/drag	Translate scene along x or z-axis
Click or drag timeline	Change time
Click timestamp	Play/pause time (button is red when paused)

Mobile controls

control	description
Drag with one finger	Rotate scene
Drag with two fingers	Translate scene along x or z-axis
Pinch with two fingers	Zoom in/out
Touch/drag on timeline	Change time
Touch timestamp	Play/pause time (button is red when paused)

Further work

- Improve the colors
- Center 3D-model in view
- Add menu/UI to select a specific Well and Connection
- Add password authentication
- Interactive 3D: Click on pipe segments to get more information
- Add some numeric information to the side (color scale for pressure)
- Discrete colors (warnings and alarms)

References

Cross-Browser testing

The application has been manually tested in the following web browsers. There is no guarantee that the application will work in other/older browsers.

Added polyfills script tag. This might have fixed some problems?

Safari on iOS13 (iPhone 6) and iOS11 (iPad Air 2)

- View height units are a nightmare: https://nicolas-hoizey.com/2015/02/viewport-height-is-taller-than-the-visible-part-of-the-document-in-some-mobile-browsers.html
- https://stackoverflow.com/questions/37112218/css3-100vh-not-constant-in-mobile-browser
- https://css-tricks.com/the-trick-to-viewport-units-on-mobile/
- THREE.WebGLRenderer: EXT_frag_depth extension not supported.
- THREE.WebGLRenderer: Image in DataTexture is too big (4351x170) (Well_1)

Microsoft Internet Exlorer 11 (on Windows 10) (This is unlike any pain you've experienced before)

- Input slider styling is an absolute nightmare. 3d rendering is also not working. CSS variables are not supported, so we will be using SASS https://css-tricks.com/styling-cross-browser-compatible-range-inputs-css/
- babel transpile node_modules dependencies: https://github.com/parcel-bundler/parcel/issues/1655

Microsoft Edge (on Windows 10)

- Does not support the use of async iterators :(

Google Chrome (version 75)

Mozilla Firefox (version 70)