Report for assignment 3

This is a template for your report. You are free to modify it as needed. It is not required to use markdown for your report either, but the report has to be delivered in a standard, cross-platform format.

Project

Name: Three.js

URL: https://github.com/mrdoob/three.js/

Three.js is an easy to use, lightweight, cross-browser, general purpose 3D library.

Onboarding experience

Three.js was easy to build. The usages of the three.js project and it's internals are well-documented online, but most (if not all) units of code in the repository lacks comments.

The three.js project was easily built in it's entirety without errors using the build command `npm run build`. All examples used for testing was also easily built using `npm run build-examples`. The project was built using rollup.

After installing extra dependencies in the `<PROJECT>/test` directory, the unit-tests can be ran. Running e2e tests with `npm run test-e2e` requires chrome or a similar browser, since e2e tests use puppeteer which is running a headless chromium browser instance.

Complexity

1. What are your results for ten complex functions?
    - Did all methods (tools vs. manual count) get the same result?
    - Are the results clear?

First we counted the cyclomatic complexity of functions in Three.js using lizard. The tool produced these results:

```
LOC  CCN  Name
301  146  WebGLProgram@382-888@three.js/src/renderers/webgl/WebGLProgram.js
153  119  parse@54-295@three.js/src/loaders/MaterialLoader.js
137  110  convert@7-253@three.js/src/renderers/webgl/WebGLUtils.js
138  107  toJSON@161-385@three.js/src/materials/Material.js
170  88   setProgram@1432-1756@three.js/src/renderers/WebGLRenderer.js
164  69   parseObject@711-1061@three.js/src/loaders/ObjectLoader.js
168  59   uploadTexture@661-1004@three.js/src/renderers/webgl/WebGLTextures.js
118  57   getProgramCacheKeyBooleans@351-474@three.js/src/renderers/webgl/WebGLPrograms.js
120  44   toJSON@629-869@three.js/src/core/Object3D.js
148  40   addShape@67-685@three.js/src/geometries/ExtrudeGeometry.js
```

We decided to count the cyclomatic complexity of the bottom 2 functions by hand:

```
Adam:
CCN   Name
40    toJSON@629-869@three.js/src/core/Object3D.js
39    addShape@67-685@three.js/src/geometries/ExtrudeGeometry.js

Zino:
CCN   Name
40    toJSON@629-869@three.js/src/core/Object3D.js
37    addShape@67-685@three.js/src/geometries/ExtrudeGeometry.js
```

We did not get the exact same results, but we came quite close. We agreed on what to count and not to count. For instance, we count `if`s, and `else-if`s, as well as `for`, `while` and other control flow statements. We do not count inline function definitions, as we could not find enough information on whether these are useful to include, and their code is not executed as part of the run of the main function.

The results of the tool are quite similar, but we might have missed some branches that should actually be included, or the tool is including something it shouldn't. We were unsure about wether to add inline function definitions. Adding these would increase the CCN more than the current difference between our measurement and the measurements of the tool.

2. Are the functions just complex, or also long?

One of the functions (`addShape@67-685@three.js/src/geometries/ExtrudeGeometry.js`) is very long and also has a large CCN. Most of the complexity come from loops, which are manipulating vertecies in the shape. The other function (`toJSON@629-869@three.js/src/core/Object3D.js`) generally has very small branches (many oneliners/ternary expressions) and is therefore not as long, while having a similar CCN.

3. What is the purpose of the functions? Is it related to the high CC?

The purpose of `toJSON@Object3D.js` is to convert a 3D object to JSON, which means that the function is testing all properties of the 3d-object and applying them to a json object, and so naturally this function has a large amount of if-statements and thus CCN for its LOC.

The purpose of `addShape@67-685@three.js/src/geometries/ExtrudeGeometry.js` is to create a data object for applying transformations. Most of the complexity of this function comes from iterating through these vertecies in order to create this data object.

4. Are exceptions taken into account in the given measurements?

No try-catch blocks were found in these functions, but they are supported by the JavaScript programming language. The documentation for the code complexity tool (lizard) is not very clear on whether try-except blocks are included in the CCN, although they probably should.

If we think of an exception as a possible branch, the ammuont of possible branches should scale linearly with the number of lines of code which are able to raise exceptions.

5. Is the documentation of the function clear about the different possible outcomes induced by different branches taken?

No, the functions are generally not very clear about the possible outcomes by different branches. Some more complex branches have explanations, while simpler branches are self-explanatory. Most branches of medium complexity remain undocumented, and it is not immediately clear what happens if any branch is or is not taken.

Coverage measurement & improvement

DIY coverage measurement is done in the `f/diy` branch.

DIY coverage measurement results

Object3D.js

taken branches: `33`

taken branches percentage: `44.59%`

IDs of taken branches:

```
[1, 3, 5, 7, 9, 12, 14, 15, 18, 22, 38, 42, 48, 49, 50, 2, 4, 6,
8, 10, 16, 52, 56, 74, 57, 59, 61, 63, 65, 67, 69, 71, 73]
```

IDs of *not taken* branches:

```
[11, 13, 17, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33, 34, 35, 36, 37, 39, 40, 41, 43, 44, 45, 46, 47, 51, 53, 54,
55, 58, 60, 62, 64, 66, 68, 70, 72]
```

BufferGeometryLoader.js

taken branches: `20`

taken branches percentage: `48.78%`

IDs of taken branches:

```
[2, 3, 5, 7, 9, 11, 13, 15, 27, 29, 33, 34, 35, 39, 41, 4, 10,
12, 14, 37]
```

IDs of *not taken* branches:

```
[1, 6, 8, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 30, 31,
32, 36, 38, 40]
```

Quality of the DIY coverage measurement

Quality

The quality of the coverage measurement is both objective and subjective. Things like ternary operators are hard to measure as only one expression is allowed within a ternary expression. We replaced ternary expressions with `if-else` clauses, that way we were able to mark branches as reached. Exceptions could be taken into account in a similar way, by adding one flag-ID operation after every exception-inducing line in the `try-clause`, but fortunately the functions that we chose to measure did not include any `try-catch` clauses. We also take into account that a function could be called by other tests than its own main test, and therefore the entire test suite might increase branch coverage. We do this by keeping the reached branches in a globally reachable array, and letting the *entire* test suite finish before printing out the contents of the array.

Limitations

The limitations of our program is that it is completely manually done, and we could therefore miss things that an automated program would not. If the program is changed, by for instance changing where branches appear or adding new branches, our coverage measurement would be wrong as the amount of branches and the flagging of reached branches have been manually inserted. To allow for changes in the program while keeping coverage measurement functional, we would need to automatize the flag-insertion process.

Coverage Improvement Results

`Object3D.tests.js` as well as `BufferGeometryLoader.tests.js` in the branch `f/coverage-improvement` contain the changes to improve coverage. Four additional tests have been added for each function, which are clearly marked with comments above the added unit tests. To find them, simply navigate to the test file in question and search for `// added test #`.

Number of test cases added: four per member (P+).

Shared setup has been broken into functions in the test files, as adding more unit tests for these kinds of functions usually means a lot of repeated code.

Object3D.js

taken branches: `44` (improved from `33`)

taken branches percentage: `59.46%` (improved from `44.59%`)

IDs of additionally taken branches: `[11, 13, 17, 39, 41, 43, 44, 45, 31, 37, 58]`

BufferGeometryLoader.js

taken branches: `28` (improved from `20`)

taken branches percentage: `68.29%` (improved from `48.78%`)

IDs of additionally taken branches: `[16, 18, 26, 30, 32, 38, 40, 28]`

Refactoring

Plan for refactoring complex code:

The functions in the Three.JS project all follow a similar pattern. They are too long, and have a lot of nested branches and functions. The plan is therefore to break out functionality into functions where possible in order to reduce the length of the functions, and ultimately the CCN.

Refactored functions@files

```
parse@BufferGeometryLoader.js
toJSON@Object3D.js
toJSON@Material.js
parseObject@ObjectLoader.js
```

The plan was followed exactly, these functions could all be simplified by breaking out parts that were clearly modularizable. In some cases, the CCN could be reduced dramatically, as tens of functions were created instead of having all the same statements in one single function.

Refactoring CCN improvements

```
parse@BufferGeometryLoader.js went from CCN=23 to CCN=6 (65% improvement)
toJSON@Object3D.js went from CCN=44 to CCN=16 (63% improvement)
toJSON@Material.js went from CCN=107 to CCN=28 (73% improvement)
parseObject@ObjectLoader.js went from CCN=69 to CCN=29 (57% improvement)
```

Statement of contributions

Adam: report, project selection, complexity measurement, pair programming branch coverage,

Zino: report, pair programming branch coverage, refactoring

Self-assessment: Way of working

The current state according to the Essence standard is “In Place”. We have thus progressed from “In Use” to the next level, since all team members are now part of using the way-of-working.

To improve our way-of-working, we need to more consistently apply the way-of-working in a way that fits all team members. That means communicating more clearly when to do work, and which work to do. Although team members can now be trusted to do their work, it is not certain *when* they will do it. This means we need to adapt the way-of-working while the work progresses and reevaluate tools and communications to better fit us. Currently, we still “think about” the practices, which one should not actively do in the “Working Well” state.

Overall experience

The main take-away from this project is that a large project with many dependents can very well exist and work well without full test coverage. While three.js has OK coverage, most of the tests are lazily made and do not test functionality properly and seem to be made to pass.

Although we managed to write the tests for P+, testing is hard and often requires deep knowledge about the project, which we did not have.

P+

- We have carried out the refactoring (see `f/refactoring`)
- We have written four new unit tests each (see `f/coverage-improvement`)
- We are using issues and systematic commit messages (linked to issues) to manage this project