Energy and time performance comparison between JavaScript and C++ engines

Monday 5th June, 2023 - 21:34

Leonint Amarantos
University of Luxembourg
Email: leonint.amarantos.001@student.uni.lu

This report has been produced under the supervision of:
Alfredo Capozucca
University of Luxembourg
Email: alfredo.capozucca@uni.lu

Abstract

For the last 50 years, computers enjoyed enormous performance increases thanks to the decreasing size of transistors - a trajectory called "Moore's law". However, as processing units become smaller and draw more power, they become harder to cool. It is now all the more important to analyze on a software level how different technologies affect the power draw of the processors. Therefore, this paper analyzes the effect on energy and time performance of different languages (JavaScript and C++). On the technical side, this paper is responsible for developing fair testing environments and collecting test data. The scientific part is then focused on the analysis and explanation of the differences (if any) in the collected results.

1. Introduction

The following paper seeks to improve understanding of the effect of different programming languages on the energy and time efficiency of software applications.

This is done in the paper through a comparison of two applications in the two modern and popular programming languages JavaScript and C++. Although the two chosen languages fit their own roles and niches and don't usually compete with each other. C++ - a powerful and robust low-level language used for games, high-performance computation, and software development. JavaScript - multidimensional tool for web and server-side development. It is an even more valuable comparison, due to their design choices and trade-offs.

The two deliverables produced with this paper are: technical - perform the energy and time comparison including developing and setting up the engines and automating the data collection, and scientific - analysis of the two applications based on the results collected in testing data.

2. Project description

2.1. Domains

2.1.1. Scientific domains. Software analysis - A field that focuses on examining behavior, performance, and functionality of the software, done through testing, statistics, static and dynamic analysis. Which is then used to detect vulnerabilities, performance pitfalls, bugs, etc.

2.1.2. Technical domains. Programming languages - A field that encompasses syntax, semantics, and pragmatics that allow for the software development of various applications. Sets of the described components then make up languages that fit different roles due to trade-offs taken.

2.2. Targeted Deliverable

2.2.1. Scientific deliverable. The aim of this deliverable is the analysis of the two languages and the two engines written on them. Even it is not so much important the result of the comparison due to their different use cases, but the analysis and understanding and explaining why the differences are.

The analysis is performed through the following steps: analysis of the languages themselves and their expected performance, looking at the state-of-the-art in the field, and statistical analysis recorded through the use of tools and data developed and collected in the technical deliverable.

2.2.2. Technical deliverable. The focus of this deliverable is to create a fair environment for data collection of energy and time performance differences between the two chosen languages JavaScript and C++ engines.

This firstly involves the development of the two test subjects - 3D engines in JavaScript and C++. It is important that the developed engines are to be as similar as possible if not identical to ensure fairness. This is ensured with matching architecture apart from language-specific advantages, identical methods of computation, as well as the use of alternative rendering APIs WebGL (JS) and OpenGL (C++).

The developed subjects are then set up in a fair environment, suitable for performance testing.

3. Pre-requisites

3.1. Scientific pre-requisites

- Strong knowledge about qualities and behavior of different programming languages (at least C++ and JavaScript) and software architecture. Terminology like compiled, interpreted, strict typing, and dynamic typing is a must.
- Sound understanding of mathematics, in particular, application of normal distribution and statistical tests like the Shapiro-Wilk test for normality, T-test, and Wilcoxon signed-rank test of significance.

3.2. Technical pre-requisites

- Minimal prior experience with C++
- Strong experience with the OOP (Object-Oriented Programming) paradigm.
- Good understanding of the inner workings of graphical engines and graphical API (WebGL, OpenGL).
- Strong experience with JavaScript and Python
- Minimal experience with Unix-based operating systems and shell scripts.

4. What are the energy and time performance differences between JavaScript and C++-based applications

4.1. Requirements

- **Terminology**: present vocabulary and terminology needed for a better understanding of the topic.
- **Compare JS and C++ engines**: present a sound comparison of the JavaScript and C++ engines that includes real-world data and its analysis.

4.2. Design

4.2.1. Design differences. To answer said question it is first important to explain and give an understanding of what the two languages are, their characteristics, and most importantly for the question the differences. The following are the characteristics that directly affect the performance of programming languages that are covered in the paper:

- Compiled vs Interpreted
- Dynamic vs Static
- Garbage collector vs Manual memory management

4.2.2. Stating hypothesis. The next step before performing the experiment is to state the hypothesis, whether there are no differences between JS and C++, or whether one is more performant than the other. The hypothesis that is to be stated must firstly be based on the differences in the key characteristics of the JavaScript and C++ languages that are discussed in the Design differences. Secondly, it is important to back up the claims with the existing state-of-the-art [1] [2] research in the field.

4.2.3. Data collection. The comparison of the engines is done by running a set of predefined tasks that are further referred to as scenarios.

The data that is being collected in the experiment are:

- Execution time (total).
- Energy consumption (CPU + RAM only)

To perform sound and complete comparisons the defined set of scenarios must be representative of the majority of the use cases/performance loads. Therefore at last three scenarios must be defined: trivial (low load), viable (medium load), and one that is pushing the limits (heavy load). Also, scenarios must be made sure to stress a multitude of hardware components and not be bottle-necked.

The scenarios are defined on a visual basis from the perspective of the user, where the medium load is acceptable to a target user's visual fidelity with emphasis on video stutters and tearing. High alternatively must reach the point of visually unacceptable with noticeable stutter and frame delay.

The experiment must be run repeatedly in order to reduce external interference and error in the experiment.

The data from the experiment is organized in a single .CSV file containing the following fields:

- Engine - what engine the experiment was run on, JS for JavaScript and CPP (to avoid special characters) for C++ engine.
- Scenario - scenario run, low for low load, mid for medium load, and high for heavy load.
- Time - execution time of the scenario in seconds.
- Energy - total energy consumed to execute the scenario in joules.

The following is an example demonstrates a possible structure of the .CSV file:

Engine	Scenario	Time	Energy
JS	low	val	val
JS	mid	val	val
C++	low	val	val
...	...	...	...

TABLE 1: Data structure example

4.2.4. Statistical Analysis. To prove or disprove the hypothesis the claims must be challenged by the experiment data. In order to correspond to the question and the hypothesis the analysis must present the following:

- Quantitative difference between execution time and energy consumption of C++ and JS engine.
- Confirm normal distribution of data with a goodness of fit test.
- Confirm the significance of the resulting difference.

A simple and effective way to quantitatively compare the scale of the difference between the engines is to compare the mean values between the two engines for each of the scenarios. This can be visualized well on a bar plot, with energy/time on the y-axis and scenarios on the x-axis. The bar plot could also show the correlation between energy consumption and the execution time for engines.

There are two goals in confirming the normality of the data. First, check the quality of the data, a non-normal distribution would indicate that there is a large external interference on the tests. In particular, a skewed to the right data would indicate interference of test repetition in between each other as the hardware does not cool down enough between different runs.

The second goal is to be able to determine the significance test that may be performed, as some tests are better suited for distributed normal data than others. A two-tailed T-test will be used under the assumption that the data is distributed normally. However, if the data turns out to be skewed from normal a Wilcoxon signed-rank test will be performed as it is much better suited for non-normal data. The goodness of fit test used is Shapiro–Wilk test as it can handle tailed data, which is in this case. The following are the null and alternative hypotheses for the tests:

Shapiro-Wilk test [3]

H_0 - The data distribution is not different from normal.

H_A - The data distribution is different from normal.

T-test and Wilcoxon signed-rank test [4]

H_0 - The true difference between the group means is zero.

H_A - The true difference between the group means is not zero.

For both of the tests, a P-value below 0.05 is considered sufficient to reject the null hypothesis.

4.2.5. Conclusion. The conclusion must bring all sections of the scientific deliverable together and explain the observed results. The result of the statistical analysis must be compared with the hypothesis and the arguments for it. In case the hypothesis is not confirmed the data must be looked at for suggestion of a reason.

4.3. Production

4.3.1. Compiled vs Interpreted languages. Before a piece of code written in a human-readable language (source code) described by semantics and punctuation can be run by a computing unit it needs to be translated into a machine-readable one (machine code). Compilation, Interpretation, and JIT compilation (Just-In-Time compilation) are some of the methods, that perform the described translation. The following will describe said methods including their benefits and disadvantages.

Compilation Performs the translation of the source code once involving the entire project before it can be executed. The result of the compilation is a binary file that can be directly read and run by a machine.

Interpretation Preforms the translation at run-time, the interpreter reads, translates, and executes the code line-by-line.

JIT compilation is an approach that combines compilation and interpretation together. The code is run by an interpreter, however, some of the code, for example, a loop is compiled just before its execution and executed by the interpreter.

Benefits and drawbacks Compiled code is expected to consume less energy and execute quicker as translation is already performed upfront once. A well-built compiler will also perform optimization on basic and common features. However, the compilation also means that any changes in code would need to be compiled again to take effect. Also, a compiled program is platform-specific as the platform resources like system libraries are not updated when the code is executed on a different machine, making it importable.

Interpreted code runs slower as translation is performed during runtime. However, contrary to the compiled code, it is highly portable and platform-independent as issues are resolved at runtime.

JIT lands in between in terms of performance significantly outpacing interpreters in recurring scenarios, but not as quickly as compilers.

4.3.2. Static vs Dynamic. Static and dynamic typing refers to whether or not the type of a variable is known at the compile-time or run-time. With static typing, a type of a variable is set when the program is compiled and then is unchanged at any point of the execution, with dynamic the type can be changed and reassigned. The benefits of statically typed language are a highly structured and predictable code and also more efficient execution as a type that is dynamic must be double-checked and if needed translated when executing. Dynamic typing allows for more flexible code but is less efficient.

4.3.3. Garbage collector vs Manual memory management. Memory management is a task of allocating memory and freeing memory, when a variable is created it must be stored in space, but when it is not needed it needs to be deallocated in order to make room for more. Garbage collector refers to a built-in language memory manager that attempts to optimally perform said task. With manual management, this task is performed by a programmer. A benefit of manual memory management is higher control and therefore possible optimizations. A well-managed memory would not only mean lower RAM (random access memory) consumption but also in cases with high memory demand it might avoid the memory to be fully utilized and need to be cashed and stored in a buffer image on a local drive, which is costly to use and is later slow to retrieve. Manual management puts the responsibility onto the programmer to avoid memory leaks (when a memory is being permanently occupied, but it is not needed or possible to use). However, in trivial cases like primitive data types, a garbage collector will perform just as well as a manual and will not avoid leaks without burdening the coder.

4.3.4. Hypothesis. Firstly the following table 2 shows the design choices made for C++ and JavaScript for described above design differences:

A note on C++ memory management, while it does have a built-in garbage collector, it is only responsible for "default"

C++	JavaScript
Compiled	JIT
Static	Dynamic
Both	Garbage collector

TABLE 2: C++ vs JavaScript design differences

cases like primitive data types and structures, but not for collections, complex classes, etc.

The expectation is clear from the presented differences in the design of the languages that C++ is made to be highly computationally efficient, whereas JavaScript is made to be portable and dynamic. C++ is even used for computation in other languages like the NumPy library in Python, TensorFlow, etc.

The following table 3 presents data concerning only C++ and JS applications that are taken from benchmarks performed in the following study [1], it shows C++ be 5-19 times faster and 2-7 times more energy efficient:

Algorithm	C++ Energy (J)	C++ Time (ms)	JS Energy (J)	JS Time (ms)
Binary-tree	41.24	1129	312.14	21349
Fannkuch-redux	219.89	6123	413.90	33663
Fasta	34.88	1164	64.84	5098

TABLE 3: C++ and JavaScript performance comparison [1]

Combining discussed reasoning, the hypothesis is that the C++ engine is going to outperform the JS engine in both energy and time efficiency.

4.3.5. Data Collection. The experiment is run on a Debian 11 OS in a machine that is equipped with:

- CPU - Intel® Core™ i9-7900X CPU @ 3.30GHz × 20
- GPU - 2xGeForce RTX 2080 Ti Rev. A
- RAM - 8x16 DDR4 2400 MHz
- DISK - HDD 7200 RPM

The two engines each are executed on a single CPU thread in a "host-server" relationship with the GPU through the use of OpenGL [5] and WebGL [6] API. The CPU performs preliminary rendering operations such as positioning and supplying data to the buffers. The GPU then synchronously performs the rendering of the supped data, taking the 3D picture projecting it on a 2D camera plane, and calculating the RGB values for individual pixels.

The execution scenarios were defined using Blender 3.4 3D software [7]. The defined scenarios for the experiment have the following characteristics:

Scenario	Vertices	Portals	Animated objects	Lights
low	720	12	19	16
mid	1088	24	29	16
high	4320	96	109	16

TABLE 4: Independent variables for defined scenarios

The experiment was run with a repetition of 100 times for each scenario for each engine with 60 seconds pauses in between the runs to cool down. The total duration of the experiment was about 13.5 hours.

4.3.6. Statistical analysis. The statistical analysis is performed using the R language [8] with the help of RStudio. The following describes the R code used 9:

- **Lines 1-7** - Loading and ordering the CSV data.
- **Lines 12-32** - Performing Shapiro-Wilk goodness of fit test. The normality_test function accepts three arguments d the data to test, engine - the engine ("CPP" for C++, "JS" for JavaScript) and the variable that is selected ("Time" or "Energy"). Line 19 performs the test on the selected engine and variable for each of the defined scenarios. The rest of the function constructs the tables from the test results. On lines 29-32, the function is called for each engine/scenario combination.
- **Lines 37-66** - Performing significance test. The significance_test function accepts three arguments d - the data to test, the variable that is selected ("Time" or "Energy"), and testChoice - the significance test to be performed. The Wilcoxon signed-rank test is used if the data is normally distributed and the t-test if the data is not, the argument for the function is then "wilcox" and "t" respectively. The normality_test function on lines 44 - 51 performs the test for each of the scenarios comparing the JS engine with the C++ one.
- **Lines 79 - 92** - Construction of bar plots for the loaded data, using the ggplot2 library [9] .

Normality test - Energy results. The table 5 shows that some of the energy data is not normally distributed as the P-value for low, mid of C++ scenario and mid of JS engine is above 0.05.

Scenario	W	P
low	0.979	0.109
mid	0.982	0.207
high	0.954	0.001

(a) C++

Scenario	W	P
low	0.841	5.406e-09
mid	0.980	0.141
high	0.900	1.387e-06

(b) JavaScript

TABLE 5: Energy Shapiro–Wilk

Normality test - Time results. The table 6 shows that some of the time data is not normally distributed as the P-value for the low and mid scenario of C++ and JS engines respectively is above 0.05.

Scenario	W	P
low	0.986	0.371
mid	0.968	0.017
high	0.949	0.001

(a) C++

Scenario	W	P
low	0.698	4.900e-13
mid	0.983	0.228
high	0.916	8.458e-06

(b) JavaScript

TABLE 6: Time Shapiro–Wilk

Goodness of fit test results. As it was concluded that some of the data is not normally distributed, therefore the

Wilcoxon signed-rank test is used. The result table 7 shows values below 0.05 for all of the data points, therefore the H_0 is rejected meaning that there is a significant difference between the group means of the data.

Scenario	W	P
low	0.000	2.562e-34
mid	0.000	2.562e-34
high	10000	2.562e-34

(a) Time

Scenario	W	P
low	0.000	2.562e-34
mid	0.000	2.562e-34
high	9986	3.899e-34

(b) Energy

TABLE 7: Wilcoxon test JS vs C++

Data bar-plots. The following data plots, 1, 2 show the data collected for C++ and JS engines against each other in red and blue respectively. The plots show that the C++ engine consumed 7% and 35% less energy for the low and medium scenario respectively, however, it consumed 10% more for the high scenario. Concerning execution time, C++ has run 62% and 26% quicker for low and medium scenarios respectively, although it ran 20% slower for the high scenario.

The two plots show a very apparent similarity, which is to a degree expected as the longer experiment is run, the more energy is to be consumed. Also as was stated before the program runs on a single core in a "server-client" relationship with the GPU synchronously. It is important that the execution is synchronous as that means that the used core is actually occupied or is busy 100% of the time as it awaits the response from the GPU that it has finished the rendering process. And as the core is constantly busy the energy consumption grows proportionally to the time for the CPU.

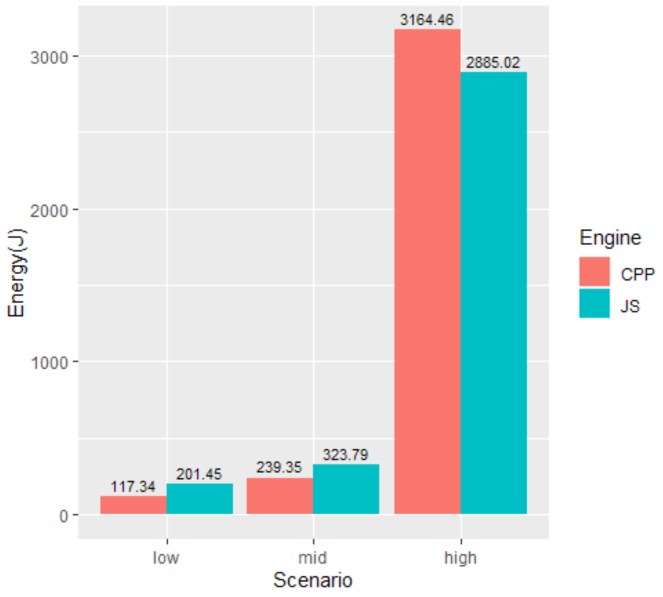

Fig. 1: Energy consumption JS vs C++

4.3.7. Conclusion. The collected data has not entirely supported the provided hypothesis. Moreover from the results,

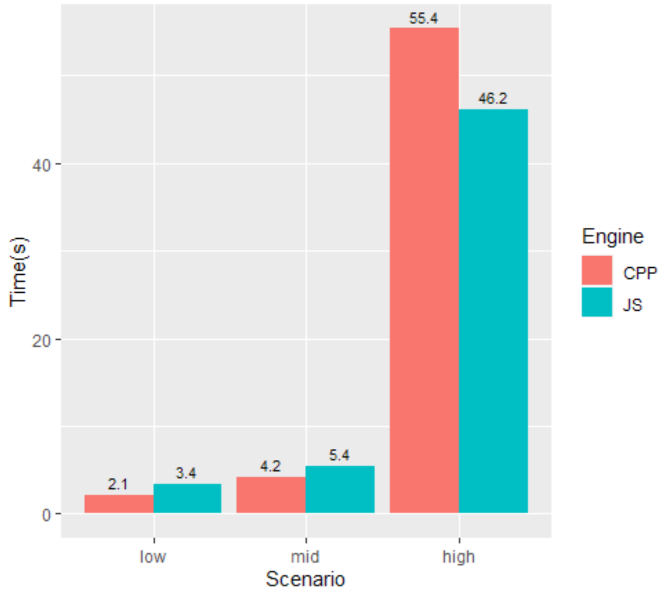

Fig. 2: Execution time JS vs C++

it is possible to conclude that the JavaScript engine is more performant at run-time as on a longer test (high) it outperforms the C++ engines after being behind on the start-up (low, medium).

C++ engines perform significantly better on the low and medium scenarios which did follow the hypothesis that was made. As was stated C++ as a compiled language has a significant start-up advantage having made a lot of work prior to the execution, whereas JavaScript must perform translation to machine code on the fly. This shows on both of the graphs, for energy and for time.

4.4. Assessment

The scientific deliverable has completed the laid-out requirements. Needed for comparison terminology was presented, and the comparison process was streamlined, performed, and described in detail.

However, the given hypothesis was not accepted, at least not entirely, and the following descriptions show some of the improvements that would be very useful to implement for better insight:

- The energy consumption of the GPU was not recorded during the experiment. First, it is possible that the JavaScript engine has been using hardware acceleration available in browsers and consumed energy on the GPU that was not accounted for, changing the results. The observed performance was only measured from the "client" side (CPU). However it is unknown from the results of this experiment as to what has been happening on the "server" side (GPU) and as the following [2] clearly shows in Fig. 10 that rendering is a very significant task in terms of execution time, depending on the implementation is potentially way larger than the "client"

side computation. Therefore with the CPU and GPU running synchronously, it is very difficult to be certain of the results without collecting energy consumption on the GPU side.
An alternative to the inclusion of the GPU energy consummation could be testing engines in an asynchronous manner, which however would imply the need for some changes in the design of the engines.

- Scenario definition is not perfect, from the data collection point it is usually it is wise to create definitions that are static and are scaled as magnitudes. Static meaning that the definition would not change from one machine to another, as was the case for this experiment with the definitions made on a visual basis which would be different if a different user or machine would have created the definition.
  A better proposition would be to define a base scenario which would be 1x and then scale it proportionally across all of the independent variables (portals, animated objects, etc.) that are set when defining the scenarios. Therefore the second or mid scenario could be 5x and the third or high could be 25x. This creates better consistency for the experiments.
- Not all of the resulting data has turned out to be normal. As by the central limit theory if we were to significantly increase the amount of data collected it would gradually converge towards a normal distribution.

5. Energy and time performance comparison between JavaScript and C++ engines.

5.1. Requirements

5.2. Functional requirements

- **Two Functionally Identical Engines**: The two developed engines must be developed in an identical manner in terms of their features. The architecture engines and the level of optimization must be as close as possible except for language-specific situations and advantages.
- **Automated Bench-marking**: Bench-marking process must be automated to a point where the user can define a multitude of test-execution scenarios and through an automated process run performance tests on it.
- **Data parsing**: Just like the Bench-marking the collected data must be parsed and collected in a meaningful analysis way.

5.3. Non-functional requirements

- **Fair Benchmark environment**: The environment of the execution must be as fair as possible. Meaning that both engines must be run on the same hardware over the same scenarios and bench-marked using the same methods.
- **Reproducible**: Steps from the data collection process must be reproducible following the exact steps described in the project from the given starting point. Cross-compatibility between platforms of similar architecture is not required but is an asset toward the success of the project.

5.4. Design

This section goes over the design choices taken in order to implement the targets described in the Requirements section.

5.4.1. Two Functionally Identical Engines. This section corresponds to the development of the C++ engine on the basis of the JavaScript engine. The source code for both of the implemented engines is available in a public GitHub repository [10], created for this project.

Starting point. As the starting point an existing 3D engine made for a prior project is taken with the following characteristics and features:

- Platform – Windows, browser-based (Google Chrome V8 [11])
- Language – JavaScript (main) [12], GLSL [5] (shaders)
- Rendering API – WebGL 2.0 [6] (based on OpenGL ES 3.0)
- Rendering – 3D perspective camera, PBR shader (physically based rendering), point lights.
- Assets – glTF2 [13] (scenes, PBR materials, meshes, objects, lights), OBJ (objects)
- Controls – keyboard and mouse support
- Other – Non-Euclidean objects (portals), collision detection (not resolution)

The engine is taken as is except for migration to Linux which allows for a cleaner bench-marking environment and automation of the data collection.

C++ engine. The choice was made to translate the JavaScript engine rather than take an existing one. Because it allows for a functionally and logically identical implementation to its JavaScript counterpart, which is much needed for a fair comparison between the two. With features defined as to be identical to the one described for the first engine, the following are the fundamental differences:

- Platform – Linux
- Language – C++ (main) [14], GLSL [5] (shader)
- Compiler – g++ (GNU)
- Rendering – OpenGL 4.2 [5]

The development process is split into the following stages to maintain a code structure that is testable at all points of the development:

- Setup - When using WebGL the window context and the linking of the API is handled by the browser. With OpenGL, that process is not given and a set-up process with a linking library is a part of the development. The chosen linker of OpenGL API is GLAD and the window context manager is GLFW for its ease of use and reliability.

- Math - Classes glMath, Matrix4, and Vector2, 3, and 4, are all responsible for representing geometric shapes present and performing operations on them, such as translation, rotation, cross product, etc.
- OpenGL helpers - Classes Shader, Mesh, Asset loader, and shaders responsible for the creation of window context, data loading, buffering, shader compilation, and any other part of initialization that takes part before scenes are displayed.
- Scene objects and portals - Classes SceneGraph, Transform, and implementations of Empty Object, responsible for organizing objects on the scenes and then passing the image to the GPU for rendering.

5.4.2. Automated scenario execution. This section corresponds to the creation of an automated way of defining testing scenarios and their execution. The two engines implement a portion of glTF2 specification [6]. Therefore, for the sake of consistency and simplicity to define testing scenarios the choice was made to implement a part of the specification that is referred to as Animation. Animation is a collection of values that can be recorded and then assigned to variables like translation or rotation during execution.

Refer to the following diagram 3 for a better understanding of the following explanation. Said animations are stored in a list of JSON-like objects in the glTF2 files together with other structures implemented in the JS engine [15][5.4.2 Data import]. Each of the animation objects contains a list of Channels and Samplers.

Samplers describe what and how the data is going to be used. The input field references the data containing a timestamp for each point of data and the output contains each of said points of data. The interpolation field defines what algorithm is used for interpolation (computation of data that is in between existing samples). The way arrays of binary data are loaded is common across all of the glTF objects. The accessor object is referenced by its index which describes data type and points to a buffer view, which in turn describes the length and position in the buffer of data and also points to the buffer used.

Channels describe how to use the data from a referenced sample using the target object. In the target object, the node field points to the index of the node (an entity that appears on a scene) that is targeted and the property of the node that is controlled by the animation.

The resulting implementation of the specification allows using available 3D modeling tools to create and record scenarios that are to be run for bench-marking. Therefore creation is a straightforward process of using a 3D tool of choice, the one used is Blender 3.4, and exporting to the engines in the glTF2 format. An important part is to make sure that the defined scenarios test the engines in a multitude of characteristics and are not bottle-necked by a single component.

5.4.3. Bench-marking. To what is referred to as benchmarking is a process of measuring and collecting performance characteristics for any arbitrary target, in a standardized manner, that can then be used for a cross-evaluation with a target of a similar subject, a broader and deeper discussion can be found in [16].

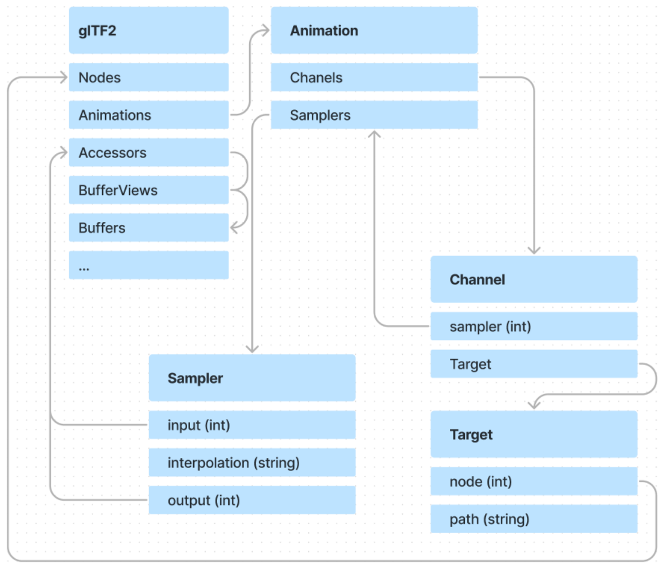

Fig. 3: glTF2 Animation structure, bold top label - data block, non-bold labels - fields, In bracket if exist - the primitive type of the field, otherwise is a list of object, arrow - a reference to an object

In this deliverable, the benchmarking targets are the JavaScript and C++ engines. The characteristics of their performance that are recorded are energy consumption and execution time. The energy collected in this deliverable is limited to the CPU and RAM. The measurements of these two are provided in the Running Average Power Limit (RAPL) [17] interface included with the Intel CPU used. The tool used for recording and querying these targets is perf [18] stat available for all common Linux distributions. The recorded values for the analyzer are energy consumption (CPU - energy/package, RAM - energy/ram, execution time). All of the benchmarking is performed on a single machine with an identical performance analyzer.

The benchmarking process is automated with a simple shell script. The script accepts several executable targets and iteratively launches N times the perf stat command in the directory of the target. Each iteration is followed by an N-second sleep to let the hardware cool down in order to reduce the interference towards following tests. It is important to increment the file names with every iteration so the results are saved in separate files. Otherwise, by default, all of the files would be overridden. Parameters like repetition count, sleep duration, and targets are kept separately in order to be able to change for the defined needs of the experiment.

After the experiment, the data is gathered and organized in a single file with a Python script. This is done separately from both bench-marking and the data analysis to keep the project modular and avoid interfering with the test.

5.5. Production

The main goal of the technical deliverable is to benchmark two engines written in C++ and JavaScript. Then using said, being to develop said engine (s) and the environment for benchmarking.

5.5.1. Targeted platform. Firstly the operating system chosen for this deliverable is UNIX-based (Debian 11). The reason for that choice is first it has a low performance overhead compared to other platforms like Windows and Mac OS, which reduces resulting on bench-marking interference. Also, a more open-style OS allows for deeper and more accurate results.

5.5.2. JavaScript engine. As the JavaScript engine was made before the beginning of this deliverable there are very few changes that are taken during the development of this project, however here are some that were introduced, mostly in the way the engine is launched for the purpose of easier and fairer bench-marking:

Engine.js

- **Lines 47-53** - A frame count function was added, as the benchmark scenarios are run in a loop for N frames.
- **Lines 55-68** - Private shutdown method that redirects to the local shutdown page and closes the active window. The server that the application is run on then handles the request. This function is run when the scenario is finished, so the perf stat can stop and store the recorded data.

server.py

- **Lines 7-11** - added support for flags that define scenarios that is to be run by the program. This makes it possible to call perf stat with –scenario "your scenario name" instead of changing the source code to choose the scenario.
- **Lines 13,14,50** - defined the launching environment as well as added flags passed to the browser to disable the limits it imposes on the running program. A defined environment enforces an equivalent environment when the experiment is reproduced. Disabled limits allow the engine to perform the best it can and not have a speed limit enforced by the browser, making it fair with the C++ engine.
- **Lines 26-30** - disabled caching, this makes sure that every run of the benchmark is as if the program is being launched for the first time.
- **Lines 32-41** - added handling of the server shutdown. Previously the server had to be shut down manually with either a keyboard-interrupt or by closing the running thread. Automating the manual process allows for automated testing.

5.5.3. C++ engine. As the works of the C++ engine are based on the previous implementation (JavaScript engine), this section of the report would only go over the nuances of the translation process. For a better insight on the engine itself refer to [15].

CMake [19] As discussed in Compiled vs Interpreted languages compiled languages are not very portable, before one can be compiled, all of the tools used for compilation must be found on a local machine. A tool that helps to automate such a process is CMake. A CMakeLists.txt must be included in a CMake project in order to build it. The file may contain libraries, directories, packages, as well as the source code and build options that need to be present and included. The following 1 describes individual commands used for the project.

- **Line 1** specifies a minimal version of CMake that is required for the build. This is mostly related to the version of C++ standard that the version of CMake supports, as well as some features of CMake that became relevant. A minimum version is set and not the used one currently, because the tool, as well as C++, is backward compatible but not entirely forward, therefore setting a minimum lets a greater range of devices compile the project.
- **Line 2** sets the name of the project.
- **Lines 4-8** describes what packages and modules are required and must be found for the project to compile and work.
- **Line 10** sets the C++ standard that is used in the project.
- **Line 12** sets the path to the compiler that must be used.
- **Lines 14 - 21** states which executable source files are a part of this project.
- **Line 23** specifies the directories that must be added to the project at compile-time.
- **Line 23** specifies the libraries that must be added to the project after it has been compiled.

```
1  cmake_minimum_required(VERSION 3.22)
2  project(OpenGLEngine)
3  
4  find_package(PkgConfig REQUIRED)
5  pkg_check_modules(JSONCPP jsoncpp)
6  pkg_search_module(GLFW REQUIRED glfw3)
7  link_libraries(${JSONCPP_LIBRARIES})
8  find_package(GLEW REQUIRED)
9  
10 set(CMAKE_CXX_STANDARD 20)
11 set(CMAKE_VERBOSE_MAKEFILE ON)
12 set(CMAKE_CXX_COMPILER "/usr/bin/g++")
13 
14 add_executable(OpenGLEngine
15         glad.c
16         GL_Helpers/AssetLoader.cpp
17         GL_Helpers/AssetLoader.h
18         Math/Vector2.cpp Math/glMath.cpp
19         Math/glMath.h GL_Helpers/Shader.cpp
20         ...
21 )
22 
23 target_include_directories(${PROJECT_NAME} PUBLIC ${
       GLFW_INCLUDE_DIRS})
24 target_link_libraries(${PROJECT_NAME} PUBLIC ${
       JSONCPP_LIBRARIES} PRIVATE glfw3 GL X11 pthread
       Xrandr Xi dl)
```

Listing 1: CMakdeLists

Headers files C++ classes are commonly split into two separate files: .h and .cpp. The header file (.h) is an interface-like structure that contains mostly non-implemented classes and attributes that a class will be used. The implementation class (.cpp) as the name suggests contains the implementations of said headers.

At this point in time, this architectural choice is rather more stylistic as the claimed benefits are not as relevant or significant in the modern day. For example, it could potentially reduce the amount of memory used during compilation time as files can be viewed by the compiler in smaller and more separate chunks.

The result is an architecture that uses a folder structure to organize the code like in the JavaScript engine, which however now includes two files instead of one for each class, as shown below.

- ...
- Math
—+ glMath.h
—+ glMath.cpp
—+ Vector2.h
—+ Vector2.cpp
—+ Vector3.h
—+ Vector3.cpp
—+ Vector4.h
—+ Vector4.cpp
—+ Matrix.h
—+ Matrix.cpp

Operator Overloading A feature included in the C++ language that allows overloading for classes just like method. This is very useful especially for this project due to the extensive use of mathematics in defined classes for matrices and vectors. The following 2 shows headers of the overloaded functions on the example of Vector2 class. Following 3 is the usage example, where multiple matrices are operated as numeric values, alternatively multiple nested functions would be needed to implement the same functionality.

```cpp
    ...
    Vector2 operator+(const Vector2 &b) const;
    Vector2 operator-(const Vector2 &b) const;
    Vector2 operator-() const;
    Vector2 operator*(const Vector2 &b) const;
    Vector2 operator*(const float &b) const;
    Vector2 operator/(const float &b) const;
    ...
```

Listing 2: Operator Overloading in Vector2 class

```cpp
Matrix4 Camera::relativeMirror(const Matrix4& from,
    Matrix4 to) {
    return this->
        viewMat() *
        from *
        (Matrix4::identity().rotateY(glMath::toRad
    (180))) *
        to.inverse();
}
```

Listing 3: Operator usage example in Camera class

OpenGL Most of the modern browsers naively support the latest version of WebGL API, which handles tasks like interaction with a graphical window (context management), loading and compiling shaders, etc. Therefore all of these are directly available in the JavaScript implementation. For C++ the same process has to be done manually, or with the help of existing established tools, a process described deeper here [20].

The first one is glad [21], a project for a multi-language generation of loaders for various graphical APIs. Said tool generates a glad.c file as well as system lib files that perform linking with OpenGL functions on the GPU. Second, is GLFW [22], which provides a context manager for rendering. The following code 4 describes the window initialization process. This static function is executed before the main rendering loop and returns a reference to a newly initialized window.

- **Lines 2-6** Initialize the instance of the GLFW library itself in a global manner, therefore it must be the first line of GLFW code being used. The following after "glfwInit();" are hints that are optional, but are recommended to be set explicitly, context version and profile refer to the version of OpenGL used in this instance 4.2 CORE. GLFW_DOUBLEBUFFER sets whether or not the program will be using an additional buffer that will be rendered to the next frame, while the original one would be used for displaying the current frame. Double buffering helps to parallelize the processing.
- **Lines 8-14** Create the new instance of a window with a passed in the parameters size of the window and the window title. Or in case of failure in the initialization process log an appropriate message.
- **Line 15** Sets the frequency of the vertical synchronization. Vertical synchronization is a method of avoiding visual artifacts called tearing by synchronizing the refresh rate of a monitor and the frame rate at which the GPU is rendering, resulting in frames being delivered as they are ready to be displayed, rather than in the middle of being drawn on a display. It is vertical as at least at the time displays drawn images in vertical strips one after another. The glfwSwapInterwal function accepts an integer that sets if the synchronization is off(0) or the number of screen updates before a new frame is fetched. In the case of this project, the value set in the configuration is 0, this is done to disable a variable factor from benchmarking - the refresh rate of a used monitor and also to allow the engines to run at their full and unrestricted speed.
- **Line 16** A function to be called on a resize of a window is passed that will manage said resize.
- **Lines 18-21** Loads glad functions and returns an appropriate message if it has failed to initialize.

```cpp
1 GLFWwindow *Engine::initGLFW(int screenWidth, int
      screenHeight) {
2     glfwInit();
3     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
4     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
5     glfwWindowHint(GLFW_OPENGL_PROFILE,
      GLFW_OPENGL_CORE_PROFILE);
6     glfwWindowHint(GLFW_DOUBLEBUFFER, GLFW_TRUE);
7 
8     GLFWwindow* window = glfwCreateWindow(
      screenWidth, screenHeight, "OpenGLEngine",
    nullptr, nullptr);
    if (window == nullptr)
    {
        std::cout << "ERROR::GLAD::
    FAILED_TO_CREATE_WINDOW" << std::endl;
        glfwTerminate();
    }
    glfwMakeContextCurrent(window);
    glfwSwapInterval(Config::glfwSwapInterval);
    glfwSetFramebufferSizeCallback(window,
    resizeWindow);

    if (!gladLoadGLLoader((GLADloadproc)
    glfwGetProcAddress))
    {
        std::cout << "ERROR::GLAD::
    FAILED_TO_INITIALIZE" << std::endl;
    }

    return window;
}
```

Listing 4: GLFWwindow function from Engine.cpp

5.5.4. Automated scenario execution. The implementation of the animation going to be demonstrated with the JavaScript version of it as the two are mostly identical, but the JavaScript one is spread across a smaller amount of source files. Also, the C++ engine uses the JsonCpp library [23] for parsing the original JSON files as such a feature is not natively supported in C++. Firstly the list of animations is parsed in the SceneGraph class together with other components in the glTF:

SceneGraph.py (only animation loading)

- **Lines 1-14** - All of the animations from the glTF file is be are being parsed as per 3 the data is extracted one after the other in the sampler is extracted first, function dataFromAccessor (described next) extracts the data from the binary file. Chanel class is created with the created sampler. Finally, the animator is created and pushed into the animations list inside of the SceneGraph.
- **Lines 16-26** - dataFromAccessor function extracts the data from the buffer that is linked in the glTF file following again this diagram 3. **Line 18** buffer, byteLenght, and byteOffset are loaded from the buffer view. Buffer refers to the index of the buffer to extract data. Buffer itself is a raw binary file, however, in the case of JavaScript, it is loaded earlier during runtime and is stored as a blob data type in a realized buffer type. Byte length refers to the length of the data that is copied from the buffer and byteOffset.

Animator.js

- **Lines 3-15** - Once the animator has been instanced the only role it has is to provide an interface that calls registered channels and then a sampler to perform needed transformations. In order to do so the animation function is called, it has a single parameter t, which is global time in seconds.
- **Lines 17-37** - The channel class serves as a connection between objects and data. It contains a set of available lambda mappings inside of the static path object if one would want to add other dimensions for animation ex. color change a new entry would be added below the translation, rotation, and scale. All of the objects like the paths object that contain mappings that are used at runtime are frozen (Object.freeze) to simulate immutability and disallow any unexpected changes. The node and sampler variables are the controlled object and the sampler provides the controlling data.
- **Lines 39-114** - The sampler class is responsible for actual computation that is performed for animations and controls the data to be passed.
- **Lines 40-46** - Static interpolationModes object contains lambda mappings of the interpolation functions that are available. The object includes the entries for all of the once available from the glTF2 specification, however, the CATMULLROMSPLINE and CUBICSPLINE interpolation modes do not perform any actions on the call as they were outside of the scope needed for the project.
- **Lines 88-113** - The interpolation functions described earlier. They all accept parameters start, end, and t. The start and end are the current sample point that a node uses and the next one in the data list. In this case, the t variable describes a fraction of the distance that must be traveled from the start sample to the end sample in order to get to generate the desired position.
  The lerp function stands for linear interpolation, therefore the function calculates a new sample that is at position t on a section of the line from the start sample to the end sample.
  The stepInrerp function stands for step interpolation and means no actual interpolation will occur, but the animation will not calculate an in-between sample and will use the next sample only as the timeline reaches it.
  The slerp function spherical linear interpolation is very similar to the linear one but instead of calculating a point on a flat line a line going along the sphere is used, in fact, a linear interpolation is used instead on lines 99 and 107 if the data becomes anomalous due to rounding errors or extremely small respectively.
- **Lines 65-85** - Function getOutput uses the globalT (global to the application time) to first find a start and end points in data (start - before the current time, end - after the current time) and returns a sample that is generated by a chosen interpolation function.

5.6. Assessment

The first requirement for the technical deliverable is two functionally identical engines, the C++ engine was implemented, and all of the features of the JS engine were used for benchmarking.

The JS engine contained some features that were not included in the C++ engine, due to them being at this point redundant or not needed for benchmarking. The first one was the support of OBJ files that became redundant with the introduction of GLTF objects. The second is a home screen that allows the selection of scenarios with the use of a user interface as it was in an unfinished state and was not needed for bench-marking and the process had to be automated and not user controlled. The details of said features are covered in [15].

Code structure A static description of the two engines is shown in the table 8. The engines have an equivalent number of effective (excluding blank lines and comments) lines of code, however, the C++ engine has significantly more files as most of the source code is split into .h and .cpp files, first containing the interface of a class and then the implementation.

Rendering Implemented rendering methods equivalent, except for the following notable difference. API used is WebGL for JavaScript and OpenGL for C++. JavaScript does explicitly support .GLSL files for shaders and therefore they are stored as constant strings, for C++ .GLSL files are used, otherwise, the GLSL code is equivalent. Lastly for the JavaScript engine shader attribute location are set before the execution of the main loop, requiring less costly calls to the GPU, which is not the case for the C++ version, where shader attributes are located again every time they are used.

Environment The only environment requirements for the JavaScript engine are a V8 engine built into Google Chrome(not only) and a Python 3 installation. As for the C++ engine, the C++ engine has a more laborious setup including the set-up of CMake, GLFW, GLAD, and g++ compiler.

Some of the other implementation characteristics are mentioned in the 8 table. In conclusion, the JavaScript and C++ engines were implemented very close to their possible similarity considering the specifics of these languages, with JavaScript leaning towards being a more portable one.

Feature	C++	JavaScript
Effective lines of code	1807	1761
# of source files	47 (.h + .cpp)	27
Paradigm	OOP	OOP
Rendering API	OpenGL	WebGL
Context manager	GLAD	V8 (Google-Chrome)
Shader storing method	separate .GLSL file	string
Pull shader attributes	on-demand	ahead of time
Linking	CMake	V8 (Google-Chrome)
Launcher	executable	python script & V8
Accepted assets	glTF2	glTF2 OBJ
Scenario choice	Hard-coded	Shell options or UI
Libraries	Jsoncpp GLFW GLAD	-
Compiler/Interpreter	g++	V8

TABLE 8: Implementation summary

6. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[2] R. C. Hoetzlein, "Graphics performance in rich internet applications," *IEEE computer graphics and applications*, vol. 32, no. 5, pp. 98–104, 2012.

[3] B. W. Yap and C. H. Sim, "Comparisons of various types of normality tests," *Journal of Statistical Computation and Simulation*, vol. 81, no. 12, pp. 2141–2155, 2011.

[4] R. Bevans, "An introduction to t tests | definitions, formula and examples," Jan 2020. [Online]. Available: https://www.scribbr.com/statistics/t-test/

[5] K. A. Mark Segal. (2012) A specification(version 4.2 (core profile) - april 27, 2012). [Online]. Available: https://registry.khronos.org/OpenGL/specs/gl/glspec42.core.pdf

[6] The Khronos™ 3D Formats Working Group. (2021) Webgl® 2.0 specification. [Online]. Available: https://registry.khronos.org/webgl/specs/latest/2.0/

[7] B. O. Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. [Online]. Available: http://www.blender.org

[8] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2021. [Online]. Available: https://www.R-project.org/

[9] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. [Online]. Available: https://ggplot2.tidyverse.org

[10] L. Amarantos, "Energy and time performance comparison between JavaScript and C++ engines," Jun. 2023. [Online]. Available: https://github.com/Lakedemon/BSP-2023-S2-Leonint-Power-performance-comparison-between-JS-and-C-

[11] V8 project authors. (2006) V8 javascript engine. [Online]. Available: https://chromium.googlesource.com/v8/v8.git

[12] ECMA International, "Standard ecma-262 - ecmascript language specification," 2022. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[13] The Khronos® 3D Formats Working Group. (2021) gltf™ 2.0 specification. [Online]. Available: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#foreword

[14] ISO, *ISO/IEC 14882:2020 Information technology - Programming languages — C++*, 6th ed. pub-ISO:adr: pub-ISO, Dec 2020. [Online]. Available: https://www.iso.org/standard/79358.html

[15] L. Amarantos, "Roomfininty - the biggest vr room in the smallest amount of physical space," 2023.

[16] T. Stapenhurst, *The benchmarking book*. Routledge, 2009.

[17] V. Weaver, "Linux support for power measurement interfaces," 2016. [Online]. Available: https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html

[18] "Perf firefox source docs documentation," 2023. [Online]. Available: https://firefox\protect\discretionary{\char\hyphenchar\font}{}{}source\protect\discretionary{\char\hyphenchar\font}{}{}docs.mozilla.org/performance/perf.html

[19] Kitware, *CMake*, Kitware, 2000. [Online]. Available: https://cmake.org/

[20] J. de Vries. (2014) Learnopengl. [Online]. Available: https://learnopengl.com/Getting-started/Creating-a-window

[21] D. Herberth, "Multi-language gl/gles/egl/glx/wgl loader-generator(glad)," 2013. [Online]. Available: https://github.com/Dav1dde/glad

[22] C. L. Marcus Geelnard, "Glfw," 2002. [Online]. Available: https://github.com/glfw/glfw

[23] Baptiste Lepilleur and The JsonCpp Authors, "JsonCpp," 2007. [Online]. Available: https://github.com/open-source-parsers/jsoncpp/

[24] "[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)."

[25] "[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021)."

[23]

7. Appendix

7.1. Images

7.2. Source Code

```
import {SceneGraph} from "./SceneGraph.js";

export class Engine {
    #timer = 0;
    #startTime;
    activeScene;
    #framesElapsed = 0;

    constructor(mainShader, scene, runForNFrames = 0) {
        window.gl = canvas.getContext("webgl2", {stencil:true});
        gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);

        this.activeScene = scene ?? new SceneGraph(mainShader);
        this.update = new Event("update");
        this.#startTime = Date.now();
        this.runForNFrames = runForNFrames;
    }

    start() {
        this.#renderLoop();
    }

    #renderLoop() {
        this.#resizeCanvas();

        this.#updateTime();
        this.activeScene.animate(this.#timer / 1000);

        this.activeScene.updateScene();
        this.activeScene.portalDraw();

        this.#updateFrameCount();
        window.requestAnimationFrame(this.#renderLoop.bind(this));
        window.dispatchEvent(this.update);
    }

    #resizeCanvas() {
        canvas.width = window.innerWidth;
        canvas.height = window.innerHeight;
        gl.viewport(0, 0, canvas.width, canvas.height);
        this.activeScene.mainCamera.resetProjection();
    }

    #updateTime() {
        this.#timer = Date.now() - this.#startTime;
    }

    #updateFrameCount(){
        this.#framesElapsed += 1;
        console.log(this.#framesElapsed);
        if(this.#framesElapsed >= this.runForNFrames && this.runForNFrames !== 0){
            this.#shutdownServer();
        }
    }

     #shutdownServer() {
        fetch('http://localhost:8000/shutdown')
            .then(response => {
                if (response.ok) {
                    console.log('Server shutdown successful.');
                    window.close();
                } else {
                    console.error('Server shutdown failed.');
                }
            })
            .catch(error => {
                console.error('Error sending shutdown request:', error);
            });
    }
}
```

Listing 5: Engine.js

```python
from http.server import HTTPServer, SimpleHTTPRequestHandler
import webbrowser
import threading
import os.path
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--scenario', type=str, help="string name of gltf scene to be launched")
args = parser.parse_args()

RequestScene = args.scenario

ChromePath = "/usr/bin/google-chrome-stable"
ChromeFlags = "--disable-gpu-vsync --disable-frame-rate-limit --window-size=1920,1080"

PORT = 8000

class CORSRequestHandler(SimpleHTTPRequestHandler):
    def end_headers(self):
      #Allow CORS(cross-origin)
        self.send_header('Set-Cookie', 'pathToDemo=' + RequestScene)
        self.send_header('Access-Control-Allow-Origin', '*')
        self.send_header('Access-Control-Allow-Methods', 'GET, POST, OPTIONS')
        self.send_header('Access-Control-Allow-Headers', 'Content-Type')

        #Disable caching
        self.send_header("Cache-Control", "no-cache, no-store, must-revalidate")
        self.send_header("Pragma", "no-cache")
        self.send_header("Expires", 0)
        SimpleHTTPRequestHandler.end_headers(self)

    def do_GET(self):
        if self.path == '/shutdown':
            self.send_response(200)
            self.send_header('Content-type', 'text/plain')
            self.end_headers()
            # this is not the way to do it, but it works...
            print("-------------------------------- CLOSING SERVER --------------------------------")
            self.server.shutdown()
        else:
            SimpleHTTPRequestHandler.do_GET(self)

server = HTTPServer(('localhost', PORT), CORSRequestHandler)
def run():
    server.serve_forever()

thr = threading.Thread(target=run, daemon=True)
thr.start()

webbrowser.get(ChromePath + " %s " + ChromeFlags).open('http://localhost:{PORT}/src/a_demo/Stranichka.html'.format(PORT=PORT), new=1)
```

Listing 6: server.py

```javascript
import {glMath, Vector3, Vector4} from "../Math/index.js";

export class Animator{
    channels;

    constructor(channels){
        this.channels = channels;
    }

    animate(t){
        this.channels.forEach(channel =>{
            channel.updateNode(t);
        })
    }
}

export class Channel{
    static paths = Object.freeze({
        translation     : (node, v) => node.setPosition(v),
        rotation        : (node, v) => node.setRotation(v),
        scale           : (node, v) => node.setScale(v)
    });

    node;
    path;
    sampler;

    constructor(node, path, sampler){
        this.node = node;
        this.path = path;
        this.sampler = sampler;
    }

    updateNode(t){
        this.path(this.node, this.sampler.getOutput(t));
    }
}

export class Sampler{
    static interpolationModes = Object.freeze({
        LINEAR          : (start, end, t) => Sampler.lerp(start, end, t),
        STEP            : (start, end, t) => Sampler.stepInterp(start, end, t),
        CATMULLROMSPLINE: (start, end, t) => null,
        CUBICSPLINE     : (start, end, t) => null,
        rotation        : (start, end, t) => Sampler.slerp(start, end, t),
    });

    input;
    output;
    mode;
    elementSize;

    constructor(input, output, elementSize, mode = Sampler.interpolationModes.LINEAR){
        this.input = input;
        this.output = output;
        this.mode = mode;
        this.elementSize = elementSize;

        this.duration = input[input.length - 1] - input[0];
        this.frameLength = this.duration / input.length;

        this.input[0] = 0;
    }

    getOutput(globalT){
        let localT = globalT % this.duration;
        let index = Math.min(Math.floor(localT / this.frameLength), this.input.length - 1);

        while (!(this.input[index] <= localT && this.input[index + 1] >= localT)){
            if(this.input[index] <= localT){
                index += 1;
            } else {
                index -= 1;
            }
            if(index >= this.input.length - 1){
                index = 0;
            }
        }
        let t = (localT - this.input[index]) / (this.input[index + 1] - this.input[index]);

        return this.mode(
            new (this.elementSize === 3 ? Vector3 : Vector4)(...this.output.slice(index * this.elementSize, (index + 1) * this.elementSize)),
            new (this.elementSize === 3 ? Vector3 : Vector4)(...this.output.slice((index + 1) * this.elementSize, (index + 2) * this.elementSize)),
            t
        );
    }

    static lerp(start, end, t) {
        return start.scale(1 - t).add(end.scale(t));
    }

    static stepInterp(start, end, t) {
        return start;
    }

    static slerp(start, end, t){
        let dot = Vector4.dot(start, end);

        if (dot >= 0.9995){
            return Vector3.QuaternionToEuler(...Sampler.lerp(start, end, t).normalized);
        }

        let s = Math.sign(dot);
        let a = Math.acos(Math.abs(dot));

        if(a <= glMath.EPSILON){
            return Vector3.QuaternionToEuler(...Sampler.lerp(start, end, t).normalized);
        }

        let r_sa = 1 / Math.sin(a);
        let vt = start.scale(Math.sin(a * (1 - t)) * r_sa).add(end.scale(s * (Math.sin(a * t) * r_sa)));
        return Vector3.QuaternionToEuler(...vt);
    }
}
```

Listing 7: Animator.js

```
if(gltf.hasOwnProperty("animations")) {
    gltf.animations.forEach(animation => {
        let channels = [];

        animation.channels.forEach(channel => {
            let samplerData = animation.samplers[channel.sampler];
            let targetPath = channel.target.path;
            let interpolation = Sampler.interpolationModes[targetPath === "rotation" ? "rotation" : samplerData.interpolation];
            let sampler = new Sampler(dataFromAccessor(samplerData.input), dataFromAccessor(samplerData.output), Mesh.typeCount[gltf.accessors[samplerData.output].type], interpolation);
            channels.push(new Channel(getNodeFromNIndex(channel.target.node), Channel.paths[targetPath], sampler));
        });
        scene.animations.push(new Animator(channels));
    });
}

function dataFromAccessor(accessorIndex){
    const accessor = gltf.accessors[accessorIndex];
    const {buffer, byteLength, byteOffset} = gltf.bufferViews[accessor.bufferView];

    const typedArray = glMath.typedArrayToWebGLType[accessor.componentType];
    const bpe = 1 / typedArray.BYTES_PER_ELEMENT;

    return new typedArray(gltf.realisedBuffers[buffer], byteOffset, byteLength * bpe);
}
```

Listing 8: SceneGraph.py (only animation loading)

```
#load csv data
data <- read.csv("parsed_data.csv")
grouped_by_ES_list <- split(data, list(data$Engine, data$Scenario))

desired_order <-c("low", "mid", "high")
data$Scenario <- factor( as.character(data$Scenario), levels=desired_order )
data <- data[order(data$Scenario),]




#Test normality with Shapiro-Wilk test
library(plotly)
library(dplyr)

normality_test <- function(d, engine, variable) {
  split_data <- split(d, d$Engine)[[engine]][,-1]
  grouped <- split(split_data, split_data$Scenario)
  test_results <- lapply(grouped, function(x) shapiro.test(x[[variable]]))
  table <- data.frame(W = sapply(test_results, function(x) x$statistic), P = sapply(test_results, function(x) x$p.value))
  table <- table %>% mutate_if(is.numeric, round, digits=3)
  s <- unlist(lapply(row.names(table), function(x) gsub(".W", "", x)))
  plot_ly(
    type = "table",
    header = list(values = c("Scenario", "W", "P")),
    cells = list(values = rbind(s, table$W, table$P))
  ) %>% layout(title = sprintf("%s_%s", engine, variable))
}
normality_test(data, "CPP", "Energy")
normality_test(data, "JS", "Energy")
normality_test(data, "CPP", "Time")
normality_test(data, "JS", "Time")



```

```
#Test significance with a two-tailed t test
library(plotly)
library(dplyr)

significance_test <- function(d, variable,
    testChoice) {
  scenarios <- unique(d[,'Scenario'])
  test_results <- list()
  for (level in scenarios) {
    cpp_col <- paste0("CPP.", level)
    js_col <- paste0("JS.", level)

    cpp <- unlist(grouped_by_ES_list[[cpp_col]][[
    variable]])
    js <- unlist(grouped_by_ES_list[[js_col]][[
    variable]])
    test_results <- append(test_results, list(wilcox
    .test(cpp, js, alternative = "two.sided")))
  }
  table <- data.frame(W = sapply(test_results,
    function(x) x$statistic), P = sapply(
    test_results, function(x) x$p.value))
  #table <- table %>% mutate_if(is.numeric, round,
    digits=3)
  plot_ly(
    type = "table",
    header = list(values = c("W", "P")),
    cells = list(values = rbind(table$W, table$P))
  ) %>% layout(title = sprintf("JS-CPP_%s", variable
    ))
}

# Non-normaly distributed data
significance_test(data, "Time", "wilcox")
significance_test(data, "Energy", "wilcox")
# Normally distributed data
# significance_test(data, "Time", "t")
# significance_test(data, "Energy", "t")



#plot data
library(ggplot2)
library(dplyr)

data_sum <- data %>% group_by(Engine, Scenario) %>%
  summarise(Energy=mean(Energy), Time=mean(Time),.
    groups = 'drop') %>%
  as.data.frame()

ggplot(data_sum, aes(fill=Engine, y=Time, x=Scenario
    )) +
  geom_bar(position="dodge", stat="identity") +
  geom_text(aes(label = round(Time, 1)),
            position = position_dodge(width = 0.9),
            vjust = -0.5,
            size = 3) +
  labs(x = "Scenario", y = "Time(s)")

ggplot(data_sum, aes(fill=Engine, y=Energy, x=
    Scenario)) +
  geom_bar(position="dodge", stat="identity") +
  geom_text(aes(label = round(Energy, 2)),
            position = position_dodge(width = 0.9),
            vjust = -0.5,
            size = 3)+
  labs(x = "Scenario", y = "Energy(J)")
```

Listing 9: statistical_analysis.R