Web technologies for cross-platform desktop applications–a feasible option?

Petrus Lidholm

October 27, 2017

Abstract

Traditionally, desktop GUI applications have been built using languages and frameworks more commonly associated with desktop platforms, such as Qt, a framework for building cross-platform applications on C++; Swing and the more recent JavaFX, which are Java counterparts; and Tkinter for Python. But recent years have seen the release of tools like Electron and NW.js for building desktop applications using web technologies such as HTML, CSS, and JavaScript. Seeing as vast numbers of developers are familiar with these technologies, there is reason to investigate these new tools and and out what the benefits and disadvantages of using them are and see if it is feasible to use them.

To answer this we first need to investigate what using these tools will mean for the performance, functionality, source protection and front-end of a potential application.

We find that, compared to C++ and Java, Node.js is a slow performer. Node.js has support for C++ through something called addons, and therefore should be as capable as C++ is. Source protection is possible if using NW.js. And finally, front-end using web technologies has some benefits over the way it is done in Qt and JavaFX.

We conclude that using web technologies is a feasible option, but whether or not it is appropriate it depends on what kind of application you want to make and which of these technologies you are experienced in.

Contents

1 Introduction

Recently, the addition of the frameworks NW.js and Electron have enabled developers to create desktop applications using web technologies. Given that– according to a survey conducted by Stack Overflow–JavaScript is the most common programming language among developers [1], there is reason to investigate these new tools and find out what the benefits and disadvantages of using them. We will try to find out what the implications of using web technologies are, how they compare to existing solutions and if they are a feasible option for building desktop applications.

1.1 Focus and clarifications

The term ”Web technologies” is vague and encompasses a lot. In our work we use it to mean Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS). These are the web technologies we will be looking at.

By ”applications” we mean those that have a graphical user interface (GUI). There are many platforms that can run software capable of displaying a GUI. Seeing as even some fridges now do this, it might help clarifying what the platforms are included in our use of the term ”cross-platform”. We mean computers running one of the two most common operating systems (OS): Windows (Windows 7 and later), macOS (10.9 and later) [2], and some distributions of Linux (Ubuntu 12.04, Fedora 21, Debian 8). All other GUI-capable devices such as various medical equipment, game consoles, and the aforementioned fridges are all left out in the cold.

We cannot, or in any case will not, make comparisons between all languages and frameworks. If it becomes relevant to compare specific frameworks we will look at Electron and NWJS for web-technology based frameworks in opposition to the more classic Qt and JavaFX. Similarly, concerning languages, we will compare JavaScript to C++ and Java.

1.2 Motive

Web technologies such as HTML, CSS, and JavaScript are well-known throughout the development world. According to another survey by Stack Overflow carried out in 2016, JavaScript was the number one "Top Tech" among developers, with HTML 9th place and CSS on 11th [3]. The knowledge is there and now so are the tools for applying this knowledge for developing desktop applications. 2013 saw the release of Electron [4], a framework for building cross-platform desktop applications using web-technologies. It was preceded by NW.js, a similar framework first available in 2012 [5]. We have the tools and the knowledge. So far so good.

There is still the question whether or not these tools should be used, though. Unlike languages like C++ and Java, JavaScript isn't traditionally used for desktop. As the name suggests, JavaScript is a scripting language, and it was created in order to enable scripting in HTML. It was not made to do any heavy lifting we expect from some desktop applications. Whilst Electron and NW.js utilize Node.js, a runtime environment that allows server-side JavaScript code execution that may enable some speedier result than regular browser-run JavaScript, it is no C++. According to a benchmark by Isaac Gouy that compares the speed of C++ and JavaScript (ran with Node.js) for different tasks, JavaScript is most often slower, in one example even $\sim 2273\%$ slower (while at the same time eating ~ 8 times the memory) [6]. There are clear disadvantages with JavaScript.

All in all, there are a large number of developers familiar with web technologies, and CSS is a capable tool for most things front-end, but there are drawbacks. A major one being performance. This means the choice between either NW.js or Electron and others such as Qt or JavaFx is not a simple one, and by addressing performance and other concerns we can help guide the decision-making of developers interested in taking this new route.

1.3 Worth

Answering important questions about performance and source protection will help those concerned to quickly decide whether or not this the appropriate tools for them. Identifying appropriate and inappropriate uses of web technologies for desktop application development could help a lot of developers make an informed decision.

Following are a few examples of whom this paper may be of use to:

- Those with little to no knowledge of and experience with existing tools, looking to make a desktop application
- Those who intend to make a desktop application and want to apply their knowledge of JavaScript, CSS, and HTML, rather than learn new languages
- Those who already have a web-based application and are looking to port it to other platforms

2 Questions

I will try to answer the following:

1. What are the implications of using JavaScript for back-end in the following areas:
   (a) Performance
   (b) Functionality
   (c) Source protection
2. Based on the findings in 1, when is *not* suitable to use web-technology based approach?
3. How is front-end done in the different frameworks? Are there any limitations for layout, styling and animation?

2.1 Motivation

JavaScript is different from C++ and Java in many ways, and it is important to know how. We assume that for many, performance is a concern. Some may not even know what JavaScript is capable of, so we should take a look what functionality Node.js run JavaScript has. Others may be put off by using an interpreted language, which might mean their source code is easy to get hold of, hence the possibility of source protection is something to investigate. All of these are addressed in question 1.

Connected to 1, 2 is aimed at those who want a few quick examples of when it is inadvisable to use web-technology for desktop applications.

Lastly, 3 is for those interested in the superficial, who want to know what is possible in each framework in regards to appearance.

2.2 Hypothesis

Regarding performance: given that JavaScript is interpreted whereas C++ and Java are compiled, I assume JavaScript will perform worse in terms of speed for most if not all tasks. As for functionality, I am confident JavaScript is not nearly as capable as the very low-level C++. Source protection is interesting, as JavaScript being interpreted should mean the source is trivially easy to get hold of.

Based on what I think currently, I assume that it is not appropriate to use web-technology to build applications where performance is of great concern, nor for when source protection is important. I am not currently aware of what specific things JavaScript cannot achieve that may be of interest, but I suspect it is very limited, especially compared to C++.

Having worked with HTML, CSS, and JavaScript, I know that it is quite capable when it comes to styling. Elements can be freely positioned anywhere, and there are plenty of styling options. Animation was previously done in JavaScript, but some animation support has been introduced in later CSS standards as well. You can achieve any look you want with them. I'm suspecting that Qt and JavaFX may be more limited in comparison. They both have their own CSS-like styling language (QSS and JavaFX CSS respectively), and I've found them to be lacking. I do not know about animations in Qt and JavaFX.

3 Research methods

3.1 Search strategy

I will search for information by using certain keywords, alone or in combination. Searching for keywords such as: ”Electron”, ”NW.js”, ”JavaFX”, ”Qt”, ”JavaScript”, ”C++”, ”Node.js”, ”Java”, without being particular about periods or case. Some times in combination with each other with phrases like ”vs”, ”versus”, ”compared to” when looking for comparisons. When looking for particulars of a tool/language, I will get information from their own documentation whenever possible.

3.2 Criteria

To find information that can be deemed relevant I will only look sources that fulfil these criteria:

- Statistics on the popularity/usage of a tool/language should be no older than 2015
- For performance comparisons and information on functionality, sources must be based on Node.js run JavaScript
- For information on Electron and NW.js, only sources from 2012/2013 and later are relevant, since they were not released before then

4 Literature Review

4.1 Performance

In order to gauge whether or not JavaScript is an option, one might want to know how it performs relative to other languages. Hence, this section is focused on just that. We will present data that shows how Node.js run JavaScript compares to C++ and Java when it comes to speed and memory usage for a set of algorithms.

4.1.1 Performance comparison

To see how Node.js run JavaScript compares to C++ and Java I have compiled two graphs in regards to execution time and memory usage.

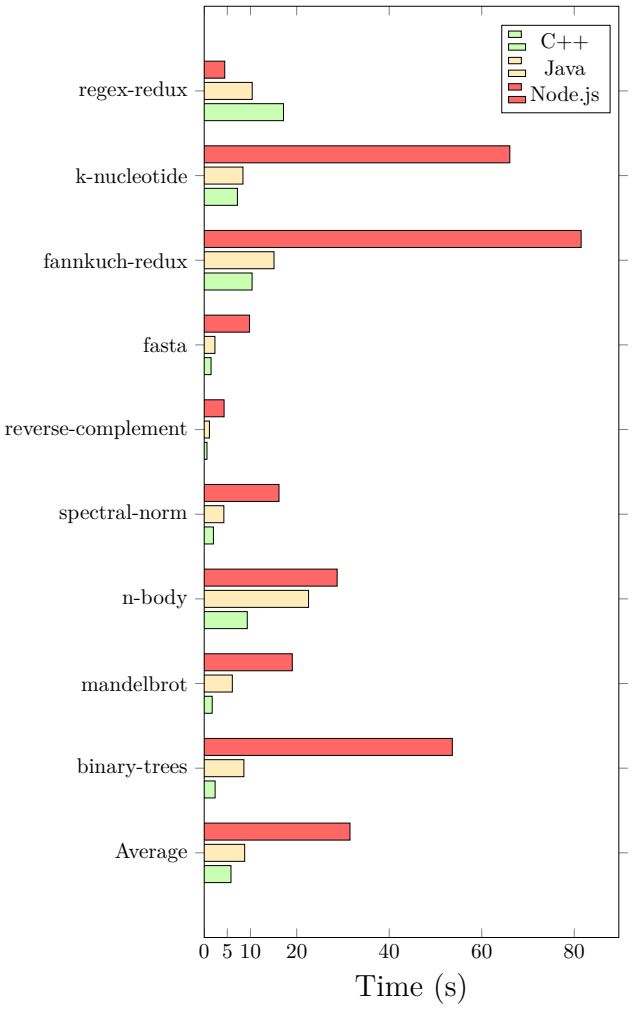

Figure 1: Time to complete certain programs (lower is better). Source: "The Computer Language Benchmarks Game" [7], C++ measurements taken 2017-04-06, Node.js measurements taken using latest version, at the time of writing (2017-10-23) v8.7.0, released 2017-10-11 [**node870**].

As we can see in figure 1, C++ is the fastest at 5.79, with Java and Node.js taking ~ 1.5 and ~ 5.4 times as long respectively.

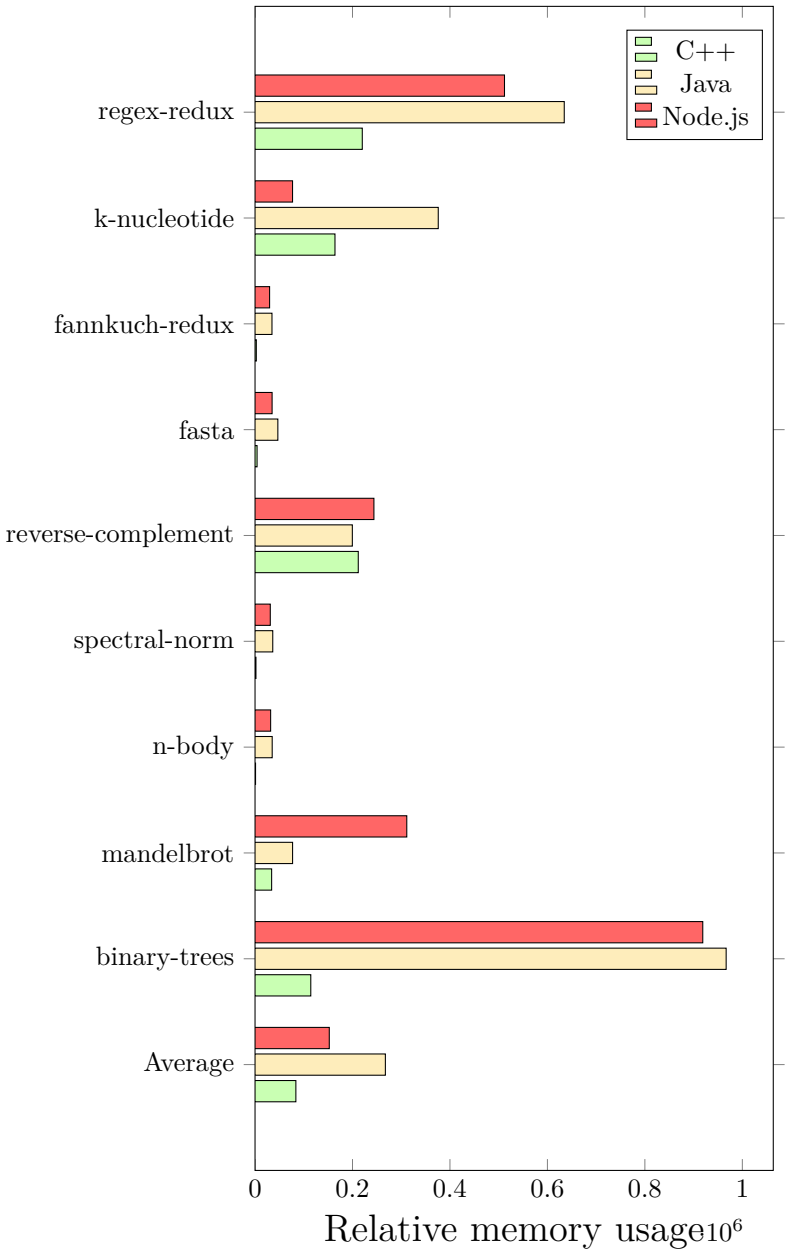

Figure 2: Memory used to complete certain programs (lower is better). Source: "The Computer Language Benchmarks Game" [7], C++ measurements taken 2017-04-06, Node.js measurements taken using latest version, at the time of writing (2017-10-23) v8.7.0, released 2017-10-11 [**node870**].

Looking at figure 2 we see that, although Node.js takes longest to execute, it is better than Java memory-wise. Still, C++ once again holds first place, Node.js using ~ 1.8 times the memory that C++ uses, and Java about ~ 3.2 as much.

4.1.2 Concurrency

One common way to speed up programs, or keep them from being slow, is to make code run concurrently. We will take a close look doing just that, bringing up the concept of threads. If you are familiar with all this, you can skip ahead to section 4.1.4.

You're hungry and you decide to cook some rice and beans. You need to boil the rice as well as the beans—separately—on your stove top which has four burners. Now, you could choose to use just one burner, first cooking the rice, which takes twenty minutes, and then the beans, which takes fifteen minutes. This would take a total of 35 minutes, but this would be a waste of your time—you have other burners and pans available and the beans can me made independently of the rice. You could do it in just twenty minutes by using another burner to cook both simultaneously. This is akin to running an application on a single thread, with independent computations being run sequentially instead of in parallel. In this analogy, the CPU is the stove top, and the burners are in turn the different cores of the CPU.

Instead of beans and rice, you are waiting for your computer to render a video file. Each frame you render takes some amount of time t_{frame}. Like using a single burner, the rendering process is run on a single core, thus only allowing for a single frame will be rendered at a time. If you have some number n_{frames} to render then it will take some time $t = n_{frames} * t_{frame}$. If the CPU has four cores (which are not already busy doing something else), you could complete the process four times as fast by rendering 4 frames at a time.

4.1.3 Threads

Having learnt about multiple cores, how do we make use of them? How do we put the beans on another burner? We use threads. Taking straight from the Wikipedia page on threads, a thread (of execution) is "the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system" [8].

If we are lucky when starting a thread, there is a separate core that is not busy and the scheduler will run our thread on that core, executing in parallel with our main thread, just like using multiple burners. If we should not be so lucky, then the thread will run on the same core as our main thread (or some other core which is already busy). The thing with threads running on the same core is that, whilst they are not run in parallel they are not run sequentially either. The core switches back and forth between the threads, taking turn executing them for some amount of time before switching to the other. This is where the stove top analogy breaks down, as this is like using one burner and constantly shifting different pans on and off the burner, which is not something you would commonly do.

But multi-threading even on a single core has its benefits, just not for our render

or cooking example. Since we are switching between executing different threads, it means that even if we are doing some heavy computation process on the same core as some other thread, neither of them will halt.

4.1.4 Concurrent execution in Node.js

So how does it all apply to Node.js?

NodeJS is designed without threading [9]. Node.js cannot start threads, but it can start a new process and communicate with it. This is done using `child_process.fork` [10].

What is the difference between starting a process and a thread? Well, unlike threads, processes do not share the same memory space, so you cannot read/write from the same locations, meaning that you will have to send copies of data between processes. Hence, ”spawning a large number of child Node.js processes is not recommended” [10].

On a related note, should you want to do a lot of parallel computations, you have the option to do it using JavaScript, since it does allow access to the Graphics Processing Unit (GPU). It does this through WebGL, which is made for ”rendering interactive 3D and 2D graphics” [11]; and WebCL, which is designed for GPU computation [12]. WebCL is not natively supported in any browsers as of now, but WebGL is. And, while WebGL is not designed for general computation, there are libraries for it aimed at parallel computation [13, 14, 15].

4.2 Functionality

Being that JavaScript was originally meant to run in a browser, does that entail that it can only do what is possible in a browser? Is there some things that languages more associated with desktop, such as C++, can do, but Node.js run JavaScript cannot? This is what we will investigate next.

4.2.1 Packages & addons

Perhaps you have a specific idea you want to implement and you wonder if it is even possible to achieve with Node.js. You maybe want to draw 3D elements, play sound or do some low-level operations. Can you do all that with Node.js? We already know we have support for 3D rendering using WebGL, sound we can play through the HTMLAudioElement interface [16], and we will deal with low-level operations in a bit. Both WebGL and HTMLAudioElement usually requires that our application is running in a window. For the intended use case of WebGL—displaying graphics—this is not a problem—no sense in drawing graphics that cannot be seen. It is a problem if we simply want to do computations, without any user interface, and this, along with playing audio, is a use case that is sensible

without a window. Are we out of luck? Well, Node.js comes with a package manager called `npm`, which has an online repository of several packages/modules and it claims to be the "world's largest software registry" [17], including one for creating a WebGL context in a outside of the browser context [18] and several related to audio. If we want to do some arbitrary task that is we find JavaScript cannot help us with, and we do not find a module for it, have we then, finally, ran out of luck? Not quite. As it turns out, what Node.js cannot do it can let C++ do for it.

Addons in Node.js are "are dynamically-linked shared objects, written in C++, that can be loaded into Node.js . . . and used just as if they were an ordinary Node.js module" [19]. Node.js itself comes with some of these addons created with C++ to solve certain tasks [19].

We need to keep in mind that whilst JavaScript is interpreted, C++ needs to be compiled, and thus, if we write our own C++ addons for Node.js, then we must keep in mind that the C++ code needs to be compiled for each platform during deployment.

4.3 Source protection

JavaScript is a "interpreted or JIT-compiled programming language" [20], *JIT* meaning "just-in-time". This means that before we run it, the source is just as you wrote it and any user could get hold of your code.

So is it impossible? Here's where NW.js and Electron differ. NW.js does offer source protection through a tool called `nwjc` Note that, although execution of JavaScript compiled this way used to be about 30% slower, it is no longer the case [21]. Do not forget that using web technologies, your front-end code (that which is written in HTML and CSS) will be exposed, unless you choose to do create all elements and do all styling from compiled JavaScript.

4.4 Front-end

We have looked at what it means to use JavaScript for back-end, and now we're going to look at how web technologies do visuals. We want to know what CSS and HTML are capable of, and how they differ from their Qt and JavaFX cousins: *QSS* (Qt Style Sheet) and XML or QML, and JavaFX's JavaFX CSS and FXML. We will, in dedicated sections, be looking at how layout, styling, and animation is handled in each. The focus will be to find the restrictions of each.

4.4.1 Layout & Styling

Yet another way these frameworks are different but similar is how styling is handled. The web technology for this is CSS, whilst Qt has QSS and JavaFX has JavaFX CSS. As NWJS and Electron both use Chromium for rendering, the

CSS functionality they can use is only limited by Chromium. According to Qt's documentation, QSS has all the selectors of CSS2 [22]. JavaFX CSS is based on CSS 2.1, with some CSS 3 functionality [23]. It is easily recognized as it prefixes most attributes with "-fx-". It lacks support for some pseudo-classes, and has its own names for some. A list of limitations can be found in the documentation [24].

4.4.2 Animations

In some cases you may want to breathe some life into your GUI. You might want to bring attention to something, like a notification, or present a progress bar. If you go with web technologies you have two options:

- Use JavaScript to manipulate the layout and styling of elements, perhaps making use of one of the many freely available animation libraries [25, 26, 27]
- Use CSS transitions [28] and animations [29]

You can of course use a mixture of both, and if you do want to do more advanced animations you may have to use JavaScript as it is does not have inherent limits of what you can do, seeing as you can manipulate the DOM freely.

Being based on CSS 2.1, JavaFX does not provide the ability to animate through CSS. Instead, you can animate elements through Java code, using for example the provided `Animation` class [30].

With animations done in JavaScript or CSS, you can make changes to animations in runtime, while in JavaFX and Qt you will have to recompile to see changes.

5 Discussion

5.1 Performance

We should be able to use C++ for multi-threaded operations, and since we can from within C++ spawn new threads, we get away with lower memory usage as we do not need to keep duplicates of data within the C++ addons. However, we will still need to send any data to or from our Node.js process, so this will perform best when we need to make heavy computations that does not require us to send large amounts of data between the two.

We should note that, since the performance statistics cited in section 4.1.1 depend on each implementation of the algorithms in the different languages, they may not be wholly representative of what is the language is capable of.

5.2 Source protection

If source protection is your only qualm with using a web based approach I do not think it is reason enough to avoid it, given a few things.

I doubt there is reason to be worried about protecting front-end code. After all, the visual aspect is not hidden; it is there for all your users to see. If someone wants to copy the aesthetics of a program the only thing that is stopping them is knowledge of how to use tools how to build an imitation of that look. This is especially true if what you are building is a desktop version of a service that already exists on the internet, since the HTML and CSS is freely accessible by the user.

5.3 Front-end

5.3.1 Styling

Switching from either QSS or JavaFX CSS to plain CSS might feel liberating as it is more capable. And it should not be too hard either, they do share a near identical syntax. However, for an experienced CSS user switching to either of these may be irritating, as some selectors and pseudo-elements they are used to might have other names or not be present at all.

5.3.2 Animation

For simple animations—those that involve transformations and color—CSS transitions and animations are enough, and they allow you to separate animation logic completely from back-end code. For more advanced things, you will want to use JavaScript. Since animations in JavaFX and QT are only available through

code, such clear separation of animation and other code is not quite possible. I assume that, given JavaScript's huge developer base and its close relationship to HTML and CSS, there is a lot more to be found on animation. There are several different libraries, and developers with knowledge of how to animate using this technique is likely more common than those of JavaFX or QT.

An advantage with using web technologies is that you can almost instantly see changes made to an animation, since the code is interpreted. You can even make changes during runtime to see how it changes. This should make it easier to tweak animations and get them just to your liking.

6 Result

6.1 Performance

If we trust the results of the benchmarks, C++ is the best of the 3 languages overall when it comes to both memory usage and execution time. Node.js came last on execution (by quite a lot) and second on memory usage.

6.2 Functionality

Trusting npm's statement that it is the world's largest software registry, you may find someone has already made a package/module for what you want to do. And with C++ addons, not much is impossible. You just have to know how to write it.

6.3 Source protection

Given that one does not care about exposing the front-end code, worries about source protection should not be an issue. Just make sure you're using a tool that allows for it, such as NW.js.

6.4 Front-end

In regards to styling, the regular browser-run CSS is the more complete CSS experience, with Qt's QSS and JavaFX's CSS being less capable copies.

6.4.1 Animation

If creating a living, breathing user interface is the focus, it seems using web technologies is the way to go, given the multitude of libraries and the presumably comparably large amount of developers with knowledge in the area.

7 Conclusions

Going back to the questions we asked, what are some of the implications of using JavaScript for back-end regarding performance? Well, just using JavaScript will make for a program that you can expect to be considerably slower than one written in Java or C++. If speed is the biggest deciding factor then using web technologies is not the best choice. Also, you cannot spawn threads, only processes, so if you know your application would need to spawn many processes it might not be the best choice, given that the documentation advises against it. If your computations are relatively simple and big in number, then parallel computation might be possible through WebGL/WebCL.

How about functionality? Well, you will have access to npm's large repository of modules, and if you cannot do what you want in regular Node.js JavaScript, you can probably do it through C++ addons. Source protection is possible, at least if using NW.js. The only implications are that your front-end code will be exposed.

For styling and animation, CSS is more capable than QSS or JavaFX CSS, since they are simply their own copies with less features, especially JavaFX CSS. Neither QSS or JavaFX CSS has animation or transition support, instead relying on animations to be written in C++ or JavaFX using special classes. With web technologies, you can do animation with both JavaScript and CSS.

Is it a feasible approach to use web technologies for desktop applications? That depends on what you are after, and what languages you are fluent in. Those with experience with JavaScript, CSS, HTML, and C++ can pick and choose, blending all technologies to get the best of each. If you already have a web application then it should be trivially easy to make a desktop client, and easy to keep the web and desktop versions identical. The styling capabilities are also larger and so, yes: it is a feasible approach, that may not be appropriate if you are already familiar with Qt/JavaFX and you are aiming solely for performance.

8 Future work

Some suggestions for future research:

- Licenses. We did not bring up licensing or pricing of the different tools. It is relevant to know what the terms are for using a certain tool, so comparisons could be made.
- Development speed. Given that JavaScript is interpreted or compiled at run time, how does this affect development time, given that recompilation is not necessary after making adjustments to code.
- For those desktop applications based on an existing web application, is it perhaps an option that operating systems treated certain web pages like they were a separate application, for instance with its own icon in the icon in the menu bar, for easy access that separates them from less application-like web pages like web searches?
- In-depth analysis of the time and computational costs of transferring information between Node.js and C++ addons or forked child processes.

9 References

[1] S. Overflow. *Developer Survey Results 2017*. Visited on 17th of September, 2017. URL: https://insights.stackoverflow.com/survey/2017#technology-programming-languages.

[2] StatCounter. *Desktop Operating System Market Share Worldwide*. Visited on 18th of September, 2017. URL: http://gs.statcounter.com/os-market-share/desktop/worldwide/.

[3] S. Overflow. *Developer Survey Results 2016*. Visited on 17th of September, 2017. URL: https://insights.stackoverflow.com/survey/2016#technology-top-tech-on-stack-overflow.

[4] *Electron GitHub repository*. Visited on 17th of September, 2017. URL: https://github.com/electron/electron/releases/tag/v0.1.0.

[5] *NW.js GitHub repository*. Visited on 17th of September, 2017. URL: https://github.com/nwjs/nw.js/releases/tag/v0.2.0.

[6] I. Gouy. *The Computer Language Benchmarks Game*. Visited on 18th of September, 2017. URL: https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=node&lang2=gpp.

[7] I. Gouy. *The Computer Language Benchmarks Game*. Visited on 23rd of October, 2017. Statitics can be found by navigating to the comparisons of the different languages. URL: https://benchmarksgame.alioth.debian.org/.

[8] *Thread (computing)*. Visited on 23rd of October, 2017. URL: https://en.wikipedia.org/wiki/Thread_(computing).

[9] *About Node.js*. Visited on 23rd of October, 2017. URL: https://nodejs.org/en/about/.

[10] *Node.js v8.7.0 Documentation*. Visited on 23rd of October, 2017. URL: https://nodejs.org/api/child_process.html#child_process_child_process_fork_modulepath_args_options.

[11] *The WebGL API: 2D and 3D graphics for the web*. Visited on 23rd of October, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[12] *WebCL Overview*. Visited on 23rd of October, 2017. URL: https://www.khronos.org/webcl/.

[13] *GPU.JS*. Visited on 23rd of October, 2017. URL: http://gpu.rocks/.

[14] gnonio. *gl-compute repository on GitHub*. Visited on 23rd of October, 2017. URL: https://github.com/gnonio/gl-compute.

[15] *TensorFire*. Visited on 23rd of October, 2017. URL: https://tenso.rs/.

[16] *HTMLAudioElement*. Visited on 23rd of October, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/HTMLAudioElement.

[17] *npm*. Visited on 23rd of October, 2017. URL: https://www.npmjs.com/.

[18] *gl module on npm*. Visited on 23rd of October, 2017. URL: https://www.npmjs.com/package/gl.

[19] *Node.js v8.7.0 Documentation*. Visited on 23rd of October, 2017. URL: https://nodejs.org/api/addons.html.

[20] *JavaScript*. Visited on 23rd of October, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[21] *JavaScript Source Code Protection Performance Vastly Improved*. Visited on 23rd of October, 2017. URL: https://nwjs.io/blog/js-src-protect-perf/.

[22] *QSS Selectors*. Visited on 23rd of October, 2017. URL: http://doc.qt.io/qt-5/stylesheet-syntax.html#selector-types.

[23] *JavaFX CSS Version*. Visited on 23rd of October, 2017. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#introscenegraph.

[24] *JavaFX Limitations*. Visited on 23rd of October, 2017. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#introlimitations.

[25] *Velocity.js*. Visited on 23rd of October, 2017. URL: http://velocityjs.org/.

[26] *Bounce.js*. Visited on 23rd of October, 2017. URL: http://bouncejs.com/.

[27] *Move.js*. Visited on 23rd of October, 2017. URL: https://visionmedia.github.io/move.js/.

[28] *CSS transition*. Visited on 23rd of October, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/transition.

[29] *CSS animation*. Visited on 23rd of October, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/animation.

[30] *JavaFX Animation Class*. Visited on 23rd of October, 2017. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/animation/Animation.html.