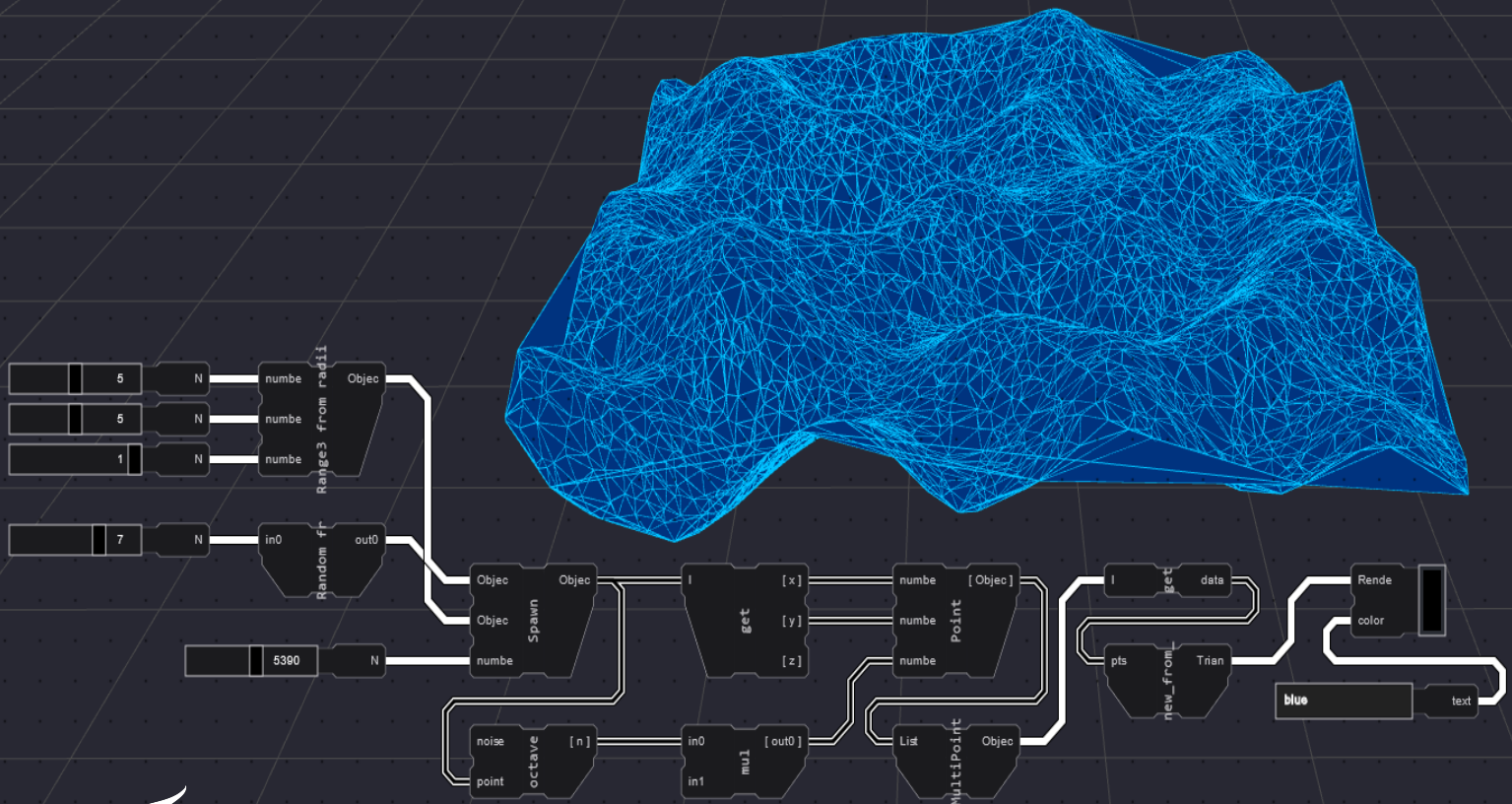


MSc thesis in Geomatics

Geofront: Directly accessible GIS tools using a web-based visual programming language

Jos Feenstra

2022



MSc thesis in Geomatics

**Geofront: Directly accessible GIS tools
using a web-based visual programming
language**

Jos Feenstra

November 2022

A thesis submitted to the Delft University of Technology in
partial fulfillment of the requirements for the degree of Master
of Science in Geomatics

Jos Feenstra: *Geofront: Directly accessible GIS tools using a web-based visual programming language* (2022)

© This work is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

The work in this thesis was carried out in the:



3D geoinformation group
Delft University of Technology

Supervisors:	Dr. Ken Arroyo Ohori Dr. Giorgio Aguiaro
External Supervisor:	Ir. Stelios Vitalis
Co-reader:	Dr. Hugo Ledoux

ABSTRACT

In the field of Geographical Information Science (GIS), geodata transformation and analysis tools often take the shape of software libraries written in system level programming languages. However, the societal impact of these tools is often limited, as most end users only access these libraries via indirect means: Through bindings in other languages, through plugin in applications, or both. Additionally, the tools end-users end up with are often not composable, and may contain other hurdles like installation or configuration.

The goal of this study is to make core GIS libraries more directly available and composable to end-users. This study presents and prototypes a novel method, centered around a visual programming language to host the functionalities of GIS libraries from within an application, and in a composable manner. Additionally, the visual language is used to connect these libraries to a user-definable Graphical User Interface (GUI). This prototype is implemented as a static web application, so that these libraries are directly accessible to end users without installation. GIS libraries are compiled to WebAssembly, making the library usable in any language, including this web based visual language by using a 'no boilerplate' plugin system. Finally, both scalability to handle sizable datasets, and a rich GUI (3D viewers, file inputs, sliders), are primary design considerations and assessment criteria.

The results show that this specific web-based VPL appears to be a feasible method for providing direct access to some native GIS libraries, and does offer a unique set of features not found in comparable visual languages. The significance of this method, compared to other web-based geometry VPLs, lies in the fact that it offers a lenient plugin system, in combination with a range of different GUI nodes, certain "dataflow VPL" properties, and a proposed zero-cost abstraction runtime. All of these features combined lead to a VPL which is able to directly connect GUI components with native GIS libraries, all while remaining scalable in principle.

On a practical level, more work remains to proof this feasibility. The methodology developed by this study is only *theoretically* accessible and composable, based on achieved features. User-testing is required to confirm if this method indeed improves workflows, and actually saves time and energy of developers and end users. Moreover, the prototypical software implementation used is limited and not production ready. Both the fact that the 'no-boilerplate' plugin system cannot be used with C / C++ GIS libraries, and that backend execution is not possible yet, must be improved upon in future work. Despite this, visual programming, distribution using WebAssembly, and Rust-based geocomputation, all proved to be valuable directions of future GIS research.

ACKNOWLEDGEMENTS

The completion of this thesis would not have been possible without the generous support of the people around me. The least I can offer back is my most sincere gratitude. First of all, I want to thank Ken Arroyo Ohori and Stelios Vitalis for going above and beyond in supporting me during all phases of this thesis. They have invested many personal hours along the full scope of this journey, and I couldn't have asked for better guidance. Thank you Giorgio Agugiaro, for seamlessly stepping in during the later stages of the process, and for steering me towards finalization. Thank you Hugo Ledoux for your proof-reading, and especially the suggestions and comments regarding the *cloud-native* aspect of this study.

My employers and colleagues also played a vital role. Thank you Sybren de Graaf, my previous employer at Sfered, for your interest, and the opportunity to develop visual programming, which greatly influenced the subject of this thesis. I want to thank Martin Kodde, my current employer at Geodelta, for the opportunity to learn geodesy in practice, and my colleagues there for their continued interest. In particular, thank you Roeland Boeters, for hosting the dataset used in the Potree demo.

I want to thank my fellow students Laurens, Maarit, Mels, Siebren, Alex & Max, for all the fun and interesting conversations we have had during our studies. My friends from D.S.J.V. Groover also greatly supported me. In particular I want to thank Tim Boot for proofreading my P2. I also cannot forget my mom, dad, brother, and the rest of the family for their encouragement.

Finally, and most importantly, I would like to thank Nadja for her patience and unwavering support throughout the entire period.

Thank you all!

CONTENTS

1	Introduction	1
1.1	Research Objective	5
1.2	Research Questions	6
1.3	Scope	6
1.4	Reading Guide	8
2	Background	9
2.1	WebAssembly & (Static) Web Applications	9
2.1.1	Rich Clients	9
2.1.2	The WebAssembly standard	10
2.2	Visual Programming	13
2.2.1	Usability	15
2.2.2	End User Development & Low Coding	15
2.2.3	Dataflow programming	16
2.2.4	Disadvantages and open problems	17
2.2.5	Conclusion	19
3	Related works	21
3.1	Browser-based geocomputation	21
3.1.1	Examples	22
3.1.2	Commercial web-based geocomputations software	23
3.2	Visual programming and geocomputation	25
3.3	Browser-based visual programming	30
3.4	Browser-based geocomputation using a VPL	31
4	Methodology	33
4.1	Definitions	33
4.2	Requirements	34
4.3	Overview	35
4.4	Framework Application	36
4.5	Web VPL	36
4.5.1	Components	37
4.5.2	Widely supported browser features	38
4.5.3	Design	39
4.6	Plugin System	43
4.6.1	Design	43
4.6.2	Components	44
4.6.3	Plugin Model	45
4.7	Tests	48
4.7.1	Compilation Tests	48
4.7.2	Demo applications	49
4.7.3	Feature Comparison	49
4.7.4	Usage tests	50

5	Implementation	53
5.1	Introducing: Geofront	53
5.1.1	Model	53
5.1.2	View	54
5.1.3	Controller	56
5.2	The plugin system	58
5.2.1	The plugin loader	58
5.2.2	Achieved Workflow	60
5.2.3	Automation and portability	61
5.3	CLI, headless runtime	62
6	Testing	65
6.1	Plugin Compilation & Utilization	65
6.1.1	Rust: minimal plugin	65
6.1.2	Rust: startin plugin	66
6.1.3	C++: Minimal plugin	68
6.1.4	C++: CGAL plugin	71
6.1.5	Comparison	76
6.2	Demo applications	79
6.2.1	Demo One: Perlin noise & startin	79
6.2.2	Demo Two: DTM & DSM extraction	82
6.2.3	Limitations	85
6.3	Feature Comparison	85
6.3.1	Plugin comparison	86
6.4	Utilization assessment	89
7	Conclusion	95
7.1	Conclusion	95
7.2	Contributions	99
7.3	Limitations	100
7.4	Discussion	100
7.5	Future work	101
7.5.1	Cloud-native deployment & scalability	101
7.5.2	Streamed, on demand geocomputation	102
7.5.3	Rust-based geocomputation & cloud native geospatial	103
7.5.4	FAIR geocomputation	103
7.6	Reflection	104
A	Software Implementation	107

ACRONYMS

CDN Content Delivery Network	44
CLI Command Line Interface	34
DAG Directed Acyclic Graph	13
DSM Digital Surface Model	82
DTM Digital Terrain Model	82
ETL Extract Transform Load	26
EUD End User Development	15
GEOCOMPUTATION Geospatial data computation	23
GIS Geographical Information Science	v
GUI Graphical User Interface	v
IDE Integrated Development Environment	13
MVC Model View Controller	39
OGC Open Geospatial Consortium	4
TIN triangular irregular network	79
UI User Interface	3
UX User Experience	23

vpl Visual Programming Language	3
wasm WebAssembly	4

1

INTRODUCTION

The field of GIS commits itself to the betterment of collecting, processing, storing, and viewing geodata. By doing so, it offers the world priceless information about the land we build on, the seas we traverse, the air we breath, and the climates we inhabit. This information is foundational for many applications, including environmental modelling, infrastructure, urban planning, governance, navigation, the military, and agriculture. As such, the field of GIS is continuously looking for new ways to provide these fields and industries with both the data and tools they need to succeed.

Problem Statement and Goal

This thesis concerns itself with the latter: Providing tools. The problem this study seeks to address, is that the core transformation and analysis tools found in certain GIS software libraries, are normally not directly accessible by practitioners in the fields mentioned above: governance, infrastructure, urban planning, etc. A small number of native software Libraries, like CGAL or GDAL [Fabri and Pion, 2009; Dohler, 2016], play a foundational role in many GIS tools. However, end users with a profession different than software developer, are often unable to directly access these libraries. The tools can only be used indirectly, and only when a software developer has incorporated these functionalities in an application. Similarly, if research leads to a new GIS library, end users are at the mercy of a software developer implementing the functionalities of said library as a usable application, or as a plugin for an existing GIS environment like QGIS [QGIS Community, 2022]. Moreover, even if these capabilities are added to an application, the tools are almost always less feature rich, and non-composable: The output of one procedure cannot automatically be used as input for another. This leads to labor intensive procedures and repetitive workflows, as opposed to automated, re-usable procedures. At the same time, maintainers of a GIS library often find themselves in the situation of having to maintain and synchronize a great number of bindings and plugins, which limits innovation (Figure 1).

It is safe to say that the limited reach of these libraries translate to a reduced societal impact, and with it, the GIS research these libraries are based upon.

The overarching goal of this study is to allow end-users more **direct** access to core transformation and analysis capabilities found in native GIS libraries (Figure 2), in a format which allows **composability**.

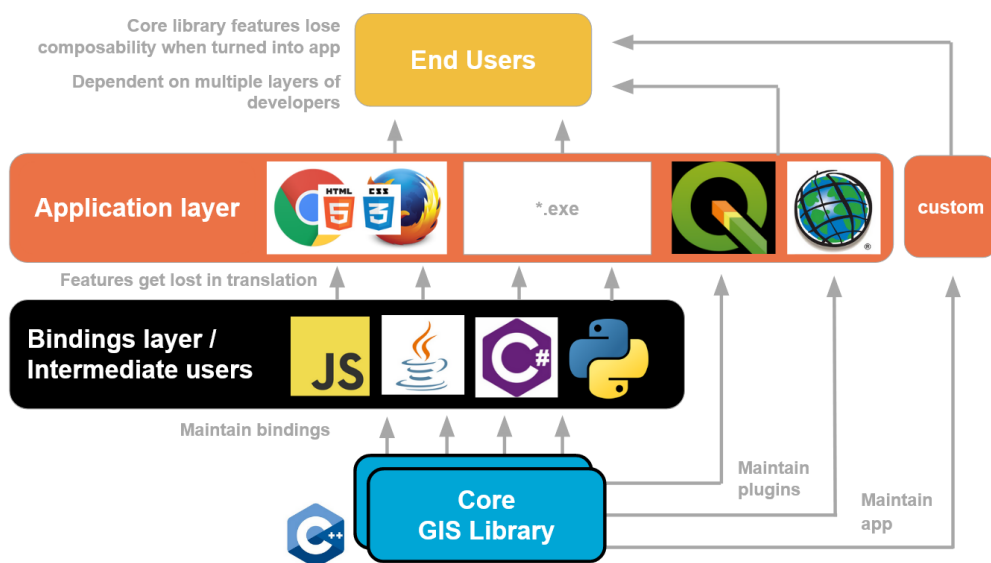


Figure 1: The layers of indirection between end users and core GIS functions

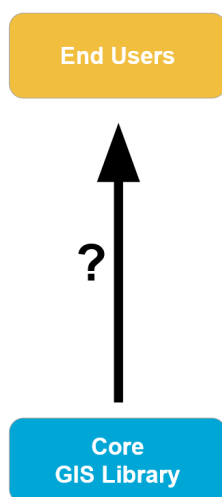


Figure 2: The goal: More direct access



Figure 3: Geoflow: An example of a graph-based VPL [Peters, 2019]

The study attempts to meet this goal by studying two domains in the field of **GIS**, and developing a possible solution from them. The goal of **composability** is addressed by using the knowledge found within the field of **visual programming**. The goal of **direct accessibility** is met by studying the field of **static web applications**. To make the research goals of this study more precise, these two domains must briefly be addressed.

Composability: Visual Programming

A Visual Programming Language (**VPL**) is a type of programming language represented by a **GUI**, rather than a text-based source code. A **VPL** 'script' might take the shape of for example a graph (Figure 3), or a block-based instruction set.

The goal and purpose of a VPL can be framed in multiple ways. This study draws from the perspective presented in Elliott [2007], as it bears resemblance to the goal of this study. Elliott [2007] focusses on the discrepancies between software libraries, and software applications, and notes that in certain situations, properties of both libraries and applications are desired. Software libraries often contain expressive, re-usable functionalities, but do not provide a User Interface (**UI**) of themselves, and must be turned into an application before utilization. Software applications on the other hand, can offer a rich **GUI**, but lose the ability to be re-composed into new software like libraries. Additionally, library functionalities presented through an applications are often reduced and less feature rich compared to the functionality of the library itself. A VPL can be seen as an attempt to extract the best properties of both software libraries and applications. It can offer composability to applications, and **GUIs** to libraries.

Because of these reasons, a VPL can be a desirable format for a GIS application. Multiple examples exist, like FME [Safe-Software, 2022b], Modelbuilder [Esri, 2022] and Geoflow [Peters, 2019]. However, if a VPL is meant to operate on geodata, it will have to attend to the nature of geodata. The size of geodata makes scalable computation a necessity, and as such, GIS VPLs often contain tools to run these scripts on a server, without the GUI. The spatial nature of geodata means that visualization is key in understanding its quality. Also, in most cases a visualization represents the end product of a GIS process. Both make it so GIS VPLs emphasize data visualization. Finally, GIS Procedures sometimes require empirically defined parameters, which could be provided in a VPL using GUI features like sliders.

Direct access: Static web applications

The maps and analysis tools used to share geo-information often take the form of web applications. Web applications offer distribution advantages over native applications [Kuhail et al., 2021; Panidi et al., 2015]. It allows the same source code to be used across different platforms without alteration, including windows, mac, linux, and mobile devices. Moreover, web applications do not need to be installed, and can often be directly accessed using a link.

Two major developments have occurred with great relevance to web GIS applications.

The first development concerns static file hosting. Various geodata formats are increasingly becoming available as singular, statically hosted files, as opposed to the active Open Geospatial Consortium (OGC) web services which query databases, and use dynamic routing [Open Geospatial Consortium, 2015]. Examples of these static formats are the Cloud Optimized GeoTiff [Sarago et al., 2021], and the COPC file format [Bell et al., 2021]. Static file hosting is orders of magnitude more cheap in terms of performance and literal cloud-host pricing, and scales more easily to accommodate high amounts of web requests [Sarago et al., 2021].

The second development is the introduction and adoption of the WebAssembly language [Haas et al., 2017]. WebAssembly (*wasm*) is a binary compilation target meant for a virtual runtime. *wasm* binaries can be run in any environment and language, as long as such a runtime is implemented. Such a runtime has been added to all major web browsers since late 2019 [w3c, 2019]. Among many purposes, it offers a method to run native software as (part of) a web application. WebAssembly partially mitigates the need for web-based, JavaScript alternatives of native software applications and libraries. An example of this is how Ammann et al. [2022] built one map viewer, which was both able to serve as a native and web application.

Both these developments taken together leads to a paradigm shift which is important to recognize. With both used in conjunction, GIS applications can be conceived which use no active servers at all: Statically hosted websites, which can still be feature-complete as regular GIS applications.

Taken Together

Given both phenomena together, it becomes clear that the interface of a VPL might present a solution to the problem of composability, for it allows both automation and a rich GUI within one application. In addition, a combination of WebAssembly and a static web application could allow for direct access and execution of native GIS libraries without installation, using statically hosted geodata as its data source. However, if a VPL wishes to 'qualify' as a GIS VPL, it needs to:

1. be scalable to handle sizable datasets.
2. provide a rich GUI, capable of visualizing geodata, and quickly exploring different parametrization of various procedures.

1.1 RESEARCH OBJECTIVE

The goal of this study is to allow GIS practitioners without a background in software development, to access the full potential of core transformation and analysis capabilities found in native GIS libraries.

The study attempts to meet this goal by designing and implementing a novel method based on the fields of visual programming, and static web applications. This method is thereafter used to load various GIS libraries, and used in demo applications, after which an assessment can be made on its quality and the extent of its achieved functionalities. The study concludes by addressing if the method meets its overarching goal.

While VPLs and even Web-based VPLs have been studied before in the field of GIS, the focus on direct utilization of native GIS libraries on the web, combined with bringing those in direct contact with various GUI elements, all without resorting to active web servers, is a combination unique to this study, at least to the best of the authors knowledge at this point in time. A comparative study on VPLs found in the field of GIS, Computer graphics, the web, and generic VPLs, is performed to ensure this. Additionally, assessment criteria are used to ensure this remains true after having performed the study.

The overall design of the prototype outlined contains a number of technical challenges. The compilation of native libraries to Webassembly is known to contain drawbacks, and achieving scalability with a VPL requires exploration of novel methods. Therefore, seeking and analyzing solutions to these challenges is required to making this prototype meet the goals of the design mentioned.

1.2 RESEARCH QUESTIONS

The objectives outlined above lead to the following main and supporting research questions:

Is a web based VPL a viable method for directly accessing native GIS libraries with a composable interface?

Supporting Questions

- *What GUI features are required to facilitate this method, and to what extent does the web platform aid or hurt these features?*
- *To what extent does this method intent to address the discrepancies between software applications and libraries, as described by Elliott (2007)? Does it succeed in doing so?*
- *What are the differences between compiling a GIS library written in C++ to WebAssembly, compared to compiling a GIS library written in Rust?*
- *What measures are taken to make this VPL scalable to large geo-datasets, and how effective are these measures?*
- *How does this method compare to existing, alternative VPLs and browser-based geo-computation methods, regarding the properties relevant to the goal of direct accessibility?*

Assessment

In order to proof if the proposed method is viable or not, tests will be performed primarily based upon feature completeness. Within the context of this study, whether or not a feature was able to be implemented given the constraints of the method (web based, visual programming), was often deemed a more insightful indicator compared to measuring aspects like performance and memory footprint. The thesis sets out to proof or disproof technical viability. Future research is required to test *How* viable it might be compared to other, comparable methods.

1.3 SCOPE

The scope of this thesis is bounded in the following ways:

Only frontend geocomputation

There is a nuance between 'web-based geocomputation' and 'browser-based geocomputation'. 'Web' can refer to both frontend and backend computation methods, 'browser' refers purely to frontend computations. This study focusses on browser-based geocomputation, and as such, excludes any *backend* based geocomputation.

Adding backend-based geocomputation to a web VPL would be an excellent follow-up investigation to this study, following in the footsteps of studies like [Panidi et al. \[2015\]](#).

No user testing

The introduction mentions *accessibility* as a motivator, which is a subjective concept. However, user testing is not part of this study, due to scope limitations.

To solve this, this study assesses accessibility without regarding subjective, psychological accessibility aspects: "*does x feel nice to use?*". Instead, an accessibility assessment is made by only regarding feature completeness: "*Is it possible to do X with Y?*". Moreover, certain presuppositions are made. It is safe to assume that a web application is more accessible than a native application, for it mitigates installation needs. Additionally, a VPL is assumed to be more accessible than normal programming. This last aspect is safe to assume based on the findings of [Kuhail et al. \[2021\]](#), stating that based on 30 independent studies on the accessibility of VPLs, VPLs are generally considered more accessible than their textual counterparts. These studies involved user tests with professional programmers, amateur programmers, and non-programmers alike.

Thus, these reasons together are why this study believes an adequate assessment can be made without user testing. Nevertheless, a follow up study to reinforce or revoke these results would be valuable.

Only WebAssembly-based containerization

This thesis examines a WebAssembly-based approach to containerization and distribution of geocomputation functionalities. Containerization using Docker is also possible for server-side applications, but is not (easily) usable within a browser. For this reason, Docker-based containerization is left out of this studies' examination. And to clarify: Docker and WebAssembly are not mutually exclusive models, and could be used in conjunction on servers or native environments.

Only GIS libraries written in C++ & Rust

The study limits itself to native libraries written in C / C++ and Rust. C++ was chosen, since almost all relevant GIS libraries are written in C or C++, like GDAL [Dohler, 2016]. Rust was chosen, for its extensive WebAssembly support. It also appears to be the most popular language for writing WebAssembly [Eberhardt, 2022]. It contains a number of relevant GIS libraries, but not to the same extent as C / C++.

1.4 READING GUIDE

The remainder of this study is structured as follows:

Chapter 2, Background, provides an overview of the theoretical background that is used in the rest of this study.

Chapter 3, Related Work, provides a review of studies comparable to this one.

Chapter 4, Methodology, explains precisely in what way the research-questions will be answered. In addition, the main design decisions are described and justified in this part of the study.

Chapter 5, Implementation, presents the implementation of the methodology.

Chapter 6, Testing, tests the results from this implementation in various ways described by the methodology.

And finally, Chapter 7: Conclusion & Discussion, concludes to which extent the study was able to satisfy the main research question, and discusses unaddressed aspects of the thesis. It also includes the envisioned future works and a reflection on the quality of the study.

2 | BACKGROUND

This chapter offers an overview of the theoretical background that this study builds upon. The study takes place at the intersection of three prior bodies of work:

- Native GIS transformation & analysis libraries
- WebAssembly & Static Web applications
- Visual Programming Languages

Since giving a full overview of all aspects of these bodies of work is too extensive, only key elements within these bodies will be mentioned and elaborated, namely visual programming, and WebAssembly.

2.1 WEBASSEMBLY & (STATIC) WEB APPLICATIONS

From all browser-based features, WebAssembly turned out to be a deciding factor of this study. This makes it important to be aware of the state of WebAssembly and its performance considerations. This section offers a background on WebAssembly, and how this technology is currently used in frontend web applications.

2.1.1 Rich Clients

The significance of the WebAssembly standard for the point of view of web applications, can be best understood in conjunction with the Rich client phenomenon. This is why this explanation starts out by framing WebAssembly within this context.

Since 2012, a trend of rich web-clients can be widely recognized [Hamilton \[2014\]](#); [Panidi et al. \[2015\]](#); [Kulawiak et al. \[2019\]](#). Around this time, browser engines had become performant enough to allow more decentralized client-server models. By reducing servers to just static file servers, and adding all routing and rendering responsibilities to the client, the interactivity of a web application could be maximized. This led to models like "single page application", which are facilitated by JavaScript frameworks like Angular, React and Vue. However, the real facilitator of these developments are the browsers vendors themselves, as these frameworks would not be possible without the performance increase granted by improvements of the various JavaScript Just In Time compilers.

This growth has also led to web applications being used 'natively'. Tools like Electron [Electron Contributors, 2022] allow web applications to be installed and 'run' on native machines by rendering them inside of a stripped down browser. Many contemporary 'native' applications work like this, such as VS Code, Slack, and Discord. Additionally, tools like React Native [React Contributors, 2022] are able to compile a web application into a native application without a browser runtime. It becomes clear that rich web clients and surrounding tools are starting to blur the line between native and web software.

2.1.2 The WebAssembly standard

If the line between web application and native application was starting to blur, WebAssembly makes this line almost invisible.

`wasm` is a binary instruction format for a conceptual, stack-based virtual machine [WebAssembly Contributors, 2022]. By combining this low-level format with features like a system of incremental privileges, `wasm` makes for a performant compilation target which can be run containerized, and thus safe. `wasm` is officially dubbed the fourth type of programming language supported by all major web browsers, next to HTML, CSS, and JavaScript [w3c, 2019]. It can be utilized to run a native application or library in a web browser, regardless of the language used to create it, be it C/C++, Python, C#, Java, or Rust. This means that in order to create a web application, developers can now in principle develop a normal, native application instead, which can then be compiled to WebAssembly, and served on the web just like any other web application.

Applications

These features together offer a reverse workflow compared to the now popular Electron based applications described in Section 2.1.1. Applications can now be written natively, and subsequently published to the web, instead of writing software as a web application, which may be packaged as a desktop application.

The WebAssembly format has several other use cases. Contrary to its name, WebAssembly has no specific link to the web or assembly language, its name being a remnant of its initial use-case. A cross-platform binary format which is designed to be lightweight, fast and safe, together with a versatile runtime implemented in several languages, makes WebAssembly applicable for other use cases. It is currently seen as a plugin system, as a runtime for serverless cloud-compute services, and as a lightweight runtime for IoT devices, according to a small-scale survey [Eberhardt, 2022]. Still, compiling libraries and application to the web remains the main, post popular application of WebAssembly. This study is focussed on the web usage of WebAssembly, and will treat it with mainly that use-case in mind.

Limitations

In principle, and if the appropriate compilers exist, any application and library written in any language can be compiled to WebAssembly. In practice, there are quite a few caveats to the format.

First of all, WebAssembly is required to adhere to containerization restrictions. There is no 'os' or 'sys' it can call out to, as it cannot ask for resources which could be a potential security risk, like the file system. Secondly, WebAssembly is in its early phases as a language, and is intended as a low-level compile target. This leads to limitations like how only simple, numerical values can be used when interfacing with `wasm` functions. In the future, the WebAssembly Interface Types proposal will also allow interfacing with more complex types [Wagner, 2022]. Lastly the current version of WebAssembly does not support concurrency features like multithreading.

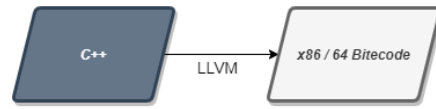
This last shortcoming can be mitigated by calling JavaScript using HTML5 features from WebAssembly. This is how many current WebAssembly projects are set up. However, this layer of JavaScript 'boilerplate' or 'glue code' is inefficient, as it leads to duplication and redirection. Additionally, platforms wishing to support WebAssembly must now also support JavaScript.

Performance

The initial performance benchmarks look promising. The majority of performance comparisons show that WebAssembly only takes 10% longer than the native binary it was compared to Haas et al. [2017]. A later study confirms this by reproducing these benchmarks [Jangda et al., 2019]. It even notices that improvements have been made in the two years between the studies. However, Jangda et al. [2019] criticize the methodology of these benchmarks, stating that only small scale, scientific operations were benchmarked, each containing only 100 lines of code. The paper then continues to show WebAssembly is much more inefficient and inconsistent when it comes to larger applications which use IO operations and contain less-optimized code. These applications turn out to be up to twice as slow compared to native, according to their own, custom benchmarks. Jangda et al. [2019] reason that some of this performance difference will disappear the more mature and adopted WebAssembly becomes, but state that WebAssembly has some unavoidable performance penalties as well. One of these penalties is the extra translation step, shown in Figure 4, which is indeed unavoidable when utilizing an in-between compilation target.

Some studies have taken place evaluating `wasm`'s performance for geospatial operations specifically. Melch [2019] performed extensive benchmarks on polygon simplification algorithms written in both JavaScript and WebAssembly. The study concludes by showing WebAssembly was not always faster,

Native compilation trajectory (ommiting LLVM intermediate representations)



WebAssembly compilation trajectory



Figure 4: Comparison of compilation trajectories

but considerably more consistent. The performance of the JavaScript implementation [Melch \[2019\]](#) had this to say: "To call the WebAssembly code the coordinates will first have to be stored in a linear memory object. With short run times this overhead can exceed the performance gain through WebAssembly. The pure algorithm run time was always shorter with WebAssembly.". These findings match [Jangda et al. \[2019\]](#), showing that the duplication of data into the webassembly memory buffer is a considerable bottleneck.

A recent study concerned with watershed delineation [[Sit et al., 2019](#)] also concluded client-side WebAssembly to be more performant than server-side C, which, as a side effect, enabled their application to be published on the web without an active server.

Lastly, the sparse matrix research of [Sandhu et al. \[2018\]](#) must be mentioned. It shows again that WebAssembly's performance gain is most notable when performing scientific computations. It states: "For JavaScript, we observed that the best performing browser demonstrated a slowdown of only 2.2x to 5.8x versus C. Somewhat surprisingly, for WebAssembly, we observed similar or better performance as compared to C, for the best performing browser.". It also shows how certain preconceptions must be disregarded during research. For example, it turned out that for WebAssembly and JavaScript, double-precision arithmetic was more performant than single-precision, probably due to byte spacing.

Even though geocomputation can fall in the category of scientific computation, these performance considerations will still have to be taken into account. The most important conclusion to take away from prior research on WebAssembly is that [wasm](#) must not be regarded as a 'drop-in replacement', as [Melch \[2019\]](#) puts it. Just like any language, WebAssembly has strengths and weaknesses. While [wasm](#) is designed to be as unassumptious and unopinionated about its source language as possible, the implementations of host environments do favor certain programming patterns and data structures over others, and this will have to be taken into account when using the compile target.

2.2 VISUAL PROGRAMMING

This section offers an overview on the topic of visual programming, after which Section 3.2 and Section 3.3 cover uses of visual programming in geocomputation and on the web, respectively.

Visual programming languages

A **VPL**, or visual programming environment, is a type of programming language represented and manipulated in a graphical, non-textual manner. A VPL often refers to both the language and the Integrated Development Environment (**IDE**) which presents this language in an editable way, by means of a **GUI**. A visual programming language allows users to create programs by adding reconfigured components to a canvas, and connecting these components to form programs.

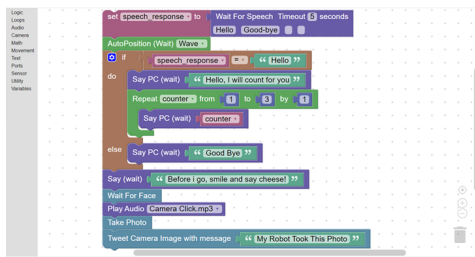
Multiple types of **VPLs** exist, but also multiple taxonomies of these types. This study bases itself on the classifications presented in [Kuhail et al. \[2021\]](#), stating four different types of visual programming languages:

1. **Block-based languages**, in which all normal programming language features, like brackets, are represented by specific blocks which can be ‘snapped’ together (see Figure 5a).
2. **Diagram-based languages**, in which programming function are represented by nodes, and variables are represented by edges between these components (see Figure 5b). This makes the entire program analogous to a Graph.
3. **Form-based languages**, in which the functioning of a program can be configured by means of normal graphical forms (see Figure 5c). This approach enhances the stability and predictiveness compared to other types, at the cost of expressiveness.
4. **Icon-based languages**, in which users are asked to define their programs by chaining highly abstract, iconified procedures (see Figure 5d).

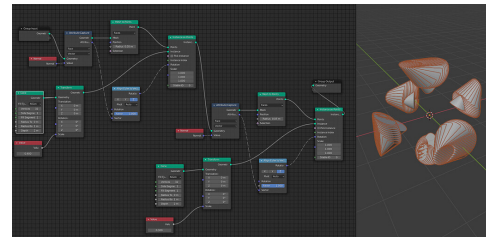
The meta analysis of [Kuhail et al. \[2021\]](#) shows a great preference among researchers for block- and diagram-based languages. Only 4 out of 30 of the analyzed articles chose a form-based VPL, and only 2 chose an icon-based approach.

This study requires to introduce a fifth type of VPL. A **Dataflow** VPL is a subtype of a diagram based VPL which only uses pure functions as computation nodes, only uses immutable variables, and which disallows cyclical patterns. This makes this VPL not only a graph, but a Directed Acyclic Graph (**DAG**). More on this in Section 2.2.3.

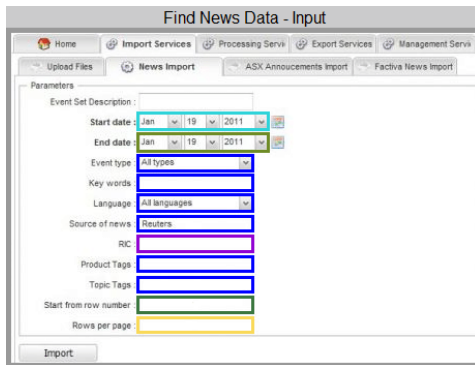
Visual programming languages are used in numerous domains. The VPLs of the 30 studies examined by [Kuhail et al. \[2021\]](#) were aimed at domains such as the



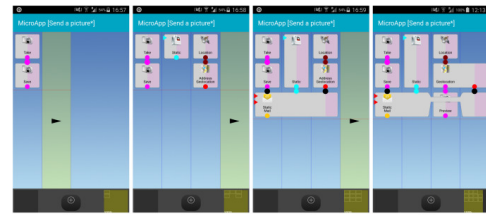
(a)



(b)



(c)



(d)

Figure 5: Four different types of visual programming languages: Block-based [Resnick et al., 2009], diagram-based [Blender Foundation and Contributors, 2022], form-based [Weber et al., 2013], and icon-based [Francese et al., 2017], respectively

Internet of Things, robotics, mobile application development, and augmented reality. Within the domain of systems control and engineering, The Ladder Diagram VPL [Control Automation, 2018] is the industry-standard for programming Programmable Logic Controllers (PLCs). VPLs are also widely used within computer graphics related applications, including the field of GIS, These will be covered in Section 3.2. Lastly, VPLs also have great educational applications. Harvard’s introduction to computer science course, CS50, famously starts out with Scratch, a block-based visual programming language normally targeted at children, to teach the basics of computational thinking [Yu, 2021].

2.2.1 Usability

Studies on VPLs indicate that generally speaking, VPLs make it easy for end users to visualize the logic of a program, and that VPLs eliminate the burden of handling syntactical errors Kuhail et al. [2021].

The locally famous Cognitive Dimensions study Green and Petre [1996], states that *“The construction of programs is probably easier in VPLs than in textual languages, for several reasons: there are fewer syntactic planning goals to be met, such as paired delimiters, discontinuous constructs, separators, or initializations of variables; higher-level operators reduce the need for awkward combinations of primitives; and the order of activity is freer, so that programmers can proceed as seems best in putting the pieces of a program together.”*. Indeed, a VPL UI can be used to eliminate whole classes of errors on a UI level by, for example, not allowing the connection of two incompatible data types.

2.2.2 End User Development & Low Coding

A VPL has the potential to make automation available to a large audience, and this is exactly its purpose. Visual Programming is part of a larger field, named End User Development (EUD). The field is concerned with allowing end users who are not professional software developers to write applications and automate processes, using specialized tools and activities.

Kuhail et al. [2021] point out two serious advantages of EUD. First, end users know their own domain and needs better than anyone else, and are often aware of specificities in their respective contexts. And two, end users outnumber developers with formal training at least by a factor of 30-to-1. This however, does not mean that experienced developers have nothing to gain from this research. Lowering the cognitive load of certain types of software development could save time and energy which can then be spend on more worthwhile and demanding tasks.

Not all EUD applications are VPLs. A good example of this is Elliott [2007] constructed a GUI algebra system to allow non-cli-based application to ‘pipe’ data

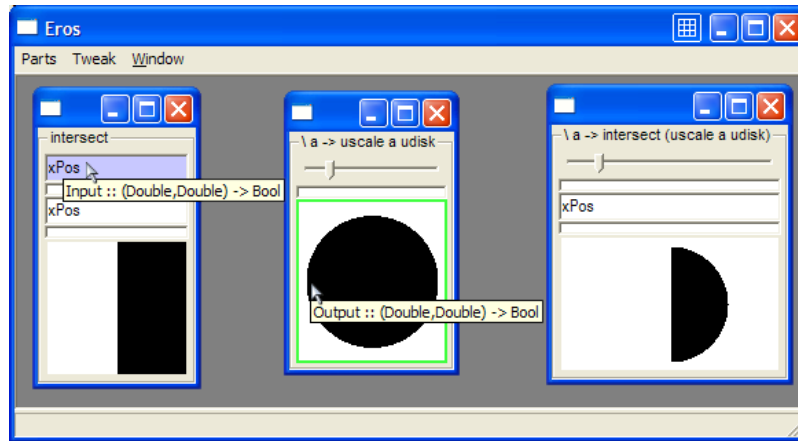


Figure 6: End User development by means of GUI algebra [Elliott, 2007]

between applications, just like how UNIX-based programs can be composed into pipes Figure 6.

In the private sector, EUD is represented by the "low code" industry. Technology firms such as Google and Amazon are investing at scale in low-coding platforms [Kuhail et al., 2021]. The market value was estimated at 12.500 Million USD, and with a growth rate between 20 and 40 percent, the value may reach as high as 19 Billion by 2030 [Itd, 2021]. Despite this being just market speculation, it does give an indication in a general need for end-user development solutions.

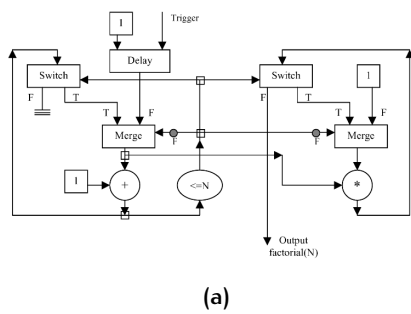
2.2.3 Dataflow programming

An important aspect of the dataflow-VPL is the connection to the field of dataflow programming, which is also a more general field than VPLs in particular.

Dataflow programming is a programming paradigm which internally, represents a program as a DAG [Sousa, 2012]. A graphical, editable representation of a dataflow program would result into a Dataflow VPL.

The big computational advantage of this model, is that it allows for implicit concurrently [Sousa, 2012]. In other words, every node of a program written using dataflow programming can be executed in isolation of any other nodes, as long as the direct dependencies (the inputs) are met. No global state or hidden side effects means no data-race issues, which allows parallel execution of the program by default. When using other paradigms, programmers need to manually spawn and manage threads to achieve the same effect.

This leads into a valuable side-effect of using dataflow programming / a diagram-based VPL: By only permitting pure, stateless functions with no side-effect, and only immutable variables, end users automatically adopt a functional programming style (albeit without lambda functions). Functional programming has many



(a)

```

LOOP WITH i = N, fact = 1
  NEW fact = fact * i;
  NEW i = i - 1;
WHILE i > 1;

```

(b)

Figure 7: A factorial function, written in a VPL, and in a textual language [Sousa, 2012]

benefits of its own besides concurrency, such as clear unit testing, hot code deployment, debugging advantages, and lending itself well for compile time optimizations [Akhmechet, 2006; Elliott, 2007].

All that to say, creating a VPL is not just a matter of designing a stylistic, user-friendly GUI alternative to regular programming. This might be true for other types of VPLs, but not for diagram-based ones. By closely resembling dataflow itself, and because of its functional programming nature, diagram-based VPLs may actually lead to faster and more reliable software.

2.2.4 Disadvantages and open problems

VPLs and dataflow programming en large have got certain disadvantages and open problems:

Iteration and conditionals

A problem described in almost all reviewed VPL literature [Green and Petre, 1996; Sousa, 2012; Kuhail et al., 2021], is that the DAG model of diagram-based VPLs are ill-suited for representing even the most basic flow control statements: `if`, `else`, `for`, `while`. Even if the acyclic quality of the dataflow graph is omitted, the resulting models are significantly more complicated compared to their textual counterparts, as shown by Figure 7.

Encapsulation & reusability

Similar and yet different is the topic of encapsulation, or, how Green and Petre [1996] names this problem: ‘visibility’. It is widely known that as a program scales in size, the complexity of managing the application scales exponentially with it. In textual languages, reducing this complexity is often achieved by means of encapsulating sub-routines and re-usable parts of the program into separate functions. Inner functionality is then hidden, and operations can be performed on a higher

level of abstraction. This hierarchy of abstraction is just as achievable for VPLs as described by Sousa [2012]. However only a select number of VPLs offer a form of encapsulation, and even less allow the creation of reusable functions, or creating reusable libraries from VPL scripts. It appears that VPL researchers and developers are either not aware of the importance of encapsulation, or have encountered problems in representing this feature in a graphical manner.

Subjective Assessment

Additionally, the claims that VPLs lend themselves well for end-user development is problematic from a technical perspective. Usability is a nebulous phenomenon, and challenging to measure empirically. As often with more subjective matter, researchers have yet to form a consensus over a general evaluation framework. There is, however, a reasonable consensus on the 'qualities' a VPL should aspire to. This is different from a full assessment framework, but nonetheless useful for comparing VPLs. The dimensions given by the cognitive dimensions framework [Green and Petre, 1996] have acquired a somewhat canonical nature within VPL research. The number of citations of this work is relatively high, and indeed, almost all VPL studies the author was able to find referred back to this work. In so far as this study needs to address the usability of the prototype VPL, we will thus follow this consensus, and base any assessment on this framework.

Life-cycle support

Finally, Kuhail et al. [2021] names the 'life cycle' of applications created by VPLs as one of the most overlooked aspects within VPL research. Out of the 30 studies covered by the meta analysis, only one briefly touched the topic of life cycle. Life cycle in this context refers to all other activities besides "creating an application that does what it needs to do". Examples of these activities are version control, extending an existing application, debugging, testing the codebase, and publishing the application to be used outside of an IDE. These operational aspects are important to making any application succeed, and EUD research should not be limited to purely the aspect of creating functionalities.

This literary study agrees with the findings of Kuhail et al. [2021]: Not one of the open source VPLs mentioned by this chapter or the upcoming Section 3.2 or Section 3.3, contained life-cycle aspects like Git version control, or integration / delivery pipelines. For paid VPLs, deployment and version control is sometimes possible for a fee (See Safe-Software [2022b]). However, in such a situation, users are limited to the publication tools, version control tools, and package / library managers offered to them by the vendors.

And on the topic of publication, only 16 out of 30 of the tools analyzed by Kuhail et al. [2021] were available publicly with some documentation. It seems the lack of publication tooling might be partially due to a lack of publication in general.

2.2.5 Conclusion

The background literature clearly indicates many advantageous properties of VPLs, both in terms of (end) user experience and the dataflow programming properties. Additionally, the studies showed important considerations which have to be taken into account in the design of any VPL. Lastly, the studies agree on several open-ended issues of which a satisfying answer is yet to be found.

3 | RELATED WORKS

This chapter offers a review of related and comparable studies and projects. While almost no studies exist at the intersection of all three of these fields, we do find many related studies and projects which intersect two of these fields, represented by the edges of:

- Section 3.1 reviews related works on browser-based geoprocessing
- Section 3.2 reviews related works on VPLs used for geo-computation
- Section 3.3 reviews related works on VPL web applications

3.1 BROWSER-BASED GEOCOMPUTATION

This section is dedicated to related works on client-side geocomputation, or browser-based geocomputation. this study prefers to use "browser-based geocomputation" in order to circumvent the ambiguity between native clients like QGIS [QGIS Community](#) [2022], and web clients like Omnibase.

Browser-based geocomputation has seen some academic interest throughout the last decade [Hamilton](#) [2014]; [Panidi et al.](#) [2015]; [Kulawiak et al.](#) [2019]. Interactive geospatial data manipulation and online geospatial data processing techniques have been described as "current highly valuable trends in evolution of the Web mapping and Web GIS" [Panidi et al.](#) [2015]. An example of this is the Omnibase application [[Geodelta](#), 2022] in Figure 8, used by Dutch municipalities to measure buildings and infrastructure based on point clouds and oblique multi-stereo imagery.

browser-based geocomputation, compared to native GUI or CLI geocomputation, allows geocomputation to be more accessible and distributable. Accessible, since geocomputation on the web requires no installation or configuration, and distributable, since the web is cross-platform by default, and poses many advantages for updating, sharing, and licensing applications. Lastly, by performing these calculations in the browser rather than on a server, server resources can be spared, and customly computed geodata does not have to be resent to the user upon every computation request.

However, browser-based geocomputation poses multiple challenges. Browsers & JavaScript are not ideal hosts for geocomputation. As an interpreted language, Javascript is slower and more imprecise compared to system-level languages like C

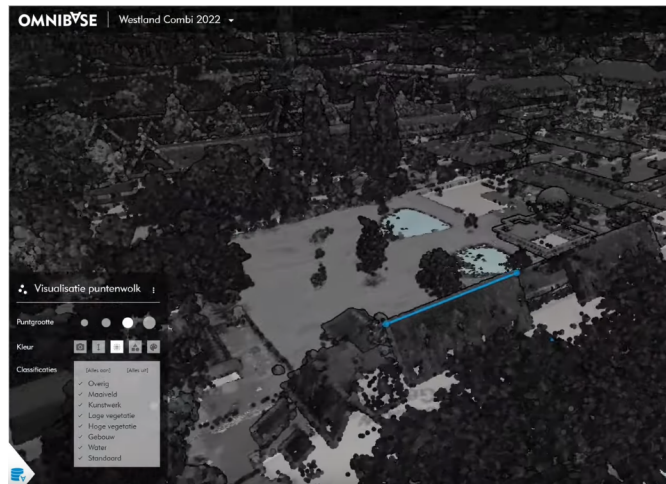


Figure 8: Omnibase: An example of browser-based geocomputation [Geodelta, 2022]

/ C++. In addition, it has limited support regarding reading and writing files, and does not possess of a rich ecosystem of geocomputation libraries. Novel browser features like WebAssembly may pose a solution to some of these open questions, but this has not seen substantial research.

3.1.1 Examples

Hamilton [2014] created a 'thick-client', capable of replacing certain elements of server-side geoprocessing with browser-based geoprocessing. The results of this study were unfavorable. The paper states how "the current implementation of web browsers are limited in their ability to execute JavaScript geoprocessing and not yet prepared to process data sizes larger than about 7,000 to 10,000 vertices before either prompting an unresponsive script warning in the browser or potentially losing the interest of the user." [Hamilton, 2014]. While these findings are insightful, they are not directly applicable to the efforts of this study proposal. Three reasons for this:

- The paper stems from 2014. Since then, web browsers have seen a significant increase in performance thanks to advancements in JavaScript JIT compilers [Haas et al., 2017; Kulawiak et al., 2019].
- The paper does not utilize compile-time optimizations. The authors could have utilized 'asm.js' [Mozilla, 2013] which did exist at the time.
- The paper uses a JavaScript library which was never designed to handle large datasets.

The same statements can be made about similar efforts of Panidi et al. [2015]. However, Panidi et al. [2015] never proposed browser-based geoprocessing as a replacement of server-side geoprocessing. Instead, the authors propose a hybrid approach, combining the advantages of server-side and browser-based geoprocessing. They

also present the observation that browser-based versus server-side geoprocessing shouldn't necessarily be a comparison of performance. "User convenience" as they put it, might dictate the usage of browser-based geoprocessing in certain situations, despite speed considerations [Panidi et al. \[2015\]](#).

This concern the general web community would label as User Experience (UX), is shared by a more recent paper [Kulawiak et al. \[2019\]](#). Their article examines the current state of the web from the point of view of developing cost-effective Web-GIS applications for companies and institutions. Their research reaches a conclusion favorable towards browser-based data processing: "[Client-side data processing], in particular, shows new opportunities for cost optimization of Web-GIS development and deployment. The introduction of HTML5 has permitted for construction of platform-independent thick clients which offer data processing performance which under the right circumstances may be close to that of server-side solutions. In this context, institutions [...] should consider implementing Web-GIS with client-side data processing, which could result in cost savings without negative impacts on the user experience."

From these papers we can summarize a true academic and even commercial interest

in browser based geoprocessing over the last decade. However, practical implementation details remain highly experimental, or are simply not covered. The implementations of [Panidi et al. \[2015\]](#); [Hamilton \[2014\]](#) were written in a time before WebAssembly & major JavaScript optimizations, and the study of [Kulawiak et al. \[2019\]](#) prioritized theory over practice. Additionally, to the best of the authors's knowledge, all papers concerned with browser-based geoprocessing either tried to use existing JavaScript libraries, or tried to write their own experimental WebAssembly / JavaScript libraries. No studies have been performed on the topic of compiling existing C++/Rust geoprocessing libraries to the web.

3.1.2 Commercial web-based geocomputations software

Despite the earlier statement of the general lack of Geospatial data computation ([geocomputation](#)) within browsers, there are exceptions. A select number of web-based GIS applications are starting to experiment with empowering end-users with geocomputation. These applications will briefly be mentioned.

GeoTIFF ([\[Dufour, 2017\]](#), [Figure 9](#)), is a web-based, open source, geoTIFF processing tool. It offers basic operations such as taking the median or & mean of a certain area, color band arithmetic, and can plot histograms, all calculated within the browser using customly written JavaScript libraries.

The modelLab application by Azavea, is also a GeoTIFF / raster based web processing tool, in which basic queries and calculations are possible [\[Azavea, 2011\]](#). This tool offers more advanced types of geocomputation, like buffering / minkowski sums, and even multi-stage processing via a simple but clear visual programming

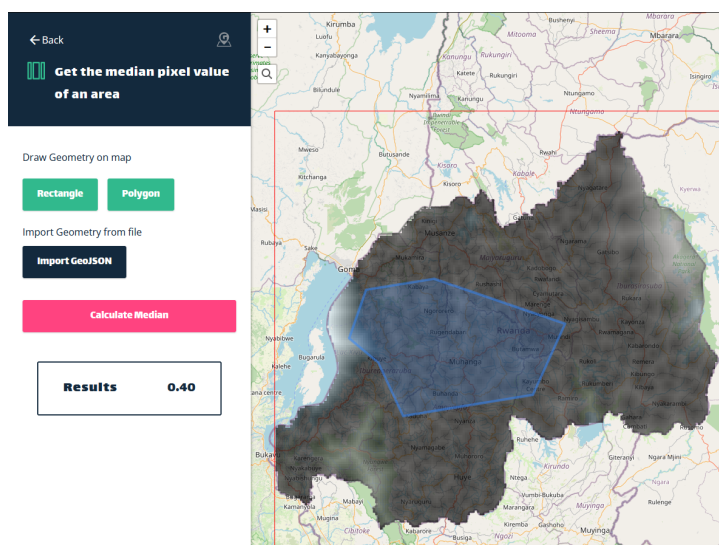


Figure 9: The geoTIFF.io application [Dufour, 2017]

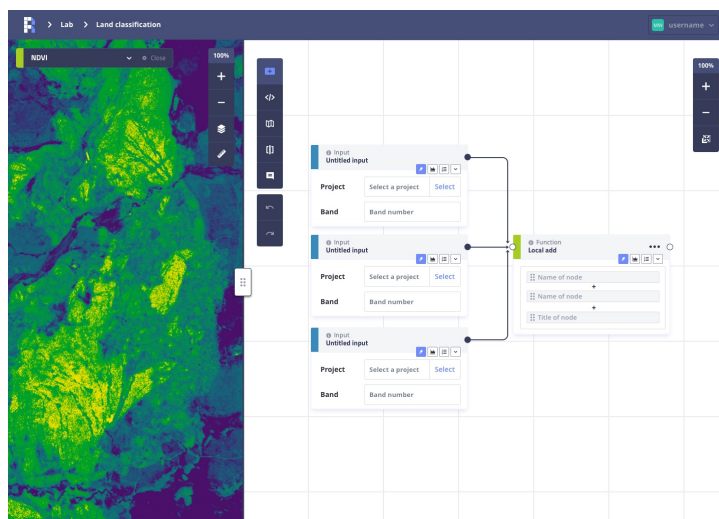


Figure 10: The ModelLab application [Azavea, 2011]

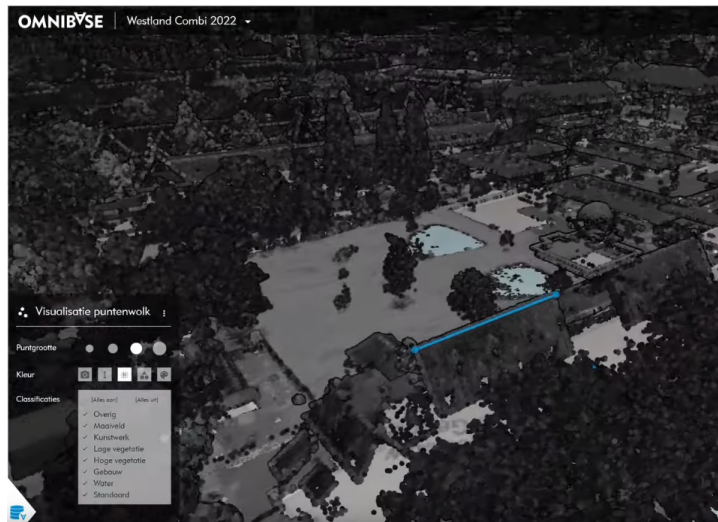


Figure 11: The Omnibase application [Geodelta, 2022]

language (see Figure 10). However, the tool uses mostly server-side processing, making this application less relevant to this study.

The last web-based geocomputation platform this study would like to mention is Geodelta's Omnibase application [Geodelta, 2022] (see Figure 11). Omnibase is a 3D web GIS application for viewing and analyzing pointclouds and oblique multi-stereo imagery. It offers client-side geocomputation in the form of measuring distances between locations, and calculating the area of a polygon. It also offers photogrammetry-techniques such as forward incision of a point in multiple images, but these are calculated server-side.

3.2 VISUAL PROGRAMMING AND GEOCOMPUTATION

This section is dedicated to giving an overview of related works on VPLs related to geocomputation.

Figure 12 offers this overview of some of the more significant VPLs present in not only GIS, but also the neighboring domains based on computer graphics.

VPLs in GIS

Within the field of geo informatics, VPLs are not a new phenomenon. VPLs have been used for decades to specify geodata transformations and performing spatial analyses.

Domain	Name	Related Software	Author	License	Cost	Purpose	
1	GIS	FME	-	Save Software	Proprietary	€ 2,000 one time	ETL
3	GIS	The Graphical Modeler	QGIS	QGIS Contributors	Open Source	-	Geoprocessing
4	GIS	Model Builder	ArcGIS	Esri	Proprietary	\$100 p.y. (ArcGIS)	Geoprocessing
2	GIS	geoflow	-	Ravi Peter	GNU Public License Version 3.0	-	ETL for 3D geoinformation
5	Photogrammetry	MeshRoom	-	AliceVision	Mozilla Public License Version 2.0	-	3D Reconstruction & photomodeling
6	Procedural geometry	Geometry Nodes	Blender	Blender Foundation & Contributors	GNU Public License Version 3.0	-	Procedural Modelling & Special effects
7	Procedural geometry	Grasshopper	Rhino	David Rutten / McNeel	Proprietary	€ 995 one time (Rhino)	Procedural Modelling / BIM
8	Procedural geometry	Dynamo	Revit	Autodesk	Proprietary (Revit is needed in execution)	€ 3,330 p.y. (Revit)	BIM
9	Procedural geometry / Shader programming	Houdini	-	SideFX	Proprietary	€ 1,690 p.y.	Procedural Modeling, material modeling, Special Effects
10	Shader programming	Shader nodes	Blender	Blender Foundation & contributors	GNU Public License Version 3.0	-	Textures and material modelling & animation
12	Shader programming	Substance Designer	-	Adobe	Proprietary	\$ 240 p.y.	Textures and material modelling & animation
13	Shader programming	Material Nodes	Unreal Engine	Epic Games	Proprietary	Semi-free / \$1,500 p.y. (UE)	Textures and material modelling & animation
14	Shader programming	Shader Graph	Unity	Unity Technologies	Proprietary	Semi-free / \$400 p.y. (Unity)	Textures and material modelling & animation
15	Game engine programming	Blueprints	Unreal Engine	Epic Games	Proprietary	Semi-free / \$1,500 p.y. (UE)	Game programming
16	Game engine programming	Bolt	Unity	Unity Technologies	Proprietary	Semi-free / \$400 p.y. (Unity)	Game programming

Figure 12: An overview of VPLs in the field of GIS and adjacent domains

The most well-known visual programming language within the field of GIS is the commercial Extract Transform Load (ETL) tool FME [Safe-Software, 2022b], (see Figure 13a). This tool is widely used by GIS professionals for extracting data from various sources, transforming data into a desired format, and then loading this data into a database, or just saving it locally. FME is most often used within GIS to harmonize heterogenous databases, and as such specializes in tabular datasets.

The two major GIS applications ArcGIS and QGIS also have specific VPLs attached to their applications. The main use-case for these VPLs is to automate repetitive workflows within ArcGIS or QGIS.

Lastly, Geoflow is a much newer VPL meant for generic 3D geodata processing [Peters, 2019]. While this application is still in an early phase, it already offers a powerful range of functions. It offers CGAL processes like alpha shape, triangulation and line simplification, as well as direct visualization of in-between products. Geoflow was used to model the 3D envelope of a building based on a pointcloud, which was subsequently scaled up in the creation of the 3D BAG dataset [Peters, 2019].

VPLs in neighboring domains

Figure 12 shows a great number of non-GIS VPLs. While these do not explicitly cover GIS, their close ties to computer graphics are still highly relevant to GIS and the activity of geocomputation.

The choices of which VPL to include in Figure 12 are based upon popularity. The particular ones chosen see a lot of use, evident by the sheer number of courses and tutorials which cover these VPLs, and the popularity of the software packages these applications are attached to. In fact, many of the mentioned VPLs are popular enough that it is safe to say that VPLs are common in the wider field of

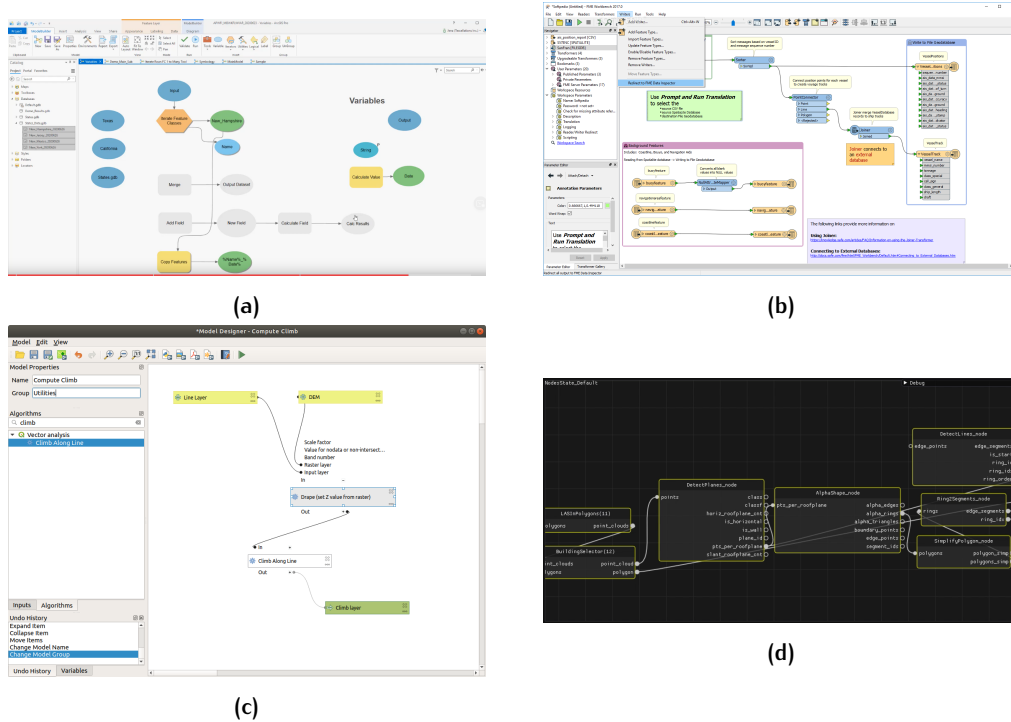


Figure 13: Four VPLs used in the field of GIS: ArcGIS's Model Builder [Esri, 2022] (a), Save Software's FME [Safe-Software, 2022b] (b), QGIS's Graphical Modeler QGIS Community [2022] (c), and Geoflow [Peters, 2019] (d).

computer graphics. This study limits itself to four sub-domains relevant to geocomputation:

- VPLs to calculate materials, shaders and textures
- VPLs to calculate geometry
- VPLs for photogrammetry
- VPLs to calculate behavior and logic

Commonalities

One interesting fact is that we see a great number of parallels among all these VPLs.

- All are diagram-based VPLs.
- All offer inspection of in-between products. Some even visualize data being parsed between nodes.
- All emphasize a process of "parametrization": parameters of various functions can be configured using sliders, curves, and other GUI elements. This allows quick experimentation of different settings.

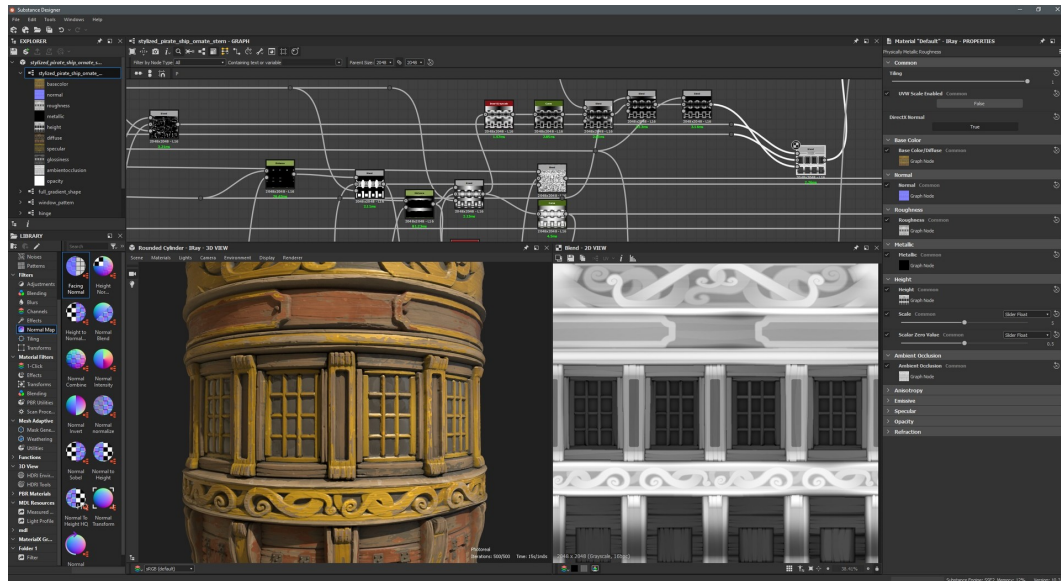


Figure 14: Substance designer, a VPL for textures [Rutten, 2012]

Moreover, the persistence of visual programming within these computer graphics fields, suggests that visual programming languages are advantageous for calculations dealing with 2D and 3D data.

One hypothesis of why this might be, is that all these VPLs, with exception to the behavior VPLs, are essentially dealing with "functional data pipelines". These VPL pipelines are straightforward calculations, and do not require to handle distributed systems or use event driven architectures. However, the sheer amount of possible steps within these pipelines, together with the challenges of fine-tuning many relevant parameters, and the importance of inspecting in-between products visually, may lead to VPLs being a good fit.

Material VPLs

One of the most commonplace types of VPL present in computer graphics is the material VPLs. These types of VPLs can be found in special effects applications, modelling applications, and game engines. In this context, the concept "material" often refers to a combination of procedural textures and shaders. These may include PBR settings, normal maps, bump maps, and / or custom shader programs. The repetitive and time-consuming nature of manually creating textures, and the fact that some of these material properties can be inferred from each other, lead many CG applications to develop VPLs for this particular purpose. 3D artists use these VPLs to create procedural materials.

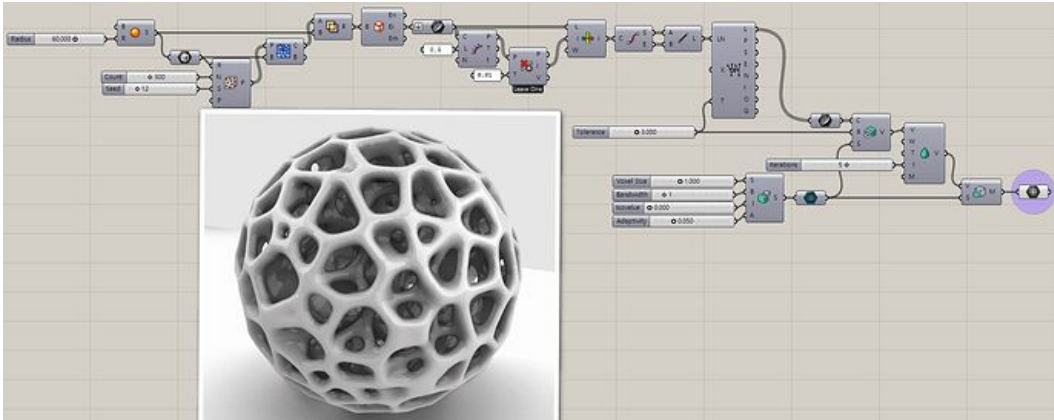


Figure 15: Grasshopper, a VPL for geometry [Rutten, 2012]

Geometry, and photogrammetric VPLs

Procedural Geometry VPLs are not far behind the material VPLs in terms of popularity. Applications like Blender's geometry nodes [Blender Foundation and Contributors, 2022], Rhino's Grasshopper [Rutten, 2012], or Houdini [SideFX, 2022], are all widely used to automate the creation of geometry. Where Houdini and Blender's VPLs are primarily used in games and special effects, Grasshopper sees usage in the Architecture, Engineering and construction industry. In this field, procedural geometry is often referred to as "parametric design".

Alicevision's Meshroom application must also be mentioned [Alicevision, 2022]. While this can be regarded as procedural modelling, the complexity and computation involved in photogrammetry make a VPL offering it a class in of itself. The VPL inside of Meshroom can be used to fine tune all stages of the 3D reconstruction process.

Behavioral VPLs

The behavioral and logical VPLs found in applications such as Unreal's Blueprint [Epic Games, 2022] and Unity's Bolt [Unity Technologies, 2021] are less relevant to the activity of geocomputation. However, one interesting property worth mentioning, is that these languages have actually designed a way for end-users to define imperative flow statements, since these could not be overlooked for behavior and logic. Section 2.2 named conditions and loops as one of the challenges of diagram-based VPLs. These languages both attempted to solve this problem by introducing a special "flow state" variable. It represents no value, but simply the activity of 'activating' or 'doing' the node selected. Figure 16 showcases these flow-state variables in both languages using conditionals. flow-state variables have their own set of rules, completely separate from connections carrying data. For example, they can be used cyclically, offering users looping functionality and are allowed to have multiple sources. Despite these functionalities, one might wonder if these aspects are

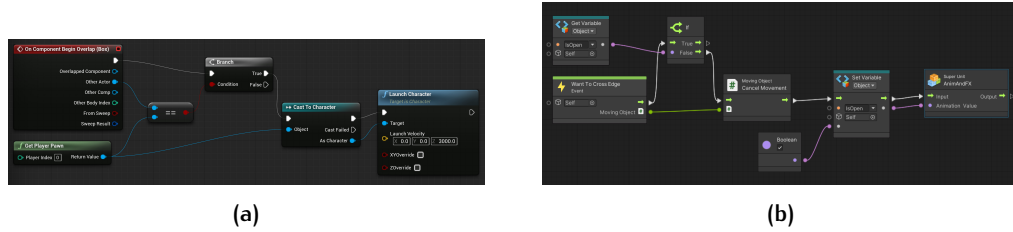


Figure 16: Two VPLs for logic, showing "flow-state" variables. Left: Unreal's Blueprints, Right: Unity's Bolt. [Epic Games, 2022; Unity Technologies, 2021]

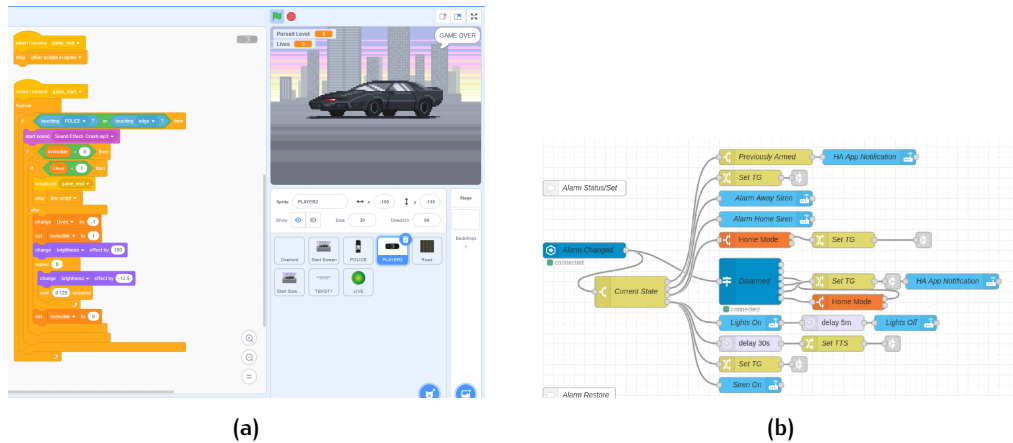


Figure 17: Two VPLs used on the web: Scratch (a) [Resnick et al., 2009], and nodeRED (b) [OpenJS Foundation, 2022].

worth these extra complications. Especially since these flow-state variables are effectively GOTO statements, which are widely known as an anti-pattern in large-scale software projects.

3.3 BROWSER-BASED VISUAL PROGRAMMING

This section is dedicated to visual programming applications running in a browser. It must be emphasized that of all the various VPLs named in Section 3.2, none are browser-based. This is likely the case because most of those VPLs are computationally intensive, C / C++-based applications.

Nevertheless, if one looks in other domains, we quickly see many VPLs which are web-based. Out of all 30 VPL studies covered by the meta analysis of Kuhail et al. [2021], 17 were web based, 7 were mobile based, and only 6 were desktop applications. Kuhail et al. continue by noting that most of these 6 desktop applications were build during or before 2013. The reason Kuhail et al. give for this stark difference is in line with research covered in Section 2.1: "This can be explained by the fact that desktop-based tools are cumbersome to contemporary users. They must be downloaded and installed, are operating-system dependent, and need frequent updates."

This study wishes to present two web based visual programming languages, which each use the web in a meaningful way. The first web-vpl is "Scratch" [Resnick et al., 2009] (See Figure 17a). Scratch is well-known as an educational, block-based VPL, targeted at children and young adults to teach the basics of computational thinking. As noted by the authors of CS50, scratch is, despite this target audience, surprisingly close to any normal programming language, with for and while loops, if statements, and even event handling and asynchronous programming. Scratch used to be a desktop application. The web environment this VPL now occupies allows its users some life-cycle support. Users can immediately publish their work, search for and run the work of others, and even "Remix / clone / fork" the source code of these other projects. This encourages users to learn from each other.

The second exemplary web VPL this study wishes to bring to the readers attention is the "nodeRED" application [OpenJS Foundation, 2022] (see Figure 17b). This is a feature-rich diagram-based application, created to serve the domain of IoT. This VPL uses the browser-based platform not only for the aforementioned Section 2.1 reasons, but also for the exact same reasons a router, NAS or IoT device often opts for a browser-based interface: Servers, either small or big, explaining how they desire to be interfaced, is more or less the cornerstone all web clients are based upon. If the server serves its corresponding client, users do not need to find some compatible interface themselves. For this reason the "nodeRED" application is a web application, even though it is mostly run on local networks.

3.4 BROWSER-BASED GEOCOMPUTATION USING A VPL

To the best of the author's knowledge, only one publicly available visual programming language exist which is both able to be configured and executed in a browser, and is able to be used for geodata computation. This application is called the Möbius Modeller [Janssen, 2021], and is the closest equivalent to the geo-web-vpl proposed by this study. Though it only uses JavaScript, the tool is able to be successfully used for a number of applications, including CAD, BIM, urban planning, and GIS. It uses a combination of a 'bare-bones' diagram-based VPL, together with a rich block-based VPL (See Figure 18). In fact, the block-based VPL is so rich that is almost ceases to be a VPL altogether, and starts to be python-like language with heavy IDE support.

The VPL proposed by this study still differs from the mobius modeller in the following ways:

- This study explores the usage of a pure dataflow VPL, as opposed to the multiple types of VPLs used by the Möbius Modeller. This is done to allow for the dataflow programming advantages described in Section 2.2.3.
- This study explores the usage of WebAssembly to hypothetically improve performance and to use existing geocomputation libraries.

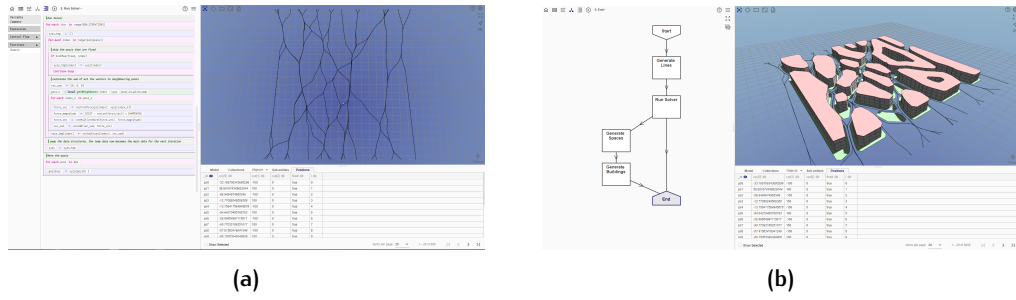


Figure 18: Images of the Mobius modeller application [Janssen, 2021]

- This study addresses some of the life-cycle issues of VPLs stated in Section 2.2.4.

4 | METHODOLOGY

This chapter present the design of the proposed method. It first sums up and clarifies all requirements of the method, followed up by presenting an overview, and then the various aspects to the design of the method. It finishes by elaborating on the assessments used to test the environment.

4.1 DEFINITIONS

Throughout this and subsequent chapters, the following terms and phrases will be used.

Pipeline / Script: A program 'written' within a VPL is referred to as a 'script' or a 'pipeline'.

VPL as Programming Language or Application: An ambiguity exist of calling a VPL an application or a programming language. Arguments can be made for both. This study would describe its VPL as an application, which so happens to possess a composable [GUI](#), and not a programming language. Nevertheless, it does possess language characteristics, and depending on context, the VPL will be referred to as both.

Plugin vs Library: A plugin represents additional, optional functionalities for a certain application. A library represents additional functionalities to be used within a programming language. Due to aforementioned ambiguity of a VPL, 'additional functionalities' for a VPL could be called both. To make matters worse, the upcoming plugin system will blur the lines between library and plugin. To offer some clarification, the term 'plugin' will be used when referring to the 'functionalities' from a point of view of the VPL. The term 'library' will be reserved for referring to a software library itself.

Node: a component or 'block' within a VPL pipeline. It is analogous to a function in a normal language.

Boilerplate: An informal term to refer to code which is not part of the core logic of an application. It is in a sense irrelevant code, but still required in order to use a certain system or feature.

Binding: Bindings refer to the code a maintainer of a library may add to allow it to be used in another language. For example, a C++ library can have python bindings. Bindings may sometimes be regarded as boilerplate.

Headless: Headless execution of a program means that it is run without using its normal GUI. Often, this means a Command Line Interface (CLI) is used instead.

4.2 REQUIREMENTS

Three supporting research questions ask to clarify requirements for the design of the methodology. These questions can now be partially answered based on the literature presented at Chapter 2 and Chapter 3, and based on insights granted by these works. The answers to these questions form the core requirements of the proposed method, and steer its features and design considerations.

What GUI features are required to facilitate this method? (and to what extent does the web platform aid or hurt these features?)

Two layers of GUI features are required:

Framework application: Firstly, a base web application is required to host the visual programming language. This needs to provide a GUI for all basic application features, like saving, loading, undoing and redoing.

The VPL: Secondly, UI elements are required to form the interactive elements of the visual program itself. This forms a layer independent from the framework application, and will require a custom set of GUI features. That is to say, a text field within the framework is not the same as a text field within the visual program. Existing VPLs differ in their level of support of certain 'GUI nodes'. This study has established that extensive support is required, for the sake of visualizing geodata, and parametrization of GIS operators.

To what extent does this method intent to address the discrepancies between software applications and libraries, as described by Elliott (2007) (Does it succeed in doing so?)

All three discrepancies described by the introduction may be addressed by incorporating the following aspects:

Generic GUI: The VPL must serve as a generic GUI to serve practically any GIS library. As such, it requires a wide range of buttons, sliders, text fields, and other inputs.

Composable applications: In order to make web applications themselves more composable, the GUI of the VPL itself should incorporate other web applications, via for example a <i>iframe</i> or popup.

Direct utilization: To allow direct utilization of a library, the solution must be distributed as a static web application. To make sure library capabilities do not get

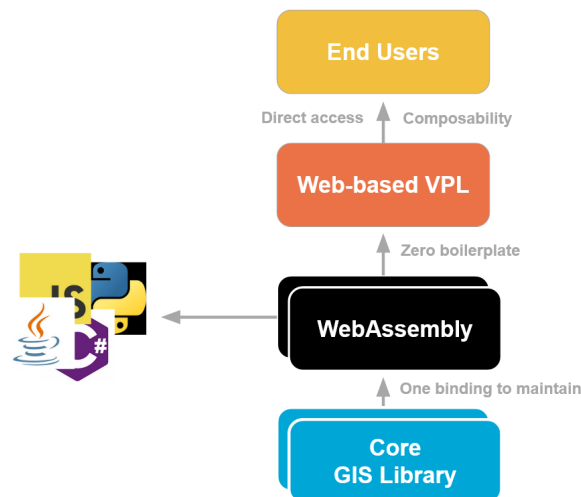


Figure 19: The proposed methodology

lost when used in an application, the VPL must be able to accept a wide range of libraries as plugins, with as little bindings or glue code as possible.

What measures are taken to make this VPL scalable to large geo-datasets? (and how effective are these measures?)

Three measures can be taken to offer scalability in principle:

Portability: A Backend execution of a VPL pipeline must be predictable. As such, GIS libraries run on the frontend in the VPL must behave exactly the same as if they were run on a backend.

Zero-cost abstraction: A pipeline created with the VPL should be able to run headless, to ensure a scalable backend can use such a pipeline.

Locality: The VPL should be designed as a Dataflow VPL, which shares characteristics with functional programming. This leads to source code which can be reasoned about in a local manner, instead of a global one. This allows for parallelization when scaled in a backend environment.

4.3 OVERVIEW

The design of this method is as visualized by Figure 19: A Prototype VPL is used to host the functionalities of GIS libraries from within an application, and in a composable manner. Additionally, it is used to connect these libraries to various GUI features. The format of web application is used to allow this prototype to be directly accessible to end users without installation or configuration. This prototype is statically hosted, to minimize operational costs. GIS libraries are loaded within this platform by first compiling them into WebAssembly, and then loading this binary

as a plugin directly. This method is dubbed a 'no-boilerplate' method, and explores how libraries can be used with as little in-between layers as possible. WebAssembly is also used so that libraries written in native languages can be used without resorting to active backend web services. Finally, as this prototype is intended for GIS usage, both scalability to handle sizable datasets, and rich GUI support (3D viewers, file inputs, sliders), are primary design considerations and assessment criteria.

4.4 FRAMEWORK APPLICATION

To serve the requirements of a **Framework application** and **direct utilization**, a statically hosted web application will be required, to serve as a framework in which the VPL can be embedded. A UI library akin to `imgui` [Cornut, 2022] or `egui` [Ernerfeldt, 2019] would be optimal in providing such an environment. However, due to the absence of such a desktop application-like framework in the JavaScript ecosystem, a custom implementation will be needed. Using HTML5 features like Web Components, custom components such as multiple windows, dropdown menu's, side menu's etc. can be created without resorting to JavaScript frameworks.

4.5 WEB VPL

Secondly, a web-based VPL is required as a host for all subsequent steps of the methodology. This VPL must serve the requirements of **The VPL**, a **Generic GUI**, **Composable applications** and **Locality**.

A novel, custom VPL is needed to fit all requirements. A dataflow VPL must be designed for web usage, which offers a range of GUI nodes to serve as a generic GUI, and to allow applications to be composable.

While the intention was to re-use an existing web VPL, the related works review of Chapter 3 showcased that no existing browser-based geocomputation VPL would be an appropriate fit. The Mobius Modeller [Janssen, 2021] came closest, but offered no dataflow VPL, no GUI components, and no plugin support. Additionally, the sizable nature of the mobius modeller project made aligning its goals with the goals of this study challenging. Building a custom implementation would allow more degrees of freedom.

The following approach was deemed as the most fitting method for implementing this VPL. First, the components of a dataflow-VPL handling geometry have to be made clear. Secondly, in order to know what tools may be used to implement this VPL, a small analysis of "widely supported browser features" is made. Then, with both these constraints known, a design for a web VPL can be laid out, which can be subsequently implemented.

4.5.1 Components

The components of a dataflow-VPL implementation can be subdivided in components of a dataflow VPL in general, and a VPL for geo-computation specifically. Based on the literature study of Section 2.2, any dataflow-VPL must at least contain the following aspects:

- a base 'programming language model'
 - A representation of the 'variables' and 'functions' of the language
 - With all computations being pure functions
 - With all variables being immutable
- a 'graph-like' visualization of this data model
- an interface to create and edit this graph
- a way to provide input data
- a way to execute the language
- a way to display or save output data

The implementation of these aspects would result in a 'baseline', general purpose, dataflow VPL. To specialize this implementation further, A visual programming language handling geometry should have:

- Type safety
- A way to load or to create geometry data
- A way to export geometry data
- A method to preview geometry data in 3D
- A standard set of geometric types and operations

These requirements need further explanation. First, regarding type safety. In this context, type safety refers to: The input and output of a function should have a type stated, and users should be notified of incorrect usage of types, or 'invalid connections'. Geometry VPLs in particular need this, as many data representations of geometry are required to be precise about their data usage. A VPL used to construct geometry should reflect this. Additionally, when these types are clear and clearly communicated, users must have ways to provide these types as inputs or outputs. This will require specialized parsers to become part of the VPL, such as an obj and geojson reader and writers.

Regarding visualization, A hallmark of VPLs is the ability to inspect the data of in-between steps, so this must be provided for. This is also a good fit, since the immutable nature of dataflow VPL variables make these variables ideal for caching.

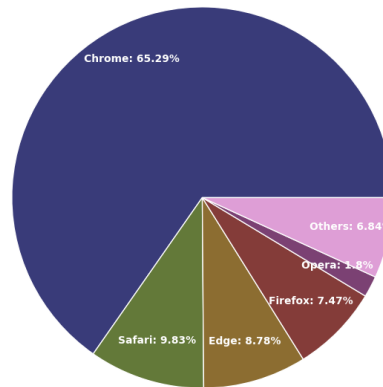


Figure 20: Fall 2021 desktop browser usage statistics. Data averaged over ([Dashiki, 2020; W3Counter, 2020; The NetMarketshare Team, 2020; StatCounter Global Stats, 2020])

Finally, a geometry VPL should contain a set of 'internal', basic types and operations. All aforementioned features are difficult to implement without defining some set of internally recognized data types. Basic operations are needed in particular to transform between the types.

4.5.2 Widely supported browser features

This study defines "widely supported browser features" as the set of default features implemented by the browser engines of 'major browsers'. Based on the desktop browser market shares of Figure 20, the chromium based browsers (Chrome, Edge, Opera) have the majority. This is followed up by Firefox, based on the Gecko engine, and Safari, based on webkit. By supporting these three engines, the vast majority of end-users can be served.

The set of features common in these three browser engines are well-documented on websites like MDN web docs [Mozilla, 2022]. This set includes the following features relevant for the 3D VPL:

- WebGL & WebGL2 (WebGPU is not widely supported yet)
- 2D Canvas API
- Web Workers
- Web Components
- WebAssembly

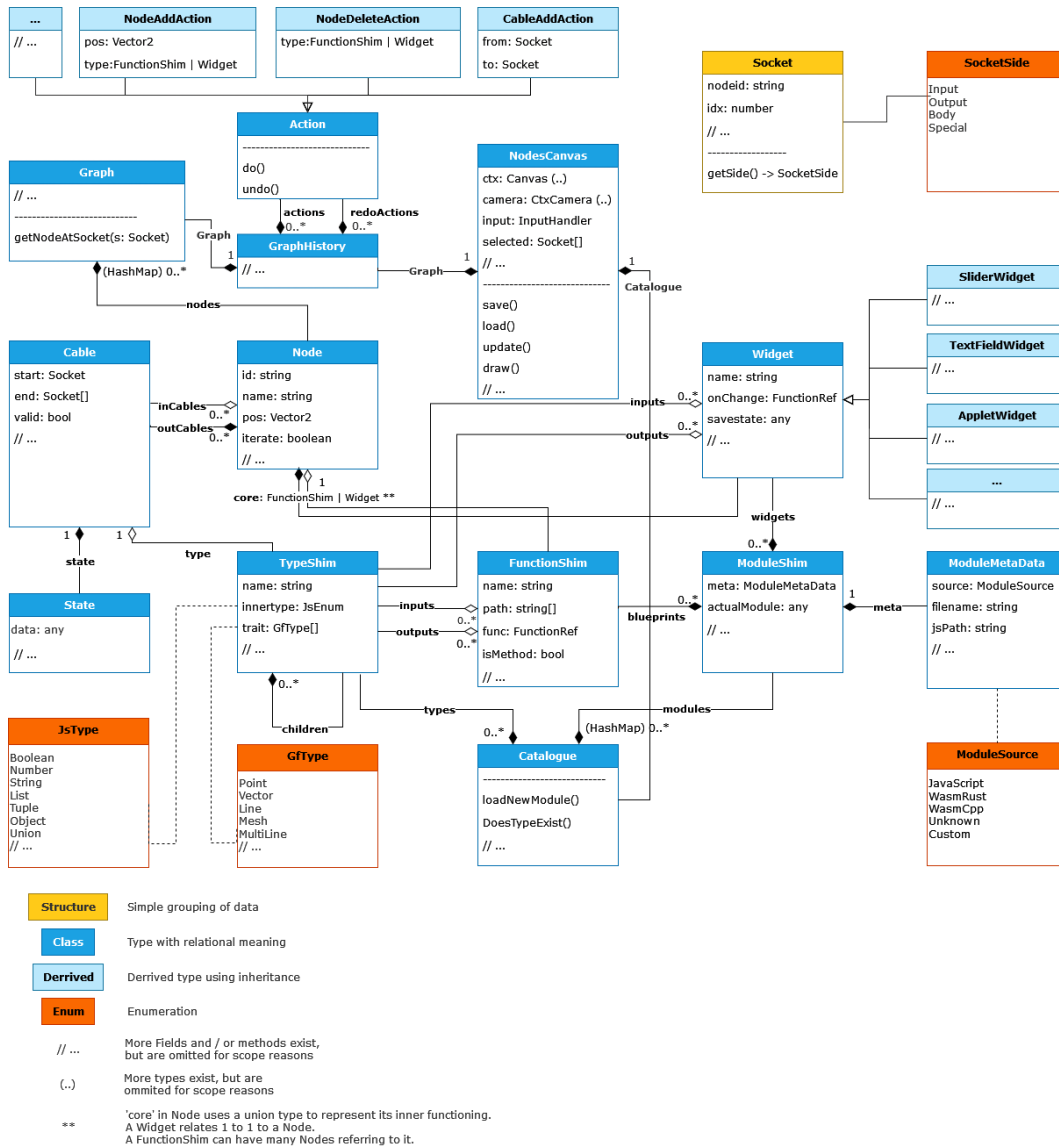


Figure 21: A UML diagram of the proposed VPL

4.5.3 Design

A software application of a VPL adhering to the specifications mentioned can be implemented in several ways. The design chosen is a Model View Controller (MVC) setup written in JavaScript. The MVC is a common model for interface-focused applications, and allows us to reason about the model of the VPL language as a separate level from the editor / viewer. The JavaScript language will be used instead of webassembly alternatives, in order to limit the usage of webassembly to just the libraries. Using WebAssembly too much at too many different locations will make the results of this study less clear.

JavaScript is a multi-paradigm language. This study chose an object-oriented approach, and will use some of the design patterns layed out in [Gamma et al., 1994].

This design is further elaborated in the subsequent sections, first by covering the Shim classes, followed up by design details corresponding to the model, view and controller:

Shim Types

Firstly, since a VPL is partially a programming language, a model is needed to reason about some of the features of a programming language, such as functions, types, variables, and modules / libraries / plugins. For example, we desire to store a description of a function, how many input parameters it needs, and which variable types each input requires.

These needs led to the design of classes serving as equivalents of these language features, called shims. The UML diagram (Figure 21) shows a TypeShim and FunctionShim, and ModuleShim.

The shim classes are designed using the Object Type design pattern [Gamma et al., 1994]. This means that these objects are used as types. For example, a loaded function corresponds to exactly one FunctionShim instance, and that this instance is shared as a read only with any object wishing to eventually use the function. This is also useful for defining recursive types. TypeShims can be structured recursively to define a List of List of strings for example.

Model

From the Shims, the main model of a VPL pipeline can be conceptualized. This model is at its core a DAG. This DAG should be an object-oriented, graph-like representation of the data flow of a regular programming language. This design can be implemented by writing a Graph class, containing Node and Cable objects.

In this model, Nodes are analogous to function *invocations* of normal programming languages. As such, a Node knows about the function they represent through a FunctionShim reference. The node contains a number of input and output sockets based on this information, and each socket contains exactly one optional reference to a Cable. As the name implies, these Nodes form the nodes of the DAG. However, they differ from a pure DAG implementation, in that they also provide pointers back in the reverse direction, forming essentially a normal graph, or a doubly linked list. This is required for keeping track of all references pointing to a Node, so that upon the deletion of a node, all pointers can quickly be identified and nullified.

The Cables of this model are an analogy to the variables of regular languages. Cables know about the type they represent through a TypeShim. A Cable must have exactly one origin, which is an output socket of a Nodes, and must have one or more destinations, which are the input sockets of other Nodes. This is required for the same reasons as the doubly linked nature of the Cables.

To reason about the graph as a whole, a overarching Graph class will be needed. This is what would be called a 'program' or 'script' in a regular language. Because of the way Cables and Nodes reference each other, the graph has characteristics of a doubly linked list data structure. Using normal references in these types of situations could easily lead to memory management issues such as Dangling Pointers. For this reason, centralizing the graph logic is desirable over adding complex logic to individual Nodes. This will make it possible to substitute references with id integers to prevent these types of problems.

View

The view aspect of the VPL will require three main components. First, the graph itself will need to be visualized in some manner. A graph based visualization will be used, based on the node-cable connections of the graph model. Important to this view is that it will need to be redrawn often. Users will want to add, select, change, and delete nodes, and these interactions should be clearly represented. This makes the HTML5 'canvas API' an ideal fit to this component.

Secondly, since not all actions and interactions will be done by clicking on the graph itself, a GUI surrounding this graph visualization is required. This will also need to house common application features, like 'new', 'save', 'load', 'export', etc. The browser context means that this aspect will need to be facilitated by HTML. Styling is required to make what is essentially a website look and behave like an application.

Finally, the VPL requires some way of visualizing 3D geometry, so that in-between products containing spatial data can be viewed. a custom 3D engine, specialized to the needs of the VPL, would be best option for this aspect.

Controller

Finally, a controller will be needed to modify and manipulate the VPL (NodesCanvas in Figure 21). It will need to house all types of interactions, such as loading and saving a VPL script, model manipulation and updating the view only when necessary. Two important aspects require further explanation: Keeping track of history, and calculating the graph.

History

In order to support all these interactions, especially undo / redo support, we are required to explicitly track the history of the graph. A Command Pattern [Gamma et al., 1994] makes for a good fit in this regard. Instead of directly editing the graph, all manipulation actions should be represented as Action objects. Each Action can 'do' and 'undo' a specific action, and the data needed to make this do and undo are stored within the action. By then introducing a GraphHistory class (see Figure 21), the model and controller can be separated, only allowing interaction with

```

1 Step -1:
2   Make an 'order' list
3 Step 0:
4   Make a 'visisted' counter, initialized at 0
5 Step 1:
6   Make a 'dependency' counter for each node, initialized at 0
7 Step 2:
8   Add 1 to this counter for each input edge of this node.
9 Step 3:
10  Fill a queue with all dependency 0 nodes.
11  These are the starter nodes.
12 Step 4:
13  Remove a node from the queue (Dequeue operation) and then:
14  add the nodes' id to the 'order' list.
15  Increment 'visisted' counter by 1.
16  Decrease 'dependency' counter by 1 for all dependent nodes.
17  If one 'dependency' counter reaches 0,
18  add it to the queue.
19 Step 5:
20  Repeat Step 4 until the queue is empty.
21 Step 6:
22  If 'visisted' counter is not equal to the number of nodes,
23  then the graph was degenerate, and probably cyclical.

```

Figure 22: Khan's algorithm in pseudo code

the model by serving this bridge Action objects. The GraphHistory maintains a stack of undo and redo actions, which represents this history.

Calculation

When regarding the graph model, or any other programming language, we see many functions requiring variables which are the result of other functions. This is why a graph like this can also be called a dependency graph. If one wishes to calculate the result of a VPL script, then these dependencies must be taken into account. The functions the graph muse be sorted in such a way that all dependencies are known before a function is calculated. Such a problem is known as a topological sorting problem, and can be solved using Kahn's algorithm (Section 22):

Using this algorithm for calculating a VPL has several important qualities. First of all, it detects cyclical graph patterns without getting trapped within such a loop. VPLs implemented on the basis of an event-system suffer from this drawback, and models such as those must continuously check their own topology to avoid loops.

Secondly, by sorting the *order* of calculation before actually performing the calculations, we can use the algorithm for more than just the calculation. For example,

in theory this could be used to compile a VPL Script to Javascript at runtime, by composing a list of functions based on this order.

Finally, if all intermediate calculation results are cached, this same algorithm can also be used for performing partial recalculations of the graph. The starting positions of the algorithm then simply become the altered parameter, after which only the invalidated functions will recalculate.

Mutability

Recall that a variable in this VPL model always has one origin, and one or multiple destinations, just like variables a regular language. The calculation system requires to know the mutability of the destination function parameters. This has two reasons. First, to allow concurrent calculations, all functions using a variable should only be allowed to use a immutable references to this variable, in order to prevent data races. And secondly, to prevent unnecessary copies of variables, only one function should be allow to 'claim ownership' of a variable, as in, modify the data and pass it along as output, or delete it and free the memory. This is inspired by the Rust language model [Rust Contributors, 2022]. In such a system, concurrency can still occur between all other immutable references to this variable. However, only when all these calculations are done, may this final transformative step occur.

All this to say, the mutability of function parameters must be known in order to create a performant and memory efficient graph calculation system.

4.6 PLUGIN SYSTEM

The third step of the methodology involves developing a method to use GIS libraries from native sources within the VPL model outlined above. In other words, a plugin system is required, existing of a plugin model these libraries will need to adhere to, together with a design for an importer of those libraries at the side of the VPL.

4.6.1 Design

The design of the plugin system is centered around leveraging the ability of WebAssembly to be used as a generic interface. Certain `wasm` compilers, like Rusts `wasm-pack` compiler [wasm-pack Contributors, 2018], support interface types in the shape of TypeScript type declarations, or embedded within the WebAssembly binary itself. This follows a design proposal which is soon to be part of the `wasm` Standard itself [Wagner, 2022]. These interface types allows `wasm` to be used as an interoperable binding: One binary which could be used in any language, such as Python, C#, Javascript, or Java. Python and Java already have support for a `wasm` runtime [Clark, 2019].

The core concept behind the proposed plugin system, is to make the VPL utilize these bindings as well, and directly. For this to work, a to-be-imported GIS library needs to be compiled to WebAssembly, published to a public Content Delivery Network (CDN), and then referenced by the plugin importer, within the proposed VPL. The importer must then translate the exposed functions and types into nodes which can be used in a VPL pipeline. During execution, these nodes will then call functions from within the [wasm](#) binary, which is possible since modern browsers possess of a WebAssembly runtime.

This setup may provide answers to the requirements of **Portability**, **Zero-cost abstraction** and **Direct utilization**:

- **Direct utilization:** This setup leads to a low maintenance setup. Per supported GIS library, only one binding project may be required to serve all needs in all languages, including the proposed VPL. Not requiring any additional 'wrapper' or 'boilerplate' codebases in between the core GIS library and the application makes it so its inner functionality may be more directly accessed, and may lead to less features getting 'lost in translation'. This property has led this design to be called a 'zero-boilerplate' plugin system.
- **Portability:** By using WebAssembly, the exact same binary can be run on the frontend, and a potential scalable backend. Using the same binary in multiple locations leads to predictable behavior.
- **Zero-cost abstraction:** Additionally, because of the above two properties, a native, headless execution of a VPL pipeline can be envisioned, without reference to the VPL altogether. This is visualized by Figure 23. A pipeline could be compiled to any plain script, like JavaScript or python, which can then be executed on a backend, utilizing the functionalities from within these portable binaries. This could make the VPL a 'zero cost abstraction', as the abstraction imposed by using nodes and GUI components, will not lead to any difference in performance, compared to if a script was created using a 'normal' programming language. This headless runtime will not be part of the implementation, but is part of the design of this plugin system.

All in all, to allow for zero-cost abstraction in the future, and allow the zero-boilerplate concept currently, The plugin system of the proposed method will be designed as mentioned above.

4.6.2 Components

The components mentioned above fit together to propose the following workflow to use a native GIS library in the proposed Web VPL:

1. Write or find a GIS library written in either C++ or Rust.

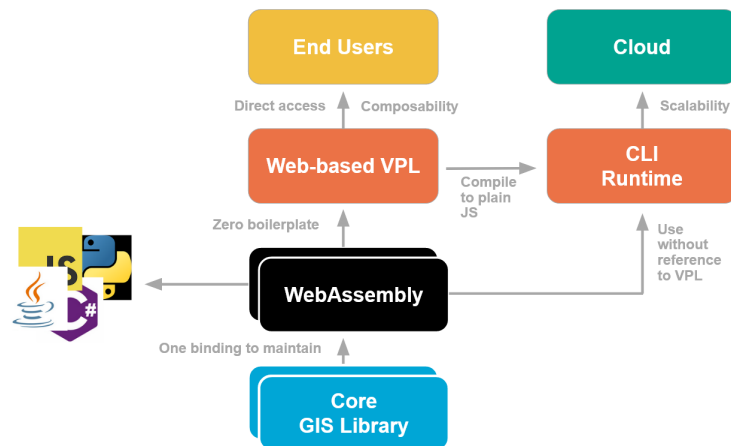


Figure 23: The methodology expanded beyond the scope of this study, to provide a method for backend, cloud-based execution

2. Create a second library, in which a subset of this library is flagged and wrapped as 'functions usable on the web'.

Optional: Include metadata to add additional functionality to the library

3. Compile this library with a compatible compiler.
4. Publish the results of these compilers to a [CDN](#).
5. Within the VPL: Reference the CDN address to the plugin loader.
6. The plugin loader now loads and converts the exposed functions, and includes them in the list of VPL components, ready to be used in the VPL.

For the scope of this system, we will refer to a native GIS library compiled to WebAssembly as a 'plugin', even though these projects are in essence normal JavaScript libraries, and can be used in regular JavaScript projects.

What follows is an elaboration on the side of the plugin model:

4.6.3 Plugin Model

The plugin model serves three purposes:

Compilation

One, it needs to form a bridge between the language in which the GIS library is written, and WebAssembly. In other words, the requirements of the WebAssembly compiler compatible with the language in question must be adhered to.

```

1
2  class SquareCalculator {
3
4     my_number = 0;
5
6     set_input(some_number: number): void {
7         this.my_number = some_number;
8     }
9
10    do_process(): void {
11        this.my_number = this.my_number * this.my_number;
12    }
13
14    get_output(): number {
15        return this.my_number;
16    }
17 }
18

```

Figure 24: An API which can only be used in an imperative, non-functional manner

The aforementioned WebAssembly interface types proposal is not ready to use for C or C++ based GIS libraries. However wrapping WebAssembly using JavaScript types is something C++ does support. This can also leverage the infrastructure of JavaScript: Existing JavaScript tooling, such as [CDNs](#), can be utilized for distribution, updates and version control. Therefore, the WebAssembly binaries will be accompanied by a Javascript wrapper file.

Both the C++ and Rust compilers require functions to be flagged explicitly for compilation. Additionally, for a library to be compilable, all dependent libraries must also be able to compile to `wasm`. For C++, the Emscripten compiler can be used to compile to WebAssembly [[Zakai, 2011](#)].

For Rust, the 'wasm-pack' and 'wasm-bindgen' toolkits enable `wasm` compilation [[wasm-bindgen Contributors, 2022](#); [wasm-pack Contributors, 2018](#)].

Wrapping

Secondly, in certain situations, some level of function wrapping may still be required. Some libraries are written in an imperative style, troubling its conversion to the functional model used by a dataflow VPL (See Figure 24). This is why the implementation of this plugin model must take cases like these into account. This comes down to wrapping the functionality of the GIS library as pure functions, by making copies of inputs / outputs, or grouping a series of imperative steps.

Flagging

Lastly, a plugin model will need to communicate the content of the plugin to the VPL. We distinguish between *required* data, and *optional* data. The central idea for

this aspect is to generate this information automatically from the wasm binary and related files. Only when that is impossible, should the information be manually added within the plugin.

The following information is *required* for the VPL to load a GIS library, and convert it into visual components:

- A list of all functions present in the library, uniquely named.
- A list of all custom types (structs / classes) present in the library, also uniquely named.
- Per function:
 - A list of all input parameters, name and type.
 - An output type.

The following information is *optional*, but it would improve the functionality and usability of the library:

- Per function:
 - A custom, human-readable name.
 - A description to explain usage.
- Per type:
 - A custom, human-readable name.
 - A description to explain usage.
 - Functions for serializing and deserializing this type (binary, json)
 - Functions for rendering this type in 2D or 3D.
 - A 'constructor' and 'deconstructor', to convert this type from and to basic types present within the VPL.

On the side of the plugin loader within the VPL, all the compiled, wrapped and flagged information within the plugin needs to be extracted. The automated extraction of all **required** information can be done by utilizing TypeScript Declaration 'd.ts' files. A 'd.ts' file can be understood as a 'header' file generated by the TypeScript compiler, exposing the types required by all functions found in a corresponding JavaScript file. By using the typescript compiler in the VPL, this header file could be loaded and interpreted to find all basic information, including the names, the namespace where to find a functions, and all input and output types. This extraction of types was required, since these are not present in JavaScript source code, and types are needed in explaining to the end-user how to use a function, and in making the VPL typesafe.

With the extracted information from the ".d.ts" file, a corresponding Javascript file can be traversed and loaded as a VPL plugin. javascript's nature as a scripting language can be utilized for this: Firstly, its dynamic nature allowed a library to be loaded and incorporated at runtime without any special alterations. Hot-loading libraries in C++ for example, can't usually be done without significantly altering the way a program runs. Secondly, JavaScript's prototype-based classes and its support for reflection allows a plugin loader to localize and collect all functions within a library. And lastly, the "first-class function support" allowed these functions to be referenced and called by the nodes of the VPL Graph.

Because the VPL will be implemented as a Dataflow-VPL, the loader seeks to extract only (pure) functions. However, many libraries also include classes, as these can make an API more clear to use using regular languages. The plugin loader will need to support classes by converting them to a series of normal functions. Static methods and constructors can be converted directly, and methods are converted into functions with the object as the first argument.

The **optional** data can be exposed by flagging functions with a standard prefix. These functions are then loaded by the VPL, but will not be converted into visual components. Instead, these functions are programmatically called when the VPL engine or the user requires this optional aspect.

4.7 TESTS

With both the VPL and the plugin system in place, the final step of the methodology is introduced to gather the results and data needed for properly answering the second, third fourth sub-research questions.

4.7.1 Compilation Tests

To test the ability of the VPL and the plugin system to host different libraries and different languages, four demo plugins are compiled and loaded within the VPL. The Rust language and C++ are tested. Per test, the workflow layed out in Section 4.6 is followed. Per language, a minimum plugin is first created, assessing if and how well a simple function, method, and class can be exposed to the VPL. After this demo, a more sizable, existing geocomputation library is compiled.

These tests are meant as a qualitative comparison between compiling a full-scale library written in Rust, to a full library written in C++. This way, the tooling and workflow can be compared for a realistic use-case. The study is conducted by compiling both libraries using their respective [wasm](#) toolsets, and noting the differences in workflow, supported features, and the resulting plugins.

The libraries must be compiled without 'disruption': They must be kept the exact same for normal, native usage.

C++ Library: CGAL

The library tested for C++ is CGAL, compiled using emscripten. For one, this library is well established and relevant to geoprocessing as a whole, as multiple other C++ geo-libraries depend on it. Moreover, it is a sizable and complex project, making it highly likely the problems described by Section 2.1.2 are encountered. We could choose more simple libraries, but this is not representative of most C++ geoprocessing libraries.

Rust Library: Startin

The second library tested is the Startin library, written in Rust, compiled using wasm-bindgen. This library is smaller in scope than CGAL. Ideally, a library with a size comparable to CGAL should have been chosen, to make for a balanced comparison. However, Rust is still a relatively unknown language in the field of GIS, making libraries like these difficult to find. Startin was chosen, for the triangulation functionalities it provides makes for a good comparison against CGAL's Triangulator. It also, just like CGAL, makes use of a high precision kernel, and offers geometric robustness.

4.7.2 Demo applications

Two demo applications are to be created for the purpose of testing the practical capabilities of the methodology. The first demo focusses on the intended use case of 'library publication'. We wish to see to what extent a library published using WebAssembly can directly be tested from within the proposed environment.

The second demo expands upon this first demo, and is intended to examine to what extent such a library can actually be composed as part of a 'real' application. In this test, the interaction between this library and the composable GUI will be elaborated upon.

4.7.3 Feature Comparison

In this third set of tests, the achieved functionality of the VPL is compared with the functionality of existing VPL methods. This comparison is made on the basis of features relevant to the goal of this study:

- Does the VPL allow for plugins: Third party 'nodes'?

- In which language must a third party plugin be written?
- Can third party nodes use customly defined types, such as a new class or struct?
- Can the VPL be run headless, that is to say, without the GUI?
- Is the VPL accessible as a static web application?
- Does the VPL contain GIS-specific nodes by default?
- Does the VPL contain GUI nodes, so that users can build custom interfaces?

4.7.4 Usage tests

This final set of tests are meant to analyze the usability of the visual language itself. It allows us to understand the advantages and disadvantages of the more granular language design decisions.

For the assessment criteria, the cognitive dimensions framework of [Green and Petre \[1996\]](#) will be used. The framework is useful for its focus on language features. Also, as commented on in [Section 2.2](#), the study has acquired a canonical nature among many VPL researchers for its elaborate examination of the "Psychology of Programming".

The framework presents the following 13 dimensions and accompanying descriptions [[Green and Petre, 1996](#)]:

1. Abstraction gradient: What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
2. Closeness of mapping: What 'programming games' need to be learned?
3. Consistency: When some of the language has been learnt, how much of the rest can be inferred?
4. Diffuseness: How many symbols or graphic entities are required to express a meaning?
5. Error- proneness: Does the design of the notation induce 'careless mistakes'?
6. Hard mental operations: Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?
7. Hidden dependencies: Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
8. Premature commitment: Do programmers have to make decisions before they have the information they need?
9. Progressive evaluation: Can a partially-complete program be executed to obtain feedback on 'How am I doing'?

10. Role- expressiveness: Can the reader see how each component of a program relates to the whole?
11. Secondary notation: Can programmers use layout, colour, other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
12. Viscosity: How much effort is required to perform a single change?
13. Visibility: Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

As stated by the authors; the purpose of this framework is to make the trade-offs chosen by a language's designer explicit. It is not meant as a 'scoring' system [Green and Petre, 1996].

5 | IMPLEMENTATION

This chapter presents the implementation of the methodology. It discusses the extent of the achieved functionality, as well as certain ways it fell short of the proposed design.

5.1 INTRODUCING: GEOFRONT

This section discusses the extent of the prototype [VPL](#) implementation. The prototype is titled "Geofront", as a concatenation of "geometry" and "frontend".

The Geofront Application is implemented according to the design layed out in [Section 4.5](#), and uses TypeScript as its main language. `webpack` is used to compile this codebase into a singular JavaScript file, and this file practically serves as the full application. the repository spend around 11.000 lines of code, divided into core categories and functionalities. What follows is a clarification of the implementation of some of these categories.

5.1.1 Model

The visual programming language model as described by [Section 4.5.3](#) and [Figure 21](#) could be fully implemented on the web. Both the shims as well as the model itself was implemented in TypeScript (TS). This model allows Geofront to internally represent the data structure and logic of a dataflow-VPL program.

Type safety was fully implemented by essentially creating a new 'layer' of hierarchical types on top of TS / JS, as proposed by the methodology.

The type system can be extended by types found in third party Plugins. In theory, this can be used to prevent all incorrect type usage during creation of a Geofront graph, and before calculation. In practice, to support iteration, some runtime type checking was still required. Additionally, the type system had to be disabled in situations were plugins did not produce interface types (such as projects compiled using `emscripten`).

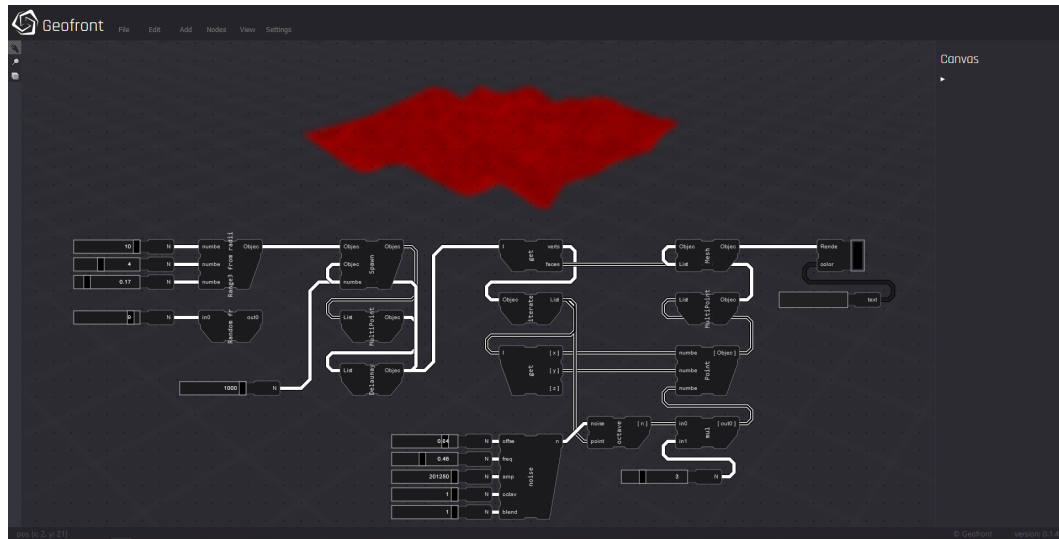


Figure 25: The Geofront Application

5.1.2 View

The graph

The graph data model must be rendered to the screen so users can comprehend and edit the graph. This visualization is achieved by using the HTML5 Canvas Api. The Canvas API is a raster-based drawing tool, offering an easy to use, high-level api to draw 2D shapes such as lines, squares, circles, and polygons. The Nodes Canvas uses this API to draw polylines and polygons at runtime, to represent the cables and nodes respectively. These basic shapes and their styles change dynamically, based on features like the length of a cable, how many input sockets a node requires, or whether or not a node is selected.

Like other HTML5 features, the main advantage is that this API is included and implemented within the browser itself. This method is fast thanks to its C++ based implementation, and can be used without the need to include anything within the source code of the application.

Additionally, the Canvas API provided granular control over how and when shapes should be rendered. This feature was instrumental in providing Geofront with a wide range of widgets, containing sliders, text fields, image viewers, file load, fetch, and save widgets, and even a component capable of opening up third-party applications as a pop-up (Figure 26c).

The downside of this implementation, is that all features the HTML renderer normally accounts for, like picking, conditional styling, and performant rendering of repetitious elements, are lost. These had to be re-implemented in typescript, which will never be as performant as the codebase of the browser engines themselves. An additional limitation is that the draw calls are primarily CPU based, making it less

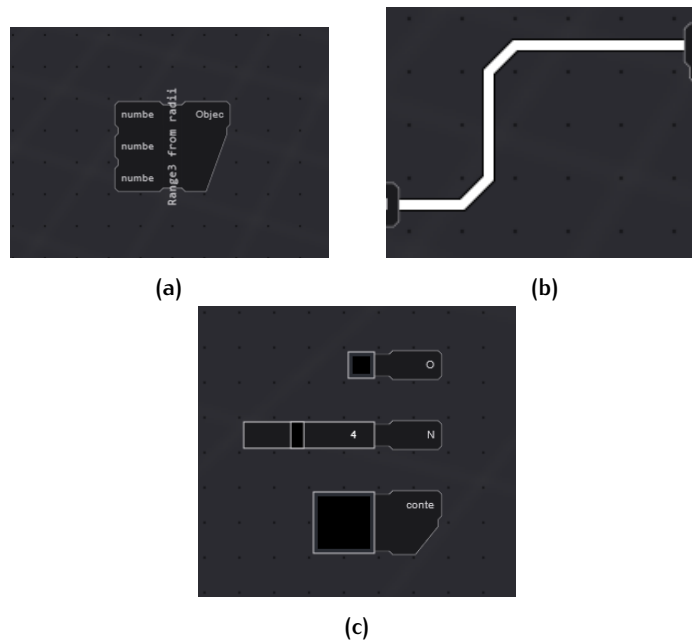


Figure 26: The basic canvas components of Geofront: a Node (A), a Cable (B), and multiple Widgets (C)

performant than a pure WebGL implementation would have been. lastly, the implementation chose to redraw the full canvas on every registered change to the graph, instead of partial redraws.

These limitations come together to a performance linear to the amount of nodes and cables drawn. For the current implementation and scale of Geofront, this performance is acceptable. Still, the application slows down when rendering a large amount of components.

This performance hit is partially due to the implementation, and partially due to browser feature limitations. The browser does not offer a 'middle ground' between html-like rendering and 2D canvas-like rendering required for a visualization like the dataflow graph. Still, this implementation could have experimented more with a HTML + SVG based render method.

Presentation

Special care has been put into the presentation of the implementation. The layout takes inspiration from various geometry VPLs mentioned at Section 3.2, such as Blender's GeometryNodes, McNeel's Grasshopper, and Ravi Peter's GeoFlow. A few notable exceptions, however. Firstly, the entire graph is placed on a grid, and all nodes adhere to this grid (see Figure 27). For example, a node with three inputs will always occupy three grid cells in height. This grid is applied for much of the same reason as terminals & source code are displayed using monospaced fonts. Consistent sizing encourages organization and clarity, for this makes it easy for

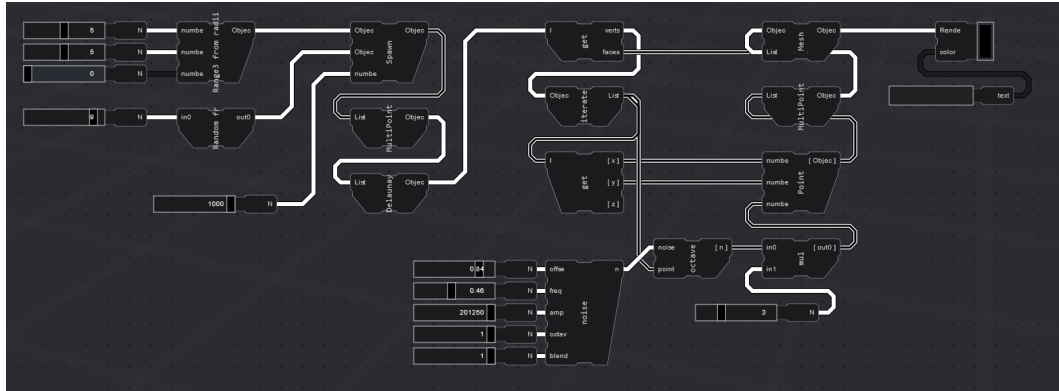


Figure 27: A complete Geofront script

components to line up, and predict how much space something requires. Cables also adhere to the grid. They alter their shape in such a way to remain as octagonal as possible, in an attempt to make connections between nodes more readable.

3D Viewer

The 3D viewer attached to the geofront application is also implemented in typescript. It uses WebGL and the OpenGL Shading Language (glsl) as its graphics API. The viewer can be used to represent and visualize various geometries, such as points, lines, meshes, bezier curves, and bezier surfaces. Images can also be rendered, which are represented as a quad mesh with a texture.

The useful aspect of WebGL is the fact it does not have to be included within the source code of a program. WebGL supports all render demands basic, small-scale 3D geodata visualization might need, such as point clouds and DTMs. large scale visualization is not possible, as the visualizer does not support idioms like frustum culling, or dynamic levels of detail. Additionally, the viewer does not support

These visualization options open the possibility of visualizing a great number of different geodata types, such as DTM / DSM, GEOtiff, Point clouds, and OGC vector data. However, specific visualization convertors are not implemented, for these are reliant upon the compilation of existing geocomputation libraries.

5.1.3 Controller

Interaction

User interaction is made possible through the HTML DOM Events. This API provides ways to listen to many events, including keyboard and mouse events. When the mouse is moved, its screen-space position is transformed to a grid position, which allows the user to select one or multiple objects.

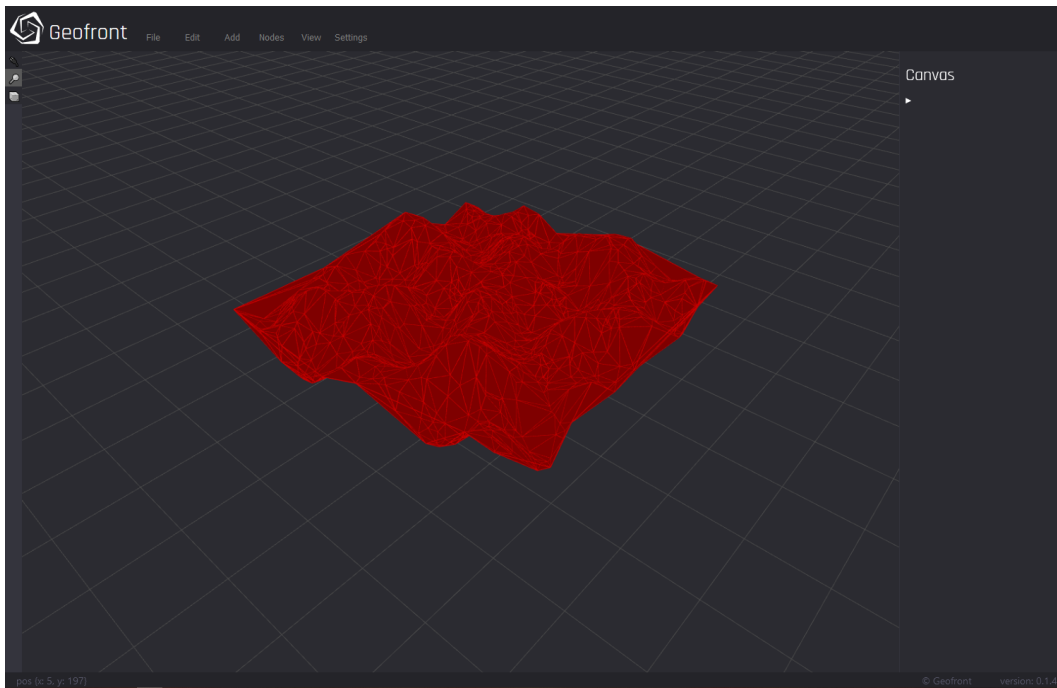
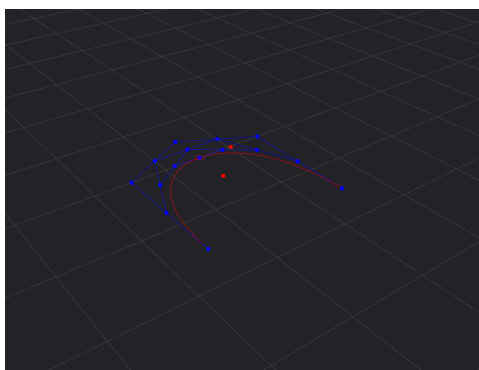
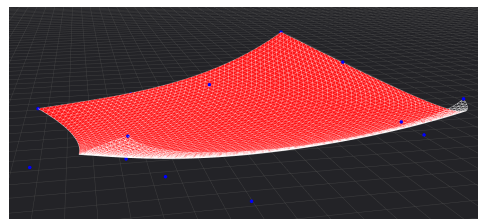


Figure 28: The Geofront Viewer



(a)



(b)

Figure 29: A bezier curve and surface visualized using the geofront viewer

Geofront's user interface aims to mimic features of regular desktop applications. As such, the Geofront Graph supports features like undoing, redoing, duplication, copying, and pasting. These functionalities can be used with the expected keyboard combinations (Ctrl + C / Ctrl + V). However, the implementation does lack touch & mobile support.

In general, these editing features are complete, but there are a few caveats caused by the browser environment. Namely, the browser has need of its own controls and shortcuts. For example, the right mouse button brings up the browser context, and the Ctrl + W shortcut closes a browser tab, which cannot be overwritten. While there are some workarounds, these aspects make it so Geofront, and web applications mimicking desktop applications in general, may behave unexpectedly.

Input and Output

The base dataflow VPI component of Geofront support input and output UI elements, like sliders, buttons, and text fields. These form special nodes on the canvas, called 'widgets'. Widgets simply are nodes with side effects. By making this a different type of node, the behavior of the graph becomes more predictable.

The fact that the Geofront implementation opted for a canvas API-based visualization made it so HTML could not be used for these aspects, and all these features had to be created within the constraints of the Canvas API.

Files can be used as inputs and extracted as outputs using the 'file load' and 'file save' widgets. These files are then loaded as raw text or raw binary, which can be parsed by using a parser appropriate for that file type. The problem with this implementation, is that it requires a full file to be loaded into memory. Most native parsers make use of streaming to avoid this. There are ways of supporting incrementally reading files in a browser, but these methods are not supported by all browsers yet.

5.2 THE PLUGIN SYSTEM

The plugin system is implemented according to design discussed in Section 4.6. The implementation comes down to a plugin loader written inside of the Geofront application, with an accompanied workflow of how to create such a plugin.

5.2.1 The plugin loader

The plugin loader implemented in Geofront can load a JavaScript / typescript library, and convert it into appropriate visual components.

As prescribed by the methodology, Typescript declaration files are loaded to determine the name, location, parameter types of functions. This works similar for libraries which include a WebAssembly binary.

While in theory any JavaScript / typescript library can be used, in practice some limitations are in place due to the specific implementation used:

Files

Firstly, the current loader accepts only one Javascript file, and one Typescript Declaration file per library. A library without a 'd.ts' declaration cannot be used. If additional files are used, such as `wasm` files, these will have to be explicitly fetched and run by the JavaScript file. For the purposes of this study this is acceptable, as the used `wasm` compilers work in a manner compatible to these limitations. Javascript bundlers also help to adhere to these limitations. Still, this does mean that not just any JavaScript library can be imported.

Library Structure

Secondly, while the loader does support namespaces and classes, not all types of libraries and programming styles are supported. Functions using callbacks, promises, complex types, or generics, cannot be properly loaded. Libraries utilizing "method chaining APIs" can be loaded, but are difficult to use as intended on the Geofront Canvas.

Types

The plugin loader can only load functions using acceptable input and output types. Not all input and output arguments translate well to the format of a dataflow VPL. The types may only include:

- basic JavaScript types (boolean / number / string)
- basic jsons (unnamed structs), objects, interfaces
- JavaScript ArrayBuffers like `Float32Array` (vital for performant data transfer)

The typesystem of the plugin loader will pick up on types exposed by a library, and include them within the type safety system of Geofront. However, this does mean that certain types are not supported, like asynchronous promises, or functions as variables.

Supported languages

Finally, not all languages are equally supported:

- **Javascript / Typescript:** If the Javascript and Typescript files used adhere to the limitations mentioned above, the library can be used. However, a bundler needs to be used to include all sub-dependencies of a library, as the Geofront loader does not load sub-dependencies currently.
- **Rust:** Libraries compiled to webassembly using the "wasm-bindgen" work "out of the box" in most cases. `wasm-bindgen` is able to generate JavaScript wrapper bindings for a `wasm` library, accompanied by TypeScript type definitions. This wrapper handles type conversions.

However, rust libraries compiled to the web do require a initialization step. As such, the loader now checks if the library looks like a Rust library, and if it does, it uses a slightly altered loading method.

- **C++** At the time of writing this study, the 'embind' compiler (explained in Section 5.2) does not have an option to compile a typescript declaration file. Additionally, the JavaScript generated to wrap the `wasm` binary is not a wrapper handling type conversions and memory safety like Rust. Instead, it uses a custom architecture programmatically expose JavaScript wrapper functions one by one, and leaves it to the user of the library to deal with type conversions and memory safety. These two aspects combined makes it so C++ cannot use Geofront's loader directly, and must use an additional in-between wrapper libraries.
- **Other languages** This study only experimented with Rust and C++ as non-js languages. While the loader's ability to work with WebAssembly is promising, additional testing is required before Geofront can claim to support any language.

5.2.2 Achieved Workflow

With all the above considerations in mind, the following workflow can now be used to create a Geofront Plugin, which, as explained, doubles as a normal JavaScript library. If Rust or C++ is used, this setup in a way triples as also a native geoprocessing library. The Geofront standard library is also implemented by using workflow with Rust.

-
- ```

1
2 Using Typescript:
3 1. Write or find a geoprocessing / analysis library
4 using typescript,
5 2. Compile and bundle to a 'd.ts' + '.js' file.
6 3. publish to npm

```

```

7
8 Using Rust:
9 1. Write or find a geoprocessing / analysis library using rust
10 2. Create a second library, which exposes a subset of this
11 library as 'functions usable on the web', using 'wasm-pack'
12 and 'wasm-bindgen'.
13 3. Compile to '.wasm' + 'd.ts' + '.js'.
14 4. publish to npm ('wasm-pack publish')
15
16 Using C++:
17 1. Write or find a C++ based geoprocessing / analysis library.
18 2. Create a second library, which exposes a subset of this
19 library as 'functions usable on the web', using 'emscripten'
20 and 'embind'.
21 3. Compile to a '.wasm' and '.js' file using emscripten.
22 4. Manually edit the 'js' file to wrap initialization and type
 conversions.
23 5. Manually create a corresponding 'd.ts' file, calling newly created
 methods in the 'js' file.
24 6. Publish these to npm
25
26 -----
27
28 In Geofront:
29 A. Reference the CDN (content delivery network) address of this node
 package.
30 B. Use the library.

```

---

### 5.2.3 Automation and portability

The Section 4.6 mentioned the requirements of library portability and automation. In this section we assess to what extent this implementation was able to succeed on delivering these two requirements.

#### *Automation & exposure of metadata*

Based on the results, we can state that the loader mitigates the need for explicit configuration only for the required, mandatory aspects. All optional properties like human-readable names and descriptions, must be specified explicitly using a naming convention specific to Geofront.

In practice, however, there are some more "configuration" requirements. The limitations outlined by Section 5.2.1 show that there are quite a few additional consid-

erations. Geofront does not support all types or all library structures, and certain languages require additional compile limitations.

Also, while the optional properties are just that, optional, one could argue that some of these properties are in fact required. Libraries without 'human-readable' names and descriptions are harder to utilize in a Geofront script by end users. While regular programming languages also allow the creation and publication of undocumented libraries, one can question if this should also be allowed for the more end-user focussed VPL libraries.

So, while the plugin loader can load some simple textual programming libraries almost without any special configuration, sizable libraries intended for consumption by Geofront will still need to be explicitly configured for Geofront. However, even with these requirements, this can still be considered an improvement compared to the plugin systems of geometry VPLS studied at Section 3.2, in which developers are required to create a class per exposed function.

### *Portability*

This system creates seamless interoperability between a textual programming library, and a VPL plugin to an extent. However, because of the reasons outlined above, it is also safe to say that this seamless interoperability is only one-directional: Libraries intended for consumption by Geofront double as also a 'normal' JavaScript library. The configuration demands of Geofront only impair the normal, JavaScript-based usage by forcing a functional style, and by including certain methods only intended for Geofront. Even these Geofront-specific methods might prove useful in certain scenarios, such as by providing a way to visualize data.

This seamless interoperability is less prominent in the opposite direction. A normal JavaScript library, or a JavaScript library using WebAssembly, can't be automatically used by Geofront in most cases. Most libraries use an imperative programming style, use unsupported types, or consist of multiple files. In some cases, a library is able to be loaded, but its functionality is impaired by the interface.

## 5.3 CLI, HEADLESS RUNTIME

While the zero-cost abstraction runtime, proposed in Section 4.6, was said not to be part of the implementation, small scale experimentation has been done as a proof of concept.

An early build of Geofront had the ability to compile a Geofront script to a subset of regular JavaScript (see Figure 30). All libraries were converted to normal import statements, all nodes were replaced by function calls, and the cables substituted by a variable token. Additionally, because of the no-boilerplate plugin system, this JavaScript file could be re-imported back into Geofront, to represent the exact graph

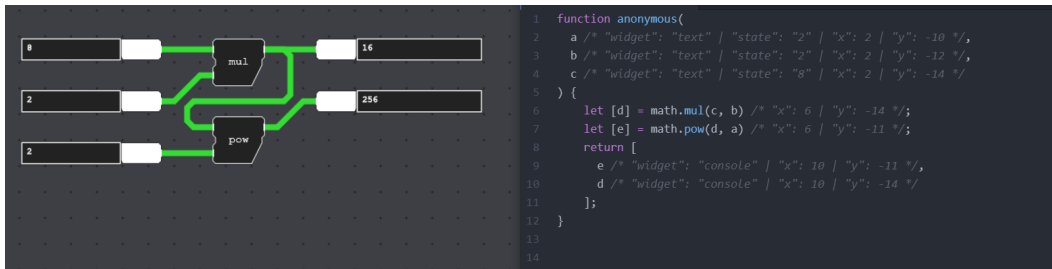


Figure 30: An early build of geofront, showcasing JavaScript interoperability

it originated from. A subset of JavaScript was supported, using inline comments to add the appropriate metadata. This allowed the setup shown above to be synchronized. That is to say, updating the textual representation led to a change in the graph, and vice-versa. This JavaScript representation could be executed headless in either a browser-based application, or using a local JavaScript runtime like Deno [Deno Contributors, 2022].

However, both the Pipeline to JavaScript compiler, and the JavaScript to Pipeline compiler, could not be maintained given the other requirements of the environment, such as complex GUI nodes. Auxiliary systems complicated the export and import workflow, like allowing a node to iterate through a list. Still, the brief success of the system does proof the proposed method is feasible, given enough time.





# 6 | TESTING

In this chapter the various software implementations and design choices made in Chapter 5 are used and tested on various aspects. This is done to gather the data needed to answer the research questions adequately. It consists of the sections Section 6.1, Section 6.2, Section 6.3, and Section 6.4.

## 6.1 PLUGIN COMPILATION & UTILIZATION

In this section, the tests outlined in Section 4.7.1 are performed.

### 6.1.1 Rust: minimal plugin

The compilation of the minimum Rust plugin, which exposes a function, a class, and a method, occurred as expected. Figure 31 Shows the source code of this plugin. The 'wasm-bindgen' library allows functions and classes to be annotated as 'expose to JavaScript' with a macro statement. This simplifies the compilation process to WebAssembly greatly. Due to Rust's macro system, it also creates compile time errors with IDE integration, if a property or class are incompatible to be exposed. This aids the construction of a wasm-compatible API.

This library was compiled to WebAssembly and JavaScript using 'wasm-pack'. This produces multiple artifacts, showcased in Figure 33. This figure also showcases how wasm-pack wraps the functionality: The `point_distance` function exposed by the wasm file is wrapped by the JavaScript file, converting it to look like a regular JavaScript class. Keep in mind that as the Interface Type proposal gets accepted as 'standard webassembly' [Wagner, 2022], The JavaScript or typescript artifacts can be omitted, as this logic will be added to the .wasm binary itself. The wasm-pack library is already able to produce a WebAssembly build with interface types included. However, as this is not standard WebAssembly yet, it hinders integration with WebAssembly produced from the emscripten compiler, so this study chose against using it.

To check if the result is valid, a small html demo was created to load and use the library (see Figure 32). Note how the JavaScript library wrapping the wasm file looks and works almost like a normal JavaScript library, the only difference being a 'init' function, which is required to be run before using the library, and the need to free the memory of used object with the `free()` method.

```

1 use wasmbindgen::prelude::*;
2
3 #[wasm_bindgen]
4 pub fn add(left: f32, right: f32) -> f32 {
5 left + right
6 }
7
8 #[wasm_bindgen]
9 pub struct Point {
10 pub x: f32,
11 pub y: f32,
12 }
13
14 #[wasm_bindgen]
15 impl Point {
16
17 pub fn new(x: f32, y: f32) -> Point {
18 Point { x, y }
19 }
20
21 pub fn distance(&self, other: &Point) -> f32 {
22 ((self.x - other.x).powi(2) + (self.y - other.y).powi(2)).powf(0.5)
23 }
24 }

```

Figure 31: Rust: Minimum WebAssembly example

To load this project into Geofront, a reference to the path of the compilation artifacts must be specified within the Geofront GUI, shown in Figure 34. A local path was used for convenience, instead of publishing this demo to npm, and accessing it via a CDN.

Figure 35a shows how all functions in this demo are loaded correctly, and Figure 35b shows that the functions indeed work as expected to create two Graphs. Note how the parameter names and Types are also loaded, indicated by the names visualized at the input and output of the nodes. This is thanks to the 'd.ts' file of Figure 33.

### 6.1.2 Rust: startin plugin

Loading the startin library also occurred as expected. startin already offered a wasm-ready library, this could directly be loaded into Geofront. However, the API exposed by this library used a non-functional style, making it hard to properly use the library on a VPL canvas. This is why a custom plugin library still had to be created, in which functions like 'new\_from\_vec' were added to support functional usage.

```

index.html X
example > index.html {} "index.html" > html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <title>Demo</title>
5 </head>
6 <body>
7 <script type="module">
8 import init, {add, Point} from "../pkg/rust_min_gf.js";
9
10 await init();
11
12 console.log(add(2.0, 40.0)) // result: 42.0
13
14 let p1 = Point.new(-1.0, -1.0);
15 let p2 = Point.new(2.0, 3.0);
16
17 console.log(p1.distance(p2)); // result: 5.0
18
19 p1.free()
20 p2.free()
21 </script>
22 </body>
23 </html>

```

```

42 index.html:12:17
5 index.html:17:17
>>

```

Figure 32: Web demo using the WebAssembly build

```

rust_min_gf.d.ts X
rust-min-gf > pkg > rust_min_gf.d.ts > ...
16 @returns (Point)
17 /
18 static new(x: number, y: number): Point;
19 /**
20 * @param (Point) other
21 * @returns (number)
22 */
23 distance(other: Point): number;
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41

```

```

rust_min_gf.js X
rust-min-gf > pkg > rust_min_gf.js > ...
91 /**
92 * @param (Point) other
93 * @returns (number)
94 */
95 distance(other) {
96 _assertClass(other, Point);
97 const ret = wasm.point_distance(this.ptr, other.ptr);
98 return ret;
99 }
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115

```

```

rust_min_gf_bg.wasm X
50441 local.get $i3
50442 i32.store
50443 local.get $i2
50444 i32.const 48
50445 i32.add
50446 global.set $fp0
50447 (func $point_distance (type $t14) (param $p0 i32)
50448 (param $p1 i32) (result f32)
50449 (local $i2 i32) (local $i3 f32) (local $i4 f32)
50450 (local $i5 f32) (local $i6 f32)
50451 block $B0
50452 block $B1
50453 local.get $p0
50454 i32.eqz
50455 br_if $B1
50456 local.get $p0
50457 i32.load
50458 local.tee $i2
50459 i32.const -1
50460 i32.eq
50461 br_if $B0
50462 local.get $p0
50463 local.get $i2
50464 i32.store
50465 local.get $p1

```

Figure 33: wasm-pack Compilation artifacts: A Typescript Declaration file, a JavaScript file, and a WebAssembly binary, visualized as WebAssembly Text (WAT).

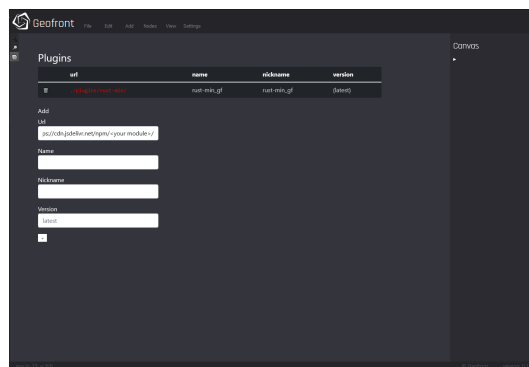


Figure 34: Loading a plugin into Geofront using the GUI



Figure 35: Usage On the canvas

Other than that, all steps performed by the minimal Rust plugin could be made with this project as well, as shown in Figure 36. This also showcases the usage of optional, additional bindings: A series of functions are given definition, which together flag the Triangulation type as 'Renderable'. This way, a variable of type 'Triangulation' can be viewed in 3D by clicking on it.

### 6.1.3 C++: Minimal plugin

Unexpected shortcomings were encountered in making a minimal C++ WebAssembly build in Geofront. It was possible to compile the plugin file to WebAssembly, and to use it in a web demo, but this build was limited in its ability to be automatically interfaced.

First of all, the construction of the C++ source code itself. Emscripten's `embind` tool uses a macro syntax to flag functions marked for JavaScript compilation, shown in Figure 39.

While it does work similar to Rust's `wasm-bindgen`, `cpp` macro's do not allow for any pre-compile-time code checking, and can produce hard to decipher error messages.

Secondly, to compile this file to the right binary with accompanied JavaScript wrapper, it had to be compiled using the `-sMODULARIZE=1` and `-sEXPORT_ES6=1` flags enabled. Otherwise, the JavaScript produced uses an import syntax too deviant from regular import statements to have any chance to be interfaced by Geofront. It must be noted that `emscripten` was behind on documentation compared to 'wasm-pack' and the rust-wasm organization [[wasm-pack Contributors, 2018](#)]. it does not offer many examples, or thorough explanations on what many of the compiler flags do or mean [[Zakai, 2011](#)]. `wasm-pack` on the other hand offers complete tutorials, a range of starter projects, and elaborate documentation of most of its functionalities.

Thirdly, the compilation with the right flags enabled resulted in a valid 'wasm' and 'js' file, but not a 'd.ts' file. `Emscripten` does not support typescript declaration files, and third-party tooling to add this is only conceptual at this point in time. This

```

9 #[wasm_bindgen]
10 pub struct Triangulation {
11 dt: startin::Triangulation,
12 }
13
14 #[wasm_bindgen]
15 impl Triangulation {
16
17 pub fn new_from_vec(pts: Vec<f64>) -> Triangulation {
18 let mut tri = Triangulation::new();
19 tri.insert(pts);
20 tri
21 }
22
23 pub fn new() -> Triangulation {
24 let dt = startin::Triangulation::new();
25 Triangulation { dt }
26 }
27
28 pub fn insert(&mut self, pts: Vec<f64>) {
29 const STRIDE: usize = 3;
30 for i in (0..pts.len()).step_by(STRIDE) {
31 self.insert_one_pt(pts[i], pts[i+1], pts[i+2]);
32 }
33 }
34
35 // ...
36
37 // impl Renderable for Triangulation
38 #[wasm_bindgen]
39 impl Triangulation {
40
41 pub fn gf_has_trait_renderable() -> bool {
42 true
43 }
44
45 pub fn gf_get_shader_type() -> GeoShaderType {
46 GeoShaderType::Mesh
47 }
48
49 pub fn gf_get_buffers(&self) -> JsValue {
50 let buffer = MeshBuffer {
51 verts: self.all_vertices(),
52 cells: self.all_triangles(),
53 };
54 sende_wasm_bindgen::to_value(&buffer).unwrap()
55 }
56 }
57 }

```

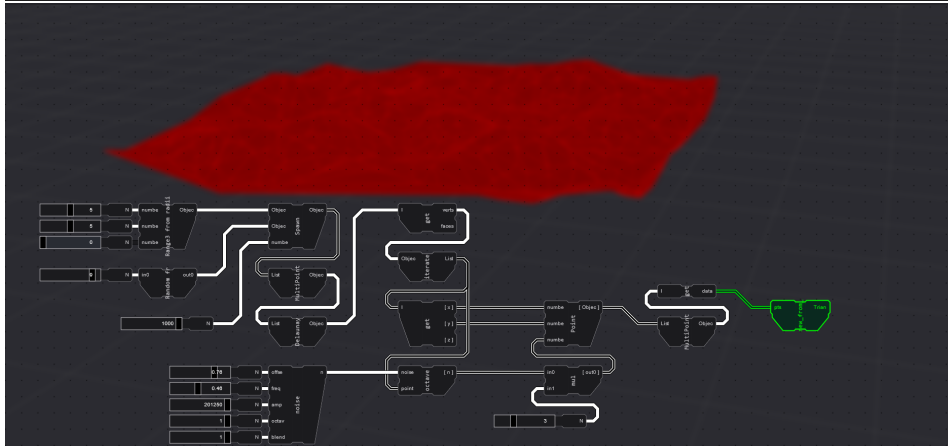


Figure 36: Startin, loaded as plugin within Geofront

```
1 #include <emscripten/bind.h>
2 #include <cmath>
3
4 using namespace emscripten;
5
6 float add(float left, float right) {
7 return left + right;
8 }
9
10 class Point {
11 public:
12 double x;
13 double y;
14
15 Point(double x, double y) :
16 x(x),
17 y(y) {}
18
19 double distance(Point other) {
20 return std::pow(
21 std::pow(x - other.x, 2) + std::pow(y - other.y, 2),
22 0.5);
23 }
24 };
25
26 EMSCRIPTEN_BINDINGS(cpp_min) {
27 function("add", &add);
28 class_<Point>("Point")
29 .constructor<double, double>()
30 .function("distance", &Point::distance)
31 .property("x", &Point::x)
32 .property("y", &Point::y);
33 }
```

Figure 37: C++: Minimum WebAssembly example

study was also unable to find any developments indicating development towards Interface Types. All of this does allow for a working web demo similar to the Rust equivalent, shown in Figure 38. However, the absence of interface types, or the substitute typescript interface types, made automated interfacing with the binary difficult.

This study did eventually achieve a workaround by using JavaScript reflection. By creating a blacklist of all 134 default functions the emscripten JavaScript wrapper comes with with the aforementioned flags enabled, we can distill the imported module down to the 2 symbols exposed by embind in this case, the add function and Point class. However, by doing this, all function types and the names of all parameters are lost, and can not be loaded into Geofront, which in turn does not allow the resulting Geofront graph to be type safe. Another solution would have been to manually declare type definitions as strings in the CPP file or as a typescript file, but was deemed a too manual of a solution to a problem which should be able to be solved programmatically, as 'wasm-pack' did.

with a custom system in place to load embind files, the plugin loader could attempt to load both the js and wasm file. It is here were an obstacle was encountered, which could not be solved within the time frame of this study. The JavaScript wrapper file dynamically fetches the WebAssembly file. This is allowed when importing a JavaScript library using ES6 module syntax. However, this is incompatible with the implementation choices of the Geofront plugin loader. To load a plugin dynamically, it fetches and interprets the source files at runtime. However, for security reasons, dynamically fetching a wasm file from within these runtime interpretations is not allowed. The wasm-pack solution does not have the same problem, for it allows the WebAssembly file to be parsed within its initialization function. A change within the emscripten JavaScript wrapper would allow this obstacle to be overcome, but this must be left to subsequent research.

A final workaround was made to still load the plugin within Geofront (Figure 41). However, this was done by forgoing the concept of a plugin, and by integrating the WebAssembly binary within the source code of Geofront itself. As this thesis is a study on direct accessibility of any GIS library, and not on creating the most feature-rich GIS web VPL imaginable, this solution was discarded. However, it does show that if one forgoes all design constraints, it is fully possible to run and use C++ libraries in Geofront.

#### 6.1.4 C++: CGAL plugin

If the minimum cpp plugin example could not be loaded into Geofront, it will be unsurprising that the entirety of CGAL could also not be compiled into a suitable format ready for VPL consumption. Several steps towards this goal were made however.

```

<!DOCTYPE html>
<html lang="en">
<head>
 <title>cpp test</title>
</head>
<body>
 <script type="module">
 import init from "./cpp_min.js";

 let mod = await init();

 console.log(mod.add(40, 2)); // 42

 let p1 = new mod.Point(-1, -1);
 let p2 = new mod.Point(2, 3);

 console.log(p1.distance(p2)); // 5

 p1.delete()
 p2.delete()
 </script>
</body>
</html>

```

Figure 38: Cpp-wasm web demo

```

@libcpp M X
plugins > cpp-min-g > src > @libcpp
1 // @idl_wasmjs.cpp
2 #include <emscripten/bind.h>
3 #include <cmath>
4
5 using namespace emscripten;
6
7 float add(float left, float right) {
8 return left + right;
9 }
10
11 class Point {
12 public:
13 double x;
14 double y;
15
16 Point(double x, double y) :
17 x(x),
18 y(y) {}
19
20 double distance(Point& other) {
21 return std::pow(
22 std::pow(x - other.x, 2) + std::pow(y - other.y, 2),
23 0.5);
24 }
25 };
26
27 EMSCRIPTEN_BINDINGS(cpp_min) {
28 function("add", add);
29 class_<Point>("Point")
30 .constructor(<double, double>())
31 .function("distance", &Point::distance)
32 .property("x", &Point::x)
33 .property("y", &Point::y);
34 }

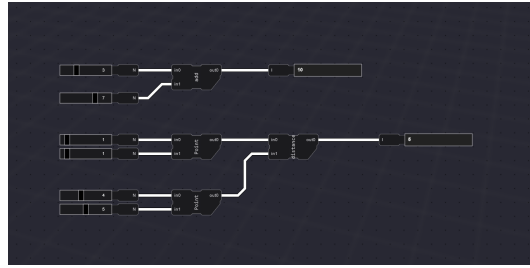
```

Figure 39: Cpp Plugin Source file

|                          |                               |
|--------------------------|-------------------------------|
| ready                    | typeDependencies              |
| preloadedImages          | char_0                        |
| preloadedAudios          | char_9                        |
| callRuntimeCallbacks     | makeLegalFunctionName         |
| withStackSave            | createNamedFunction           |
| demangle                 | extendError                   |
| demangleAll              | BindingError                  |
| wasmTableMirror          | throwBindingError             |
| getWasmTableEntry        | InternalError                 |
| handleException          | throwInternalError            |
| jsStackTrace             | whenDependentTypesAreResolved |
| setWasmTableEntry        | registerType                  |
| stackTrace               | _embind_register_bool         |
| __embind_register_bigint | ClassHandle_isAliasOf         |
| getShiftFromSize         | shallowCopyInternalPointer    |
| embind_init_charCodes    | throwInstanceAlreadyDeleted   |
| embind_charCodes         | finalizationRegistry          |
| readLatin1String         | detachFinalizer               |
| awaitingDependencies     | runDestructor                 |
| registeredTypes          | releaseClassHandle            |
| typeDependencies         | downcastPointer               |
| char_0                   | registeredPointers            |
| char_9                   | getInheritedInstanceCount     |

Figure 40: Emscripten JavaScript wrapper blacklist





**Figure 41:** The result of the workaround: including the C++ binary within the standard library of Geofront

WebAssembly compilation was partially successful. A web demo was able to utilize the CGAL kernel for basic operations, as can be seen in Figure 42. Additionally, A subsequent web demo can utilize the CGAL TIN to a limited extent (Figure 43).

However, this demo too could not be fully completed. The major obstacle preventing full usage was making sure all dependencies, like Boost, are compiled together with CGAL. Legacy makefile build systems complicate this process. To get the current demo's working, several dependencies and sub-dependencies had to be manually traversed, their makefiles had to be edited, and the projects had to be re-compiled and copied to different location, to be used by the emscripten compiler exclusively. This is an unsustainable workflow, which will complicate development.

Secondly, building a proper interface to a CGAL functionality, such as a triangulation, was difficult due to combination of CGAL's API together with the restrictions imposed by emscripten, and the functional interface of Geofront. As an example, a robust manner of providing CGAL with an array of points was not found. A single point could be provided, but instantiating a number of of points from within JavaScript turned out to be expensive, so much so that the method had to be deemed unusable. Additionally, this is difficult to interface from within Geofront, as this would mean a chain of nodes would have to be created, one for each single point (see Figure 45).

The GDAL.js project was analyzed in an attempt to find a possible solution to handling the great number of dependencies [Dohler, 2016], and interfacing. Regrettably, this project too has resorted to incorporating the source code of GDAL and all its dependencies, and has altered them heavily. Interfacing was solved by heavy use of emscripten on the side of JavaScript, to workaround the limitations regarding parsing vectors and arrays. To illustrate, a fragment was taken from the `inspect_geotiff` example, shown in Figure 44. Actions like these cannot be directly interfaced from Geofronts visual language.

All these factors together made it so a CGAL plugin could not be presented. This is not to say that a WebAssembly build of CGAL is not possible, but that many additional steps need to be taken to allow an API which is usable in practice. To work around all limitations, a more extensive time frame could have allowed this study to:

```

1 // msvc_compiler.cpp
2 #include <string>
3 #include <vector>
4
5 #include <math.h>
6
7 #include <CGAL/Carthesian.h>
8 #include <CGAL/squared_distance_2.h>
9
10 #include <emscripten/bind.h>
11
12 typedef CGAL::Cartesian<double> K;
13 typedef K::Point_2 Point_2;
14 typedef K::Vector_2 Vector_2;
15
16 // check precision
17 bool something(double x, double y) {
18 Point_2 p(x, y), q;
19 Vector_2 v = p - CGAL::ORIGIN;
20 q = CGAL::ORIGIN + v;
21
22 return (p == q);
23 }
24
25 // use kernel
26 double distance(double x1, double y1, double x2, double y2) {
27 Point_2 p(x1, y1), q(x2, y2);
28 double dis_squared = CGAL::squared_distance(p, q);
29 return std::pow(dis_squared, 0.5);
30 }
31
32 typedef double N;
33
34 N lerp(N a, N b, N t) {
35 return (1.0 - t) * a + t * b;
36 }
37
38 std::vector<double> multilerp(N a, N b, std::vector<double> vector) {
39 std::vector<double> result(vector.size());
40
41 for (int i = 0; i < vector.size(); i++)

```

Figure 42: CGAL Kernel Demo

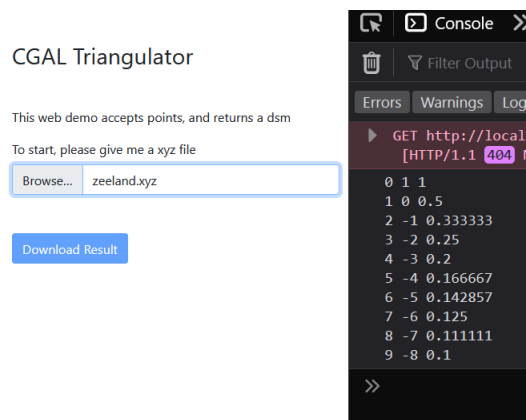


Figure 43: CGAL Triangulation Demo

1. Translate CGAL and all dependencies to a WebAssembly-acceptable format
2. Build a wrapper library on the side of C++, offering a functional interface while remaining compatible with C++, Geofront, and Emscripten's way of interfacing with WebAssembly.
3. Build a custom importer on the side of Geofront, to provide certain wrappers around emscripten compiled projects.

However, much of this would be simplified if emscripten adopts the features found in `wasm_pack` and `wasm_bindgen`, such as interface types, and support for vectors and arrays without pre-allocation.

```
1 // [...]
2
3 // This is where things get a bit hairy; the C function follows a common C
4 // pattern where an array to
5 // store the results is allocated and passed into the function, which
6 // populates the array with the
7 // results. Emscripten supports passing arrays to functions, but it always
8 // creates a *copy* of the
9 // array, which means that the original JS array remains unchanged, which
10 // isn't what we want in this
11 // case. So first, we have to malloc an array inside the Emscripten heap
12 // with the correct size. In this
13 // case that is 6 because the GDAL affine transform array has six elements.
14
15 var byteOffset = Module._malloc(6 * Float64Array.BYTES_PER_ELEMENT);
16
17 // byteOffset is now a pointer to the start of the double array in
18 // Emscripten
19 // heap space
20 // GDALGetGeoTransform dumps 6 values into the passed double array.
21
22 GDALGetGeoTransform(dataset, byteOffset);
23
24 // Module.HEAPF64 provides a view into the Emscripten heap, as an array of
25 // doubles. Therefore, our byte offset
26 // from _malloc needs to be converted into a double offset, so we divide it
27 // by the number of bytes per double,
28 // and then get a subarray of those six elements off the Emscripten heap.
29
30 var geoTransform = Module.HEAPF64.subarray(
31 byteOffset/Float64Array.BYTES_PER_ELEMENT,
32 byteOffset/Float64Array.BYTES_PER_ELEMENT + 6
33);
34 // [...]
35
```

Figure 44: A fragment from the `inspect_geotiff` example found in `GDAL.js`. Source: [Dohler \[2016\]](#)

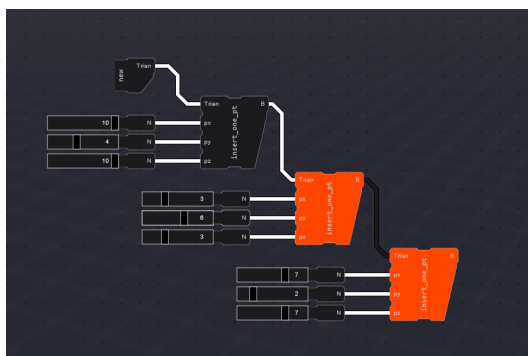


Figure 45: If only one point can be provided at a time, this would become the API in Geofront for adding three points: A long chain of additions

### 6.1.5 Comparison

A couple of additional comparisons between the four WebAssembly projects mentioned above were made, both in terms of the file size of the compiler artifacts, as well as the performance of specifically the interoperability between WebAssembly and JavaScript. These are the results:

#### *Size*

Figure 46 shows how the minimum C++ plugin compiles to a binary of 68 kb. Compared to the Rust build of 20 kb, This is around 3.4 times more. The CGAL TIN test compiled to 258 kb bytes, and compared to startin's 114 kb, is around 2.3 times more heavy.

The wasm-pack artifacts include the typescript header files, and both its typescript and JavaScript file were not optimized for size. Emscripten does not produce a typescript file, and compressed its JavaScript file.

All of these facts combined make it safe to say that the emscripten wrappers are significantly heavier.

This study speculates this difference could be because emscripten's primary use-case is compiling complete applications. This requires a more heavy wrapper, offering features like file servers. When compiling sizable C++ applications, the overhead of this wrapper can be marginalized. However, apparently, emscripten is not able to distinguish between full applications, and small, granular use-cases like this one, and must include the 'full emscripten runtime' in all cases.

#### *Performance*

To test the performance of `wasm` - JavaScript interaction, the minimum C++ and Rust plugins were again used. This makes for a good comparison, since both

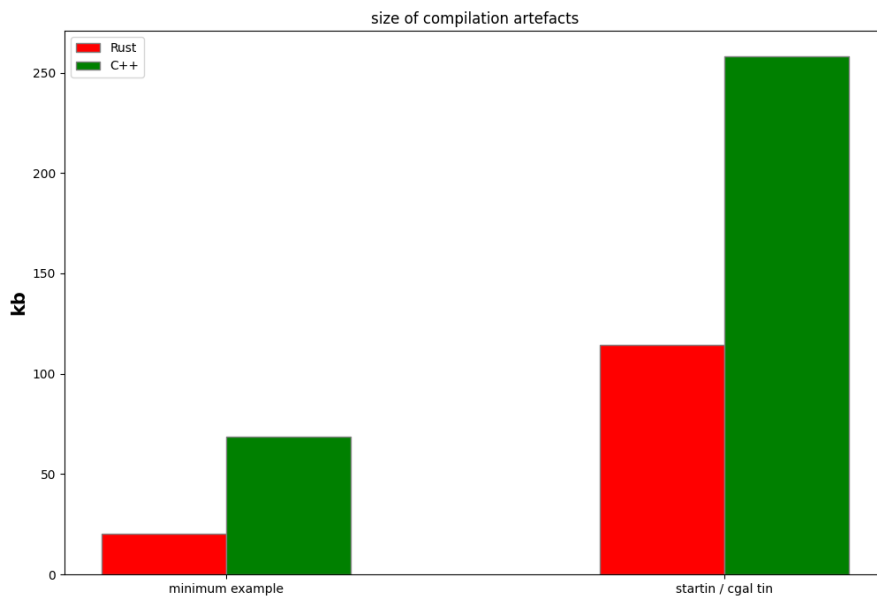


Figure 46: Combined size of all compiler artifacts produced by either wasm-pack or emscripten

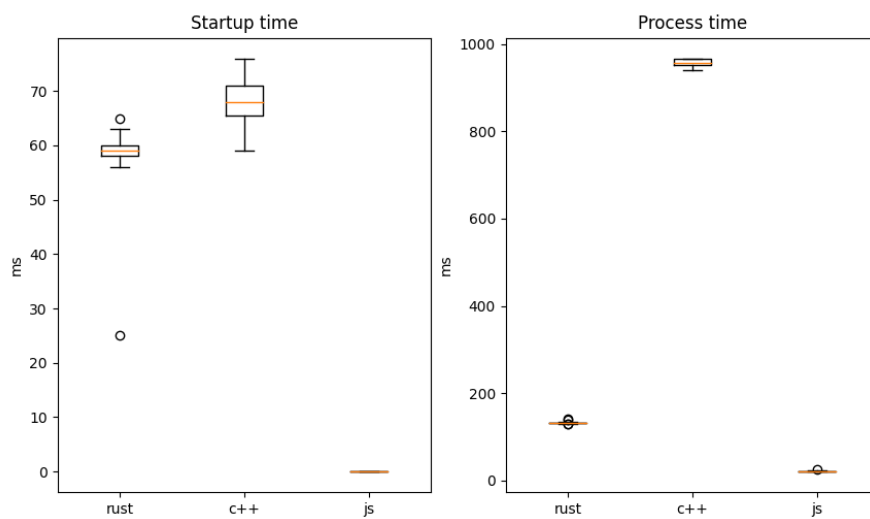


Figure 47: A benchmark using three comparable demo's handling points, given 100.000 iterations

(a)

```

1 <doctype html>
2 <html lang="en">
3 <head>
4 <title>rust test</title>
5 </head>
6 <body>
7 <script type="module">
8
9 import {add, Point} from "...(pkg/rust_bin.js)";
10
11 console.time("init");
12 const init();
13 console.timeLog("init");
14
15 let acc_distance = 0;
16 let count = 100;
17 console.time(count + " iterations");
18 for (let i = 0; i < count; i++) {
19 let p1 = Point.new(1.0, -acc_distance);
20 let p2 = Point.new(4.0, -acc_distance);
21 acc_distance = add(acc_distance, p1.distance(p2));
22 p1.free();
23 p2.free();
24 }
25 console.timeLog(count + " iterations");
26
27 </script>
28 </body>
29 </html>

```

(b)

```

1 <doctype html>
2 <html lang="en">
3 <head>
4 <title>cpp test</title>
5 </head>
6 <body>
7 <script type="module">
8
9 import {add} from "...(pkg/cpp.js)";
10
11 console.time("init");
12 let mod = await init();
13 console.timeLog("init");
14
15 let acc_distance = 0;
16 let count = 100;
17 console.time(count + " iterations");
18 for (let i = 0; i < count; i++) {
19 let p1 = new mod.Point(1, -acc_distance);
20 let p2 = new mod.Point(4, -acc_distance);
21 acc_distance = mod.add(acc_distance, p1.distance(p2));
22 p1.release();
23 p2.release();
24 }
25 console.timeLog(count + " iterations");
26
27 </script>
28 </body>
29 </html>

```

(c)

```

1 <doctype html>
2 <html lang="en">
3 <head>
4 <title>js test</title>
5 </head>
6 <body>
7 <script type="module">
8
9 function add(a, b) {
10 return a + b;
11 }
12
13 class Point {
14 x;
15 y;
16 constructor(x, y) {
17 this.x = x;
18 this.y = y;
19 }
20 distance(other) {
21 Math.sqrt(Math.pow(this.x - other.x, 2) + Math.pow(this.y - other.y, 2), 0.5);
22 }
23 }
24
25 let acc_distance = 0;
26 let count = 100;
27 console.time(count + " iterations");
28 for (let i = 0; i < count; i++) {
29 let p1 = Point.new(1.0, -acc_distance);
30 let p2 = Point.new(4.0, -acc_distance);
31 acc_distance = add(acc_distance, p1.distance(p2));
32 }
33 console.timeLog(count + " iterations");
34
35 </script>
36 </body>
37 </html>

```

Figure 48: Rust vs C++ vs js performance benchmarks

projects deliver the exact same functionality from the point of view of JavaScript. A pure JavaScript version was added however, to add a base comparison. The web pages used for these tests are presented in Figure 48.

The results are can be seen in in Figure 47. These results show the emscripten wrapper is not as performant as the wasmpack solution. From experimentation, performance hit could be narrowed down to the initialization step of the 'Point' classes.

These results may suggest two phenomena: One, just like the artifact size comparison, this difference could be because emscripten is written from the point of view of a C++ application, and use case does not require custom JavaScript - C++ interoperability. A full application seldom needs access to specific JavaScript processes the way this demo does. such an application only needs to address JavaScript through emscripten own interface, which could be more performant.

And two, the C++ builds may be suffering from 'legacy burden', as described by Ammann et al. [2022]. The emscripten solution needs to take more edge cases into account, has more complicated dependencies, and software compiled using emscripten must be more heterogenous than a younger language like Rust. This all could lead to a significant performance hit.

## 6.2 DEMO APPLICATIONS

This section demonstrates the extent to which Geofront is able to perform the main role it set out to fulfill: Accessing functions from low level [GIS](#) libraries from within a web VPL, and subjecting these functions to interactive elements, so that these functions may be used in different ways by end users.

Two related demo applications are presented, both featuring the `startin` library covered in [Section 6.1](#). The following two sections demonstrate the achieved functionality, and implications of said functionality. The last section presents the limitations encountered during the writing and usage of these demo's.

### 6.2.1 Demo One: Perlin noise & `startin`

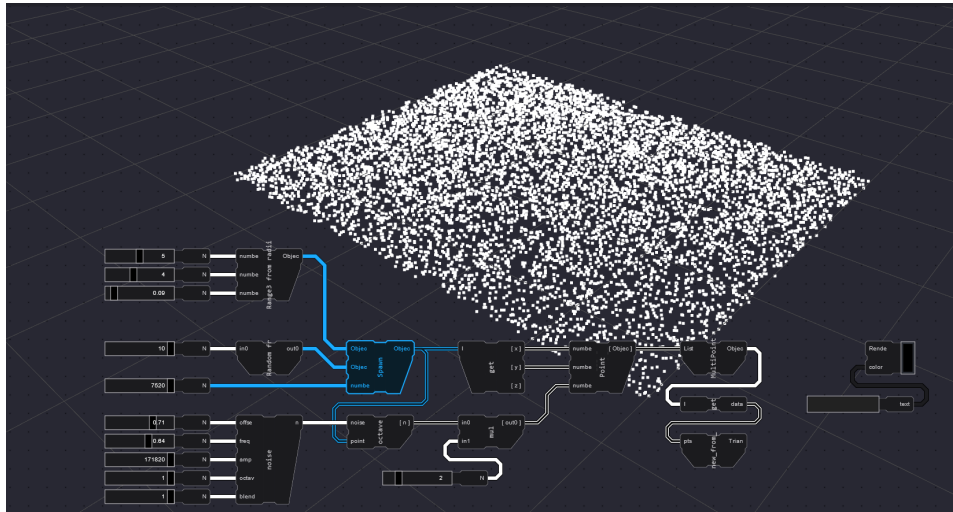
The first demo application demonstrate Geofronts core ability to connect a native library to an arbitrary UI. In this hypothetical scenario, we wish to discover how the `startin` triangulation reacts to various terrains by subjecting it to different point samples: Smooth samples, noisy samples, homogenous or heterogenous point distribution.

In reality, the `startin` triangulation serves as a normal 2.5D delaunay triangulation, so we do not expect to see any strange behavior, but one can imagine other terrain extraction and generalization algorithms where a test like this would grant valuable insights. An example of such an algorithm would be a plane fitting algorithm using RANSAC. or least squares adjustment of a polynomial surface.

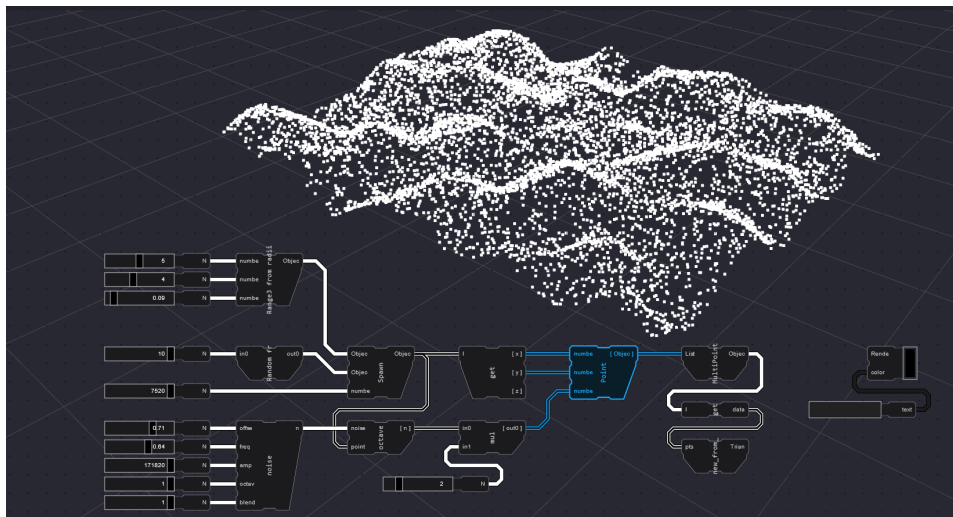
A test setup is made to generate various terrains using Perlin Noise [[Perlin, 2002](#)], so that different landscapes could be simulated. This setup not provide a way to generate different sample densities, but it will serve for the purposes of this demo.

The resulting geometry pipeline can be seen in [Figure 49](#). A UI is created by using multiple slider widget as inputs numbers. These numbers are used to create a bounding box area, which is subsequently filled with a random distribution of points ([Figure 49a](#)). The input sliders are also used to construct a noise field, which is sampled using these points. These noise values are used as the heights of the points ([Figure 49b](#)). Lastly, these points are used as input for the triangular irregular network (TIN) ([Figure 49c](#)).

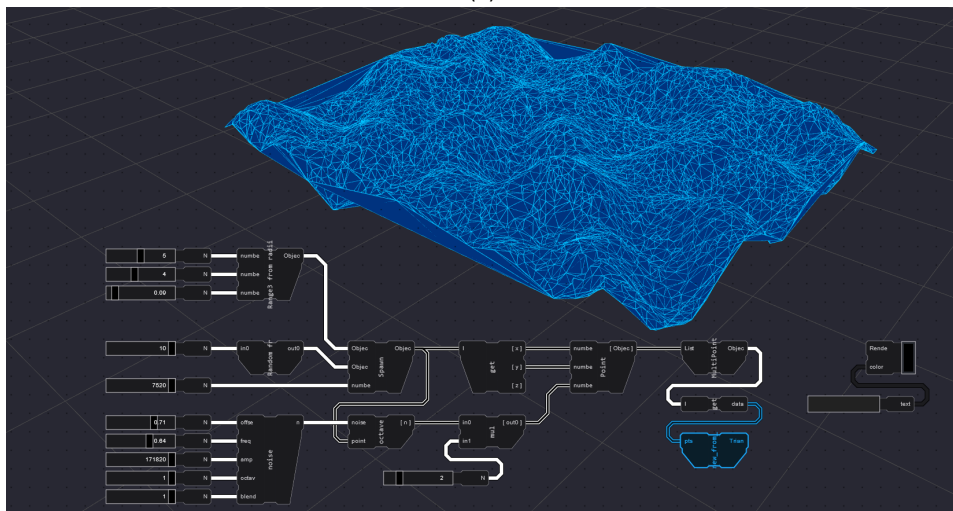
Multiple observations can be made from this demo: Firstly, note how the various visualizations of these in-between products could be inspected by selecting the corresponding operation ([Figure 49](#)). Secondly, observe how by connecting `startin` to this arbitrary GUI, the behavior of the library can now be explored with different parameters ([Figure 50](#)). Thirdly, recall that this sequence can be altered in real time,



(a)



(b)



(c)

Figure 49: Visual inspection of in-between products



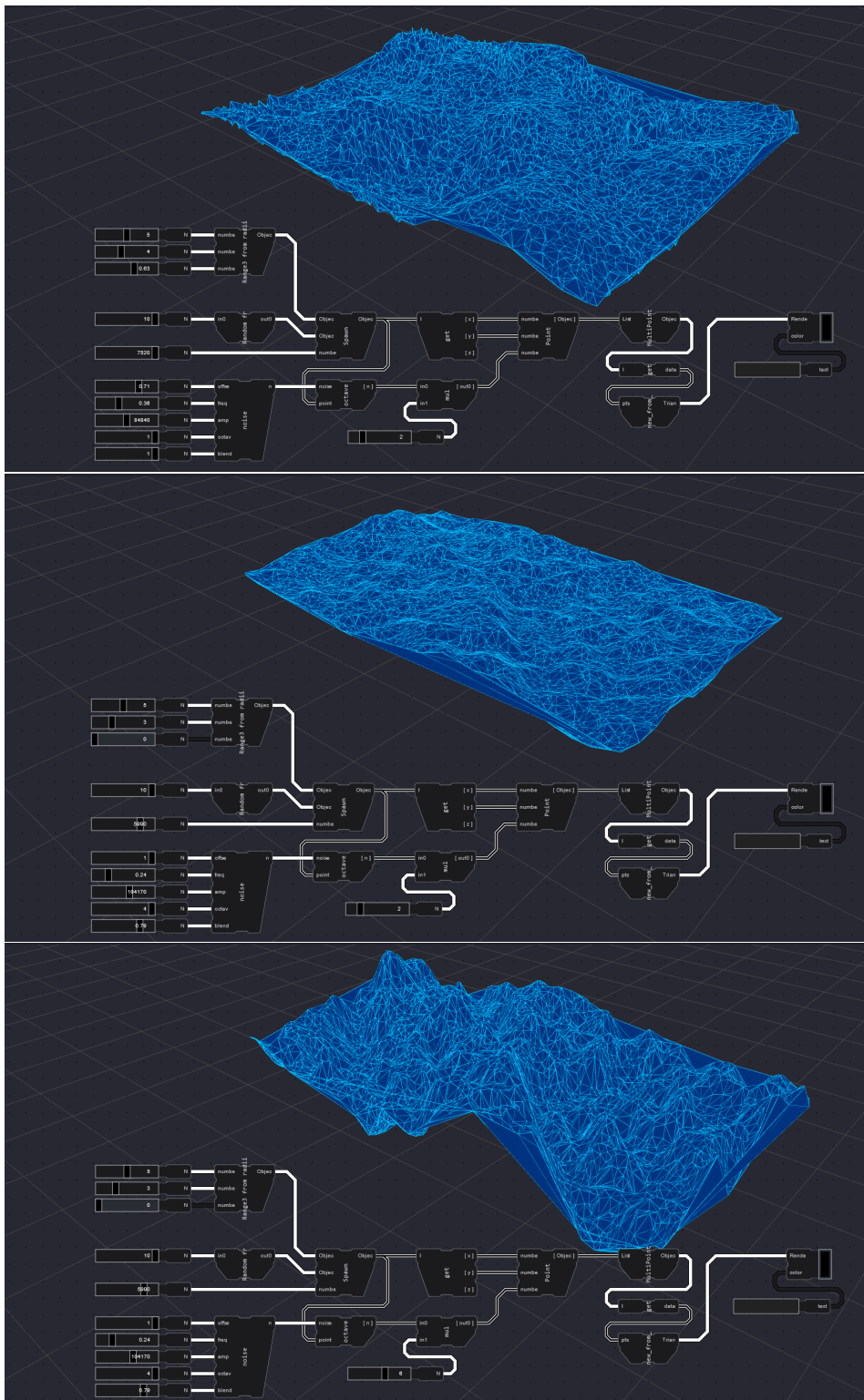


Figure 50: Exploration of the effects of different parameters

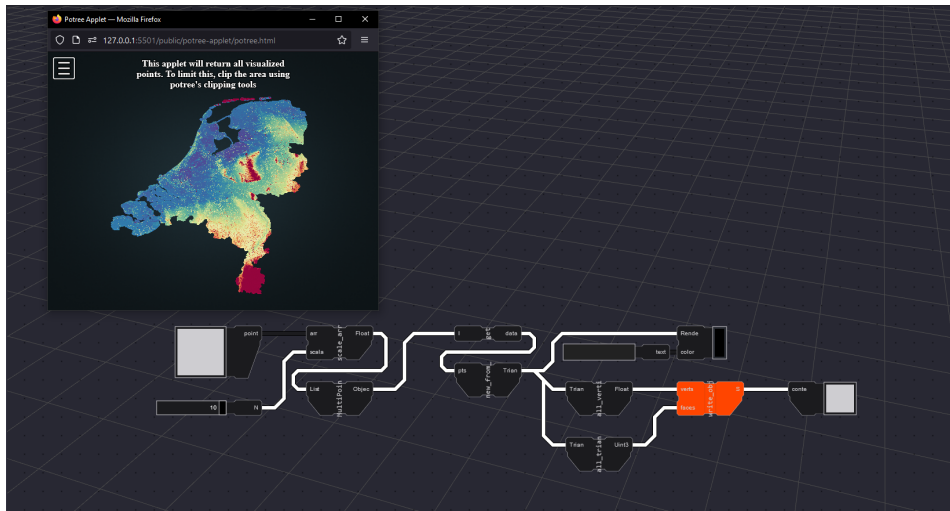


Figure 51: DSM and DTM extraction using Potree within Geofront

without any software alterations. These functionalities in conjunction are all indicators that Geofront can indeed be used to explore the capabilities and quality of native GIS libraries.

### 6.2.2 Demo Two: DTM & DSM extraction

Geofront was meant not only to be used to examine and visualize native libraries, but also to directly utilize them to fulfill real applications. This second application demonstrates the extent to which this is possible.

The hypothetical scenario used for this demo, is a situation in which a small scale Digital Terrain Model (DTM) is required from a point cloud. One could imagine a Digital Surface Model (DSM) or DTM of a construction site, required for creating an accurate render of a buildings surroundings.

the above startin demo is used as a starting point. This time, however, a TIN will be generated from a real point cloud, instead of a simulated one, and will be converted to a .obj file, so that the DTM / DSM might be used in a different process.

The resulting pipeline can be seen in Figure 51. Central to this pipeline is the 'Potree applet', found at the start of the pipeline. This is a special type of UI widget, which can be used to open a new browser window, running a second application. This application was created as part of this study, and uses the Potree viewer [Schütz, 2012]. It can be used independently, but by opening it from within Geofront, a connection between the two applications can be established. It allows Geofront to request a pointcloud, by pointing to a publicly hosted, potree-converted dataset. The same connection allows Geofront to request parts of this pointcloud, so that a



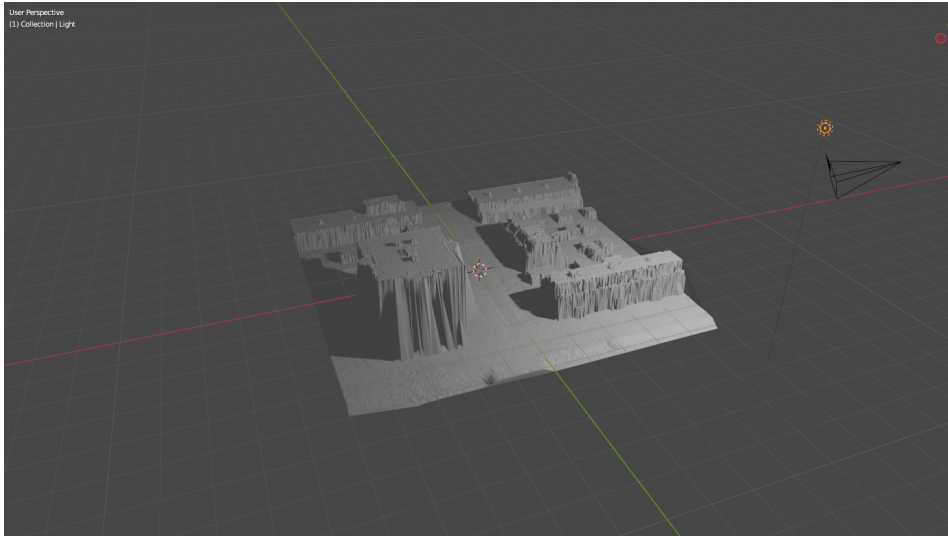


Figure 53: Resulting obj mesh, extracted from the Potree demo

sub section can be used in the Geofront pipeline. To query a specific subset, the Potree viewer itself can be used to:

- Filter the density of the pointcloud to a desired LOD.
- Filter the pointcloud to a certain bounding box or polygon.
- Filter the pointcloud based on classifications, to create either:
  - a DSM using all samples (Figure 52a),
  - a DTM with only terrain (Figure 52b), or
  - a DSM with only buildings (Figure 52c).

The pointcloud subset is then loaded into the Geofront pipeline, after it can be used as input for any loaded plugin. In this case, the points are inserted to the start in triangulator, after which the resulting mesh is converted to an .obj string, and made available for download using a 'save file' widget, so it could be used in a render (Figure 53).

Multiple observations can be made based on this demo. First, notice again how all steps mentioned correspond to the nodes of the pipeline. None of these steps are final. A different application could be created at runtime, by for example, adding a laz writer component to save the resulting subset as a point cloud itself, or by using a component to perform a solar potential analysis. This application strongly suggests that Geofront can indeed be used to directly utilize native libraries as part of a "real" use-case. Additionally, it demonstrates that Geofront allows not only composability between libraries, but also between libraries and full-scale, existing applications.

### 6.2.3 Limitations

Based on the above demo's, the following limitations in utilization were encountered:

- **Limited STD:** The standard operations and widgets present in geofront are limited. examples of these are scaling a vector, filtering lists, or getting the highest point from a set. Additionally, standard GIS functionalities, such as reprojections, are also not possible. Currently, plugin libraries add some of these basic functions as a workaround.
- **Types not interoperable:** While the type system of Geofront is extensive and supports complex aspects such as generics, no clear design is present for inter-operating between internal Geofront types, such as a Geofront Mesh, and a foreign type, such as startins triangulation. The method used in the above demo's involves breaking down a type into a basic JavaScript data formats, and using that as an interoperable model. However, this relies on unsustainable parsers which use hardcoded strings, reflection, and "blind thrust".
- **Limited visualization:** Currently, the application is only able to visualize a small number of data types, like points, lines, and meshes. Geofront's source code will have to be altered to support a library using images, for example. Another visualization shortcoming is that meshes currently cannot support over 65.535 vertices.
- **Limited performance:** The geofront runtime is not offloaded to a different thread, due to a difficulty of integrating web workers with the logic of the VPL. This means that heavy calculations performed on the canvas will freeze up the GUI of the application.

## 6.3 FEATURE COMPARISON

In Table 1, Geofront is compared against the most similar VPLs mentioned in Section 3.2. This comparison was done using the features described in Section 4.7.3.

Based on this comparison, the implementation of the proposed method appears to indeed provide a unique set of features, not found in comparable visual languages. It offers relatively extensive plugin support compared to other VPLs, it is web based, and offers a range of GUI nodes. Moreover, it is unique in its ability to accept third party Plugins written in different types of languages, and in the fact that it allows custom types within those plugins.

Geofront's drawback is that it offers no GIS nodes by default, and relies on third party plugins to provide these aspects. It can also not yet be regarded as scalable. Despite certain design decisions which might aid scalability, no native runtime exist yet which can run a Geofront script in a headless, GUI-less manner.



|                  | <i>Grasshopper</i> | <i>Blender</i> | <i>Mobius</i> | <i>Geoflow</i> | <i>Geofront</i> |
|------------------|--------------------|----------------|---------------|----------------|-----------------|
| Plugin support   | <b>Yes</b>         | No*            | No            | <b>Yes</b>     | <b>Yes</b>      |
| Plugin language  | C#                 | -              | -             | C++            | Rust/Js/Ts**    |
| Plugin types     | Partially          | No             | No            | Unknown        | <b>Yes</b>      |
| Headless runtime | No                 | No             | No            | <b>Yes</b>     | No              |
| Web based        | No                 | No             | <b>Yes</b>    | No             | <b>Yes</b>      |
| Base GIS Nodes   | No                 | No             | <b>Yes</b>    | <b>Yes</b>     | No              |
| GUI nodes        | <b>Yes</b>         | <b>Yes</b>     | No            | No             | <b>Yes</b>      |

\* Not without altering the source code of Blender itself

\*\* theoretically, any language can be used. Practically, only Rust, JavaScript and TypeScript result in valid plugins.

**Table 1:** Feature comparison of relevant VPLs

### 6.3.1 Plugin comparison

An additional comparison was made between the VPLs which accept third party plugins. Specifically, the different ways in which a new node has to be registered in Geoflow, Grasshopper, and Geofront are regarded next to each other.

#### *Grasshopper*

Grasshopper offers an object-oriented approach of registering new nodes: (Figure 54). A derived class has to be created and configured to represent an addition operation in this case. While this does lead to a verbose setup, this also makes it so metadata can be included alongside the 'function', as can be seen in the constructor: A name, nickname, and description can be provided.

#### *Geoflow*

Geoflow offers a similar object-oriented interface of registering new nodes (Figure 55). It is less verbose than the grasshopper version, but it does use hardcoded strings to access the inputs and outputs.

#### *Geofront*

As shown before, Geofronts 'no boilerplate' approach directly utilizes type and function declarations. This leads to the following minimum plugin: (Figure 56). Not only does this approach lead to no boilerplate code and classes, It also makes it so no separate binding projects are needed in principle: Functions can be annotated directly where they are declared in the core library. In practice, however, it still appears to be practical to create a separate library with annotated functions. Functions annotated with `wasm_bindgen` are constraint in certain ways, which can complicate a core library. Also, Geofront does not offer a way to add aspects like descriptions.

```

1 namespace MyPlugin
2 {
3 public class AdderNode : GH_Component
4 {
5 public ComponentNodeFromString()
6 : base("Add Integers",
7 "Add",
8 "This component adds two integer values",
9 "My Plugin",
10 "My Plugin Category")
11 {
12 }
13
14 protected override void RegisterInputParams(GH_Component.GH_
InputParamManager pManager)
15 {
16 pManager.AddIntegerParameter("a", "value A", GH_ParamAccess.item);
17
18 pManager.AddIntegerParameter("b", "value B", GH_ParamAccess.item);
19
20 }
21
22 protected override void RegisterOutputParams(GH_Component.GH_
OutputParamManager pManager)
23 {
24 pManager.AddIntegerParameter("R", "result", GH_ParamAccess.item);
25 }
26
27 protected override void SolveInstance(IGH_DataAccess DA)
28 {
29 int a;
30 int b;
31 DA.GetData(0, ref a);
32 DA.GetData(1, ref b);
33 int c = a + b;
34 DA.SetData(0, c);
35 }
36
37 public override Guid ComponentGuid
38 {
39 get { return new Guid("197d2ec4-c3b1-47ed-8355-6af3b7612f01"); }
40 }
41 }
42 }

```

Figure 54: Grasshopper plugin

```
1 class AddNode : public Node
2 {
3 public:
4 using Node::Node;
5
6 void init()
7 {
8 add_input("a", typeid(int));
9 add_input("b", typeid(int));
10 add_output("result", typeid(int));
11 }
12
13 std::string info()
14 {
15 std::string s;
16 if (output("result").has_data())
17 s = std::to_string(output("result").get<int>());
18 return s;
19 }
20
21 void process()
22 {
23 auto in1 = input("a").get<int>();
24 auto in2 = input("b").get<int>();
25 std::this_thread::sleep_for(std::chrono::microseconds(200));
26 output("result").set(int(in1 + in2));
27 }
28 };
```

Figure 55: Geoflow plugin



---

```

1 #[wasm_bindgen]
2 fn add(a: i32, b: i32) -> i32 {
3 a + b
4 }

```

---

Figure 56: Geofront plugin

Still, it is safe to assume that this method of directly using WebAssembly as plugins makes for a more simplified process compared to the aforementioned methods.

## 6.4 UTILIZATION ASSESSMENT

This final set of tests offers an analysis on specifically the usability aspects of Geofront, according to the framework described by [Green and Petre \[1996\]](#).

***Abstraction gradient: What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?***

The design of Geofront deemed the need for re-using parts of a script as components / functions as more important than the benefits of having no abstraction hierarchy (what you see is what you get). This is why a method for encapsulation was developed, by taking a part of a Geofront script, and compiling it to a JavaScript subset. This could then be loaded by the plugin loader. However, the addition of special types of nodes, and features like iteration, invalidated the geofront -> js translator. The translation is still possible, but not implemented. As such, if a user desires re-usable components and a lower abstraction level, they will need to write a plugin.

***Closeness of mapping: What 'programming games' need to be learned?***

Mapping a problem to a geofront script is intuitive for the most part. Think of the operations needed to solve a problem, find the right libraries and nodes representing these operations, and connect these nodes according to type. However, this mapping of problem and solution is hindered by the fact that Geofront needed to support iteration, shown in [Figure 57](#).

Based on the studies and experiences with existing VPLs (Section 2.2), these types of 'iteration games' are known to be a significant hinder to the closeness of mapping principle.

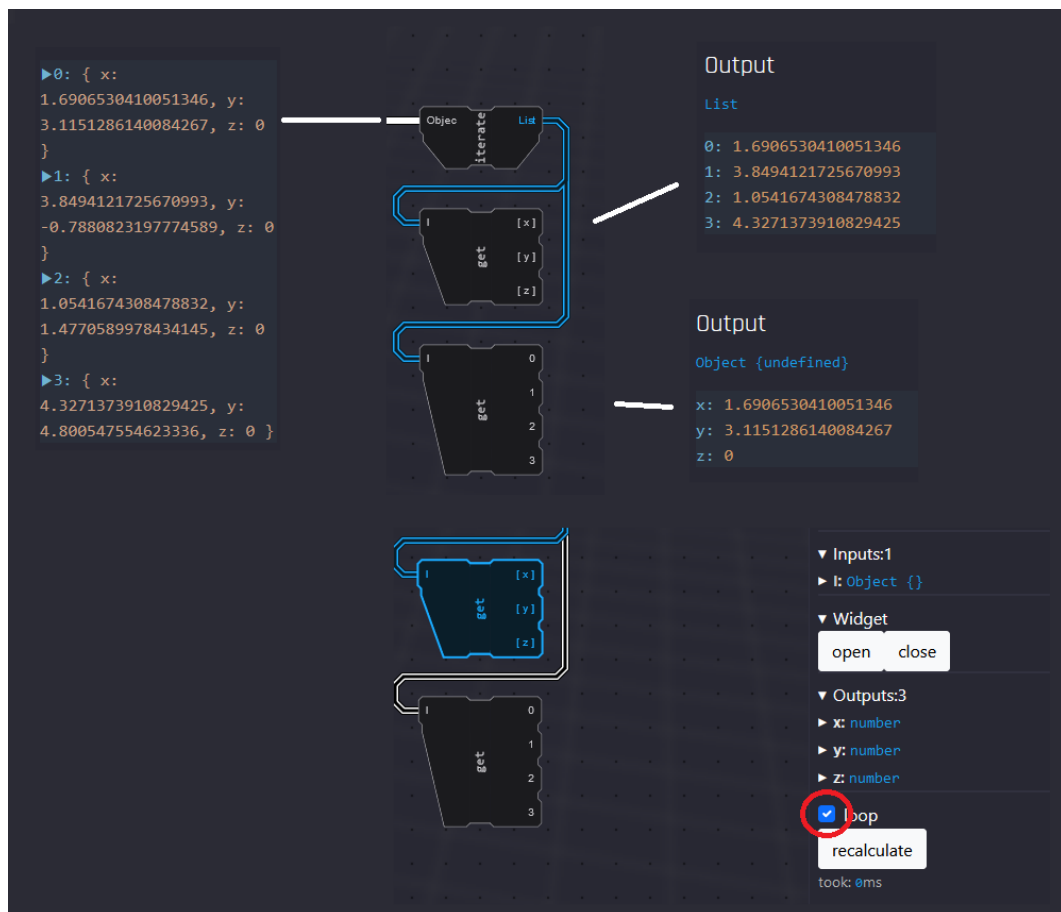


Figure 57: An example of a 'programming game' within Geofront. The 'loop' toggle indicates if an component should operate on a full list, or operate on all individual items within the list. This can be used to iterate over a 'list of objects' column-wise, or row-wise. This is considered a 'game', since this is not an obvious interaction, and must be learned before properly using Geofront.

***Consistency: When some of the language has been learnt, how much of the rest can be inferred?***

[Green and Petre, 1996] notes on the difficulty of defining 'consistency' in language design, and chose to define it as a form of 'predictability'. Geofront has introduced symbolic distinctions between graphical entities to aid this predictability. The biggest is the distinction between operation and widget components: operations are pure functions with inputs and outputs. widgets represent some IO interaction, like an input value, a file, or a web service. This way, 'special behavior' is isolated to widgets, making the rest of the script act more predictable.

In practice, certain inconsistencies within Geofront arise due to the open nature of the plugin system. the consistency of geofront is mitigated by a library with a very different notion of naming, or if the library chooses unusual input or output patterns. For example, a euclidean, 3D coordinate can be specified as a `Vector3` object, a structure, an array of three numbers, or three different `x`, `y`, `z` input parameters. Then again, it is unclear if inconsistencies between the API's of a language's libraries are to be contributed to the inconsistency of the language as a whole.

***Diffuseness: How many symbols or graphic entities are required to express a meaning?***

Geofront periodically suffers from the same 'Diffuseness' problems Green and Petre [1996] adheres to VPLs general. That is, sometimes a surprising number of 'graphical entities' / nodes are required to represent a simple statement. This is apparent when representing simple mathematical calculations.

Additionally, the flowchart can only represent linear processes. Many geoprocessing algorithms are iterative and make use of conditionals. These cannot easily be expressed in a Dataflow VPL. As such, these processes must happen within the context of a function, within a 'node'.

A widget evaluating a line of JavaScript could help improve diffuseness. For now, if a user desires to use dense mathematical statements, or functions with many conditionals and complex iteration, a Geofront Plugin should be created.

***Error- proneness: Does the design of the notation induce 'careless mistakes'?***

There are some errors the user can make in Geofront that will not be immediately obvious. The biggest one is that there are no systems in place preventing large calculations. These might freeze up the application.

To prevent this, the geofront interpreter should have been implemented to run on a separate thread, using a web worker. Besides this, in general, many systems are in place preventing errors, such as the type-safety used throughout geofront. Also, by disallowing cyclical graphs, users cannot create infinite loops accidentally.

***Hard mental operations: Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?***

Geofront is developed specifically to prevent "Hard mental operations". Following the dataflow paradigm explained in Section 2.2.3, geofront chose to disallow cyclical patterns. This greatly reduces the complexity of possible graph configurations, and also causes all in-between results to be immutable or 'final'. By then allowing these results to be inspected, and allowing the graph to be easily reconfigured, Geofront allows a workflow rooted in experimentation and 'play'. Users do not need to 'keep track' or 'guess' how things work. Instead, they can simply experience the behavior, and adjust the behavior until satisfied.

***Hidden dependencies: Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?***

The dimension of 'hidden dependencies' is another way the dataflow-paradigm is advantageous. The pure functions of a diagram-based VPL like Geofront make the language in general consistent and predictable. However, there are two exceptions to this rule: First, the widget nodes are allowed to produce side-effects, such as opening a window, asking for an input, making a web request, etc. These are required to provide geofront with interactive inputs and outputs. The distinction between widgets and

And second, the pureness of functions can only be maintained if all Geofront libraries also exclusively use pure functions. There is no fail-safe in place to prevent the usage of a library containing functions with many side-effects.

***Premature commitment: Do programmers have to make decisions before they have the information they need?***

In general, Geofront requires almost no premature commitment. Or, rather, the level of premature commitment is in line with textual programming languages, in the sense that a user is always somewhat committed to the structure they themselves build.

One practical way in which Geofront exceeds in this dimension of premature commitment, is that the application does not require a restart upon loading a new library. Users can add or remove libraries "on the fly". This is unlike any VPL studied at Section 3.2 or Section 3.3.

One particular type of commitment users must be aware of, however, is the commitment to using a VPL like Geofront in general. The current version of Geofront does not support compilation to JavaScript yet, which would mitigate this premature commitment.

***Progressive evaluation: Can a partially-complete program be executed to obtain feedback on 'How am I doing'?***

Yes. As explained at the answer for the dimension of 'Hard mental operations', this aspect, together with the ability to inspect parameters, allows a user to 'debug while they write the program'.

***Role- expressiveness: Can the reader see how each component of a program relates to the whole?***

as the authors of [Green and Petre \[1996\]](#) write: "The dimension of role-expressiveness is intended to describe how easy it is to answer the question 'what is this bit for?'"

One of the ways Geofront addressed this is by making a distinction between nodes possessing pure operations, and nodes producing side effects, like widgets.

***Secondary notation: Can programmers use layout, color, other cues to convey extra meaning, above and beyond the 'official' semantics of the language?***

No, Geofront does not offer annotations in its current state, besides the way the nodes are configured on the canvas. Geofront does provide visual indicators for types, and for if a cable / variable represents a single item, or a list of items.

This could be improved by providing a way to annotate: to create groups, to write comments, etc. Type colors, or some other way to distinguish data based on iconography, would also improve the secondary notation principle.

***Viscosity: How much effort is required to perform a single change?***

Despite these efforts, the 'mouse intensive' interface of VPLs like Geofront continues to be a hinder for viscosity. Certain situations require excessive mouse interaction, like substituting a function with another function, but keeping all inputs the same. In text, this would be as simple as a non-symbolic renaming of the called function. In geofront, this requires a lot of reconfiguration of cables.

Viscosity could be improved by creating special actions in the editor to perform these types of manipulations.

***Visibility: Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?***

All parts of the code are simultaneously visible, and geofront does not offer any way of creating new popups or screens, with the exception of special [GUI](#) widgets.



# 7 | CONCLUSION

This chapter contains the conclusion of the study. It starts out by answering the research questions posed in the introduction (Section 7.1), followed up by a summary of the most significant contributions (Section 7.2) and the limitations of these contributions (Section 7.3). It continues by addressing points of discussion about this study (Section 7.4), which is followed by a number of theorized implications of this study (Section 7.5), and lastly, a reflection on the value and quality of this study (Section 7.6).

## 7.1 CONCLUSION

This section answers the research questions. It starts with answering the sub research questions, and concludes with the answer to the main research question.

### Sub Questions

1. *"What GUI features are required to facilitate this method, and to what extent does the web platform aid or hurt these features?"*

It turned out that two layers of GUI features had to be created:

**Framework application:** Firstly, a base application was required to host the visual programming language, which needed to provide support for multiple windows, dropdown menu's, side menu's etc. For this, a custom framework using Web Components was created, due to the absence of a desktop application-like framework in the JavaScript ecosystem. However, this did not provide the same level of tooling as a library like IMGUI [Cornut, 2022] or egui [Ernerfeldt, 2019]. Because of this, in this area, the web context lead to a sub-optimal experience.

**The VPL:** Secondly, UI elements were required to form the interactive elements of the visual program itself. This GUI was given shape using a custom implementation. DOM events were used as inputs, and the Canvas API provided a way to visualize the VPL. The web context provided two advantages here. Firstly, a JavaScript program can access a large amount of features by default, such as this canvas API, but also WebGL. These features do not need to be included within the source code of the application, leading to quick load times. Secondly, by using `iframe`'s and window pop-ups, the web context could be utilized to integrate third party UIs in a

simple manner, as demonstrated with the Potree demo in Section 6.2. Summarized, for this second set of GUI features, the web platform was advantageous.

2. *"To what extent does this method intent to address the discrepancies between software applications and libraries, as described by Elliott (2007)? Does it succeed in doing so?"*

Based on the features presented at Section 6.2 and Section 6.3, it can be stated that the implementation was able to address all three discrepancies to a certain extent:

**1. Library capabilities get lost when used in an application:** Because of the plugin system of this method, all functions which are included in a wasm compilation can be accessed by a user from within an application. Additionally, the plugin boilerplate comparison of Section 6.3 showed that the no-boilerplate setup makes it simpler to add a new library as a plugin. Both features combined may lead to less features ending up 'lost in translation'.

**2. Applications are not further composable:** The implementation showed that this web-based VPL can make existing applications composable, evident from the demo application presented in Section 6.2. In this example, the Potree application was composed within a pipeline to extract a DTM or DSM, while retaining full functionality as a standalone application.

**3. A library offers no visualization or GUI by itself, and must be turned into an application before it can be used:** Because the method presented in this study succeeded in combining GUI components with low-entry barrier plugins, and because it allows the usage of unaffiliated WebAssembly projects, Geofront pipelines may act as a "custom GUI for any library" to an extent. In that sense, end users will only have to wait for a WebAssembly build, instead of a full application, before they are able to access a libraries content.

3. *"What are the differences between compiling a GIS library written in C++ to WebAssembly, compared to compiling a GIS library written in Rust?"*

The differences between C++ and Rust compilation are caused by the difference in their respective WebAssembly compilers: C / C++ requires the usage of emscripten and embind. Rust requires the usage of wasm-pack, and wasm-bindgen.

In the experiments performed by this study, significant differences were encountered between these compilers in terms of file size, supported features, and the performance of the produced 'glue code'. The results of Section 6.1 showed that the emscripten compiler produced a binary which requires more than three times the size of the same functionality compiled with wasm-pack. Interfacing this binary with JavaScript was between six and seven times as slow compared to the rust equivalent.

Moreover, emscripten lacked interface features which were required for the purpose of using WebAssembly to compile libraries for generic use by any language.



wasm-pack provides type declarations in the shape of a TypeScript file, or by embedding them in WebAssembly using the proposed interface types [Wagner, 2022]. On top of that, it offered a strong binding system, allowing complex types such as a vector of floats (`Vec<f32>`) to be easily translatable to a JavaScript equivalent (`Float32Array`).

Regrettably, none of the above features were present in emscripten. Custom interventions were necessary on both the C++ end, and the JavaScript end, to get to a similar level of functionality. However, these interventions made it so both ends need to be developed in relation to each other, which invalidates the WebAssembly binary as a *generic* interface.

Because `wasm-pack` produces WebAssembly binaries with generic, simple, typed interfaces, Rust libraries could be used within the methodology presented in this study. However, because binaries produced by emscripten did not possess these traits, C++ libraries could not be used within the methodology, meaning the goal of making core C / C++ GIS libraries more directly accessible could not be met.

All in all, it must be concluded that Rust-based WebAssembly compilation is more performant, more lightweight, and more feature rich compared to C++-based compilation.

4. *"What measures are taken to make this VPL scalable to large geo-datasets, and how effective are these measures?"*

Three measures were conceptualized and successfully implemented. However, all measures do possess certain shortcomings. It must also be noted that all these measures are only presented 'in principle'. Future work is required to verify their impact in practice.

1. **Portability:** The application uses native geocomputation libraries compiled to WebAssembly, which makes the libraries behave the same way in the front-end, and in a potential back-end.  
**Effectiveness:** The effect of this is limited by the fact that `wasm` binaries must for the time being be wrapped using JavaScript, due to the current lack of interface types. This means that currently, a backend would require a JavaScript runtime to use these libraries.
2. **Zero-cost abstraction:** The application uses a zero-boilerplate plugin system, to offer direct usage of JavaScript-wrapped libraries, and eventually WebAssembly-compiled libraries. This means that there is no difference between calling an operation in Geofront, and calling the function in the JavaScript-wrapped library. This allows a Geofront pipeline to be compiled to JavaScript, completely eliminating any explicit dependency to the Geofront platform, the VPL model, or the `GUI`. This is why Geofront can claim to use 'zero cost abstraction'.  
**Effectiveness:** A Geofront-to-javascript compiler was implemented and functional, but requires further study. For example, it does not compile complex `GUI` Widgets, and does not support iteration.

3. **Locality:** Geofront is designed as a Dataflow VPL, which shares characteristics with functional programming. This leads to source code which can be reasoned about in a local manner, instead of a global one. This allow for parallelization.

**Effectiveness:** The implementation of Geofront as a dataflow VPL was successful. However, due to limitations of the JavaScript language, the functional programming properties of the DataFlow VPL could not be strictly enforced. Plugins must be trusted not to alter input data, or call global, state-altering functions, as functions could not be declared as 'pure', and variables cannot be declared as 'immutable'.

5. *"How does this method compare to existing, alternative VPLs and browser-based geocomputation methods, regarding the properties relevant to the goal of direct accessibility?"*

Based on Table 1 presented in Section 6.3, Geofront provides a unique set of features. Comparable visual languages do not possess this exact combination. It offers relatively extensive plugin support compared to other VPLs, it is web based, and offers a range of GUI nodes. Moreover, it is unique in its ability to accept third party Plugins written in different types of languages, and in the fact that it allows custom types within those plugins.

### Main Question

*"Is a web based VPL a viable method for directly accessing native GIS libraries with a composable interface?"*

Based on the answers to all supporting questions, The answer is a reserved **yes**.

The method is indeed capable of bringing native GIS capabilities from certain libraries directly into contact with end users, from within an application, without installation or configuration, and in a further composable manner. Compared to browser-based web applications, this method is more composable thanks to the dataflow-VPL implementation, and compared to the studied native VPLs, these functionalities are more directly available because of the static, web based implementation. The method is also unique compared to studied web-based geometry VPLs, because of the plugin system, the range of different GUI nodes, the dataflow VPL properties, and the proposed zero-cost abstraction runtime. All of these features combined lead to a VPL which is able to directly connect GUI components with native GIS libraries, all while remaining scalable in principle.

On a practical level, more work remains to proof this feasibility. The methodology developed by this study is only *theoretically* accessible and composable, based on achieved features. User-testing is required to confirm if this method indeed improves workflows, and actually saves time and energy of developers and end users. Moreover, the prototypical software implementation used is limited and not

production ready. C / C++ GIS libraries cannot be used in the method presented, which may or may not be fixed when the emscripten compiler adopts the Interface Types proposal. Additionally, the zero-cost abstraction runtime is non-functional, and must be improved upon in future work.

Despite all of this, based on the presented work, it is safe to say that visual programming methods, distribution using WebAssembly, and Rust-based GIS, all remain promising, valuable directions of future GIS research.

## 7.2 CONTRIBUTIONS

The study was able to deliver two major contributions:

- **A new implementation of a web-based geocomputation VPL** This study introduces a novel JavaScript implementation of a web-based dataflow-VPL capable of both geometry processing, as well as geocomputation. Compared to existing web-based alternatives, this VPL is closer in design and functionality to common geometry VPLs like grasshopper, as it adheres to being a graph-based dataflow VPL.
- **A novel workflow of publishing and using native libraries on the web** Secondly, a new workflow was developed to allow a geo-computation function or library to be used within a visual programming environment. Moreover, this can be done with a minimum of configuration steps: Any Javascript, Typescript or Rust library which satisfies the conditions layed out in Section 5.2.1, automatically functions in Geofront.

These two contributions together lead to an environment suited for a number of use cases, including:

- **Visual debugging:** One can use this environment visualize the result of an algorithm in 2D or 3D.
- **Fine tuning parameters:** In situations where algorithms contain unintuitive or empirically derived parameters (e.g. RANSAC), a visual environment can be use to quickly try out multiple settings, and to observe their effects.
- **Benchmarking:** Geofront can be used to test the web performance of multiple algorithms written in different languages.
- **Publication:** Geofront scripts can be shared using a link. this can be used to make a native library usable online, and by doing so, it may help to lower the delta between 'my library works for me' and 'my library works for someone else'.

The combination of these features together make Geofront unique among both geo-VPLS and web-VPLS. by providing the full source code, together with all implementation details given in Chapter 5, this study aims to provide guidance for all subsequent studies on the topic of VPLs, geocomputation, or geoweb applications using WebAssembly.

### 7.3 LIMITATIONS

These contributions are bound by following limitations:

- **Only Rust, Js & Ts library support** For now, only libraries written in Rust or Javascript / Typescript can be used in Geofront. Due to the results laid out in Section 6.1, a stable method of using any C++ library can not be provided for at the current moment.
- **In practice, not all libraries can be used** Section 5.2.1 shows that not all Rust and JS/TS libraries are supported. Additionally, in order to properly communicate, visualize, and make data interoperable, special 'config' functions and methods are still required.
- **Only small-scale geodata is possible** the Geofront environment uses browser-based calculations, which does not lend itself well to process datasets larger than a certain threshold. This means it cannot be used properly for big data, or other expensive processes.
- **Implementation shortcomings** Geofront is a prototype, and has many usability shortcomings, explained in Section 6.4. In addition, many geocomputation-specific aspects are missing, such as a topographical base layer.

### 7.4 DISCUSSION

This section covers questions on the decisions made during the study, and the answer this study is able to provide as a response.

*Q: Why wasn't Geofront developed as a native application, and published to the web as a whole?*

A native-first build of Geofront would indeed be more performant, especially if the application is fully hybridized: If both a native and web build of an application are available, then users can choose for themselves if they wish to sacrifice the performance and native experience for the accessibility of a web build.

However, many features key to the solution and workflow specific to Geofront would be lost in such a setup. (Prospected) features like dynamically loading plugins, scriptable components, or the compilation to JavaScript would be lost, or would have to be regained by incorporating a browser engine *within* this native application.

However, a valid criticism can be made that this study could have opted for adding more native-first components, such as the maplibre renderer [Ammann et al., 2022]

*Q: A large reason for developing web-based VPLs is accessibility. Is this environment accessible?*

This study can only answer this question to a limited degree. Based on the analysis given at Section 6.4, it is safe to say that based on its features, Geofront is about as accessible as comparable geo-vpls, like Geoflow or Grasshopper. However, this analysis is only based on the achieved functionality and features. Actual user-testing is required to assess The true accessibility of the tool.

*Q: Is this environment a competitor to native methods of geocomputation?*

In theory, yes. Using the workflow as described, native geocomputation libraries could be used on the web at near native performance, without requiring installation. Additionally, the web offers enough functionality so that even sizable, local datasets could be processed this way. In practice, the dilemma between Rust and C++ means that in the sort term, this environment will not be used for professional geocomputation. Additionally, the tool is still in a prototypical state, and will need to be more stable and reliable before being used professionally.

## 7.5 FUTURE WORK

The many fields this study draws from mean that a great variety of auxiliary aspects were discovered during the execution of the study. Some of these aspects are listed here, and could lead to interesting topics for follow-up research.

### 7.5.1 Cloud-native deployment & scalability

As presented, an early build of Geofront had the ability to compile a Geofront script to regular JavaScript (see (Figure 30)). All libraries were converted to normal import statement, all nodes were replaced by function calls, and the cables substituted by a variable token. This way, the application could be run headless in either the browser or natively, using a local JavaScript runtime like Deno [Deno Contributors,

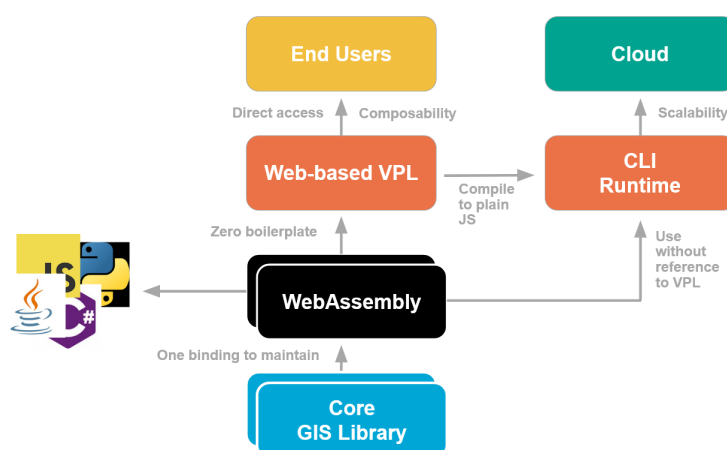


Figure 58: An expanded proposed methodology, repeated

2022]. However, time constraints led this feature to be abandoned in favor for other aspects.

A future study could re-implement this feature, opening up the possibility for deployment and scalability: Scripts created with geofront could then deployed as a web worker, as a web applications of themselves, or as a web processing services [Open Geospatial Consortium, 2015]. Also, by running this script on a server, and ideally a server containing the geodata required in the process, one could deploy and run a Geofront scripts on a massive scale.

The overall purpose of this would be to create an open source alternative to tools like the Google Earth Engine [Google, 2022], and FME cloud compute [Safe-Software, 2022a]. In conjunction with statically hosted geodata, Geofront could become a 'cloud native' platform, allowing end users both to construct geodata pipelines, and execute them natively on the cloud as well.

### 7.5.2 Streamed, on demand geocomputation

This study showed that browser-based geocomputation is reasonably viable. This might allow for a new type of geoprocessing workflow, which could replace some use-cases that now require big-data processing and storage. A big problem in the GIS industry is having to process and store sizable datasets, while only a portion of it will be actually used. A possible solution could be to take a raw dataset base layer, and process it on-demand in a browser.

This would have several advantages. First, end-users can specify the scope and parameters of this process, making the data immediately fine-tuned to the specific needs of this user. Secondly, this could be a more cost-effective method, as cloud computation & Terabytes of storage are time consuming and expensive phenomena.

This type of *on demand geocomputation* is certainly not a drop-in replacement for all use cases. But, in situations which can guarantee a 'local correctness', and if the scope asked by the user is appropriate, this may be possible. Examples of this would be a streamed TIN interpolation or color-band arithmetic.

### 7.5.3 Rust-based geocomputation & cloud native geospatial

An interesting aspect this study was able to touch on is using the Rust language for GIS processing and analysis. The reason for this was the extensive support for webassembly, which was essential for allowing GIS tools to be used in the browser. However, there are additional reasons one might want to perform geocomputation within Rust. One is that rust is widely considered as a more stable, less runtime error-prone language than C++, while offering similar performance and features. Additionally, rust Wasm binaries also tend to be smaller than C++ wasm binaries.

This could be valuable to the "cloud-native geospatial" movement. This [GIS](#) movement aims to create the tools necessary to send geocomputation to servers, rather than sending geodata to the places where they are processed. To do this, geocomputation must become more portable than it currently is, and Rust compiled to WebAssembly might prove to be a strong candidate for creating exchangeable, performant, compact, and error-proof binaries. It already sees usage on both cloud and edge servers [[Eberhardt, 2022](#)].

Therefore, studying Rust-based geocomputation for the purpose of cloud native and edge computing, would be a promising topic for subsequent research.

### 7.5.4 FAIR geocomputation

The introduction theorized on how both VPLS and web-apps could be used to make GIS tools more directly available. The study chose to pursue this on a practical, technical level.

However, a more theoretical study could also be performed. It turns out that the concepts of 'accessible GIS tools', have something in common with many of the geoweb studies on data accessibility and usability [[Brink, 2018](#)]. The ideas of 'data silo's', 'FAIR geodata', and 'denichifying of [GIS](#) data' (see [Brink \[2018\]](#)) may be used as well as well on GIS tooling: Functionality Silo's, FAIR geocomputation, denichifying of [GIS](#) computation.

Therefore, an interesting question for a subsequent study could be: "How could geocomputation become more Findable, Accessible, Interoperable, and Reusable?", or "How to integrate the function-silo's of GIS, BIM & CAD?" By focussing on data processing actions rather than the data itself, we may shed a new light on why data

discrepancies and inaccessibility exist. After all, if a user is unable to convert retrieved geodata to their particular use case, then the information they seek remains inaccessible.

## 7.6 REFLECTION

Here I reflect on possible shortcomings of the thesis in terms of value and quality, and how I have attempted to address these shortcomings.

### **Biases regarding C++ and Rust**

First of all, in the comparison between C++ and Rust, the studies conducted proved to be unfavorable towards C++. However, C++ and `emscripten` may have been used improperly, due to the authors personal inexperience with the build tooling of the language. Many complications were encountered during compilation, leading to extensive editing of makefiles, and attempts at recompiling forked subdependencies of CGAL using unsustainable workarounds. It is unknown how much of this was due to personal C++ inexperience, inflexibility of the libraries in question, or the shortcomings of the toolchain.

Despite this, the study performed steps to make the judgement as non-biased as possible. Preliminary studies were conducted with both languages, and multiple demo projects have been made with both. Additionally, before the assessment between Rust and C++, approximately the same amount of time was spent with both languages. One could even argue that this last fact may make this judgement more fair compared to an author well-accustomed to C++.

### **Scope too wide**

Additionally, the scope generated by combining geocomputation, web applications and VPLs, may have made the scope of this study too extensive. This is evident in the great number of related studies, and the sizable workload of the implementation. A better approach might have been to focus the scope of the thesis down to only 'browser-based geocomputation', 'visual programming and GIS', or 'portable GIS libraries using rust', to allow for a more in-depth analysis.

On the other hand, the core of the contribution of this thesis lies precisely in the attempt to form bonds between these subjects, especially since prior studies remained by en large closely scoped to their respective domains. The position taken by the author was that a certain synergy may exist, and that each separate domain stand to gain from the ideas and knowledge found in the other ones. In order to make this possible, the study had to acquire a scope to explore all in-between synergies and interactions, leading to geo-vpls, web-vpls, and browser-based geocomputation. Now that this study has made these connections explicit, future studies can focus on more precise aspects of these cornerstones again.

### **Too distant from the field of GIS**



Where the exact boundary of one field of study is, and where another begins, remains of course a fuzzy question. Still, the direction of this study appears to stray far from 'core GIS concerns', and appears more in one line with the field of "End User Development (EUD)", and fields like "Computer-Human interactions".

In defense of this, during the implementation of the study, it appeared that little foundation was in place for a web VPL specifically. This made it necessary to generalize, to build the missing foundation first. For example "How can *a* library be compiled and loaded into *a* VPL" is a question which had to be answered first. Then, the question could be specified to *geometry* and *Web VPL*. And only after that, the geodata and geoprocessing libraries specific to the field of GIS could be regarded. By doing so, this study wishes to provide a foundation to assist any subsequent future study in this direction, which can then be more GIS focussed.

### **Subjectivity in qualitative assessment**

Lastly, many of the assessments made by this study are qualitative assessment, and as such, might suffer from a high level of subjectivity. This is unavoidable in any assessment which does not come down to clear, quantifiable aspects, such as performance, memory usage or precision.

Nevertheless, the study has attempted to scope this subjectivity by basing its assessments heavily on prior works in the field of VPLs, showcasing clear examples, and by focusing on achieved features.





## SOFTWARE IMPLEMENTATION

All source code of the implementation is publicly available under the MIT license, and can be found on the version control platform Github, at the 'thegeofront' organization: <https://github.com/thegeofront>. Additionally, the Geofront application itself is hosted at <http://geofront.nl>, or <https://thegeofront.github.io/>.



## BIBLIOGRAPHY

- Akhmechet, S. (2006). Functional Programming For The Rest of Us. <https://www.defmacro.org/2006/06/19/fp.html> Accessed 2022-08-18.
- Alicevision (2022). Meshroom. <https://github.com/alicevision/Meshroom>. Accessed 2022-09-27.
- Ammann, M., Drabble, A., Ingensand, J., and Chapuis, B. (2022). Maplibre-rs: Towards portable map renderers. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-4-W1-2022:35–42. ISSN: 1682-1750.
- Azavea (2011). GeoTrellis. <https://github.com/locationtech/geotrellis> Accessed 2022-09-27.
- Bell, A., Butler, H., and Manning, C. (2021). Cloud Optimized Point Cloud Specification – 1.0. <https://copc.io/> Accessed 2022-10-27.
- Blender Foundation and Contributors (2022). Geometry Nodes — Blender Manual. [https://docs.blender.org/manual/en/latest/modeling/geometry\\_nodes/index.html](https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/index.html) Accessed 2022-09-27.
- Brink, L. v. d. (2018). Geospatial Data on the Web. PhD Thesis, Delft University of Technology.
- Clark, L. (2019). WebAssembly Interface Types: Interoperate with All the Things. *Mozilla Hacks: the Web developer blog*. <https://hacks.mozilla.org/2019/08/webassembly-interface-types> Accessed 2022-10-29.
- Control Automation (2018). Ladder diagram ld programming | basics of programmable logic controllers (plcs). *Automation Textbook*. <https://control.com/textbook/programmable-logic-controllers/ladder-diagram-ld-programming/> Accessed 2022-06-09.
- Cornut, O. (2022). Dear ImGui. <https://github.com/ocornut/imgui> Accessed 2022-10-28. original-date: 2014-07-21T14:29:47Z.
- Dashiki (2020). Simple Request Breakdowns. <https://analytics.wikimedia.org/dashboards/browsers/#desktop-site-by-browser/browser-family-timeseries> Accessed 2022-09-06.
- Deno Contributors (2022). Deno. <https://github.com/denoland/deno> Accessed 2022-09-27.
- Dohler, Derek, C. (2016). GDAL JS. <https://github.com/ddohler/gdal-js> Accessed 2022-10-28.

- Dufour, D. (2017). geotiff.io. <https://github.com/GeoTIFF/geotiff.io> Accessed 2022-09-27. original-date: 2017-06-30T02:12:06Z.
- Eberhardt, C. (2022). The State of WebAssembly 2022. <https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html> Accessed 2022-09-21.
- Electron Contributors (2022). Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS. <https://www.electronjs.org/> Accessed 2022-09-27.
- Elliott, C. (2007). Tangible functional programming. *International Conference on Functional Programming*. <http://conal.net/papers/Eros/> Accessed 2022-09-27. Talk: <https://www.youtube.com/watch?v=faj8N0giqzw>.
- Epic Games (2022). Blueprints visual scripting: Documentation. <https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/> Accessed 2022-09-27.
- Ernerfeldt, E. (2019). egui: an easy-to-use GUI in pure Rust. <https://github.com/emilk/egui> Accessed 2022-10-28.
- Esri (2022). ModelBuilder | ArcGIS for Desktop. <https://desktop.arcgis.com/en/arcmap/10.3/analyze/modelbuilder/what-is-modelbuilder.htm> Accessed 2022-09-27.
- Fabri, A. and Pion, S. (2009). CGAL: the Computational Geometry Algorithms Library. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '09*, pages 538–539, New York, NY, USA. Association for Computing Machinery.
- Francese, R., Risi, M., and Tortora, G. (2017). Iconic languages: Towards end-user programming of mobile applications. *Journal of Visual Languages & Computing*, 38:1–8.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. ISBN: 978-0-201-63361-0.
- Geodelta (2022). Omnibase. <https://geodelta.com/en/software/omnibase> Accessed 2022-09-27.
- Google (2022). Google Earth Engine. <https://earthengine.google.com> Accessed 2022-10-29.
- Green, T. R. G. and Petre, M. (1996). Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework. *Journal of Visual Languages & Computing*, 7(2):131–174.
- Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 185–200, New York, NY, USA. Association for Computing Machinery.

- Hamilton, E. L. (2014). Client-side versus Server-side Geoprocessing: Benchmarking the Performance of Web Browsers Processing Geospatial Data Using Common GIS Operations. Msc Thesis. University of Wisconsin.
- Jangda, A., Powers, B., Berger, E. D., and Guha, A. (2019). Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. Pages 107-120, ISBN: 978-1-939133-03-8.
- Janssen, P. (2021). Möbius Modeller. [https://mobius.design-automation.net/pages/mobius\\_modeller.html](https://mobius.design-automation.net/pages/mobius_modeller.html) Accessed 2022-09-27.
- Kuhail, M. A., Farooq, S., Hammad, R., and Bahja, M. (2021). Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access*, 9:14181–14202.
- Kulawiak, M., Dawidowicz, A., and Pacholczyk, M. E. (2019). Analysis of server-side and client-side Web-GIS data processing methods on the example of JTS and JSTS using open data from OSM and geoportal. *Computers & Geosciences*, 129:26–37.
- ltd, R. a. M. (2021). Low-code development platform market research report - global industry analysis, trends and growth forecast to 2030. <https://www.researchandmarkets.com/reports/5184624/low-code-development-platform-market-research> Accessed 2022-08-18. Note: Unreliable data.
- Melch, A. (2019). Performance comparison of simplification algorithms for polygons in the context of web applications. Msc Thesis. <https://qgit.de/melch/mt-polygon-simplification/raw/commit/9f19a7b3c2fd086d3a387a200a761e20c3d7bacd/build/mt-polygon-simplification-2019.pdf>.
- Mozilla (2013). asm.js specification. <http://asmjs.org/> Accessed 2021-12-22.
- Mozilla (2022). MDN Web Docs. <https://developer.mozilla.org> Accessed 2022-09-27.
- Open Geospatial Consortium (2015). Web processing service. <https://www.ogc.org/standards/wps> Accessed 2022-09-27.
- OpenJS Foundation (2022). Node-RED. <https://nodered.org/> Accessed 2022-09-27.
- Panidi, E., Kazakov, E., Kapralov, E., and Terekhov, A. (2015). Hybrid Geoprocessing Web Services. DOI: 10.5593/SGEM2015/B21/S8.084.
- Perlin, K. (2002). Improved Noise reference implementation. <https://mrl.cs.nyu.edu/~perlin/noise/> Accessed 2022-10-28.
- Peters, R. (2019). Geoflow. <https://github.com/geoflow3d/> Accessed 2021-12-22.
- QGIS Community (2022). Qgis. <https://qgis.org/en/docs/index.html> Accessed 2022-01-06.

- React Contributors (2022). React – A JavaScript library for building user interfaces. <https://reactjs.org/> Accessed 2022-09-27.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11):60–67.
- Rust Contributors (2022). The rust programming language : ownership. <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#what-is-ownership> Accessed 2022-09-28.
- Rutten, D. (2012). Grasshopper. <https://www.grasshopper3d.com/> Accessed 2022-09-19.
- Safe-Software (2022a). FME Cloud | Hosted Data Integration. [https://docs.safe.com/fme\\_cloud/FME\\_Cloud/Content/Home.htm](https://docs.safe.com/fme_cloud/FME_Cloud/Content/Home.htm) Accessed 2022-10-29.
- Safe-Software (2022b). FME Desktop | Data Integration and Automation. <https://www.safe.com/fme/fme-desktop/> Accessed 2022-09-27.
- Sandhu, P., Herrera, D., and Hendren, L. (2018). Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript and WebAssembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang '18*, pages 1–13, New York, NY, USA. Association for Computing Machinery.
- Sarago, V., Barron, K., and Albrecht, J. (2021). Cloud Optimized GeoTIFF. <https://www.cogeo.org/> Accessed 2022-10-27.
- Schütz, M. (2012). Potree: WebGL point cloud viewer for large datasets. <https://github.com/potree/potree> Accessed 2022-10-28.
- SideFX (2022). Houdini 3D Procedural Software. <https://www.sidefx.com/products/houdini/> Accessed 2022-09-27.
- Sit, M., Sermet, Y., and Demir, I. (2019). Optimized watershed delineation library for server-side and client-side web applications. *Open Geospatial Data, Software and Standards*, 4(1):8.
- Sousa, T. (2012). Dataflow Programming: Concept, Languages and Applications. Unpublished.
- StatCounter Global Stats (2020). Browser Market Share Worldwide.
- The NetMarketshare Team (2020). Browser market share. <https://netmarketshare.com/browser-market-share.aspx> Accessed 2022-09-06.
- Unity Technologies (2021). Bolt. <https://docs.unity3d.com/bolt/1.4/manual/index.html> Accessed 2022-09-27.
- w3c (2019). World Wide Web Consortium brings a new language to the Web as WebAssembly becomes a W3C Recommendation. <https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en> Accessed 2021-11-30.



- W3Counter (2020). Global Web Stats. <https://www.w3counter.com/globalstats.php> Accessed 2022-09-06.
- Wagner, L. (2022). Interface Types Proposal. <https://github.com/WebAssembly/interface-types/blob/main/proposals/interface-types/Explainer.md> Accessed 2022-10-27.
- wasm-bindgen Contributors (2022). `wasm-bindgen`. <https://github.com/rustwasm/wasm-bindgen> Accessed 2022-09-27.
- wasm-pack Contributors (2018). `wasm-pack`. <https://github.com/rustwasm/wasm-pack> Accessed 2022-09-27.
- WebAssembly Contributors (2022). WebAssembly. <https://webassembly.org/> Accessed 2022-09-27.
- Weber, I., Paik, H.-Y., and Benatallah, B. (2013). Form-Based Web Service Composition for Domain Experts. *ACM Transactions on the Web*, 8(1):2:1–2:40.
- Yu, B. (2021). CS50’s Introduction to Programming with Scratch. <https://cs50.harvard.edu/scratch/2021/> Accessed 2022-09-27.
- Zakai, A. (2011). Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, OOPSLA ’11*, pages 301–312, New York, NY, USA. Association for Computing Machinery.

## COLOPHON

This document was typeset using  $\text{\LaTeX}$ , using the KOMA-Script class `scrbook`. The main font is Palatino.

