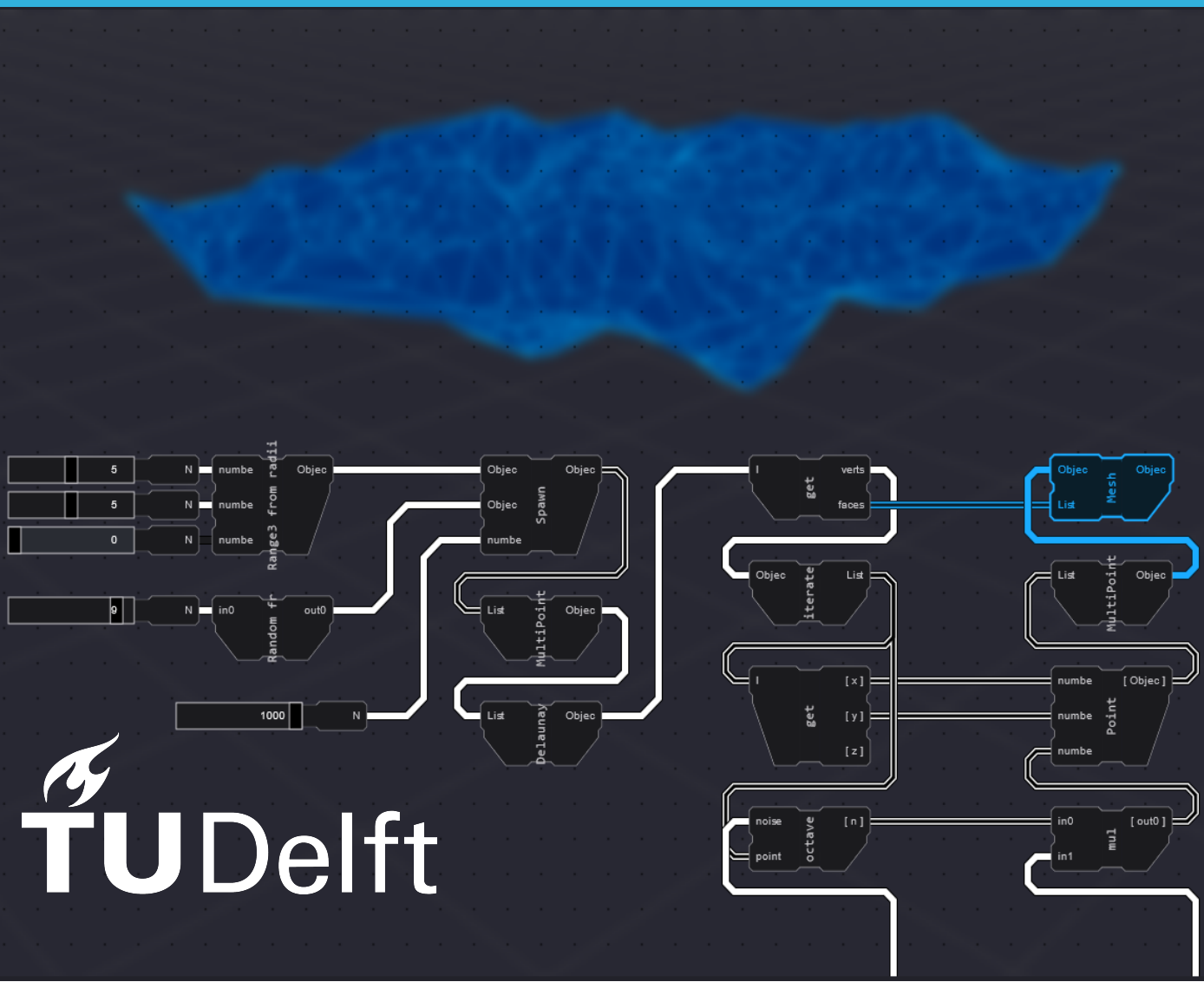
MSc thesis in Geomatics

Geofront: Library portability in a browser-based visual programming language for geo-computation

Jos Feenstra

2022



TU Delft

MSc thesis in Geomatics

Geofront: Library portability in a browser based visual programming language for geo-computation

Jos Feenstra

June 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: Library portability in a browser based visual programming language for geo-computation* (2022)

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

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

Abstract

Geocomputation is a cornerstone of GIS & modern mapping needs. While the last decades have seen major advancements, writing a geocomputational pipeline remains a complex, non-trivial exercise. Performing geocomputations by utilizing a visual programming language (VPL) within a web browser is a novel development which seeks to simplify this process, allowing more people to write these types of pipelines more successfully.

In this context, the portability of existing geocomputation libraries is a common problem. This is often addressed by maintaining duplicate JavaScript alternatives to libraries such as GDAL, PROJ, and GEOS, complicating innovation.

This study seeks to improve the state of browser-based geocomputation VPLs by attempting to bringing industry-standard geo-libraries to these environments. The study poses that this can only be done if native geocomputation libraries can be *compiled* to WebAssembly, *loaded* into a VPL, and *utilized* in a browser-based dataflow-VPL format. Discovering if and how these steps can be performed is the central question of this study.

This question is answered by testing a possible solution. A proof of concept web-based VPL is designed, implemented and evaluated, which makes use of a novel plugin system, as well as a directed, acyclic graph-based data model & interface.

Using this proof of concept, the study was able to demonstrate that the web platform was sufficiently capable of representing a dataflow VPL capable of constructing geocomputation pipelines. The functional programming-properties of this dataflow VPL also makes geo-libraries sufficiently *usable*, albeit with some well-known caveats of dataflow-VPLs, like the representation of conditionals and iteration.

The current methods of *compiling* existing C++ geocomputation libraries to the web turned out to be insufficient for the purposes of this study. This is due to the Emscripten compiler's focus on compiling full C++ applications instead of separate libraries. Despite this, the study wás able to demonstrate how a novel method can be used to sufficiently *compile* and *load* multiple Rust-libraries for usage in the VPL, thanks to more feature-rich WebAssembly tools in the Rust ecosystem. While Rusts geocomputation libraries are young, the study presents this method to either offer emscripten contributors a blueprint of a desired workflow, or to offer geocomputation library contributors a powerful use-case for the Rust language.

All in all, this means that either if the geocomputation libraries found in the Rust ecosystem mature, or if Emscripten's capabilities improve, then the code portability problem & dataflow problem of existing web-based geocomputation VPLS can be solved.

Acknowledgements

I really want to thank, in no particular order:

- Stelios
- Ken
- Giorgio
- Hugo
- Nadja
- Martin, Current employer, GeoDelta
- Sybren, Previous employer, Sfered
- Tim Boot, for proofreading
- Friends & Family

(I will write this out nicely for P5)

Contents

Acronyms

1 Introduction

The field of Geographical Information Science (GIS) concerns itself with the collection, processing, storage, and visualization of geodata. By doing so, we offer 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.

Central to these goals is Geospatial data computation (geocomputation). The term geocomputation, or geodata processing, is used to represent all types of computations performed on geographical datasets. Anything from the calculation of the area of a region, to a Coordinate Reference System (CRS) transformations, or converting a raster dataset into a vectorized dataset, can be regarded as geocomputation.

geocomputation is a cornerstone of GIS, and vital to almost all its applications. In the field of GIS, the raw *data* gathered from direct land surveyance (e.g. terrain height samples) seldom equals to the precise *information* we wish to discover about the earth (e.g. is my city in risk of flooding?). To convert data into more rich information, a refinement and analysis process by means of geocomputation is required. Geocomputation is also necessary for specializing geodata. For example, various computations are needed for converting raw cadastral records into a base map useful to urban planners.

Despite major advancements during the last decades in geocomputation and geocomputation platforms, geocomputation remains no trivial exercise. The geometric nature of geocomputation, together with the sheer scope of geodata and the variety in formats and quality, make geocomputation both computationally intensive and difficult to operate.

While researchers and developers have made considerable improvement over the last decades, the improvement of both geocomputation, and the activity of geocomputation, remains as relevant as ever before.

In recent years, a new development regarding geocomputation is emerging: browser-based geocomputation by means of a Visual Programming Language (VPL). The goal of this study is to contribute to this development. However, to specify its contribution, an explanation of this method must first be given.

The (web) VPL

A visual programming language is a type of programming language which uses an alternative model to the textual source code of regular programming languages. These alternative models are represented by a Graphical User Interface (GUI), in which the program might take the shape of for example a graph, or a block-based instruction set. The value a VPL lies in the fact that constructing or editing a program in a visual way can be a faster process than using a textual method, even for the most experienced software developers [Green and Petre, 1996;

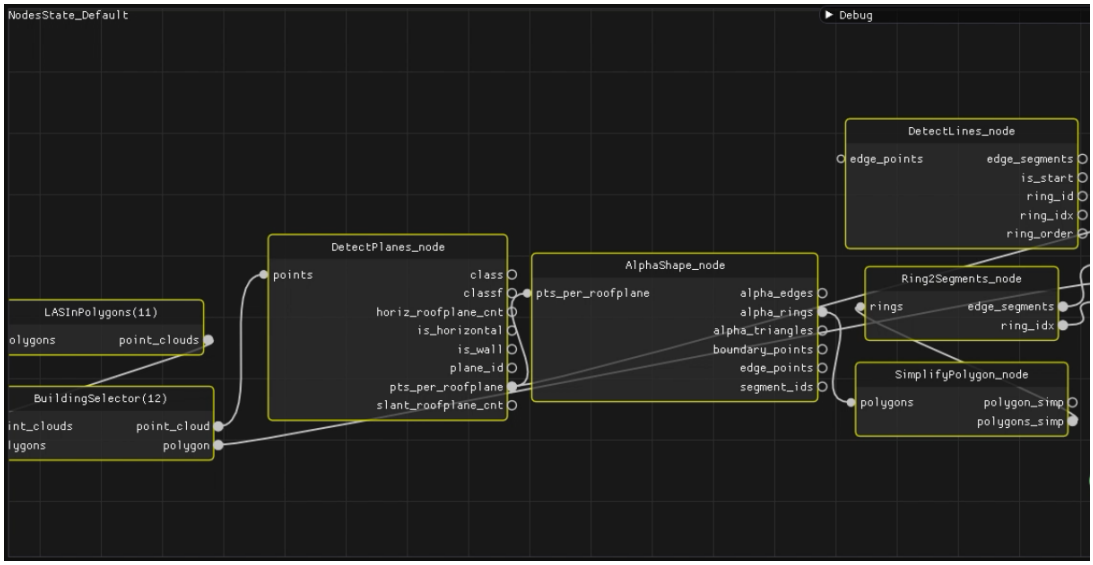


Figure 1.1: Geoflow: A geocomputation VPL. [Peters, 2019]

Kuhail et al., 2021]. Additionally, a VPL allows for so-called End User Development [Kuhail et al., 2021]. A VPL makes development more accessible to end users, as it allows practitioners to automate some processes and workflows like a software developer, but without needing a full background in software development [Benac and Mohd, 2022]. Finally, a VPL may even possess performance and reliability advantages over textual languages if it is implemented as a dataflow-type VPL [Sousa, 2012].

These qualities are beneficial to geocomputation, and perhaps as a consequence, geocomputation by means of visual programming is not an uncommon phenomena in the field of GIS. For example, Safe software's FME [Safe-Software, 2022] is a popular VPL among GIS practitioners, arguably due to the way it simplifies the process of writing a Extract Transform Load (ETL) pipeline. Another example is McNeel's Grasshopper [Rutten, 2012], often used for small-scale spatial analyses, like solar irradiation or heating demands of a neighborhood [Sadeghipour Roudsari and Pak, 2013].

A new development regarding Geocomputation VPLs is attempting to make these VPLs run in a web browser. The value of this is that web applications require no explicit installment to use the application, giving them distribution and accessibility advantages over native applications [Kuhail et al., 2021; Panidi et al., 2015]. The Mobius Modeller is a good example of a browser-based geocomputation VPL [Janssen, 2021]. The application allows users to configure a calculation by visual means in a web browser, which can then be shared as a stand-alone web application.

Problem Statement

The web based geocomputation VPL is a novel development, and contains a multitude of unaddressed challenges. This study focusses on the particular problem of library portability.

Successful geocomputation pipelines often relies on operations found in a select number of libraries. Most major, industry-standard geocomputation libraries are native, C++-based libraries. Examples of these are GDAL, PROJ, and GEOS. However, A web based geocomputation VPL like the Mobius Modeller does not make use of these libraries, and uses javascript equivalents like Turf [Contributors, 2022h]. While it is possible to port a library like GDAL to javascript, guarantying that these two versions show the same behavior will be complex and time consuming. Additionally, managing duplicate libraries like this will slow down development, or create large version discrepancies [Ammann et al., 2022].

Fortunately, on a theoretical level, it is possible to compile a native C++ library to a web-consumable format using the 'WebAssembly' binary format [Haas et al., 2017]. This way, only one library needs to be maintained to serve all platforms, both native and web. However, to the best of the author's knowledge, no example exists of a web-based geocomputation VPLs which make use of native geocomputation libraries. The theoretical possibility combined with the practical absence of any implementation, leads this study to assume that a practical inhibition is in place, preventing native geocomputation libraries to be used by a web VPL.

The source of this inhibition is unclear. There are many aspects and components to compiling a native geocomputation library, and using it on the web, in a VPL format. Given the absence of an implementation, the possibility remains that it could be any one of these aspects and components, or even an interplay between all of these factors. This study identifies three main challenges to using a native geocomputation library in a web-based VPL:

Compilation

Firstly, the challenge of compilation. As explained before, the most viable option for using a non-js library in a web browser, is by compiling it to WebAssembly [Haas et al., 2017]. Other options exist, like simply rewriting non-js languages to JavaScript, but these methods have significant drawbacks [Haas et al., 2017; Jangda et al., 2019]. However, as described in Section 3.1 compiling libraries to WebAssembly (wasm) also may pose challenges:

- wasm promises a 'near native performance' [Haas et al., 2017]. However, this can be quite situational, as multiple studies have shown [Jangda et al., 2019; Melch, 2019].
- wasm cannot compile all code. Its containerized nature means that code accessing a file system for example, does not function without workarounds.
- Compiled wasm code could be difficult to access and interface in a web browser. Without third-party tools, functions exposed by wasm can only accept primitive data types as input. There is no `string` data type, let alone a `struct` or `object` type.
- Compiling a *library* to wasm is ~~seriously~~ different from compiling a full *application* to wasm. A library requires more complicated wasm-javascript interoperability, which third-party tools may or may not be able to provide.

Loading

Secondly, even if the libraries are compiled to a web-ready format, geocomputation functionalities might not be able to **load** properly within a web-based VPL.

This is a challenge of designing an appropriate binding system or plugin loader. When writing Python bindings to a C++ library for example, one has to consider the way both languages operate, and make sure the discrepancies between the language models are accounted for, such as manual memory management. This is not different from a visual language interfacing with a web-compiled native library. However, this situation adds additional challenges, as the host VPL does not necessarily know the source language of the binary.

Utilization

Finally, It might be the case that a web-based VPL *is* able to load and run functions from native, non-js geo-computation libraries. And still, one might not be able to successfully **use** these libraries.

Central to this challenge of utilization is the aforementioned dataflow VPL format. The performance and reliability needed for geocomputation makes a strong argument that any geocomputation VPL should be implemented as a dataflow VPL. This is also evident from the fact that almost all geocomputation and geometry manipulation VPLs are implemented as a dataflow VPL (see Section 3.2).

However, it may be that native geocomputation functionalities are unfit for this model. The model requires all functions to be pure, and all variables to be immutable. This might invalidate some of the operations found in these libraries, or make other operations complicated to use in practice.

1.1 Research Objective

This study seeks to improve the state of browser-based geocomputation VPLs. This is done by studying a possible solution to the library portability problem found in existing browser-based geocomputation VPLs.

The study poses that the library portability problem can only be overcome if native geocomputation libraries are able to be properly compiled, loaded and utilized in a browser-based dataflow-VPL format. There are multitude of technical challenges present within each one of these steps, and in between these steps. Seeking and analyzing solutions to these challenges is the focal point of this study.

1.2 Research Questions

Based on this objective, the research question is formulated as follows:

> *How can native geocomputation libraries be* compiled, loaded, *and* utilized *within a browser-based dataflow-VPL?*

Supporting Questions

The following supporting questions are defined to aid in answering the main question. The first question serves as a prerequisite for attempting to answer the remaining three. The other three research questions are based upon the three main categories of challenges of the library portability problem. These questions are posed in such a way that answering them will require us to explore the extent of this challenge, and find possible solutions.

- *How to implement a browser-based dataflow-vpl for processing 3D geometry?*
- *How can geocomputation libraries written in system-level languages be **compiled** for web consumption?*
- *To what extent can a web-consumable library be **loaded** into a web-vpl without explicit configuration?*
- *How can a 'geo-web-vpl' be **used** to create geodata pipelines?*

Evaluation

In order to obtain answers to these questions, a literature review is performed, a prototype web-based geocomputation VPL is developed, and this application is accessed on various aspects.

This prototype vpl is both used as a test case to discover the extent of possible challenges, as well as a staging ground for possible solutions. The specific way in which each sub-research question is evaluated is explained in Chapter 4.

1.3 Scope

The scope of this thesis is bounded in the following seven 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 Usability Comparison

This research attempts to solve the practical problem of library portability. We must therefore access if a library can be *used* in a VPL context. But, while accessibility and usability are motivations for this study, and while usability will be analyzed to a limited extent, no claims will be made that this method of geocomputation is *more* usable as opposed to existing geocomputation methods.

If it turns out that library portability is viable enough technically, future research will be needed to definitively proof *how* usable VPL usage of a library is compared to all other existing methods. Similarly, a survey analyzing how users experience browser-based geocomputation in comparison to native geocomputation is also not part of this particular study.

This distinction is made, because browser-based, VPL-based geocomputation is too new of a method to make a balanced comparison against any other methods. Native environments like QGIS, FME or ArcGIS simply have a twenty year lead in research and development.

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.

Mostly Point Cloud/ DTM focussed geocomputation

We are also required to concentrate the scope of 'geocomputation', which is a sizable phenomenon. The term is generally used to cover all operations on geodata, from rasters, tabular datasets, highly structured datasets such as the CityJSON or IndoorGML, and point clouds. Due to time limitations, we are forced to focus on particular type of geocomputation. 3D-based geocomputation is chosen, with a particular focus on pointclouds and DTMs. The hypothesis is that these types of data may fit the small-scale geocomputation of a VPL well due to the local optimality quality of many DTM procedures (Such as the Delaunay triangulation (DT)),

Assumption: a 'Geocomputation-VPL' is a Geometry VPL able to handle geodata

For the scope of this thesis, we assume a VPL for geocomputation is practically the same as a VPL for generic geometry processing. Examples of "Geometry VPLs" are Blender's Geometry Nodes, Houdini, and Grasshopper. More on this in chapter Section 3.2.

A Geocomputation VPL differs only in the fact that it supports additional geodata types, and offers functionalities specific to processing those types. This assumption is safe to make since a geocomputation VPL will *at the very least* require the same features as a 'normal' geometry VPL, due to their common dependency on the field of computer graphics and computational geometry.

Still, in reality, a lot of differences and nuances exist between the field of geocomputation and the field of procedural modelling. However, due to time limitations, these concerns are left to future research.

Only geocomputation libraries written in C++ & Rust

The study limits itself to native libraries written in C++ and Rust. C++ was chosen, since almost all relevant geocomputation libraries are written in C++, like CGAL and PROJ. 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 geocomputation libraries, but not to the same extent as C++.

Only core browser features

Lastly, the implementation of the prototype geocomputation VPL will limit itself to core browser features, keeping dependencies at a minimum, in an attempt to generalize the results of the study. If the study would use specific web frameworks and technologies to solve key issues, questions might arise if the results of the study counts in general for browser-based geocomputation or geocomputation VPLs, or if they only count in this specific scenario. "Core browser features" is defined in Section 4.1.

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:

- Geocomputation
- 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 the visual programming language, and WebAssembly.

2.1 WebAssembly & 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 lead to web applications being used ’natively’. Tools like Electron [Contributors, 2022b] 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 [Contributors, 2022c] 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 [Contributors, 2022f]. 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 sees usage 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. These limitations can be split up into two groups: Limitations due to the web platform, and limitations due to the current state of the language and its host.

First of all, WebAssembly is required to adhere to the same containerization restrictions as javascript and the web at large. 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 simple, bare-bones, low-level

compile target. For example, the current version does not support concurrency features like multithreading.

Many of these shortcomings 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. criticize the methodology of these benchmarks, stating that only small scale, scientific operations where 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. 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 2.1, 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 performed extensive benchmarks on polygon simplification algorithms written in both javascript and WebAssembly [Melch, 2019]. It concludes by showing WebAssembly was not always faster, but considerably more consistent. Melch 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. must be mentioned [Sandhu et al., 2018]. 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

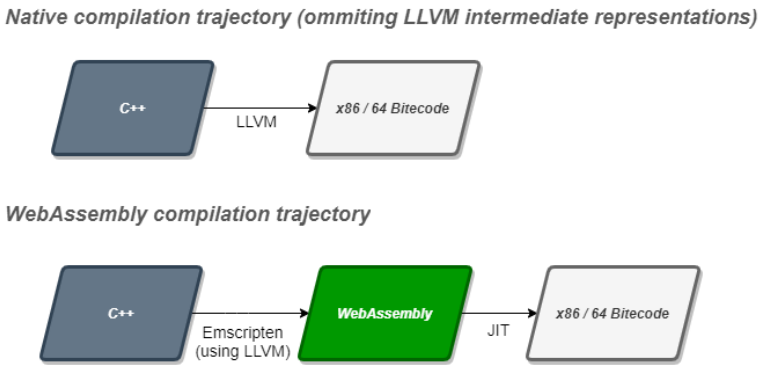


Figure 2.1: Comparison of compilation trajectories

to 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.

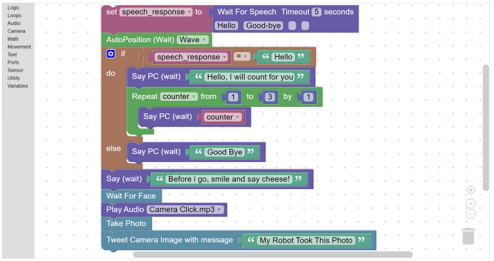
Multiple types of VPLs exist, but also multiple taxonomies of these types. This study bases itself on the classifications presented in Kuhail et al. [2021], stating four different types of visual programming languages:

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 2.2a).
2. **Diagram-based languages**, in which programming function are represented by nodes, and variables are represented by edges between these components (see Figure 2.2b). 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 2.2c). 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 2.2d).

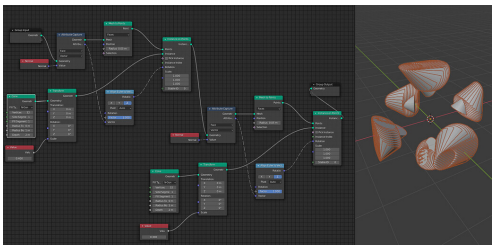
The meta analysis of Kuhail et al. [2021] shows a great preference among researchers for block- and diagram-based languages. Only 4 out of 30 of the analyzed articles chose a form-based vpl, and only 2 chose an icon-based approach.

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.

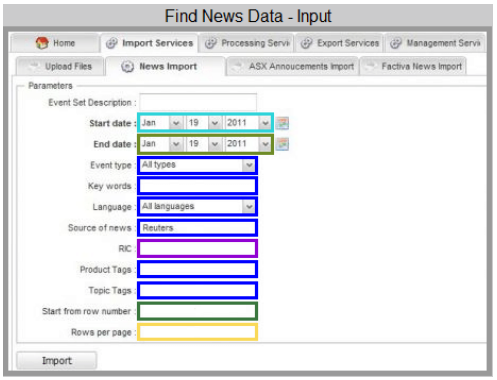
Visual programming languages are used in numerous domains. The vpls of the 30 studies examined by Kuhail et al. [2021] were aimed at domains such as the Internet of Things, robotics, mobile application development, and augmented reality. Within the domain of systems control and engineering, The Ladder Diagram vpl [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



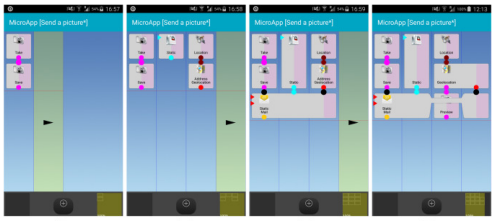
(a)



(b)



(c)



(a) (b) (c) (d)

(d)

Figure 2.2: Four different types of visual programming languages: Block-based [Resnick et al., 2009], diagram-based [Foundation and Contributers, 2022], form-based [Weber et al., 2013], and icon-based [Francese et al., 2017], respectively

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.

These properties together make visual programming also highly suitable for activities of **experimentation** and **debugability**, and not only for end users.

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 between applications, just like how UNIX-based programs can be composed into pipes Figure 2.3.

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 [ltd, 2021]. Despite this being just market speculation, it does give an indication in a general need for end-user development solutions.

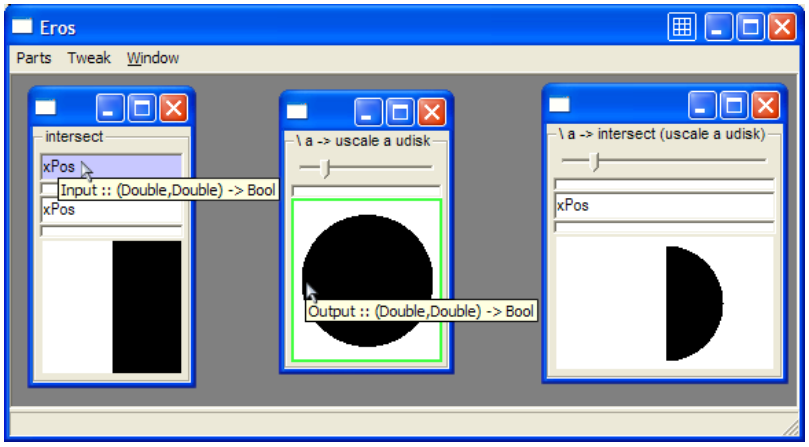


Figure 2.3: End User development by means of GUI algebra [Elliott, 2007]

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.

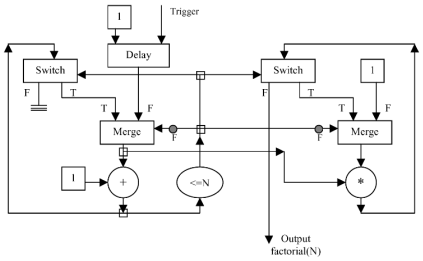
The big computational advantage of this model, is that it allows for implicit concurrently [Sousa, 2012]. In other words, every node of a program written using dataflow programming can be executed in isolation of any other nodes, as long as the direct dependencies (the inputs) are met. No global state 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 an interesting 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 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 can 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:



(a)

LOOP WITH i = N, fact = 1
NEW fact = fact ∗ i;
NEW i = i − 1;
WHILE i > 1;

(b)

Figure 2.4: A factorial function, written in a vpl, and in a textual language [Sousa, 2012]

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 2.4.

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 researches 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 [2022]). 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 Figure 3.1:

- 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 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]. The central idea is to add browser-based geocomputation to web-mapping applications, allowing users not only to view geodata, but to analyze it, and even fine-tune the data to their own custom needs. An example of this is the Omnibase application [Geodelta, 2022] in Figure 3.2, 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. The big catch is that 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++. 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.

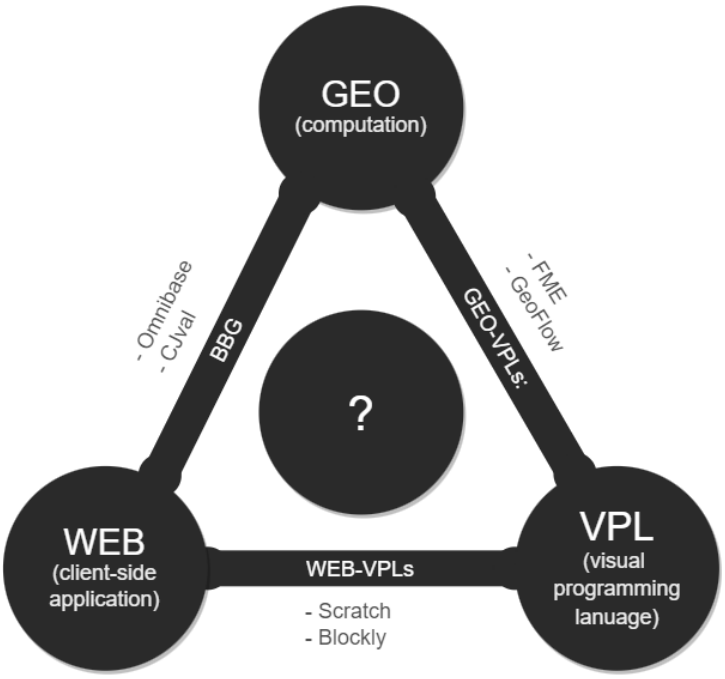


Figure 3.1: Triangle Model

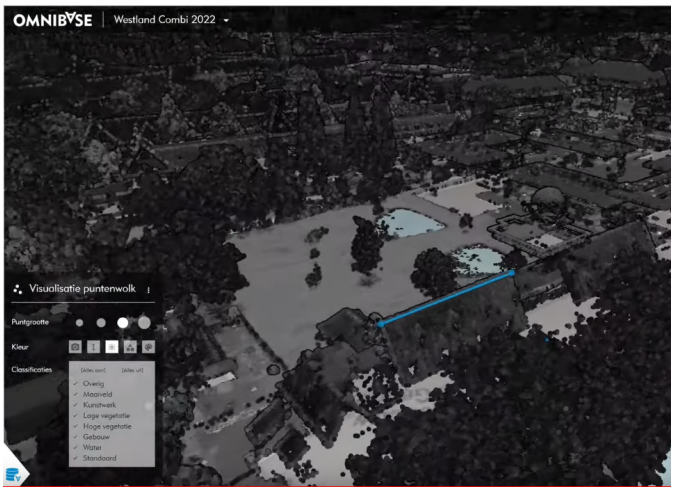


Figure 3.2: Omnibase: An example of browser-based geocomputation [Geodelta, 2022]

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. 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 compassion 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.

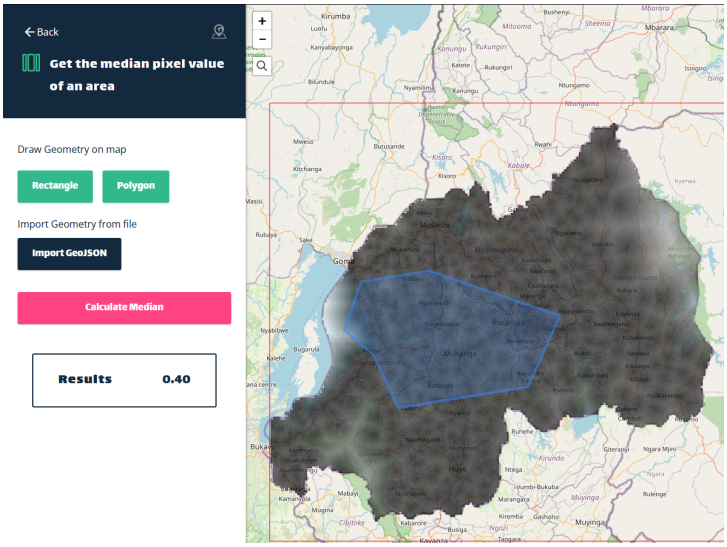


Figure 3.3: The geoTIFF.io application [Dufour, 2022]

3.1.2 Commercial web-based geocomputations software

Despite the earlier statement of the general lack of 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, 2022], Figure 3.3), 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, 2022]. This tool offers more advanced types of geocomputation, like buffering / minkowski sums, and even multi-stage processing via a simple but clear visual programming language (see Figure 3.4). 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 3.5). 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.

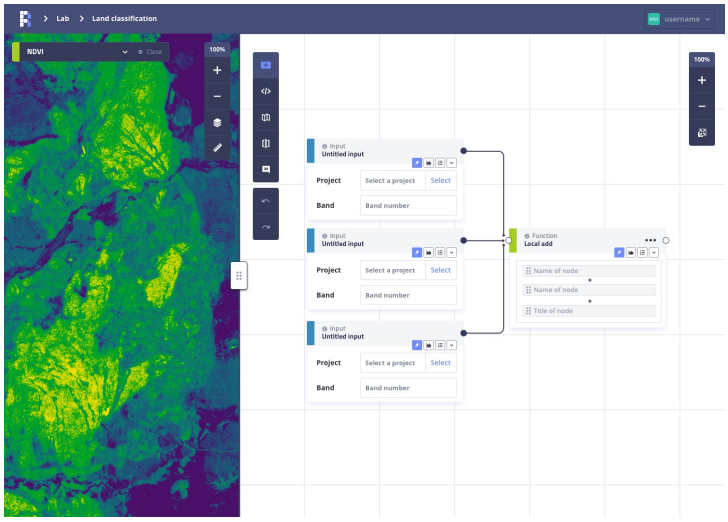


Figure 3.4: The ModelLab application [Azavea, 2022]

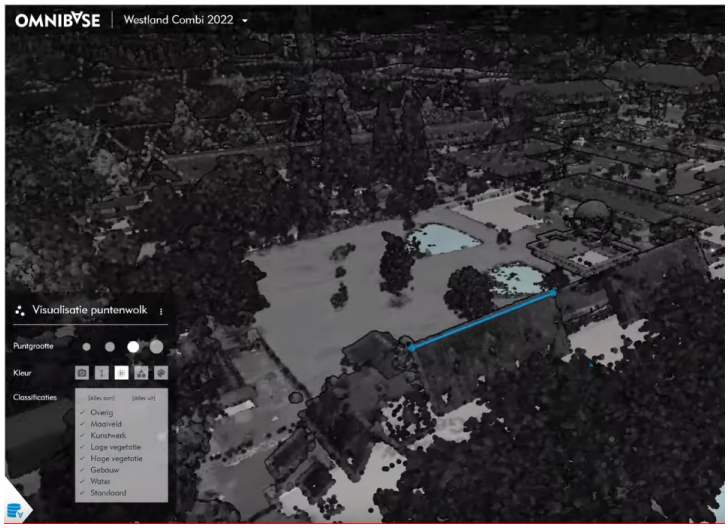


Figure 3.5: The Omnibase application [Geodelta, 2022]

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

3.2 Visual programming and geocomputation

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

Figure 3.6 offers this overview of some of the more significant VPLs present in not only GIS, but also the neighboring domains based on computer graphics.

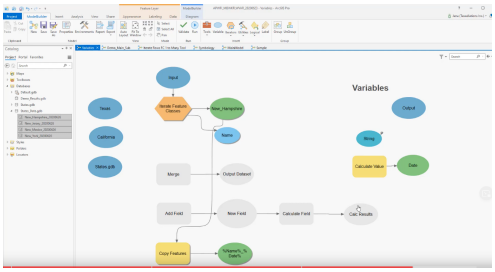
VPLs in GIS

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.

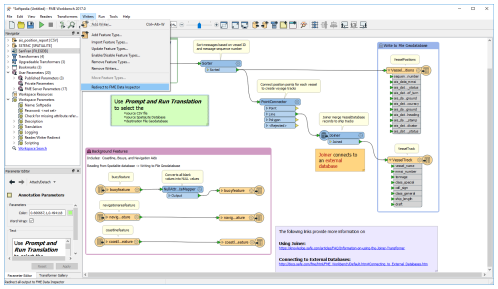
The most well-known visual programming language within the field of GIS is the commercial ETL tool FME [Safe-Software, 2022], (see Figure 3.7a). This tool is widely used by GIS professionals for extracting data from various sources, transforming data into a desired format, and then loading this data 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.

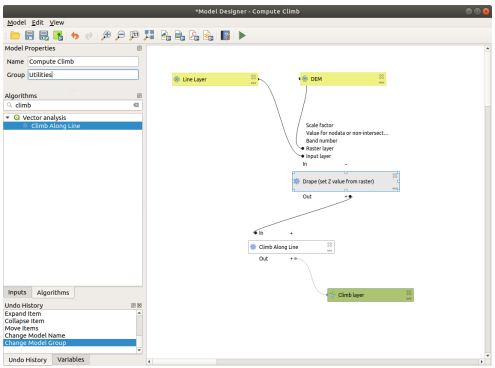
Lastly, Geoflow is a much newer VPL meant for generic 3D geodata processing [Peters, 2019]. While this application is still in an early phase, it already offers a powerful range of functions. It offers CGAL processes like alpha shape, triangulation and line simplification, as well as direct visualization 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].



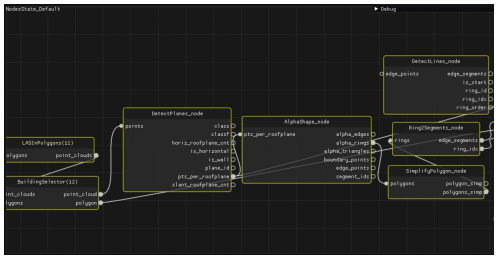
(a)



(b)



(c)



(d)

Figure 3.7: Four VPLs used in the field of GIS: ArcGIS's Model Builder [Esri, 2022] (a), Save Software's FME [Safe-Software, 2022] (b), QGIS's Graphical Modeler Community [2022] (c), and Geoflow [Peters, 2019] (d).

VPLs in neighboring domains

Figure 3.6 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 geo-computation.

The choices of which vpl to include in Figure 3.6 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 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.

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.

The study speculates that this might be the case, because all these vpls, with exception to the behavior vpls, are essentially dealing with "functional data pipelines". No distributed systems or event driven architectures, just one calculation from start to finish, to produce a desired product. 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, do not allow these pipelines to be configured by conventional UI's. However, a VPL does deliver these features.

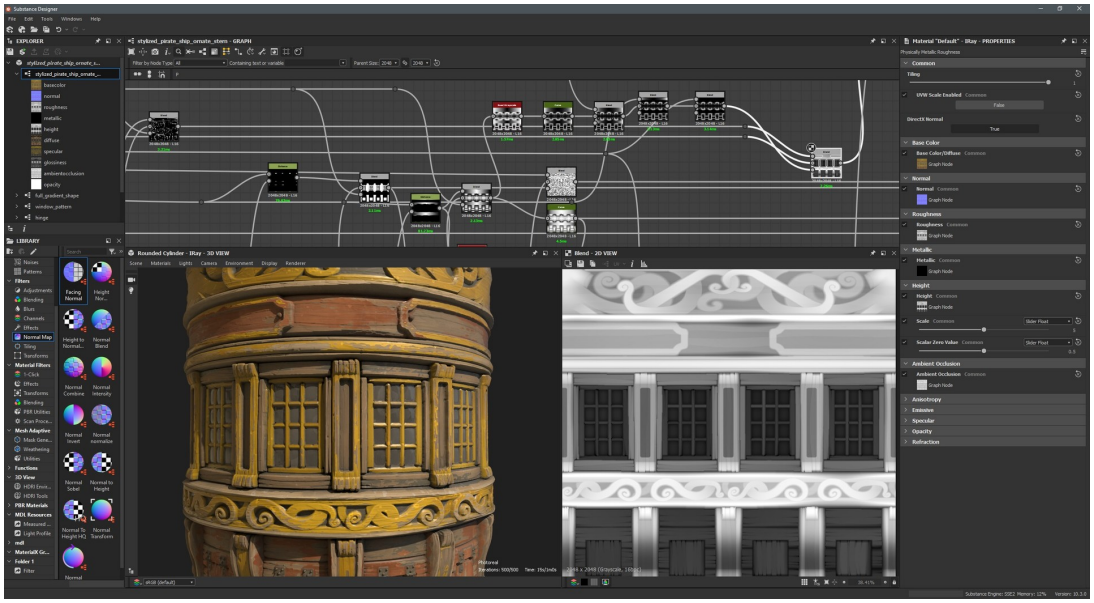


Figure 3.8: Substance designer, a VPL for textures [Rutten, 2012]

Material VPLs

By far the most commonplace type of vpl present in computer graphics are material Vpls. In this context, the concept "material" often refers to a combination of 2D textures and shaders. These 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.

Geometry, and photogrammetric VPLs

Procedural Geometry VPLs are not far behind the material VPLs in terms of popularity. Applications like Blender's geometry nodes [Foundation and Contributers, 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.

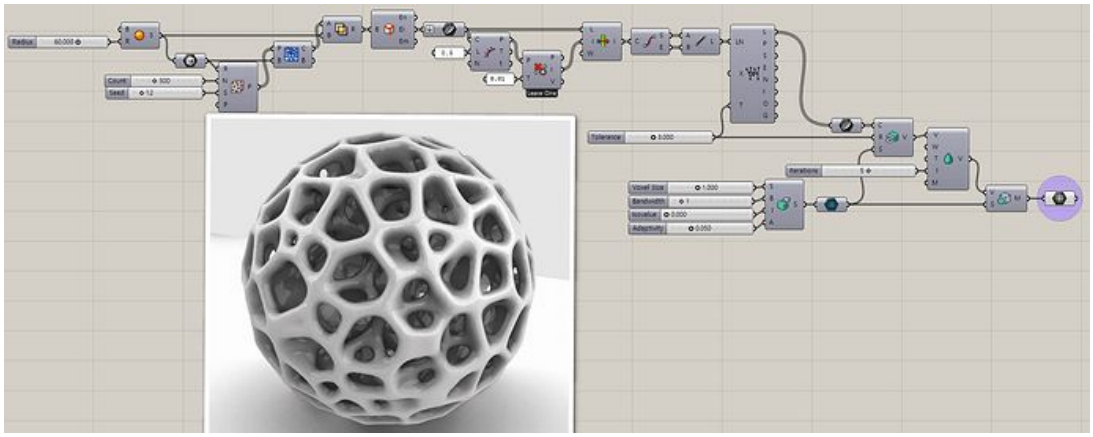


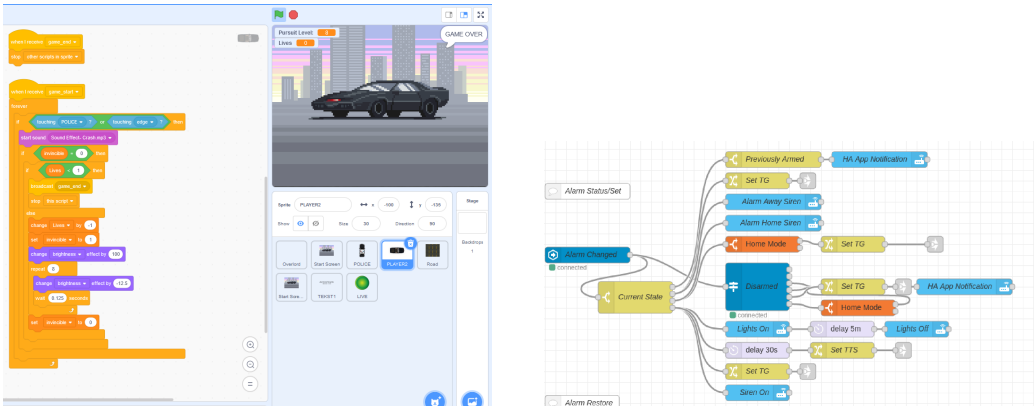
Figure 3.9: Grasshopper, a VPL for geometry [Rutten, 2012]

(a) (b)

Figure 3.10: Two VPLs for logic, showing "flow-state" variables. Left: Unreal's Blueprints, Right: Unity's Bolt. [Games, 2022; Technologies, 2021]

Behavioral VPLs

The behavioral and logical vpls found in applications such as Unreal's Blueprint [Games, 2022] and Unity's Bolt [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 3.10 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 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.



(a) (b)

Figure 3.11: Two VPLs used on the web: Scratch (a) [Resnick et al., 2009], and nodeRED (b) [Foundation, 2022].

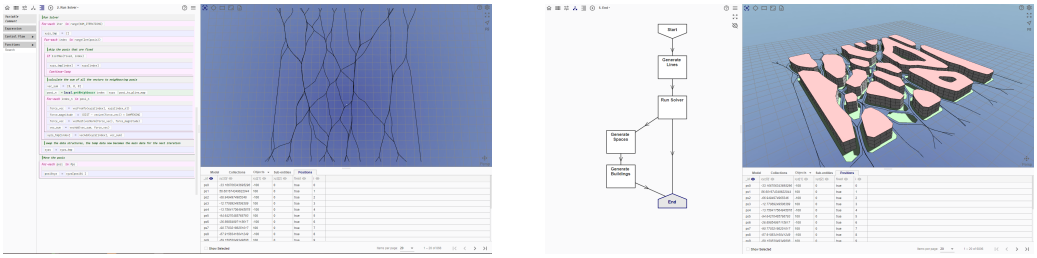
3.3 Browser-based visual programming

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++-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 3.11a). 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 [Foundation, 2022] (see Figure 3.11b). 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



(a) (b)

Figure 3.12: Images of the Mobius modeller application [Janssen, 2021]

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 3.12). 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.
- This study addresses some of the life-cycle issues of VPLs stated in Section 2.2.4.

4 Methodology

This chapter explains the methodology used in this study. The overall methodology is to first develop a base VPL (Section 4.1), followed up by developing a binding system for this VPL (Section 4.2). The final step is to set up and execute a series of tests (Section 4.3), to analyze the extent to which these implementations solve the various challenges raised by Chapter 1.

4.1 Base VPL

The first step of the methodology involves the question: *How to implement a browser-based dataflow-vpl for processing 3D geometry?*. This implementation is required as a host for all subsequent steps of the methodology. While the initial plan was to re-use an existing web VPL, the related works review of Chapter 3 showcased that no exiting browser-based geocomputation VPL would be an appropriate fit. The Mobius Modeller [Janssen, 2021] came closest, but the sizable nature of this project makes aligning its goals with the goals of this study challenging. Building a custom implementation would also allow more degrees of freedom, in terms of designing a VPL which takes hosting geocomputation libraries from multiple hosts into account from the start.

The following approach was deemed as the most fitting method for implementing this VPL. First, the requirements 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 layed out, which can be subsequently implemented.

4.1.1 Requirements

The requirements of a dataflow-VPL implementation can be subdivided in requirements 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 dataflow VPLs is the ability to inspect the geometry created in 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.

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.

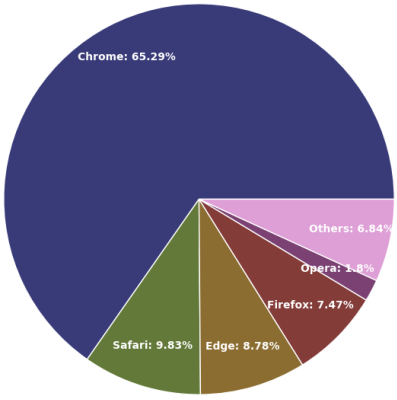


Figure 4.1: Fall 2021 desktop browser usage statistics. Data averaged over ([Dashiki, 2020; W3Counter, 2020; Team, 2020; Stats, 2020])

4.1.2 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 4.1, 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 fully covered yet)
- 2D Canvas API
- Web Workers
- Web Components
- WebAssembly

4.1.3 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-focussed applications, and allows us to reason about the model of the VPL language an a separate level from the editor / viewer. The JavaScript language will be used instead of webassembly alternatives, in order to limit

```
class TypeShim {

    traits: GFTypes[] = [];

    constructor(
        public name:  string, // human-readable name
        public type: JsType, // the actual type
        public readonly glyph?: string,  // how to visualize the type or variable in shortened form
        public readonly children?: TypeShim[], // sub-types to handle generics. a list will have an 'item' sub-variable for example
    ) {}

    // ...
}

class FunctionShim {

    constructor(
        public readonly name: string,      // human-readable name
        public path: string[] | undefined, // this explains where the function can be found in the Geofront menu tree.
        public readonly func: Function,    // the raw function this shim represents
        public readonly ins: TypeShim[], // input types
        public readonly outs: TypeShim[], // output types
        public readonly isMethod = false, // signal that this function is a method of the object type found at the first input type
    ) {}

    // ...
}
```

Figure 4.2: Shim classes. (pseudo) code is presented instead of a UML diagram, to be clear about the recursive aspects. (TODO: add modules)

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. Figure 4.2 shows a type-shim and function-shim respectively. A Shim equivalents of a module, and a VPL graph are also required.

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.

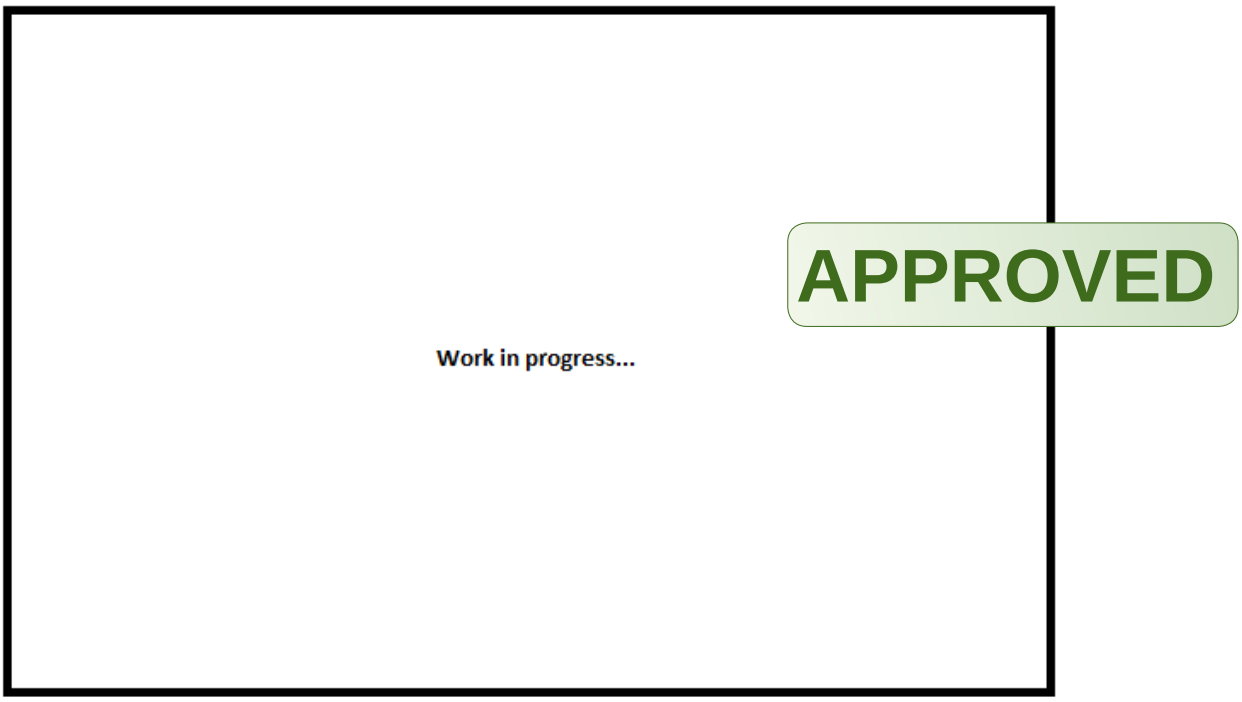


Figure 4.3: (TODO: MAKE A DAG UML)

Model

From the Shims, the main model of a VPL script can be conceived (see Figure 4.3). This model is at its core a DAG. This DAG should be a 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. 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 `Bridge` class, the model and controller can be separated, only allowing interaction with the model by serving this bridge Action objects. The `Bridge` 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

```
Step -1:
  Make an 'order' list
Step 0:
  Make a 'visisted' counter, initialized at 0
Step 1:
  Make a 'dependency' counter for each node, initialized at 0
Step 2:
  Add 1 to this counter for each input edge of this node.
Step 3:
   Fill a queue with all dependency 0 nodes.
   These are the starter nodes.
Step 4:
  Remove a node from the queue (Dequeue operation) and then:
  add the nodes' id to the 'order' list.
  Increment 'visisted' counter by 1.
  Decrease 'dependency' counter by 1 for all dependent nodes.
  If one 'dependency' counter reaches 0,
    add it to the queue.
Step 5:
  Repeat Step 4 until the queue is empty.
Step 6:
  If 'visisted' counter is not equal to the number of nodes,
    then the graph was degenerate, and probably cyclical.
```

Figure 4.4: Khan's algorithm in pseudo code

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 4.4):

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 [Contributors, 2022g]. 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.

4.2 Plugin System

The second step of the methodology involves developing a method to use libraries from native sources within the VPL outlined above. This aspect of the methodology seeks to design a base to answer the questions: *How can geocomputation libraries written in system-level languages be **compiled** for web consumption?*and *To what extent can a web-consumable library be **loaded** into a web-vpl without explicit configuration?*.

The plugin system refers to the combination of the plugin model these libraries will need to adhere to, together with the importer of those libraries in the VPL. This section will cover the design of both, as well as the proposed compilation workflow.

4.2.1 Terminology

Firstly, some confusion exists between 'libraries' and 'plugins'. The distinction between them is relevant to the design of the plugin system. Typically, one would use the terminology 'libraries' and 'bindings' when referring to expanding the functionality of a codebase with a 'package of functions'. One would use 'plugins' when referring to practically the same phenomena, but within the context of an application.

With a VPL being something in between a programming language and an application, it remains difficult to determine the appropriate terminology. This is further complicated if a plugin is able to double as a library, as will be explained.

4.2.2 Requirements

First and foremost, in order to use software libraries written in system-level languages in a web browser, these libraries will need to be compiled to the WebAssembly binary format (see Section 2.1.2). Secondly, in order to use these binaries on a VPL canvas, they must include some explanation of how to expose its inner functionality as VPL visual components. Thirdly, the core design goal for the plugin system is library portability. Portability in this context refers to: "usable in multiple locations". As such, the native geocomputation libraries we seek to support must not only be compiled to a format usable in a web-based VPL, but to a format usable on the web as a whole. And finally, the full workflow of bringing a geo-library to the web VPL should be as automated as possible.

4.2.3 Design

These requirements lead to the following design for the plugin system. Both the model of what a plugin should look like, and the plugin loader on the side of the VPL must be designed in conjunction, to make sure the models match. The plugin model:

- Must wrap a geocomputation library compiled with WebAssembly.
- Must include optional metadata about how the library may be used in the VPL.
- Must be packaged as a regular javascript library.
- Must be distributed using a Content Delivery Network (CDN) like the Node Package Manager (NPM).

Furthermore, the plugin loader of the VPL must be able to:

- accept a regular javascript library as a plugin.
- load exposed metadata about all included logic.
- convert this to an internal representation of a library.

This system utilizes the infrastructure of regular javascript libraries as much as possible. This way, existing javascript tooling, such as CDNs, can be utilized for distribution, updates and version control. Additionally, this setup addresses the portability requirement by making these libraries interoperable with unaffiliated projects on both the frontend and backend (see Figure 4.5). It might even allow libraries to be loaded which were never intended to be used by the VPL.

For the scope of this system, we will refer to a native geo-library wrapped with the appropriate VPL / javascript bindings as a 'plugin', even though these projects are javascript libraries, and can be loaded as a library into regular javascript projects.

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

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

What follows is an elaboration on the side of the plugin model, and on the side of the plugin loader.

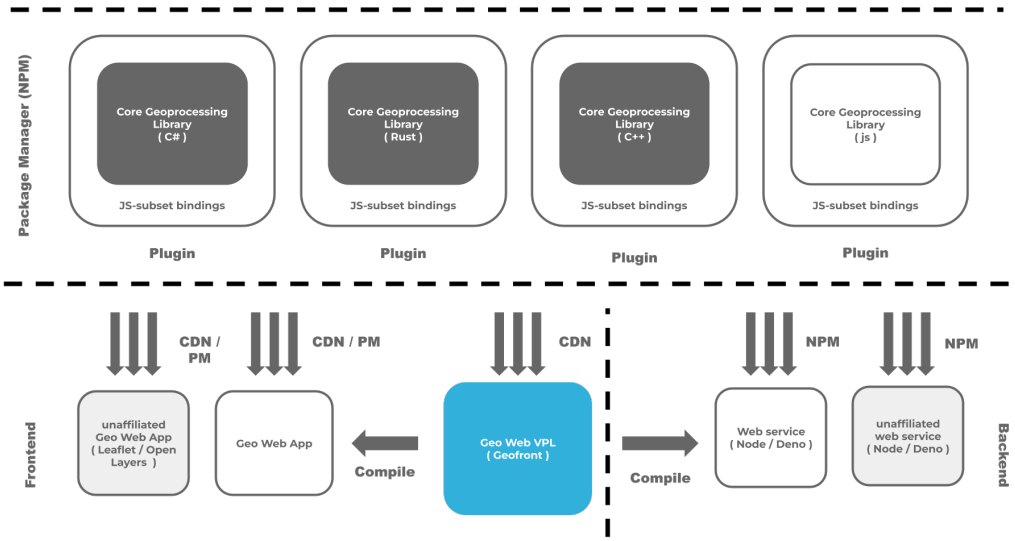


Figure 4.5: Plugin Design

4.2.4 Plugin Model

The plugin model serves three purposes:

Compilation

One, it needs to form a bridge between the language in which the geocomputation library is written, and javascript. In other words, the requirements of the WebAssembly compiler compatible with the language in question must be adhered to. 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 []. A minimum example of what a 'emscripten-ready-library' looks like can be seen in Figure 4.6.

For Rust, the 'wasm-pack' and 'wasm-bindgen' toolkits enable wasm compilation [Contributors, 2022d,e]. A minimum example of what this looks like can be seen in Figure 4.7. These toolkits do not require incremental compilation steps. What is needed, however, is to make sure all dependencies do not use incompatible features, like os file system access. Fortunately, most Rust libraries are written with 'no-std' use-cases in mind, and contain flags to easily exclude these features.

Wrapping

Two, it needs to bridge the gap between the dataflow-VPL and regular software library on a functional level. This comes down to wrapping the functionality of the geocomputation

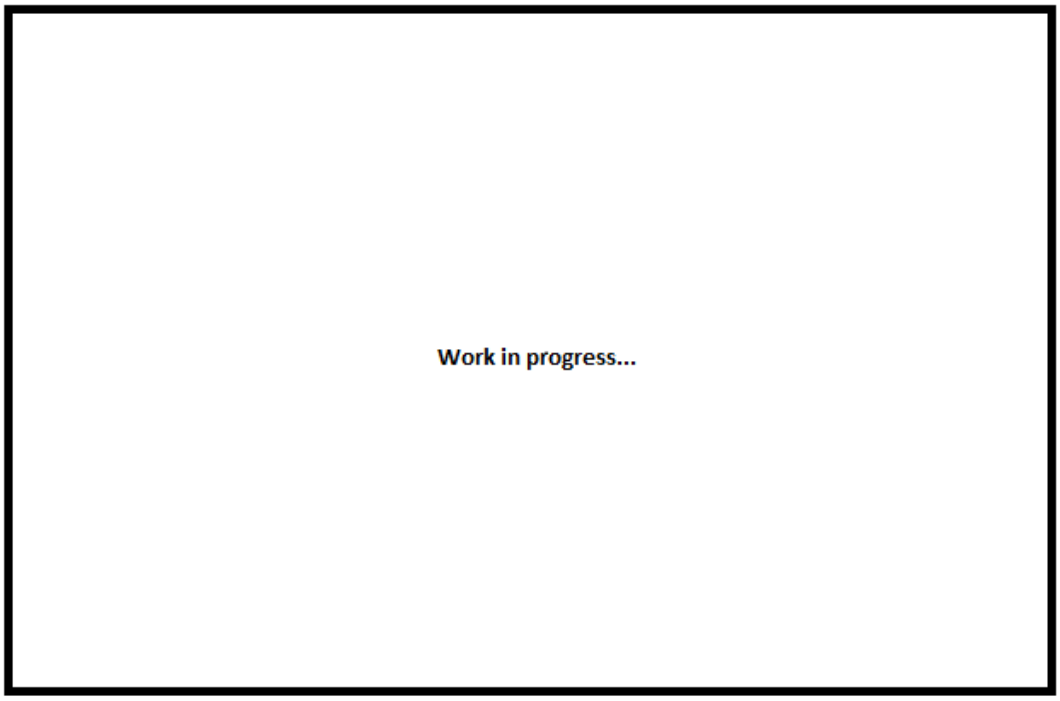


Figure 4.6: TODO: C++: Minimum WebAssembly example

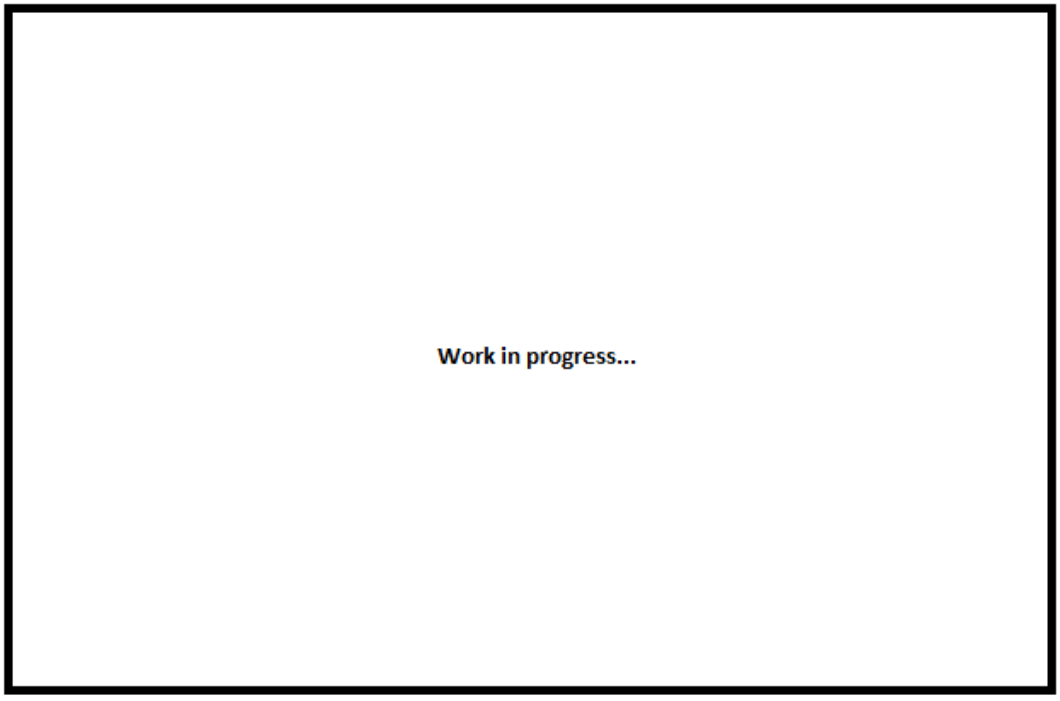


Figure 4.7: TODO: Rust: Minimum WebAssembly example

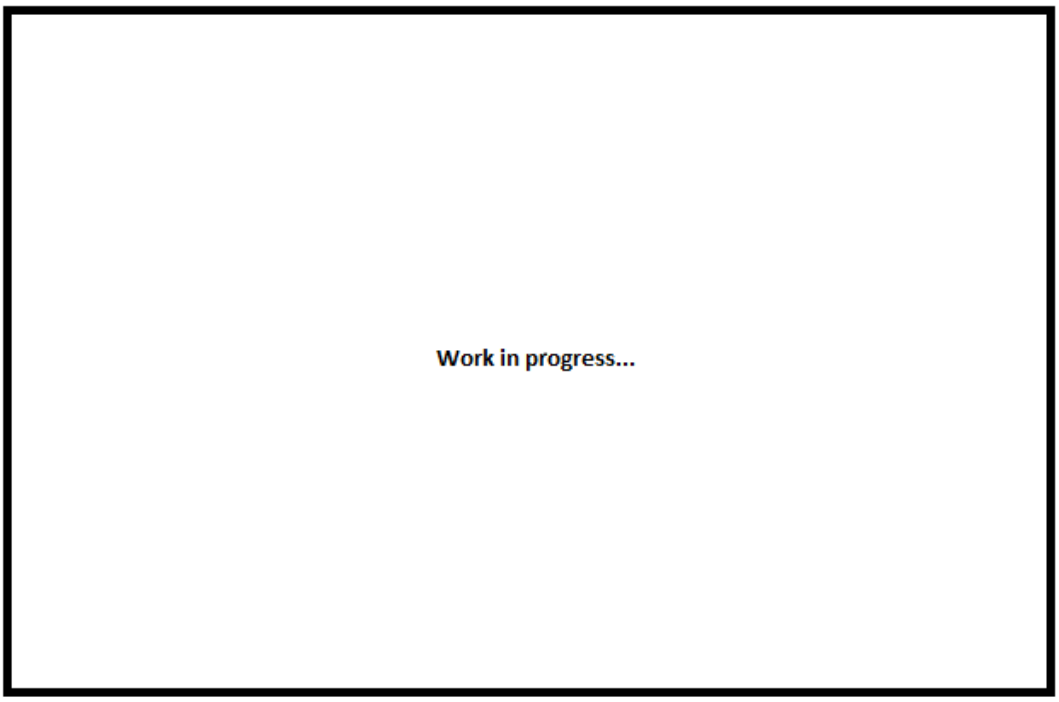


Figure 4.8: TODO: Rust: Exclude std features

library as pure functions. This often leads to making copies of inputs / outputs, or grouping a series of steps to make an imperative interface functional (See Figure 4.9)

Flagging

And Three, it 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 hardcoded within the plugin.

The following information is *required* for the VPL to load a geocomputation 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:

```
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 4.9: This typescript is written in a non-functional manner, troubling its conversion to the functional model used by a dataflow VPL

- 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.

4.3 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.

4.3.1 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 4.2 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 `emscriptem`. For one, this library is well established and very relevant to geoprocessing as a whole. Many 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 CGALs Triangulator. It also, just like CGAL, makes use of a high precision kernel, and offers geometric robustness.

4.3.2 Usage tests

The second set of tests seek the data needed to answer the question of Utilization: *How can a 'geo-web-vpl' be **used** to create geodata pipelines?*. To acquire this data, an application will be created using the VPL, which will be subjected to a qualitative assessment.

Feature Assessment

Per requirement, to what extent is it successfully implemented by this dataflow-VPL?. Per requirement, which role did the core browser features play in supporting or hindering it?.

Assessment Framework

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. This allows the assessment to be made within the scope of this study, and without performing user-testing. 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 age of the study indicates that the principles have stood the test of time.

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 it 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.

5 Implementation

This chapter presents the execution of the first and second step of the methodology. It discusses the achieved functionality, and mentions the ways this implementation fell short of the methodology.

5.1 The Geofront Application

This section discusses the extent of the prototype VPL implementation. The prototype is titled "Geofront", as a concatenation of "geometry" and "frontend". Geofront exists as a set of loosely coupled repositories, all published on the version control platform GitHub under the MIT license. These repositories are grouped under the GitHub Organization `thegeofront` .

The Geofront Application is implemented according to the design layed out in Section 4.1, 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 9.000 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 4.1.3 could be fully implemented on the web. Both the shims as well as the model itself was implemented in typescript, and no special web features were utilized in the creation, despite HashMaps and HashSets offered by the Typescript language. This model allows Geofront to internally represent the data structure and logic of a dataflow-VPL program. The programs constructed with the VPLs are dubbed 'scripts'.

Type safety was fully implemented by essentially creating a new 'layer' of types on top of typescript Figure 5.2. The type system can be extended by types found in Geofront 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.

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

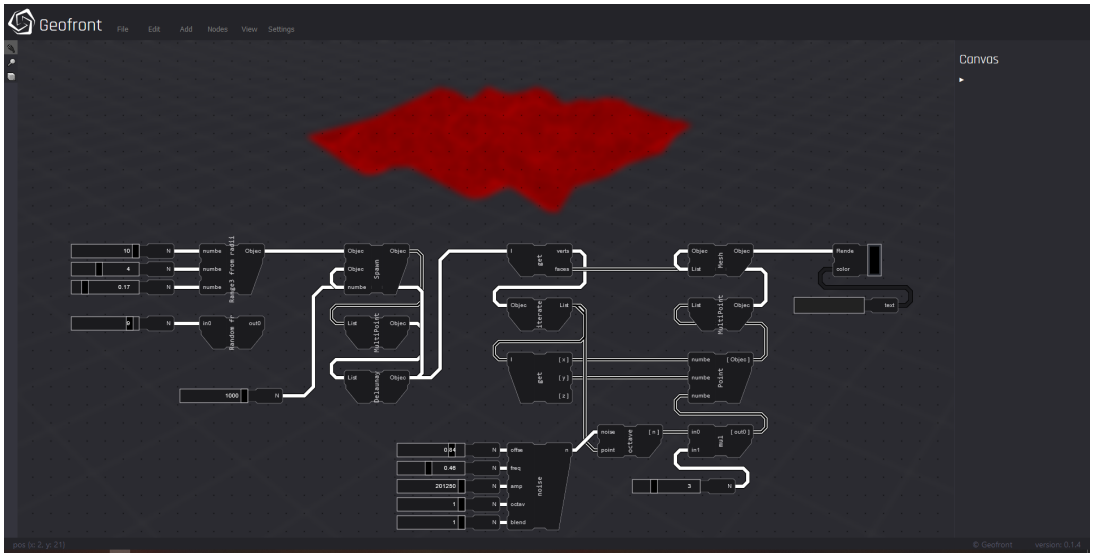


Figure 5.1: The Geofront Application

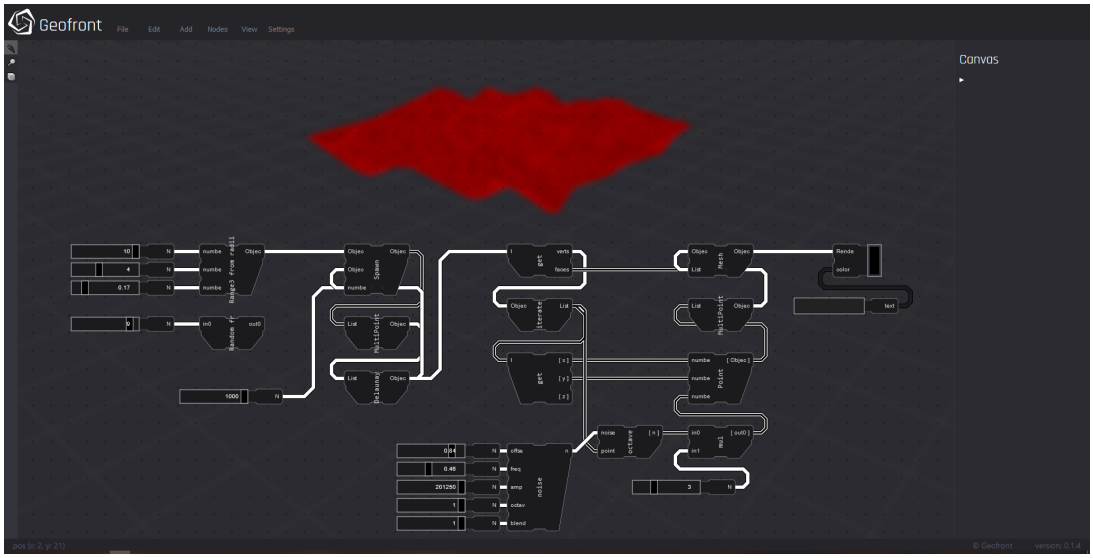


Figure 5.2: Geofront Types (TODO)

(a) (b) (c)

Figure 5.3: The basic canvas components of Geofront: a Node (A), a Cable (B), and multiple Widgets (C)

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.

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 performant than a WebGL & glsl 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 (See TODO).

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 5.4). 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

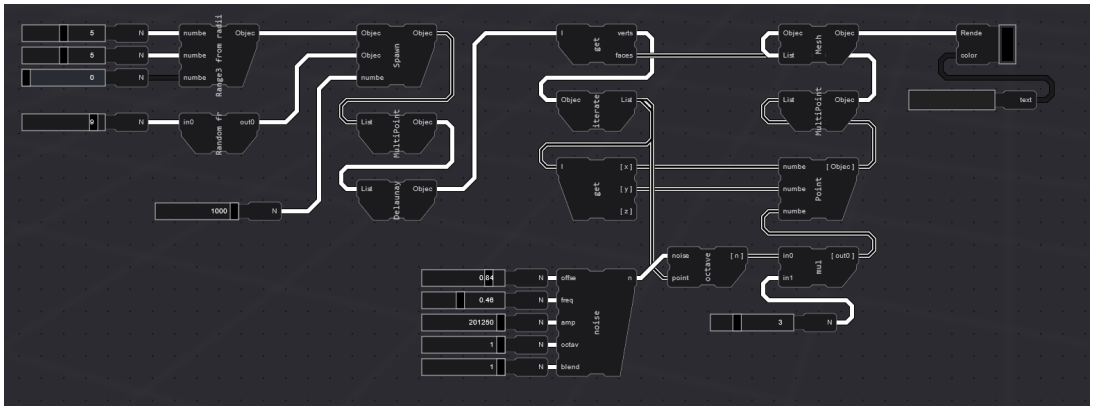


Figure 5.4: A complete Geofront script

monospaced fonts. Consistent sizing encourages organization and clarity, for this makes it easy for 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

Calculation

Execution of the graph is implemented as described by the methodology, albeit with a major setback: execution does not run on a separate thread. This means that the interface freezes up during large calculations.

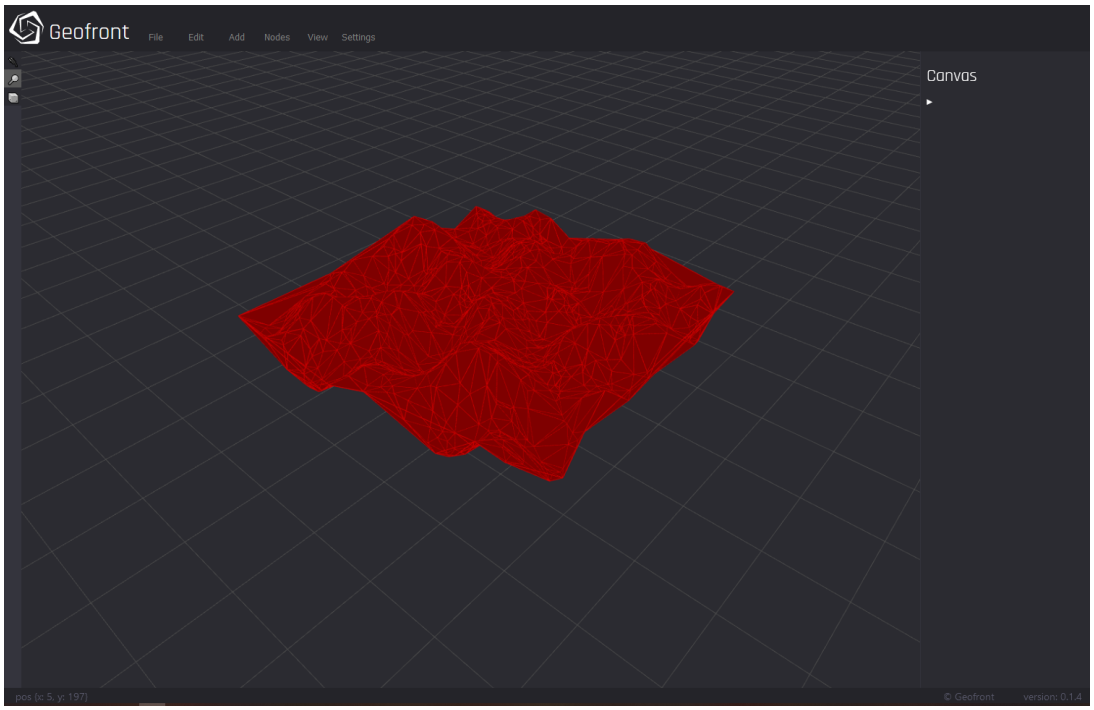
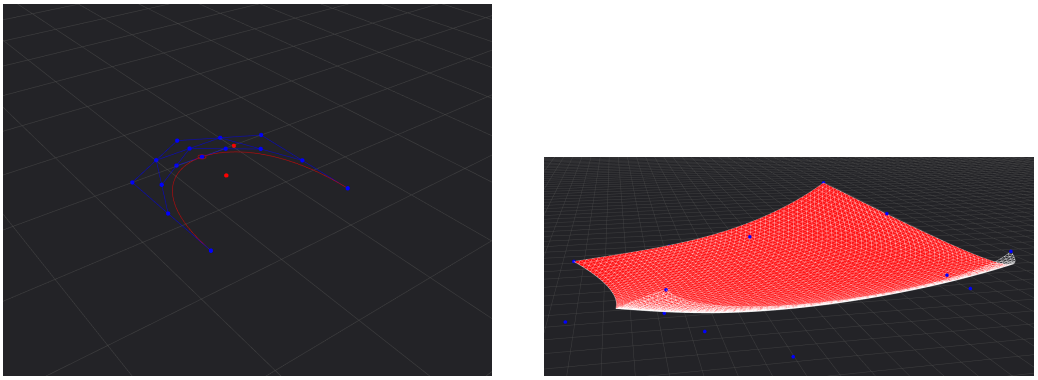


Figure 5.5: The Geofront Viewer



(a) (b)

Figure 5.6: A bezier curve and surface visualized using the geofront viewer

The reason for this is the difficulty in achieving concurrency in the browser. Web workers have to be implemented and instantiated using separate javascript files. Not only is this unusual, it forces a codebase to create separate files per multithreaded operation. The dynamic nature in which some of Geofront dependencies are loaded made splitting up the codebase like this difficult, let alone using multiple threads for calculating each node.

Additionally, both Typescript and WebAssembly do not make the mutability of variables explicit, nor was this implementation able to find a substitute method to add a mutability model on top of these languages. This made implementing the concurrency model explained in Section 4.1.3 impossible.

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.

Geofront's user interface strives to match 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 web applications more 'convoluted' to implement than would otherwise be ideal.

Input and Output

The base dataflow VPl 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 4.2. 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. Also bear in mind that the loader does not perform any checks to see if the loaded library actually uses pure functions.

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.

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

Using Rust:
1. Write or find a geoprocessing / analysis library using rust
2. Create a second library, which exposes a subset of this
   library as 'functions usable on the web', using 'wasm-pack'
   and 'wasm-bindgen'.
3. Compile to '.wasm' + 'd.ts' + '.js'.
```

```
4. publish to npm (‘wasm-pack publish‘), or use a local address.

Using C++:
1. Write or find a C++ based geoprocessing / analysis library.
2. Create a second library, which exposes a subset of this
   library as 'functions usable on the web', using 'emscripten'
   and 'embind'.
3. Compile to a ‘.wasm‘ and ‘.js‘ file using emscripten.
4. Create a third 'js' library, which wraps the functionality*
   exposed by empscripten
5. Manually create a corresponding ‘d.ts‘ file
6. Publish these to npm

_____________________________________________________

In Geofront:
A. Reference the CDN (content delivery network) address of this node package.
B. Use the library.
```

5.2.3 Automation and portability

The Section 4.2 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 considerations. 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 be able to be loaded, but is then functionality impaired by the interface.

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 second, third and fourth research question. It consists of the sections Section 6.1 and Section 6.2.

6.1 Plugin Compilation Tests

6.1.1 Rust: minimal plugin

The compilation of a minimum Rust plugin, exposing a function, a class, and a method was successful. Figure 6.1a Shows the source code of this plugin. The 'wasm-bindgen' library allows functions and classes to be annotated as 'bound to javascript'. This simplifies the compilation process to WebAssembly greatly. It also clearly states errors if a property or class are incompatible to be exposed.

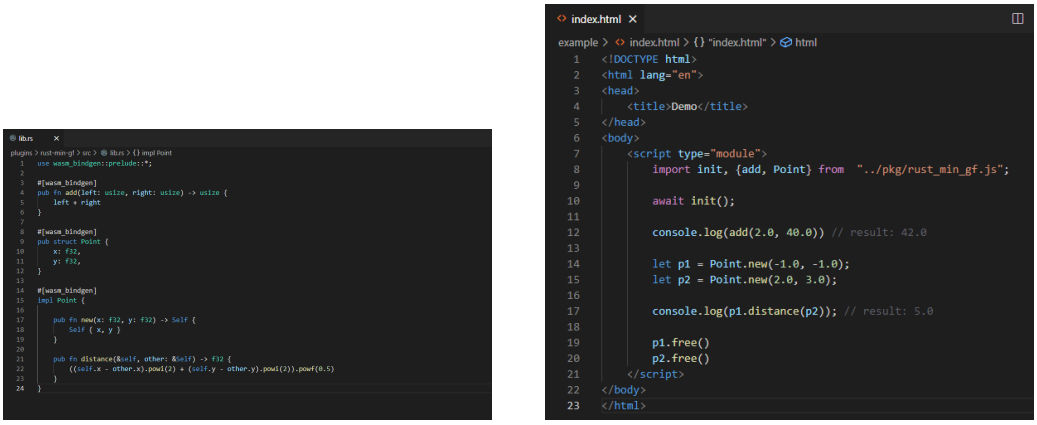
This library was compiled to WebAssembly and javascript using 'wasm-pack'. This produces multiple artifacts, showcased in Figure 6.2. This figure also showcases how wasm-pack wraps the functionality: The `point_distance` function exposed by the wasm file is wrapped by the javascript file, converting it to look like a regular javascript class.

To check if the result is valid, a small html demo was created to load and use the library (see Figure 6.1b). 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.

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 6.3. A local path was used for convenience, instead of publishing this demo to npm, and accessing it via a CDN.

Figure 6.4a shows how all functions in this demo are loaded correctly, and Figure 6.4b 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 6.2.

All in all, no problems were encountered.



(a) (b)

Figure 6.1: Rust Source code (a) and web demo (b)

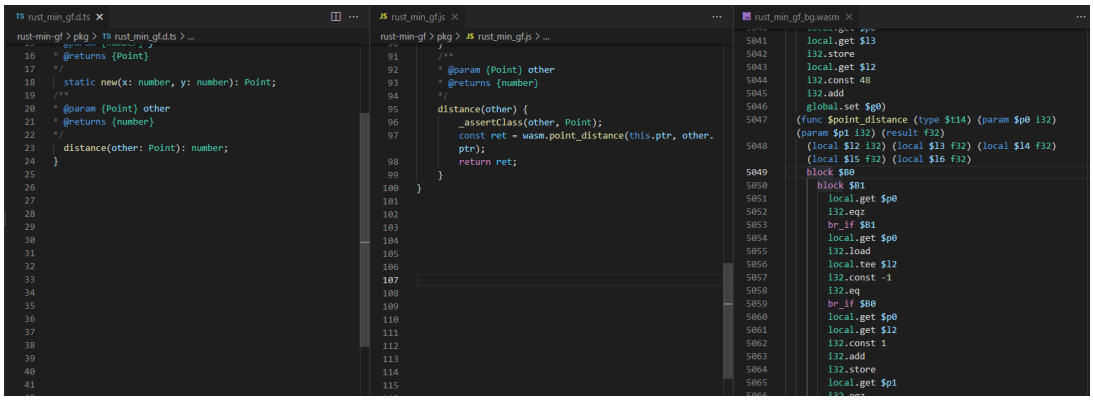


Figure 6.2: wasm-pack Compilation artifacts: A Typescript Declaration file, a javascript file, and a WebAssembly binary, visualized as WebAssembly Text (WAT).

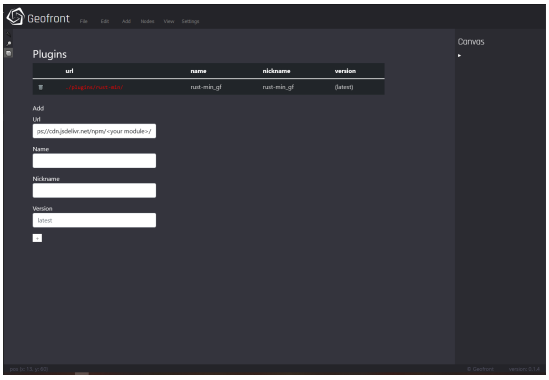


Figure 6.3: Loading a plugin into Geofront using the GUI

(a) (b)

Figure 6.4: Usage On the canvas

6.1.2 Rust: Startin plugin

Compilation of the `Startin` library was also successful. 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.

Other than that, all steps performed by the minimal Rust plugin could be made with this project as well, as shown in Figure 6.5. This also showcases the usage of the 'Renderable' bindings. This way, a variable of type 'Triangulation' can be viewed in 3D by clicking on it.

6.1.3 C++: Minimal plugin

Making a minimal C++ plugin work in a Geofront graph was unsuccessful. It was, however, possible to compile the plugin file to WebAssembly, and to use it in a web demo, but this was not without its obstacles.

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 6.7.

While it does work similar to Rusts `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 relatively unknown `-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 loaded into Geofront. It must be noted that emscripten was behind on documentation compared to 'wasm-pack' and the rust-wasm organization [Contributors, 2022e]. it does not offer many examples, or thorough explanations on what many of the compiler flags do or mean [Organization, 2022]. 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. So, while this does allow for a working web demo similar to the Rust equivalent, shown in Figure 6.6, this makes it hard

```rust
#[wasm_bindgen]
pub struct Triangulation {
    dt: startin::Triangulation,
}

#[wasm_bindgen]
impl Triangulation {

    pub fn new_from_vec(pts: Vec<f64>) -> Triangulation {
        let mut tri = Triangulation::new();
        tri.insert(pts);
        tri
    }

    pub fn new() -> Triangulation {
        let dt = startin::Triangulation::new();
        Triangulation { dt }
    }

    pub fn insert(&mut self, pts: Vec<f64>) {
        const STRIDE: usize = 3;
        for i in (0..pts.len()).step_by(STRIDE) {
            self.insert_one_pt(pts[i], pts[i+1], pts[i+2]);
        }
    }

    // ...
```

```rust
// impl Renderable for Triangulation
#[wasm_bindgen]
impl Triangulation {

    pub fn gf_has_trait_renderable() -> bool {
        true
    }

    pub fn gf_get_shader_type() -> GeoShaderType {
        GeoShaderType::Mesh
    }

    pub fn gf_get_buffers(&self) -> JsValue {
        let buffer = MeshBuffer {
            verts: self.all_vertices(),
            cells: self.all_triangles(),
        };
        serde_wasm_bindgen::to_value(&buffer).unwrap()
    }
}
```

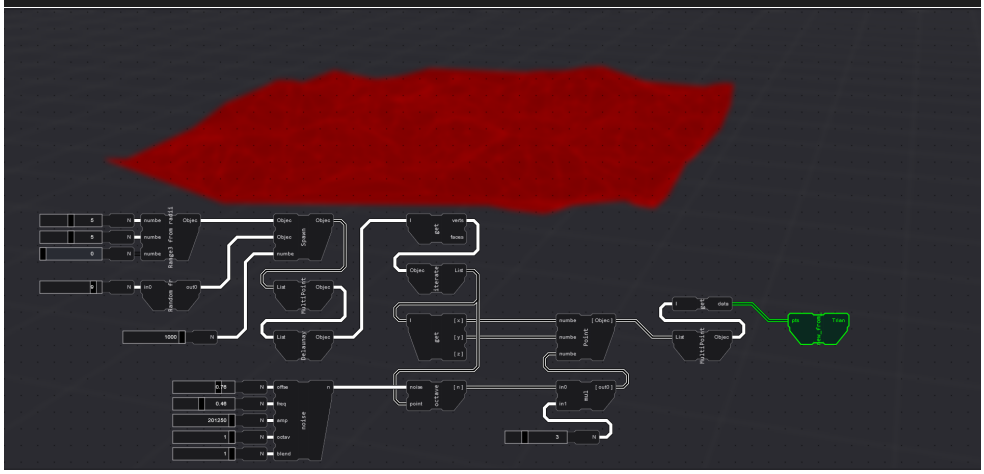


Figure 6.5: Startin, loaded as plugin within Geofront

```html
<!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 6.6: Cpp-wasm web demo

to determine the content of both files programmatically. This was partially solved 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 add static information about the functions and function types as strings in the CPP file, but this would have been a too manual of a solution to a problem which should be able to be solved automatically, as 'wasm-pack' did.

Finally, 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 in a regular, modern fashion. 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 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.

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.

lib.cpp M

plugins > cpp-min-gf > src > lib.cpp

```cpp
// quick_example.cpp
#include <emscripten/bind.h>
#include <cmath>

using namespace emscripten;

float add(float left, float right) {
    return left + right;
}

class Point {
public:
    double x;
    double y;

    Point(double x, double y) :
        x(x),
        y(y) {}

    double distance(Point& other) {
        return std::pow(
            std::pow(x - other.x, 2) + std::pow(y - other.y, 2),
            0.5);
    }
};

EMSCRIPTEN_BINDINGS(cpp_min) {
    function("add", &add);
    class_<Point>("Point")
        .constructor<double, double>()
        .function("distance", &Point::distance)
        .property("x", &Point::x)
        .property("y", &Point::y);
}
```

Figure 6.7: Cpp Plugin Source file

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

Figure 6.8: Emscripten JavaScript wrapper blacklist

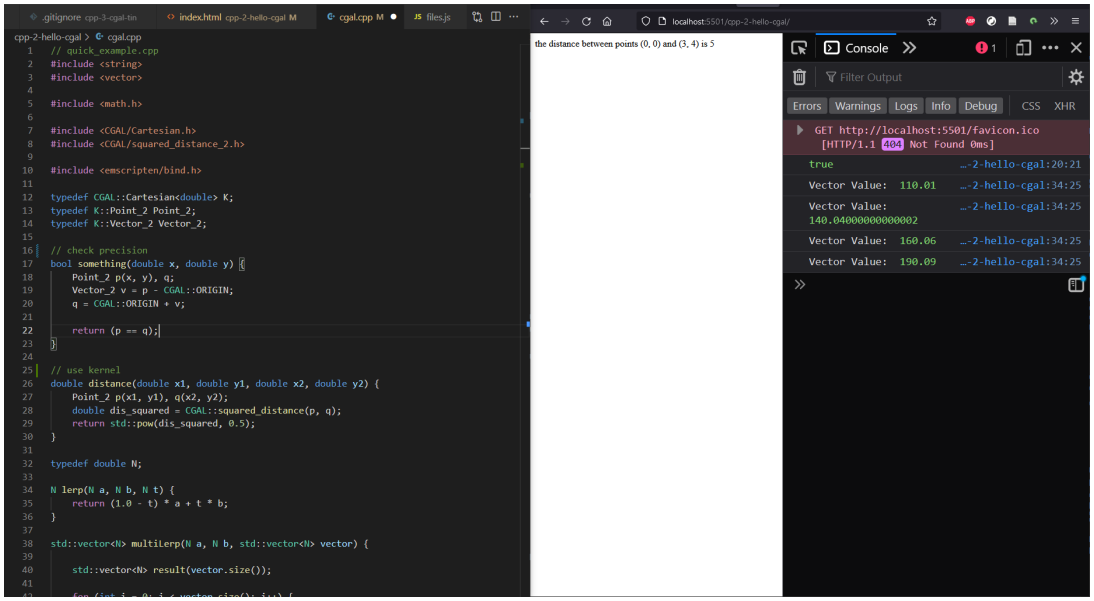


Figure 6.9: CGAL Kernel Demo

A web demo was able to utilize the CGAL kernel Figure 6.9, for basic operations. Additionally, A subsequent web demo can utilize the CGAL TIN to a limited extent Figure 6.10.

However, this study too could not be fully completed within the time frame of this study. Two problems prevented completion:

The first one has to do with rewriting inputs and outputs to CGAL functionality. The most common ways to provide CGAL functions with data, and to retrieve results, is to read and write files. While this can be used on the web, the virtual file system wrappers presented by Emscripten add irregular syntax to the plugin, again compromising any chance it can be loaded into Geofront. So, a way is needed to present CGAL with data directly from javascript or other wasm binaries, without reading or writing files. Initial tests were performed by parsing input data as a string buffer, which could then be 'read' like a file by CGAL.

The second issue with this process is to make 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.

but you can pass vectors of Point2 in the constructor. why doesn't it work?

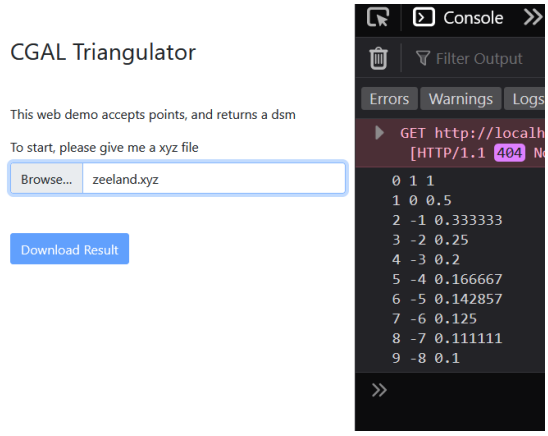


Figure 6.10: CGAL Triangulation Demo

6.1.5 Comparison

Size

Combined size of all compiler artifacts produced by wasm-back / emscripten:

```
wasm-pack:
rust min:        20140 bytes
startin:        114263 bytes

emscripten:
cpp min:         68424 bytes (3.4 times more)
cgal tin test   257994 bytes (2.3 times more)
```

It is safe to say that the emscripten wrappers are significantly heavier, especially given the fact that the wasm-pack artifacts include typescript header files and were not optimized for size, while the cpp builds where.

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

The performance benchmarks. These benchmarks primarily test how performant the javascript - WebAssembly interactions are.

```
100.000 iterations:

init:
rust min:        59 ms +/-  8 * SD
cpp  min:        71 ms +/-  6 * SD
js   min:         0 ms

process:
rust min:       131 ms +/-  5 * SD
cpp  min:       967 ms +/- 30 * SD
js   min:        22 ms +/-  2 * SD
```

The scripts run are shown in Figure 6.11.

These results also clearly show the emscripten wrapper is not able to compete with the wasm-pack solution. From experimentation, performance hit could be narrowed down to 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 compiled with emscripten only needs to address javascript through emscripten own interface, which could be more performant. A full application seldom needs access to specific javascript processes the way this demo does.

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 Usability Tests

6.2.1 Assessment

This section offers an analysis on the usability 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?

Geofront was meant to support encapsulation. The need for re-using parts of a script as components / functions was deemed more important than the benefits of having no abstraction hierarchy (what you see is what you get). An early prototype of Geofront did allow for encapsulation, by taking a subset of a Geofront script, and compiling it to a javascript subset. This could then be loaded by the library loader. However, the addition of special types of nodes, and features like iteration, invalidated the `geofront -> js` translator. The translation

(a) (b)

(c)

Figure 6.11: Rust vs C++ vs js performance benchmarks

is still possible, just not implemented. As such, if a user desires re-usable components and a lower abstraction level, they will need to write a Geofront library.

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 6.12.

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.

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 'guessability'.

Geofront has introduced certain 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 'outside world' 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 more predictable.

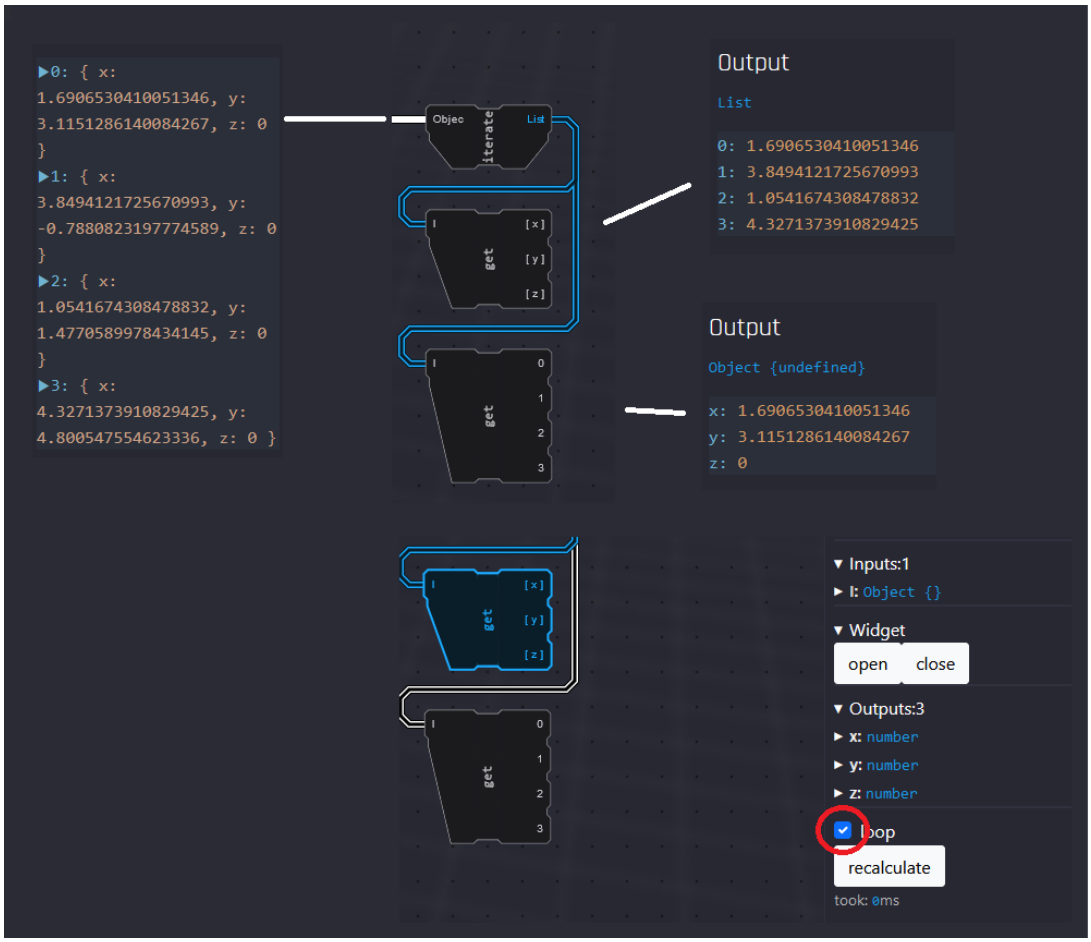


Figure 6.12: An example of a 'programming game' within Geofront. The 'loop' toggle indicates if an component should operate on a full list, or operate on all individual items within the list. This can be used to iterate over a 'list of objects' column-wise, or row-wise. This is considered a 'game', since this is not an obvious interaction, and must be learned before properly using Geofront.

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 struct, 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 disallows 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 off, 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 is a core aspect in how Geofront achieves its interactivity and debugability, together with its ability to inspect parameters.

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 it 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.

7 Conclusion & Discussion

This chapter contains the conclusion of this 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), a reflection on the value and quality of this study (Section 7.6), and, lastly, a personal reflection (Section 7.7)

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. *"How to implement a browser-based dataflow-vpl for processing 3D geometry?"*

The full answer of this question is represented by Section 4.1 and Section 5.1. Overall, the browser appears to be capable of representing a dataflow-type vpl graph to an acceptable degree, based on the implementation presented in Section 5.1. The browsers biggest advantage for an application like this is the amount of features a javascript program can use by default, like the 2D canvas api, the DOM, and WebGL. These features do not need to be included within the source code of the application, leading to quick load times. All three of these features proved to be vital, and were performant enough to support an application like this. Only the 2D canvas Api can become slow when rendering a great number of components.

TypeScript (JavaScript) was used to represent the data structure and logic needed to make a VPL functional. Javascript's flexibility proved to be useful to support features like dynamically loading and using libraries at runtime. However, TypeScript's limited type support at runtime, the absence of explicit immutability, and the limited precision in general (no integer atomic, only `number`), all hindered the implementation of the VPL model proposed.

2. *"How can geocomputation libraries written in system-level languages be **compiled** for web consumption?"*

Based on the experiments and analysis in Section 6.1, the study concludes that most contemporary, C++-based geocomputation libraries cannot be sufficiently compiled for web consumption, at least not for the purposes of loading the functionalities within a web-VPL. This is not due to wasm itself, but rather because of the focus of the emscripten compiler, combined with the implementation requirements of a web VPL. Emscripten can be used to compile full-scale C++ applications, and offers an emulation of a POSIX environment. However, it

lacks support for compiling libraries themselves, compared to other wasm-library compilers. libraries generated with emscripten's 'embind' tool use irregular syntax, which troubles its ability to be loaded programmatically. While web implementations do exist like 'GDAL-js', these solutions are required to work though Web Workers, and use the emscripten virtual file systems, which again compromises their usage for the purpose of a dataflow-type vpl requiring pure functions. Finally, the study was able to recognize some discrepancies between the novelty of the WebAssembly format, juxtaposed to 50-year-old legacy of the C++ language, leading to large wasm binaries.

Despite this, the study was able to provide a solution to these compilation shortcomings by expanding the range of 'system-level languages' beyond C++. The Rust programming language offers a performance and level of control similar to C++, and has better wasm-library support thanks to the `wasm-bindgen` toolkit. Using this toolkit, the study could successfully expose a native geocomputation library in a manner properly consumable by a web-vpl. regrettably, not many rust-based geocomputation libraries are written in pure rust, and the general pool of existing geocomputation libraries is limited due to the novelty of the language.

Thus, the conclusion is a dilemma between Rust and C++. C++ has a strong foundation of existing geocomputation libraries compared to newer languages like Rust. However, this same legacy inhibits its portability, which makes it harder to compile to web browsers. Rust is for the forseeable future a better choice for writing easily consumable, portable libraries, but does not have a fully mature ecosystem of geocomputation libraries yet.

To offer a solution, the study suggests that either the 'embind' tool must be expanded to the level of functionality of 'wasm-bindgen', or geocomputation libraries must be rewritten in Rust. This second option seems counterproductive, but as stated by Ammann et al. [2022]: "innovation often requires selectively ignoring prior work."

3. *"To what extent can a web-consumable library be **loaded** into a web-vpl without explicit configuration?"*

Based on the method described in Section 4.2 and Section 5.2, and the analysis of Section 6.1, it can be concluded that it is possible and even sufficiently usable to load a web-library into a VPL without explicit configuration. It also had the desired effect of breaking down the barrier between vpl libraries and regular text-based libraries: Using this method, only one type of library is needed to serve both. Moreover, it led to a workflow in which rapid experimentation was possible, since this method allows users to develop a library locally, and then quickly experiment and test its usage online.

The drawback of allowing this seamless interoperability and rapid experimentation, is that many important properties like descriptions and library metadata do not need to be explicitly specified, and could not be automatically extracted. These properties still had to be added to the libraries in the shape of methods with a recognizable naming convention.

Additionally, the freedom of granted by not restricting input and output types can lead to a confusing user experience, since there is no way of restricting libraries to use particular type convention. Even worse, the libraries could use references pointing to the same object, eliminating the 'immutable, no side effects' nature of a dataflow-type VPL.

4. *"How can a 'geo-web-vpl' be **used** to create geodata pipelines?"*

Based on the analysis of Geofront in Section 6.2, it can be concluded that a geo-web-VPL can be used for geocomputation to a sufficient extent. The analysis shows that many of Geofront's best aspects for the purpose of geocomputation are a consequence of the design decision to use a diagram-based, dataflow-type VPL. Examples of these are how the Functional programming paradigm leads to pure functions and immutable variables, making the graph as a whole behave in a predicable manner, allowing for the inspection of in-between data at runtime. However, the openness of the plugin system inhibits the consistency of these functional aspects. Imported libraries are not forced to exclusively use pure function. As a consequence, libraries can create functions with many side effects, or they can use inconsistent input and output datatypes, ultimately leading to confusion for the end-user.

Main Question

> "*How can native geocomputation libraries be* compiled, loaded, *and* utilized *within a browser-based dataflow-VPL?*"

A VPL can support existing geocomputation libraries if and only if these libraries are able to be *compiled*, *loaded*, and *utilized* in a dataflow VPL format.

Using a new javascript implementation of an acyclic, graph-based VPL, the study was able to demonstrate how the web platform can be used to represent a dataflow-VPL capable of hosting these libraries. The dataflow-properties of a graph-based VPL like this also makes this libraries sufficiently *usable*, albeit with some well-known caveats of dataflow-VPLS, like the representation of conditionals and iteration.

The current methods of *compiling* existing C++ geocomputation libraries to the web turned out to be insufficient for the purposes of this study. This is due to emscripten's focus on compiling full C++ applications instead of libraries. Despite this, the study wás able to demonstrate how a novel method can be used to *compile* and *load* a Rust-library for usage in the VPL. While not many contemporary geocomputation libraries are written in Rust, the study offers this method to either offer emscripten contributors a blueprint of a desired workflow, or to offer geocomputation library contributors a powerful use-case for the Rust language.

All in all, this means that either if the Rust ecosystem gains more mature geocomputation libraries, or if Emscripten's capabilities improve, then the code portability problem & dataflow problem of existing web-based geocomputation VPLS can be overcome.

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 will 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 layed 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 6.2. 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: Geodata is almost always big data. Will this web environment be scalable to handle big datasets?

The sizeable nature of geodata is a component fundamental to geocomputation. However, scaling the application up to handle big data deviated too much from the core goal of this thesis to solve the problem of library portability, and had to be left to future work.

Still, to pose a solution, this study experimented with compiling a
full Geofront script to javascript. This can be regarded as a 'release' build of the Geocomputation pipeline: It would have allowed native CLI-execution using Deno [Contributors, 2022a], without any dependency or reference to Geofront itself. Only the libraries used within the pipeline would need to be referenced, and this could have been done using regular javascript import statements, and npm.

However, this experiment turned out to be a full-sized study in itself, and so had to be left to subsequent studies due to time constraints.

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 6.2, 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.

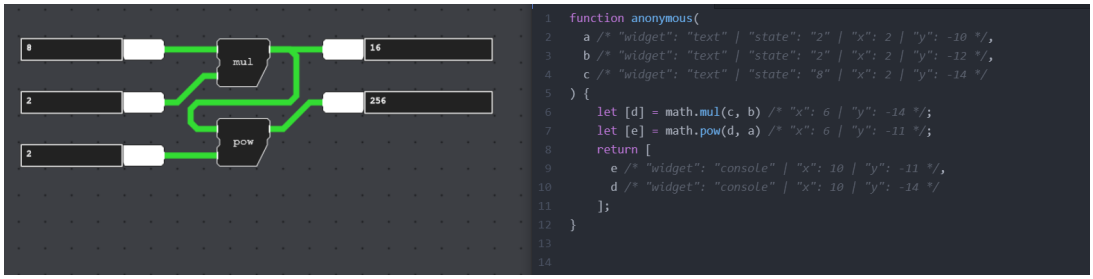


Figure 7.1: An early build of geofront, showing compilation to javascript

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 ot the study. Some of these aspects are listed here, and could lead to interesting topics for follow-up research.

7.5.1 Deployment & scalability

An early build of Geofront had the ability to compile a Geofront script to regular javascript (see (Figure 7.1)). 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 (without the GUI) either the browser or on a server, using a local javascript runtime like Deno [Contributors, 2022a].

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 [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 a Free and Open Source Software (FOSS) alternative to tools like the Google Earth Engine, and FME cloud compute.

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 used is not too large, this should be possible. Examples of this would be a streamed delaunay triangulation, 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 Rust for geocomputation. The reason for this was the extensive support for webassembly, which was essential for browser-based geocomputation. 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 very interesting 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 much more portable than it currently is, and Rust compiled to WebAssembly might proof to be a strong candidate for creating exchangeable, performant, compact, and error-proof binaries. It already sees usage on both cloud and edge servers (State of WebAssembly, 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 geocomputation less cumbersome. The study chose to pursuit this on a practical, technical level.

However, a more theoretical study could also be performed. It turns out that these ideas of 'less cumbersome geodata processing', 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]) map well to geocomputation: 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 could 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++. it could be that C++ was judged unfairly, 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 'hacky fixes'. 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 still did everything to make the judgement as non-biased as possible. Preliminary studies were conducted with both languages, and additional C++ courses were followed.

It could even be the case that this particular study is more fair than a study conducted by authors with more experience with C++, since before the assessment between Rust and C++, approximately the same amount of time was spent with both languages.

Scope too large

Additionally, the scope generated by combining geocomputation, web applications and vpls, might have been too extensive. This is evident in the number of 'supporting studies' conducted, and the sizable workload of the implementation. It might have been better to focus the scope of the thesis down to only 'browser-based geocomputation', or 'visual programming and geo-computation', or 'geocomputation using rust', to allow for a more in-depth analysis.

Then again, the core of the contribution of this thesis lies precisely in the attempt to connect these subjects, especially since prior studies remained by en large closely scoped to their respective domains. The hypothesis was that synergies exist, and that each separate domain stand to gain much 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, the field of GIS, like all research, is built on top of more foundational work which came before it. However, during the implementation of the study, it appeared that little foundation was in place for a geo-web-vpl specifically. This made it necessary to generalize, to build the missing foundation first. For example "How can *any* library be compiled and loaded into *any* web-vpl" is a question which had to be answered first. Then, the question could be specified to *geometry* and *geo-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.

Imbalance between software implementation and study

The fourth 'danger' which remained an ongoing balancing act during the execution of this study, is the balance between 'performing a study' and 'developing an application'. Indeed, many of the aspects discussed throughout this study come down to implementation aspects of the geofront application.

This is why the study has attempted to generalized its findings as much as possible. Geofront is regarded as a proxy of geo-web-VPLS in general, in the sense that whatever was encountered during implementation of the application, must be the same for any attempt at creating a web-based vpl for geocomputation.

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 vpl, and always showcasing clear examples.

7.7 Personal Reflection

A thesis is, among many things, an attempt to formulate. To clarify and structure a phenomenon, as much for yourself as for the public. This thesis can also be regarded as such an attempt. I tried to understand a phenomenon you might call: ”End User Geodata Processing”. To what extent can the activity of geocomputation truly be made ”publicly available”, besides writing open source software? I see this in much similar terms to the ”Teach a man how to fish...” saying. Providing open geodata to the public is great, but we might only be ”Giving a man a fish so he can eat for a day”. By providing geocomputation tools in a fashion usable by end users, we could give the general public more authorship and autonomy over geodata. And, more concretely, if a person themselves can compute what they want, when they want it, we don't have to pre-process several variants of the same dataset, tweaked to suit different audiences, saving storage and especially processing time.

I did not have this clear of a goal at the start of the thesis, only intuitions and a set of loosely coupled ideas. I was also unable to find a foundational body of work to base this research on. What I did know is that I did not want to stay at a theoretical level. I wanted to truly **build** a solution, to the at this point ill-defined problem.

In hindsight, I don't think it was wise to build a concrete software implementation based on multiple, loosely defined theories. This imbalance led to a time-consuming complication between research and software implementation while writing the thesis.

Nevertheless, through many iterations, I was eventually able to integrate both the ideas and the thesis at large with this software implementation. I have learned two additional major lessons from conducting this study: One, writing well is hard work. Boiling a story down to the essentials asks a great deal of time and energy. I would not call the current story this study tells concise, but it is much more precise than I was able to write at the beginning of this adventure.

And two, I must learn to rely on the work of others. This is difficult, as it is contrary to my personal conviction on the great importance of ”learning from scratch”. Both the software implementation and the written thesis had me figuring out a lot of aspects from scratch, and this had advantages and disadvantages. The advantage is that 'doing the work again yourself', is probably one of the most educational endeavors one can do. We must understand the inner workings of the systems we work with, and especially of the systems we wish to improve. This is why I took the risk of developing many aspects of this thesis anew. Even if this might turn out to be a parallel study, it would only reinforce the 're' in research.

The disadvantage of doing this, is that you prevent yourself from being able to 'stand on the shoulders of giants'. I especially felt the sting of this lesson in the written aspect of this thesis. Not really knowing up from down because of having no readily available work to build on caused countless hours of revision. Another great disadvantage is that you run the risk of alienating both yourself and your endeavors. By building from existing starting points, you connect your work with the works of others, and in doing so contributing to a wider community.

If you have read this thesis up to this point, I want to sincerely thank you for your time and interest. With this thesis, I hope to have provided you and the wider geospatial community with something of value.

Bibliography

Akhmechet, S. (2006). Functional Programming For The Rest of Us.

Alicevision (2022). Meshroom. original-date: 2015-04-22T17:33:16Z.

Ammann, M., Drabble, A., Ingensand, J., and Chapuis, B. (2022). MAPLIBRE-RS: TOWARD PORTABLE MAP RENDERERS. In *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume XLVIII-4-W1-2022, pages 35–42. Copernicus GmbH. ISSN: 1682-1750.

Automation, C. (2018). Ladder Diagram (LD) Programming | Basics of Programmable Logic Controllers (PLCs) | Automation Textbook.

Azavea (2022). GeoTrellis. original-date: 2011-12-23T14:56:54Z.

Benac, R. and Mohd, T. K. (2022). Recent Trends in Software Development: Low-Code Solutions. In Arai, K., editor, *Proceedings of the Future Technologies Conference (FTC) 2021, Volume 3*, Lecture Notes in Networks and Systems, pages 525–533, Cham. Springer International Publishing.

Brink, L. v. d. (2018). *Geospatial Data on the Web*. PhD thesis, TU Delft. original-date: 2018-10-12T08:52:14Z.

Community, Q. (2022). QGIS Homepage.

Consortium, O. G. (2015). Web Processing Service.

Contributors (2022a). Deno. original-date: 2018-05-15T01:34:26Z.

Contributors (2022b). Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS.

Contributors (2022c). React – A JavaScript library for building user interfaces.

Contributors (2022d). wasm-bindgen. original-date: 2017-12-18T20:38:44Z.

Contributors (2022e). wasm-pack. original-date: 2018-02-12T14:04:34Z.

Contributors (2022f). WebAssembly.

Contributors (2022g). What is Ownership? - The Rust Programming Language.

Contributors, T. (2022h). Turf.js | Advanced geospatial analysis.

Dashiki (2020). Simple Request Breakdowns.

Dufour, D. (2022). geotiff.io. original-date: 2017-06-30T02:12:06Z.

Eberhardt, C. (2022). The State of WebAssembly 2022.

Elliott, C. (2007). Tangible Functional Programming. In *International Conference on Functional Programming*.

Esri (2022). ModelBuilder | ArcGIS for Desktop.

Foundation, B. and Contributers (2022). Geometry Nodes — Blender Manual.

Foundation, O. (2022). Node-RED.

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.

Games, E. (2022). Blueprints Visual Scripting.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, Mass, 1st edition edition.

Geodelta (2022). Omnibase.

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*. Thesis, -. Accepted: 2016-06-02T21:00:45Z.

Jangda, A., Powers, B., Berger, E. D., and Guha, A. (2019). Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In -, volume -, pages 107–120, -. -.

Janssen, P. (2021). Möbius Modeller.

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. Conference Name: IEEE Access.

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.

Melch, A. (2019). Performance comparison of simplification algorithms for polygons in the context of web applications.

Mozilla (2013). asm.js.

Mozilla (2022). MDN Web Docs.

Organization, E. (2022). emscripten. original-date: 2011-02-12T05:23:30Z.

Panidi, E., Kazakov, E., Kapralov, E., and Terekhov, A. (2015). Hybrid Geoprocessing Web Services. In -, pages 669–676.

Peters, R. (2019). Geoflow.

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.

Rutten, D. (2012). Grasshopper.

Sadeghipour Roudsari, M. and Pak, M. (2013). Ladybug: A parametric environmental plugin for grasshopper to help designers create an environmentally-conscious design. *Proceedings of BS 2013: 13th Conference of the International Building Performance Simulation Association*, pages 3128–3135.

Safe-Software (2022). FME Desktop | Data Integration and Automation.

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.

SideFX (2022). Houdini 3D Procedural Software.

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. In -.

Stats, S. G. (2020). Browser Market Share Worldwide.

Team, T. N. (2020). Browser market share.

Technologies, U. (2021). Bolt.

w3c (2019). World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation.

W3Counter (2020). Global Web Stats.

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.

Colophon

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