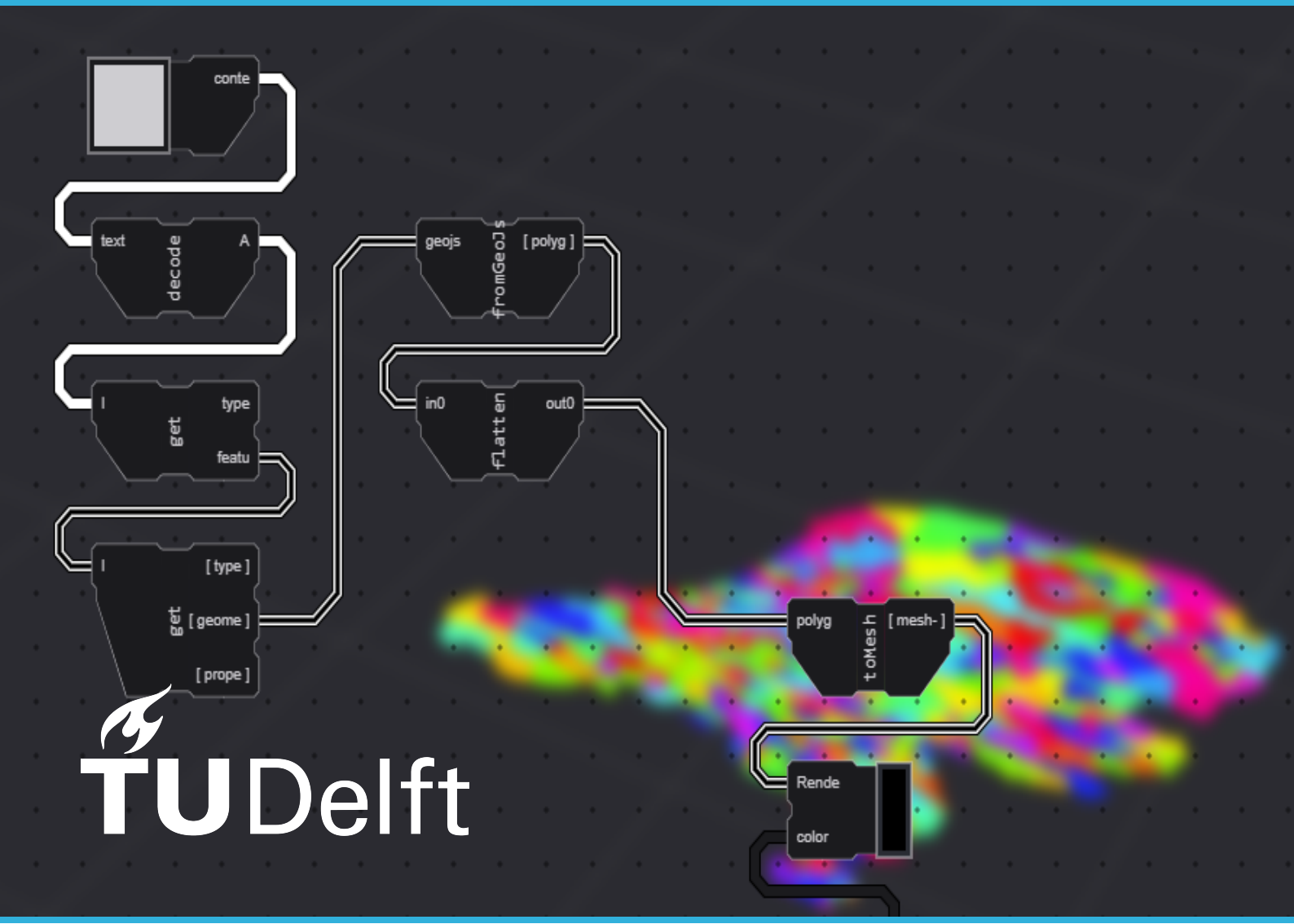
MSc thesis in Geomatics

GEOFRONT: A browser based visual programming language for geo-computation

Jos Feenstra

2022



TU Delft

MSc thesis in Geomatics

GEOFRONT: 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: 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:	Ir. Stelios Vitalis
	Dr. Ken Arroyo Ohori
Co-reader:	Associate. Prof. Hugo Ledoux

Abstract

-

Acknowledgements

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

...

Contents

Acronyms

1 Introduction

The field of Geographical Information Science (gis) concerns itself with the collection, processing, storage, and visualization of geodata. By doing do, 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, navigation, the military, and agriculture.

Central to this effort is the process of converting raw geodata into meaningful geo-information. Additionally, the effort often involves executing transformations or calculations on existing datasets. The term Geospatial data computation (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. All applications of geo-informatics require some level of geocomputation, as the raw data gathered from surveyance seldom corresponds to the precise information we wish to discover about the earth. This makes geocomputation a cornerstone of the entire field of geo-informatics, and vital to all her applications.

However, geocomputation is 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.

In an ideal world, the activity of geocomputation should be ergonomic: Operations must perform as fast as possible, and the effects of those operations must be both clear to the user, and reliable. 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, two promising developments regarding geocomputation are emerging: Visual programming, and browser-based geocomputation. these two developments require further explanation, after which the subject and goal this thesis can be made clear.

Visual programming

```
TODO: this chapter should probably be rewritten to connect more to the current
    content of the thesis
```

The Visual Programming Language (vpl) is still considered an ongoing development within the field of geocomputation, despite popular examples like FME (Source, TODO add picture). Especially in recent years, a push towards a more 'granular' vpls can be recognized: vpls offering more precise geocomputation on a smaller, more detailed level than the Extract Load Transform (etl) tools of FME. As an example, McNeels's Grasshopper (SOURCE) is increasingly being used for spatial analyses of buildings and neighborhoods, like solar irradiation or heating demands (source). Meanwhile, GeoFlow (shown in Figure 1.1) was used to model

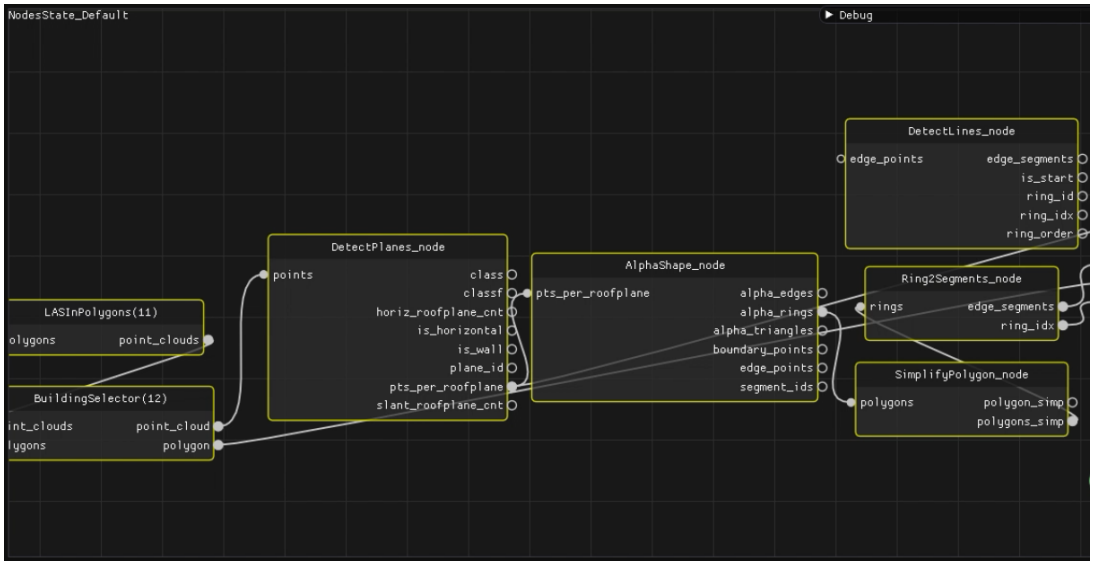


Figure 1.1: Geoflow: A geocomputation VPL

the 3D envelope of a building based on a pointcloud, which was subsequently scaled up in the creation of the 3D BAG dataset (source).

Using a vpl for geocomputation (geo-vpl) for more detailed geocomputation is still relatively novel, and contains many open ended questions. That geocomputation in certain scenarios benefits from the interface offered by vpls like these is probably true, indicated by the number of prior studies on the topic (source, source, source), and the popularity of vpls used for geodata and geometry computation. The *extend* to which vpls benefit detailed geocomputation, is still relatively unknown.

Browser-based geocomputation

Browser-based geocomputation is slowly gaining traction during the last decade Kulawiak et al. [2019]; Panidi et al. [2015]; Hamilton [2014]. 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 in Figure 1.2, used by Dutch municipalities to measure and map buildings and infrastructure.

Additionally, 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 installment or configuration, and distributable, since the web is cross-platform by default, and poses many advantages for updating, sharing, and licensing applications. 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 resend to the user upon every computation request ([back up with sources from related works]).

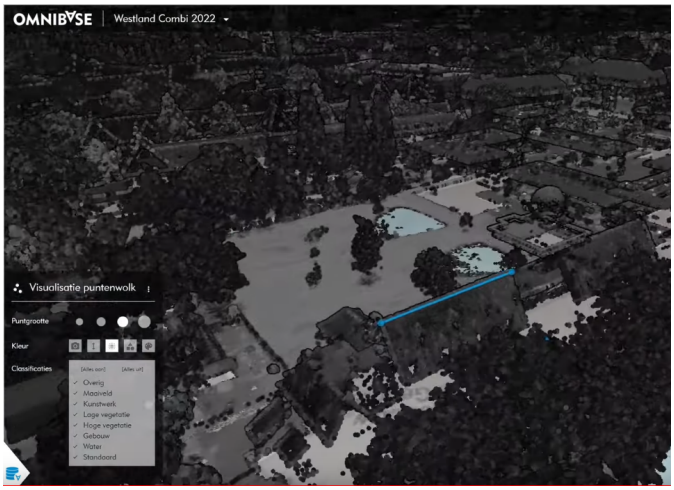


Figure 1.2: Omnibase: An example of browser-based geocomputation

However, Browser-based geocomputation also has many open-ended questions and 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.

1.1 Problem Statement

The proven usefulness of a vpl for geodata computation, together with the theorized potential of browser-based geocomputation, lead to the idea of a Browser-based visual programming environment for geocomputation (geo-web-vpl). regrettably, this topic has seen little to no prior study, even though a geo-web-vpl could be the next step in the pursuit of making the activity of geocomputation more clear and reliable. The activity of geocomputation could benefit from the experimentation and debugability advantages of a vpl, combined with the accessibility and distribution advantages of a web application. This format might even yield completely unique benefits. However, since only a very small number of geo-web-vpls exist, there are not enough examples to definitively proof or test these aspects.

Additionally, prior studies give an indication to certain disconnects between the fields of geo-VPLs and web-GIS. This is further explained in Chapter 3. Multiple existing geo-VPLs are encountered which cannot be used in a browser, while many web-based VPLs are unable to support existing, native geocomputation functionalities. From this, it can be concluded that bridging this gap is vital in making any geo-web-vpl succeed.

1.2 Research Objective

This study seeks to close the gap between geo-VPLs and browser-based geocomputation by designing, implementing, and evaluating a new prototype geo-web-vpl. The study starts from the presupposition that proper utilization of existing, native geocomputation libraries within a visual programming environment is key to making a geo-web-vpl succeed. Overcoming this technical challenge is the focal point of this study.

1.3 Research Questions

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

> *How can a VPL be used to support and execute existing geo-computation libraries in a browser?*

Supporting Questions

The following supporting questions are defined to aid in answering the main question. These are based upon various components of the main problem statements, further explained in Chapter 4

- *To what extend is the browser capable of **representing** a generic dataflow-vpl for processing 3D geometry?*
- *To what extend can geocomputation libraries written in system-level languages be **compiled** for web consumption?*
- *To what extend can a web-consumable library be **loaded** into a web-vpl without explicit configuration?*
- *To what extend can a 'geo-web-vpl' be **used** to create geodata pipelines?*

In order to obtain answers to these questions, a literature review is performed, a prototypical software application is developed, and this application is tested and accessed on various aspects.

1.4 Scope

The scope of this thesis is cornered in the following six ways:

Only frontend geocomputation

This study excludes any *backend* based geocomputation. A web application *could* be used to orchestrate geocomputation web-services, which could also deliver a form of browser-based geocomputation to end-users. However, for the scope of this thesis, we limit ourselves to purely client-side solutions, with calculations literally happening within the clients browser. This is also why this study excludes the OGC standard of the Web Processing Service (wps) OGC [2015].

Adding backend-based geocomputation to a geo-web-VPL would be an excellent follow-up investigation to this study. Future work could research the possibility of utilizing a hybrid strategy of both client-side and server-side geocomputation, following in the footsteps of Panidi et al. [2015].

No Usability Comparison

While accessibility / usability is a motivation of the development of a vpl, and while usability will be analyzed to a limited extend, no claims will be made that this method of geocomputation is *more* usable as opposed to existing geocomputation methods. This research attempts to solve practical inhibitions in order to discover whether or not browser-based, vpl-based geocomputation is *a* viable, usable option. If it turns out that this method is viable enough technically, future research will be needed to definitively proof *how* usable it is compared to all other existing methods.

Similarly, a survey analyzing how users experience browser-based geocomputation in comparison to native geocomputation must also be left to subsequent research. While this would be insightful, client-side geocomputation is too new to make a balanced comparison. 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 'geo-web-vpl' well due to the local optimality quality of many DTM procedures (Such as the Delaunay triangulation (dt)),

Assumption: a 'geo-web-vpl' is a normal Geometry VPL able to handle geodata

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

A geo-web-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 'geo-web-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 core browser features

Lastly, the implementation of the geo-web-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 very 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 geo-VPLs, or if they only count in this very specific scenario. "Core browser features" is defined in Section 4.2.

1.5 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, and shows how this relates to the background and related works chapters.

Chapter 5, Implementation, presents the implementation of the methodology. In addition, the main design decisions are described and justified in this part of the study.

Chapter 6 analyses the results from this implementation in the various ways described by the methodology.

And finally, Chapter 7: Conclusion & Discussion, concludes to which extend 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, represented by the corners of Figure 2.1:

- Section 2.1 covers the background on Geocomputation
- Section 2.2 covers the background on Web applications
- Section 2.3 covers the background on Visual Programming Languages

Each one of these cornerstones will be discussed and analyzed by themselves, after which Chapter 3 and Chapter 4 will focus on the interplay and connections between these bodies of work.

2.1 Geocomputation

```
TODO: finish this chapter
```

This section offers a brief background on the wide topic of geocomputation.

Geocomputation is a central component of the wider field of geo-informatics. The term geocomputation, or geodata processing, is used to represent all types of computations performed on geographical data. Anything from the calculation of an area of a polygon, to crs transformations, feature overlay, or converting a raster dataset into a vectorized dataset, is regarded as geocomputation.

It must be emphasized that a geocomputational procedure is always fully defined by and dependent upon its input and output data types (similar to any type of computation). This is also why geocomputation is seldom a *goal* in itself, but much rather the *means* to discovering geo-information.

```
TODO: show images of geo-computation
```

2.1.1 Similarities and differences with neighboring fields

Geocomputation can be seen as an applied field of the more general field of computer graphics. Other applications of computer graphics include computer aided design (CAD), Building Information Modelling (BIM), molecular biology, medical imaging, robotics, but also special effects, video games, and graphic design. For this reason, these applied fields can be regarded as neighbors to geocomputation & GIS.

It is important to recognize that certain geocomputational procedures fully overlap with computer graphics and these neighboring fields, while others are very specific to the field of GIS

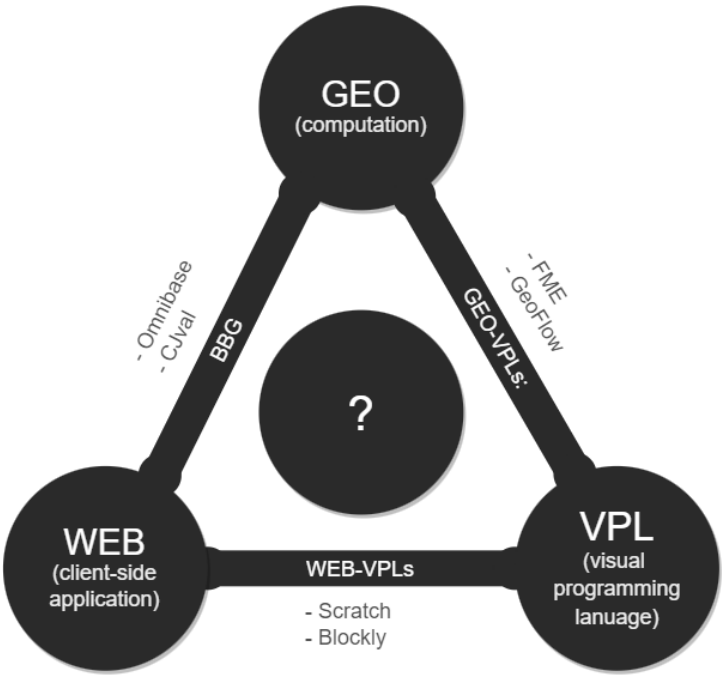


Figure 2.1: Triangle Model

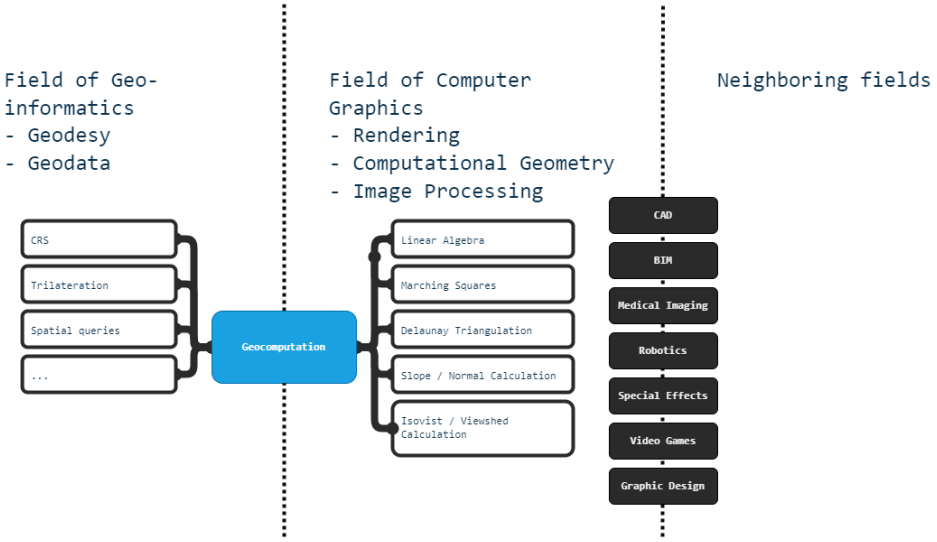


Figure 2.2: Geocomputation in relation to other fields

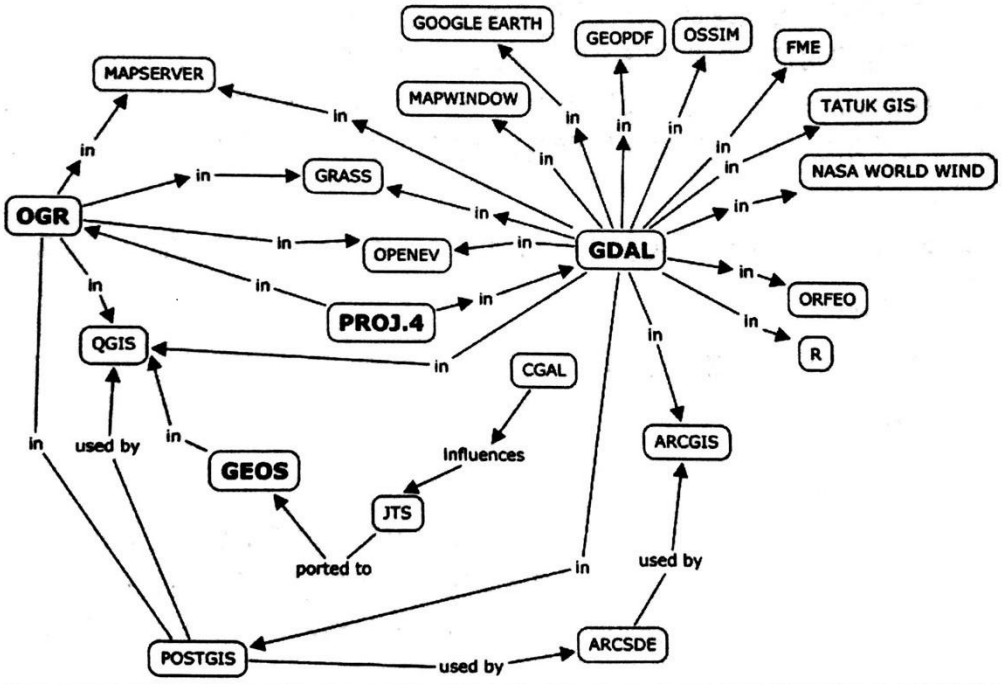


Figure 2.3: Dependency graph of common geocomputation libraries. (This needs validation)

and geo-informatics. As an example, matrix transformations and reprojections are commonplace in the wider field of computer graphics, but transformations formalized and structured in the form of crs is very specific to the field of GIS (See Figure 2.2). As such, geocomputational procedures can roughly be categorized in two fields:

- geocomputations *specific* to gis,
- *common* computational geometry procedures

This categorization can be identified by noting if an operation appears in a neighboring field, or just in the field of gis.

This *specific* category exists because of gisS foundation in the field of Geodesy, and the nature of geographical data. Geodata differentiates itself from any form of data by its sizable nature, and geospatial nature. GIS dataset sizes easily scale into terabytes of data, and each datum specifically represent a real, measured, earthly phenomenon. This makes storage and accuracy more relevant than other computer graphics applications.

2.1.2 Geocomputation libraries

Numerous geocomputational software libraries exist, written in a plethora of programming languages. Still, a certain 'Canon' can be defined based on popularity in terms of numbers of downloads, numbers of contributors, and number of dependent projects.

Out of all open-source geocomputational libraries, the ones developed and maintained by the Open Source Geospatial Foundation (osgeo) (Source) can be regarded as the most significant to the field of gis. These libraries include:

- The Geospatial Data Abstraction Library (gdal) (Source)
- The Cartographic Projections and Coordinate Transformations Library titled PROJ (Source)
- The Geometry Engine Open Source (GEOS) (Source)

The geocomputations found in these libraries operate primarily on 2D / 2.5D raster and Simple Feature datasets (Source). How these projects related together is presented by Figure 2.3. Together, these libraries represent the core computational needs specific to the field of gis.

For the more common computational geometry needs, the Computational Geometry Algorithms Library (cgal) library is also widely used.

2.1.3 Conclusion

The Take-away:

Combination of general computational geometry, and computations very specific to GIS.

Geocomputation is mostly facilitated by a very select number of C++ libraries.

2.2 Frontend web applications

```
TODO: this chapter is too large, the client-server / front-back elements should be
    trimmed down, or maybe even entirely removed. I think Its too granular.

Also, add more pictures!
```

This section offers a background on frontend web applications. Since this topic is too large in scope to fully cover, three elements are chosen which are highly relevant to this study. Finally, these three topics are linked to each other in Section 2.2.4.

2.2.1 Distributed systems

In software development of distributed systems, the following phrases exist:

- Client and Server
- Frontend and backend
- Native application and web application
- web application and website

While these phrases do overlap to an extend, years of interchangeable usage have lead to their differences often being overlooked. This study wishes to shed light on the relationships between these phenomena, as the nuances between them are vital to this studies contribution.

The client-server model

First, the client-server model. The client-server model refers to a distributed application architecture which balances storage and processing responsibilities between two cooperating types of programs: Clients and servers. In this model, a client sends a request to a server, and the server provides the response asked for (see Figure 2.4). While this model immediately invokes images of web clients and web servers, it is important to recognize that the client-server model is far older than either web applications, or the World Wide Web in general. It is an abstract computational model, of which the World Wide Web is just one example. A corresponding client and server may even exist on the same machine. A program running on a machine can act as a client, a server, or both, based on the role this program sets out to fulfill in relationship to other programs.

The client-server model is beneficial for sharing resources, both in terms of storage and processing. A distinction is made between centralized models, in which the bulk of these resources are centralized on one or more servers, and decentralized models, which distribute and offload some or all of the computational resources to the clients. A centralized model has the advantage of making clients simple and interchangeable, at the cost of making them highly reliant on the uptake of and connection to the server. This also generates more client-server traffic. A decentralized model makes clients independent and decreases traffic, at the cost of the complications caused by decentralized architectures. The choice between a centralized or decentralized client-server model is therefore highly reliant on the resources of client and server hardware, as well as the quality of the connection between the server and client.

Figure 2.4: A typical client-server interaction)

Frontend and backend

The terms frontend and backend, though closely related to clients and servers, refer to different phenomenon. Both are separations of concerns, a design principle prevalent in computer science to specialize a program into separate responsibilities. However, client and server programs are defined by their separation into ”requester” and ”responder” roles, whereas the frontend and backend are defined by their separation into ”presentation” and ”data access” functions. Presentation functions are responsible for interacting with the end-user of the application, and is concerned with aspects such as user interface, user interaction, and rendering. ”data access” interacts with the physical hardware of the machine, and is concerned with aspects such as storage methods, database management, and scalability. It just so happens that the presentation functions often corresponds with a requester role, and that data access functions often corresponds with the responder role. However, this is never a given. A server can be responsible for providing both the frontend and backend functionality, in the case the presentation of an application is rendered on this server.

Native application and web application

The nuances between a web application and a native application must also be specified, alongside their relationship to clients and servers. In this context, we make a distinction between *Programs*, which refer to individual processes on either the side of the client or the server, and an *application*, which either represents a non-distributed, self contained program, or represents the whole of corresponding client and server programs together. Client programs or server programs are also often abbreviated as clients and servers. If a client runs without the corresponding server it relies upon, we can say that the client functions, but the entire *application* does not function. In practice, however, the term 'web app' often specifically refers to the client, confusingly enough.

In any case, a program is considered native if it directly runs on the operating system of a device. A program is considered web-based or browser-based if a browser is required to run it. A web application always has a client, as it will always need to be initially served by a corresponding server. However, the extend to which the functionality of a web app is self contained or continuously reliant on this server may vary. A native program may also be a client, as there is nothing preventing a native program of making the exact same web request as a web application.

Due to this ambiguity of 'client-side' being able to refer to both native and web clients, this study makes use of the terminology 'browser-based programs' or 'browser-based applications', to point to web clients in particular.

Web applications have specific advantages and disadvantages compared to native applications. The big advantages are that web applications are cross-platform by nature, and offer ease of accessibility, since no installment or app-store interaction is required to run or update the app (src: vpl 2019, src: hybrid). As soon as a web app is found, it can theoretically be used. The containerized nature of the web also makes web applications in general more safe. For unknown native applications there is always a danger of installing malicious software, whereas an unknown web application without any privileges is practically harmless (Needs citation). The ability to share the a functional application with a link, or to embed it within the larger context of a webpage, is also not a trivial advantage.

The disadvantage is that normally, web applications can only be written in JavaScript, a very-high level interpreted programming language. Its high-level nature leads to imprecision in using computational resources. For example, it makes no distinction between integer and floating point arithmetic. Additionally, the safety and containerization demands of the web make web applications more removed from the operating system and hardware. Any type of operating system (os) interaction such as opening a window, interacting with the file system, or drawing directly to the screen buffer, is off-limits. Both these layers of indirection makes web applications traditionally unfavorable for demanding, highly specialized programs.

Web application and website

Lastly, a soft distinction is also made between websites, and web applications. Roughly speaking, a web application is a website which requires javascript in order to be functional. This makes websites more static, and web applications more dynamic, being able to change based on user input. Wikipedia (Source) can be considered a website, whereas overleaf (Source) is definitively a web application. Many border cases also exist, like Twitter (Source). Following the above definition, twitter is a web application, despite the fact that its core functionalities could be implemented without any client-side javascript.

```
Sources:

(https://en.wikipedia.org/wiki/Web_application, bad source, but this is more '
    conventional wisdom' than true 'knowledge', couldn't find a more credible
    source,

what would make a person credible on this content?)

(https://en.wikipedia.org/wiki/Frontend_and_backend)
```

2.2.2 Rich Clients

In the early days of the World Wide Web, web applications were practically impossible, and the web consisted of websites exclusively. Then, with the introduction of the javascript scripting language in 199x (Source), and browser plugins like Adobe Flash (Source), the first couple of web application slowly started to be developed. Still, these early web applications exclusively used a centralized client-server model. The clients were simple, and completely reliant on the server.

In the decades that followed, the javascript runtime of web browsers saw continuous improvements, alongside additions like HTML5, facilitating more interactive usage of webpages. As the web and web technologies matures, new ways of using these technologies are discovered. Web applications became more interactive, and frontend functionalities were slowly moved from the server to the client.

These developments continued. Since 2012, a trend of **rich web-clients** can be widely recognized Hamilton [2014]; Panidi et al. [2015]; Kulawiak et al. [2019]. At that point, the browser had become powerful enough to allow for 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 model was dubbed ”single page application”, and was and still is 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 of javascript.

This growth has also lead to web applications being used 'natively'. Tools like Electron (Source) 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 (Source) are able to compile a web application into a native application without a browser runtime.

If the applications resulting from both types of tools are to be regarded as 'web apps' or 'native apps', is left as an exercise to the reader. In any case, it becomes clear that rich web clients and their build tooling are starting to blur the line between native and web software.

2.2.3 WebAssembly

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

WebAssembly (wasm) is officially dubbed the fourth type of programming language supported by all major web browsers, next to HTML, CSS, and JavaScript. Strictly speaking however, WebAssembly not a language, but a binary instruction format for a stack-based virtual machine. (SOURCE: https://webassembly.org/) it can be used to, theoretically, run any 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.

Limitations

The sentence above uses the phrase *in principle*, since there are quite a few caveats to the format. While in theory any application can be compiled to WebAssembly, in practice, not all applications turn into functional webassembly applications, due to certain factors. 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 and HTML5 features from WebAssembly. This is what the majority of current WebAssembly projects look like. 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 2.5, 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. will 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 this study falls in the category of scientific computation, these performance considerations will still have to be taken into account. The most important conclusion 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.

Figure 2.5: Comparison of compilation trajectories

2.2.4 Conclusion

When reading Section 2.2.1, Section 2.2.2, and Section 2.2.3 together, a pattern emerges. WebAssembly blurs the line between the web-based and native development even further than the rich clients, and invites a further re-examination of our established models of distributed systems. The compile target allows web-apps to make use of native libraries, and allows native software to be run on the web. This second aspect offers a complete reverse workflow compared to the now popular Electron based applications described in Section 2.2.2.

There was a significant initial delay between the improvements of the browser, and the widespread popularity of rich web clients. This study argues that as of right now, we are in the middle of a similar situation. A new technologies exist, it is implemented by all major browsers, and offers completely new ways of working with the web platform as a whole. The question remains what this will mean for the established models of clients and servers, the frontend and backend, and the web and native contexts.

2.3 Visual Programmming

The third body of work this study draws from is works on the topic of visual programming. This section offers a brief overview on the topic itself, after which since 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 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 Graphical User Interface (gui). A visual programming language allows users to create programs by adding preconfigured 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 ().
2. **Diagram-based languages**, in which programming function are represented by nodes, and variables are represented by edges between these components (). 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 (). 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 ().

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 wishes 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 2.3.3.

Visual programming languages are used in numerous domains. The vpls of the 30 studies examined by (SOURCE) 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 (SOURCE) is the industry-standard for programming Programmable Logic Controllers (PLCs). vpls are also widely used within computer graphics related applications, including the field of gis, These will be covered in Section 3.2. Lastly, vpls also have great educational applications. Harvard's introduction to computer science course, CS50, famously starts out with Scratch, a block-based visual programming language normally targeted at children, to teach the basics of computational thinking (Source: CS50).

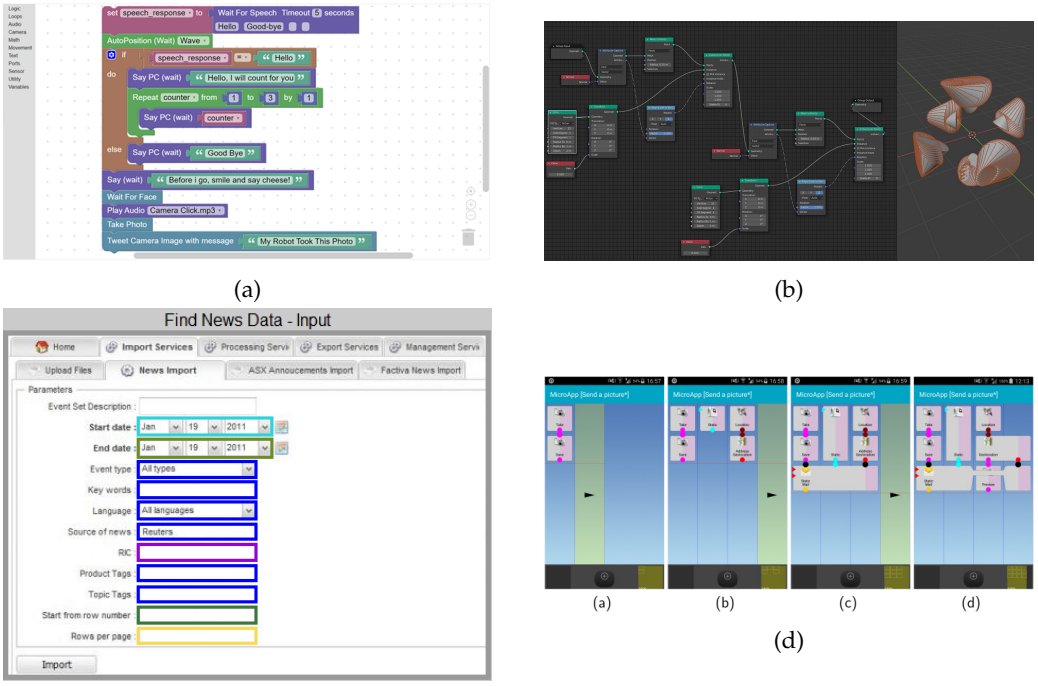


Figure 2.6: Four different types of visual programming languages: Block-based, diagram-based, form-based, and icon-based, respectively

2.3.1 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 Dimentions 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.

```
TODO: this is a weak argument. This also needs to be more nuanced
```

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

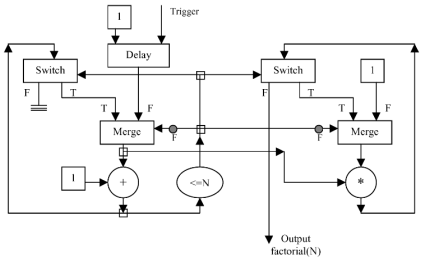
2.3.2 End User Development & Low Coding

a vpl done right can make automation available to a very large audience, and this is exactly the point. 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 software applications, using specialized tools and activities. 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.

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 (Kuhail et al. [2021]). And two, end users outnumber developers with formal training at least by a factor of 30-to-1 according to Kuhail et. al., and my suspicion is that this might be much higher.

I would like to add that offering the general public a chance to automate repetitious workflows might not only increase productivity, but can also greatly improve the quality of life in general, by focussing on the profound instead of the mundane.

In the private sector, End User Development (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 (Source:), and with a growth rate between 20 and 40 percent, the value may reach as high as 19 Billion by 2030 (SOURCE: FORRESTER)



(a)

$$\begin{aligned}&\text{LOOP WITH } i = N, fact = 1\\&\qquad \text{NEW } fact = fact * i;\\&\qquad \text{NEW } i = i - 1;\\&\text{WHILE } i > 1;\end{aligned}$$

(b)

Figure 2.7: A factorial function, written in a vpl, and textual form

2.3.3 Dataflow programming

An important aspect of the dataflow-VPL is the connection to the field of dataflow programming, which is not necessarily linked to VPLs.

Dataflow programming is a programming paradigm which internally, represents a program as a dag (SOURCE Dataflow 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 (SOURCE dataflow 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 (SOURCE: functional programming).

The important take-away is this: A vpl is not just a matter of a user-friendly UI or a stylistic choice. This might be true for block-based vpls, but not for diagram-based vpls. By closely resembling dataflow itself, and because of its functional programming nature, diagram-based vpls can actually lead to faster and more reliable software.

2.3.4 Disadvantages and open problems

there is of course, no such thing as a free lunch. vpls and dataflow programming en large have got certain disadvantages and open problems.

Iteration and conditionals

A problem described in almost all reviewed vpl literature (Source: advanced in dataflow, SOURCE: Dataflow programming, SOURCE: COGNITIVE), 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 2.7.

Encapsulation & reusability

Similar and yet different is the topic of encapsulation, or, how (SOURCE: COGNITIVE) names this problem: 'visibility'. It is widely known that as a program scales in size, the complexity of handling the application scales exponentially. 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 (SOURCE: Dataflow programming). 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 nontheless useful for comparing vpls. The dimensions given in the cognitive dimensions framework (SOURCE) 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 critical 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 (SOURCE).

Lacking Life-cycle support

Finally, (communicating 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 critical to making any application succeed, and eud research should not be limited to purely the aspect of creating functionalities.

This oversight on life-cycle aspects can be found in the vpl & low-coding industry as well.

```
TODO: Formalize these ramblings, or don't.

- no existing VPL ( that I know of ) has proper git-based version control.
  - Most vpls use proprietary storage and collaboration methods.
- vpls which are able to compile to a textual format are rare
  - vpls able to compile to a programming language, or a headless format, are even
      more rare.

- Let alone: on operational programming paradigms such as Test driven development (
    TDD), continuous integration (CI), continuous delivery (CD).

- in general, the life-cycle support of most vpls found in the low-coding industry
    are closely tied to their business models.
  - Users can only use the publication tools, version control tools, and package /
      library managers offered to them by the vendor.
```

And while we are on the topic of publication, only 16 out of 30 of the tools analysed by (communicating 2021) were available publicly with some documentation. It seems the lack of publication tooling might also partially be due to a lack of publication in general.

2.3.5 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. while almost no projects or studies exist at the intersection of all three of these fields, we do find related studies which intersect two of these fields, represented by the edges of Figure 2.1:

- 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, and web clients, as described in Section 2.2.

First, a small paragraph on the motivation behind browser based geocomputation. As stated in Section 2.2, web applications offer safety, distribution and accessibility advantages over native applications. As such, browser based gis has become a sizable component of the geospatial software landscape. However, despite the popularity of geographical web applications, the range of actual gis abilities these applications have is generally speaking very limited. geocomputation is usually not present within the same software environment as the web app. This limited range of capabilities inhibits the number of use cases geographical web applications can serve, and with that the usefulness of web gis as a whole. If web applications gain geocomputation capabilities, they could grow to be just as diverse and useful as desktop gis applications, with the added benefits of being a web application. It would allow for a new range of highly accessible and sharable geocomputation and analysis tools, which end-users could use to post-process and analyze geodata quickly, uniquely, and on demand.

This need has let to the field of Browser-based geocomputation

Browser-based geocomputation has seen some academic interest throughout the last decade Hamilton [2014]; Panidi et al. [2015]; Kulawiak et al. [2019].

Hamilton et. al. created a 'thick-client', capable of replacing certain elements of server-side geoprocessing with browser-based geoprocessing Hamilton [2014]. At first glance, the results seem 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."(SOURCE). 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 use 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. 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.

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 (SOURCE, Figure 3.1), is a web-based geoTIFF processing tool, and fully open source. 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.

Azavea's Raster Foundry and ModelLab (Powered by GeoTrellis), is another GeoTIFF / raster based web processing tool, in which basic queries and calculations are possible (SOURCE).

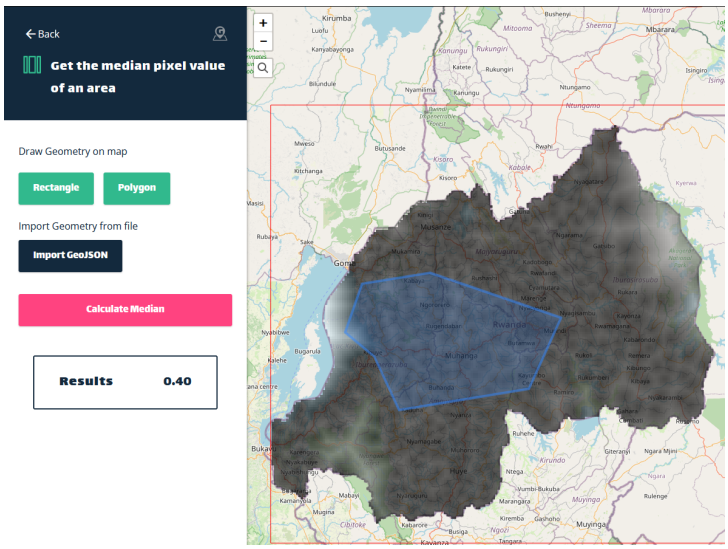


Figure 3.1: The geoTIFF.io application

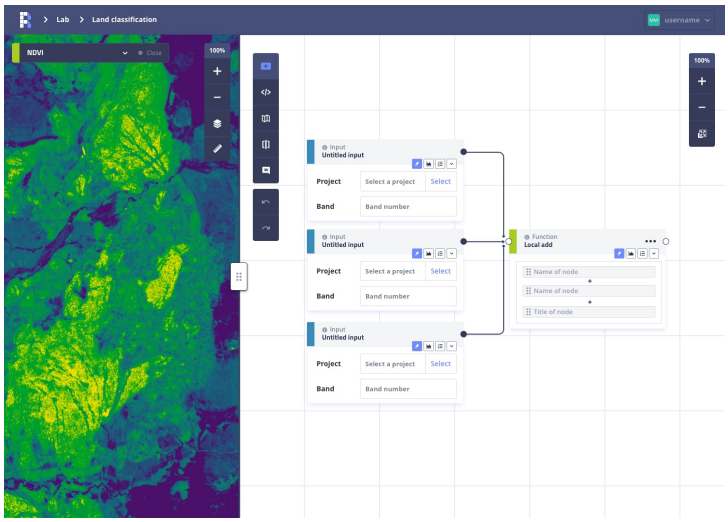


Figure 3.2: The ModelLab application

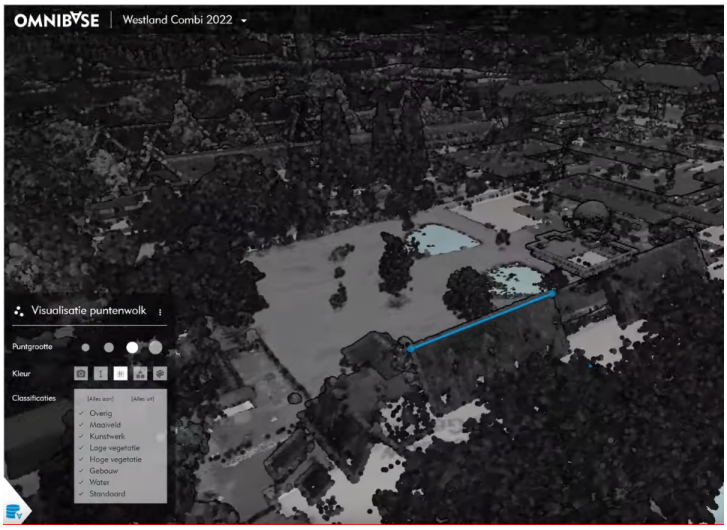


Figure 3.3: The Omnibase application

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 3.2). However, the tool uses mostly server-side processing, making this application less relevant to this study.

Also, despite their mission statement to: "help individuals and organizations to effectively access, analyze, edit, and visualize remotely sensed data in transformative new ways without years of specialized training or ongoing investments in proprietary software and technology infrastructure."(Source), The tool appears to be reliant on their own proprietary infrastructure to save and run the application. The author of this study was also not able to find a public demo of the application.

The last web-based geocomputation platform this study would like to mention is Geodelta's Omnibase application (SOURCE: GEODELTA, Figure 3.3). Omnibase is a 3D web gis application for viewing and analyzing pointclouds and areal imagery datasets. It offers client-side geocomputation in the form of measuring distances between locations, and calculating the area of a polygon. It also offers photogrammetry-techniques such as forward incision of a point in multiple images, but these are calculated server-side.

3.2 Visual programming and geocomputation

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

```
[I just wish to state: "these applications exist".
I also wish to make some comments on the interplay between geocomputation and vpl.
Are these fields complimentary?
These will also be referred back to during the methodology chapter]
```

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

Figure 3.4 offers this overview of some of the more significant vpls present in not only gis, but also the neighboring domains based on computer graphics. Why these fields are regarded as relevant are explained in Section 2.1.

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.

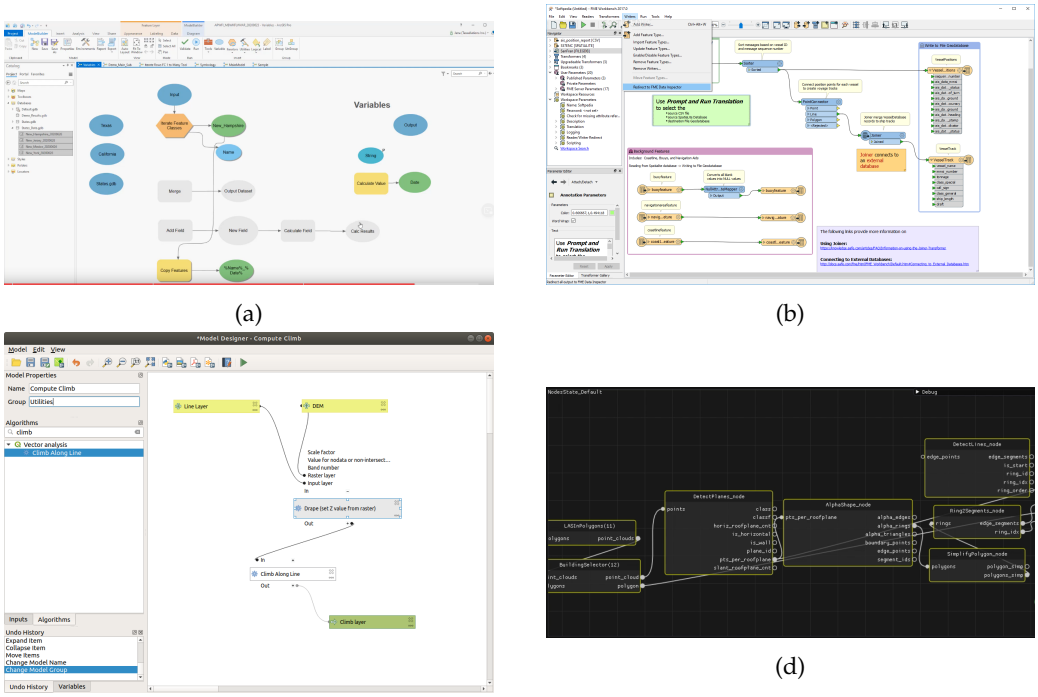
By far the most well-known visual programming language within the field of gis is the commercial etl tool FME (Source, Figure 3.5a). This tool is widely used by gis professionals for extracting data from various sources, transforming data into a desired format, and then loading this data into a database, 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. While this application is still in an early phase, it already offers very powerful range of functions. It offers CGAL processes like alpha shape, triangulation and line simplification, as well as direct visualization of in-between products.

VPLs in neighboring domains

Figure 3.4 shows a great number of non-GIS vpls. while these do not explicitly cover GIS, their close ties to computer graphics are still highly relevant to GIS and the activity of geocomputation.



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

Figure 3.5: Four VPLs used in the field of GIS: ArcGIS's Model Builder (a), Save Software's FME (b), QGIS's Graphical Modeler (c), and Geoflow (d).

The choices of which vpl to include in Figure 3.4 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 to perform photogrammetry reconstruction.
- 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”: with sliders and other UI elements, the vpls seem to be intended for rapid experimentation.

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. This might be because all these vpls, with exception to the behavior vpls, are essentially dealing with ”production pipelines”. No networking, no distributed systems, 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. Therefore, the vpl might be a perfect fit for all 2D and 3D data pipelines. vpls allow rapid debugging, rapid experimentation, and the straight-forward nature of 2D/3D pipelines mitigate the challenge vpls have with representing imperative flow statements (`if, else, for, while, break`).

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. 3D artists can now use these vpls to create procedural textures.

(a) (b)

Figure 3.6: TODO: add texture vpls

(a) (b)

Figure 3.7: TODO: add geometry vpls

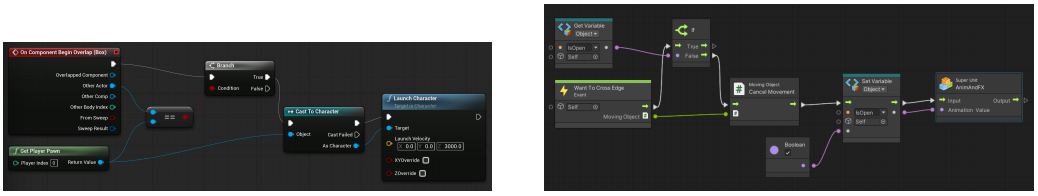
Geometry, and photogrammetric VPLs

Procedural Geometry vpls are not far behind the material vpls in terms of popularity. Applications like Blender's geometry nodes, Rhino's Grasshopper, or Houdini, 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 much 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. While this can be regarded as procedural modelling, the complexity and computation involved in photogrammetry make a vpl offering it a class in of itself. The vpl inside of Meshroom can be used to fine tune all stages of the 3D reconstruction process.

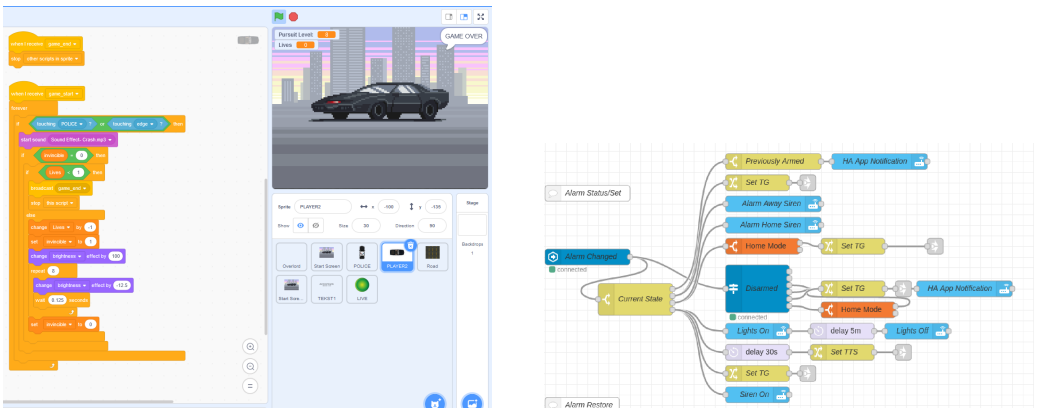
Behavioral VPLs

The behavioral and logical vpls found in applications such as Unreal's Blueprint and Unity's Bolt (https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/, https://assetstore.unity.com/packages/tools/visual-scripting/bolt-163802) 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 2.3 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 3.8 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



(a) (b)

Figure 3.8: Two behavioral vpls, showing "flow-state" variables. Left: Unreal's Blueprints, Right: Unity's Bolt.



(a) (b)

Figure 3.9: Two VPLs used on the web: Scratch (a), and nodeRED (b).

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.

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 one line with research covered in Section 2.2: *"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”, and Googles related ”Blockly” project (Source, See Figure 3.9a). Scratch is well-known as an educational, block-based vpl, targeted at children and young adults to teach the basics of computational thinking. The program is famously used as the first assignment of Harvard university's CS50 course (Source: https://www.youtube.com/watch?v=1tnj3UCkuxU). 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 excellent 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.

[blockly-¿ can be compiled to python and javascript] Microsoft makecode arcade

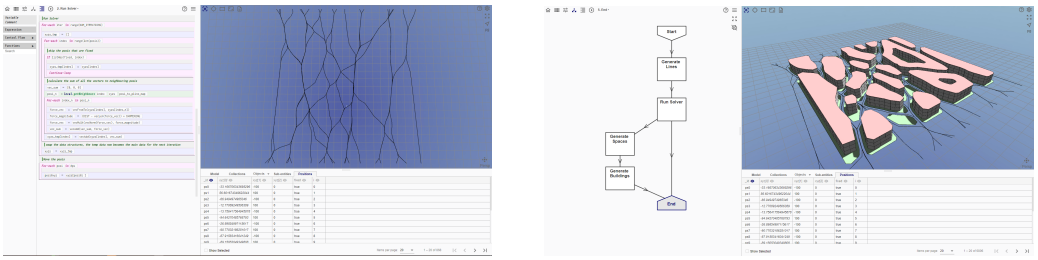
The second exemplary web vpl this study wishes to bring to the readers attention is the ”nodeRED” application (Source, see Figure 3.9b). This is a feature-rich diagram-based application, created to serve the domain of IoT. This vpl uses the browser-based platform not only for the aforementioned Section 2.2 reasons, but also for the exact same reasons a router, NAS or IoT device often opts for a browser-based interface: Servers, either small or big, explaining how they desire to be interfaced, is more or less the cornerstone all web clients are based upon. If the server serves its corresponding client, users do not need to find some compatible interface themselves. For this reason the ”nodeRED” application is a web application, even though it is mostly run on local networks.

3.4 Browser-based, visual geocomputation programming

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 (Source: https://mobius-08.design-automation.net/about), and is by far 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 an impressive range of applications, including CAD, BIM, urban planning, and GIS. It uses a combination of a very 'bare-bones' diagram-based vpl, together with a very rich block-based vpl (See Figure 3.10). 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 geo-web-vpl presented by this study still differs from the mobius modeller in the following aspects:

- This study explores the usage of diagram-based vpls as opposed to a block-based vpl, to allow for the dataflow programming advantages described in Section 2.3.3.
- 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 2.3.4.



(a) (b)

Figure 3.10: Images of the Mobius modeller application

4 Methodology

This chapter explains the methodology used in this study. It starts out with an overall structure, showing what elements this methodology precisely consists of, and how these elements came to be. The chapter then continues with four sections, each explaining one of these components. But first, the overall methodology.

4.1 Overall methodology

Nature

The methodology of this study can be characterized as practical as opposed to theoretical, and iterative compared to linear. The prior works on browser-based geocomputation and geo-vpls indicate that a strong theoretical framework for a geo-web-vpl is in place (Source). But, and this is especially evident in the prior studies regarding Browser-based geocomputation, the practical implementation of these theories were only partially successful, and limited in scope. This necessitates a practical approach in response. And, due to the investigative nature of this study, the methodology requires to iterate upon itself, instead of following a singular, linear path.

Structure

The content of this methodology is based upon the main and supporting research questions. As such, it bears fruit to explain how these specific questions were chosen.

The related studies in Chapter 3 show that only one geo-web-vpl as described by this study exists. This is in contrast to the many examples of geo-vpls in Section 3.2 and web-vpls Section 3.3. Based on this, it can be assumed that creating a visual programming environment in a web-browser must be possible. Using a vpl for geocomputation must also be possible. However, the greatly reduced number of geo-web-vpls indicates some sort of hinder, preventing this type of application to be realized. Either geo-VPLs were not able to be properly used in a browser, or web-based VPLs were unable to support geocomputation functionalities.

This study starts from the second assumption: Apparently, web-based VPLs are unable to support existing geocomputation functionalities. This could be because of several reasons, of which this study identifies four major ones.

geocomputation functionalities might not be able to be properly:

- **compiled** into a format functional on the web
- **loaded** within a web-based VPL
- **represented / visualized** by the interface of a web-based, dataflow-VPL
- **used** within a web-based dataflow-VPL

As it is unknown which one of these reasons is causing this barrier, the study must encompass all four of these possible hindrances, and access to what extend these aspects form a hinder towards the main goal of a geo-web-vpl. Moreover, the real reason might not lie in one of these areas, but in the interplay between all of these factors.

It is this study's objective to pose a solution to this barrier. The study goes about doing so, by developing a prototype geo-web-vpl accompanied by web-compiled geocomputation libraries. This prototype vpl is used as a staging ground to discover the extend of possible hindrances. Per hindrance, we document design considerations, and run experiments, all to test to what extend this prototype geo-web-vpl fails or succeeds to provide for this aspect. After this is done, we can compile a final conclusion to the main research question.

The methodology of this study is structured to facilitate this process. It is subdivided into four components, each representing a sub-research question, which is in turn based upon one of these possible hindrances. The questions are posed in such a way that answering them will require us to explore the extend of the hindrance, and find possible solutions.

The remained of this chapter covers the four components of this methodology, and how this relates to this prototype.

```
TODO: diagram: 4 research questions -> four possible barriers of geocomputation
```

```
TODO: diagram: show the 'locations' of the four research questions ( client / server
    / native, etc.)
```

4.2 Representation

The first component of the methodology involves the question of Representation: *To what extend is the browser capable of **representing** a generic dataflow-vpl for processing 3D geometry?*.

Before exploring how lifting geo-computation to the web might take place, this study first wished to discover to what extend the web browser is able to facilitate the interface of a dataflow vpl in general, both in terms of data structures and visualization. This was necessary because of the observation that successful vpls involved with computational geometry are all dataflow-vpls (see Section 3.2), but no existing web-vpl concerned with geometry uses this model (Section 3.3). Due to the many advantages posed in Section 2.3.3, but especially the concurrency advantages, a dataflow-type VPL had to be chosen for the topic of geocomputation.

A practical approach was deemed as the most fitting method to answer this question. The above question of Representationis further subdivided into 4 follow-up questions:

A *What are the requirements of a dataflow-VPL for geometry?*

B *What can be defined as 'core browser features'?*

C *Per requirement, to what extend can core browser features be used to implement it?*.

Question A and B will be answered subsequently. Question C and D are answered by Section 6.1.

A: Requirements

Based on the vpl research of Section 2.3, any visual programming language must at the very least contain the following aspects:

- a programming language model
  - a representation of the 'variables' and 'functions' of the language
- a visualization of this data model
- an interface to create and edit the language
- a way to provide input data
- a way to execute the language
- a way to display or save output data

Based on popular, existing geometry vpl's (Blender, houdini, Grasshopper) A visual programming language handling 3D data should have:

- A dataflow-type VPL implementation
  - With all computations being pure functions
  - With all variables being immutable
- A method to preview variables storing 3D data
- multiple ways to determine input data (text fields, sliders)

- multiple ways to view output data (text displays, 3D viewers, etc.)
- A method to handle types, or to guarantee type safety

A VPL also has many requirements which cannot be listed, but instead refer to the shaping of the entire application. For example, interactivity a defining factor of a vpl. GUI elements should be as interactive as possible. However, aspects like this are hard to define or measure, and will not be included as part of this component of the methodology.

B: Core browser features

This study defines "Core browser features" as the set of default features implemented by the three largest browser engines.

```
TODO: a pie chart of usage statistics
```

Based on these (FIGURE) market share statistics, the following three browsers engines appear to be the largest:

- Chromium (Chrome, Edge) (Source)
- Gecko (Firefox) (Source)
- WebKit (Safari) (Source)

The set of features common in all three browser engines are documented on (SOURCE: Mozilla). This includes the following set of features relevant for the 3D VPL:

- WebGL (WebGL2, WebGPU)
- 2D Canvas API
- Web Workers
- Web Components
- WebAssembly

4.3 Compilation

The second component of the methodology seeks an answer to the question of Compilation: *To what extend can geocomputation libraries written in system-level languages be **compiled** for web consumption?*

```
REQUIREMENT: libraries containing pure functions exclusively
- no side effects, thats the whole point
```

Making sure a geo-web-vpl is able to make use of native, system-level libraries is a key component, since it will mean access to powerful, industry standard geocomputation libraries like CGAL and GDAL. 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 wasm also may pose challenges:

- wasm promises a 'near native performance' (Source: Wasm). However, this can be quite situational, as multiple studies have shown Jangda et al. [2019] (Source: the bachelor thesis).
- 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 an *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.

Discovering the extend and relevance of these compilation challenges for geo-computation libraries is why the sub-question of Compilation was included in this study.

Two experiments are conduced to answer this supporting research question. The first focusses on making a clear, measurable comparison between compilation methods, where the second experiment focusses on compilation in a practical, realistic scenario.

Both studies limit themselves 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, since this language is likely to be a future choice for geocomputation libraries, and possesses powerful WebAssembly support.

4.3.1 First Experiment

The first experiment compares three different methods of bringing the same geocomputation procedure to the web. This way, quantitative, measurable aspects of these methods can be compared. The following three methods are tested:

- Write the procedure in normal javascript
- Write the procedure in C++, compile to wasm using the `emscriptem` toolkit (Source)
- Write the procedure in Rust, compile to wasm using the `wasm-bindgen` toolkit (Source)

These procedures are all tested within the same web application, using the same data. By taking two different languages, we can distinguish between shortcomings of wasm itself, and the wasm support of a language.

The procedure chosen is a 2D convex hull calculation of a set of sample points. The chosen procedure must be small enough to clearly reason about performance differences, and yet large enough to pose a substantial computational challenge, validating the usage of wasm.

```
Expand upon the procedure
```

The three methods will be compared in terms of:

- performance
- load times
- memory usage

```
- todo: turn features around into assessment criteria
- performance: load times, run times
- current state of webassembly & js. how much faster is it? is it even faster?
  - data translation steps, do they mitigate performance gains?
  - also given the fact that we are doing 'functions on sets'/ declarative instead
      of imperative styles, forced by the format of dataflow programming.
```

The studies on browser-based geocomputation (Section 3.1) appear to have conducted a similar experiment, by comparing the same procedure written in C++ and javascript. However, these studies compared javascript against a native, non-web compilation of C++. This experiment also differs in distinguishing between wasm itself, and a language's wasm support.

4.3.2 Second Experiment

The second experiment is 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 will be conducted by attempting to compile both libraries using their respective wasm toolsets, and noting the differences in workflow, supported features, and the resulting wasm library.

we wish to compile these languages without 'disturbing' them: they must be kept the exact same for normal, native usage. We will instead create 'wrapper' libraries.

Library One: CGAL

The first library tested is CGAL, written in C++, compiled using `emscriptem`. CGAL will be used as an exemplary C++ library. 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 related works will be encountered. We could choose more simple libraries, but this will not be representative of most C++ geoprocessing libraries.

Library Two: Startin

The second library tested is the Startin library, written in Rust, compiled using `wasm-bindgen`. This library is both smaller in scope, and less well-known than CGAL. Ideally, a library with a size and popularity comparable to CGAL should have been chosen. However, Rust is still a relatively unknown language in the field of GIS. Startin was chosen, for the triangulation functionalities it provides are comparable to that of CGAL, in terms of performance, and geometric robustness (Source).

4.4 Loading

In this third component of the methodology, we wish to discover how the web-exposed geocomputation libraries of Section 4.3 can be utilized within the 3D VPL of Section 4.2. This

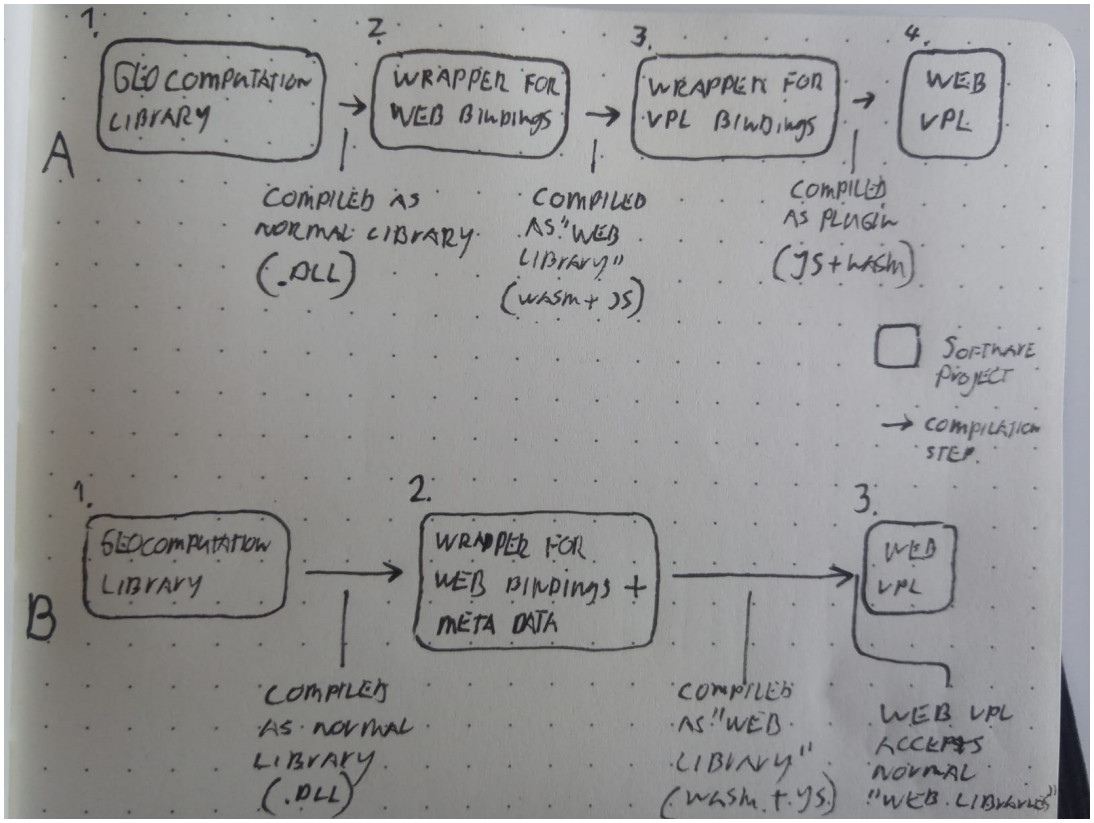


Figure 4.1: Loading Trajectories

is once more a crucial aspect for the success of the entire geo-web-vpl, and captured by the research question: *To what extend can a web-consumable library be* ***loaded*** *into a web-vpl without explicit configuration?*

Most of the 3D vpls mentioned in Section 2.3 offer a plugin system, or some other way to load external libraries. This way, the functionalities of the environments can be expanded upon. However, all of these plugin / library systems require explicit 'wrapper' libraries, to explain how the functionalities a text-based programming library map to components used in a visual manner. This turned out to be a problem during preliminary studies. If a geo-web-vpl wishes to use non-js libraries, it would mean that these libraries would have to be wrapped twice (see Figure 4.1 A): Once to expose the native library to the web using the methods described at Section 4.3, And once more to map the web library to the visual language. While this is a possibility, in practice, two layers of indirection are not acceptable in terms of a development workflow. This would be cumbersome, prone to errors, and hurting version control by having to synchronize between 4 software projects.

There is also a second reason for critically addressing the way plugins are loaded. An observation was made from studying the existing geo-vpls in Section 3.2: It seems that if a developer wants to create custom VPL components, they are required to write plugins very specific to that particular VPL. This means that practically, the library ecosystem of a VPL is entirely its own: It is separated from the wider context of textual programming libraries. End

users are at the mercy of developers implementing their libraries in the dialect their particular VPL. Meanwhile, developers are forced to implement and support a multitude of wrapper libraries for VPL platforms.

In contrast, if the library loader of a VPL was able to directly utilize textual libraries, the barrier between vpl ecosystems and regular text-based libraries would cease to exist, benefiting both developers and end users. It might even lower the barrier between visual and textual programming in general, making it easier for VPl end-users to adopt some forms of textual programming, and vice versa.

These two reasons are why this third component of the methodology is focused on exploring and assessing a method to mitigate the need for the second wrapper library. The following plan was used:

- Design a library / plugin model for the prototype geo-web-vpl
- Implement this library model
- Assess to what extend it mitigates the need for explicit configuration
- Assess to what extend this creates seamless interoperability between textual programming libraries, and VPL libraries.

The design is given subsequently, the build implementation and assessment can be found in Section 5.3 and Section 6.3.

4.4.1 Design & Method

TODO: I should elaborate the design some more

The library model consists of two components: the design for a geo-web-vpl library, and the design for a loader on the side of the VPL. The central idea for this model is to take the wasm-wrappers created in Section 4.3, and to either interpret the required information from the wasm binary and related files, or, if that is impossible, add the required information in the wasm wrapper library itself as meta data. By doing so, we make sure that at the very least, only one wrapper library is required for exposing any non-js library to the geo-web-vpl. 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, named.
- A list of all custom types (structs / classes) present in the library, also named.
- Per function:

  A list of all input parameters, name and type.

  An output type.

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

- Per function:

  A custom, human-readable name.

  A description to explain usage.

- Per type:

  A custom, human-readable name.

  A description to explain usage.

  A definition of how to serialize and deserialize this type

  A definition of how to render this type in 2D or 3D

  A definition of how to convert this type to basic types present within the geo-web-vpl.

The necessity to automate loading geocomputation libraries means that the geo-web-vpl needs to be able to extract this information automatically.

4.5 Utilization

```
CUT OUT THE USE-CASE APPLICATION ASPECT.
-> we can make a utilization assessment based on the criteria alone, We dont need
    to create an application.
```

The final component of the methodology is dedicated to overcoming the fourth and final challenge to realizing a geo-web-vpl, and involves the utilization of all aforementioned components. In this section, we wish to discover the practical usefulness of a geo-web-vpl, encompassed by the research question : *To what extend can a 'geo-web-vpl' be **used** to create geodata pipelines?*

This component of the methodology is included in the study because of the following: It might be the case that a geo-web-vpl *is* able to represented by a web-browser, and *is* able to load and run functions from native, non-js geo-computation libraries. And still, it might not be able to successfully *use* these libraries. The entire idea of a vpl might not be sensible for the operation at hand, or some other, unforeseen aspects mitigates the practical usefulness of the environment. It is therefore vital to access the actual usage of the application for accessing a geocomputation library.

To answer the question of Utilization, the various applications created within Geofront will be subjected to a qualitative assessment.

4.5.1 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 2.3, 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 section presents the (software) implementation details of the methodology.

First, Section 5.1. Second, Section 5.2. Then, Section 5.3. Finally, Section 5.4.

5.1 The Geofront Application

To implement the component of Representation, the core of the prototype application had to be created. To the best of the authors knowledge, no web-based dataflow-VPL project could be found to serve as a starting point. This is why the full application had to be created from scratch (actual scratch, not to be confused with the scratch language: (SOURCE: SCRATCH LANGUAGE)).

This section explains the significant components of this prototype application. 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` : https://github.com/thegeofront . The `app` repository holds the main Geofront application. It contains the source code for most aspects, such as the basic UI, the visual programming interface, and the standard components. The application is hosted at (https://thegeofront.github.io/). The `engine` repository is the library used for the 3D visualizations, some of the logic within the graph components, and some helper functions. The repositories prefixed with a `gfp` are Geofront Plugins. These plugins are consumed by the main `app`.

The Geofront Application (`app`) is set up using a `TypeScript` codebase. It uses `webpack` to compile this to a singular javascript file, and this file practically represents the full application. the repository used around 9.000 lines of code, divided into core categories and functionalities:

I `shims`: Models to reason about 'programming language features', such as functions and variables.

II `nodes-canvas`: Contains the model, renderer and controller of the visual programming graph itself.

III `modules`: holds all classes dedicated to dynamically loading plugins.

IV `std`: The default plugin, baked into Geofront.

V `html`: a 'web framework' of reusable html components making up the UI visuals.

VI `menu`: a menu system which takes care of the model & logic of the UI.

VII `viewer`: The 3D renderer accompanying Geofront.

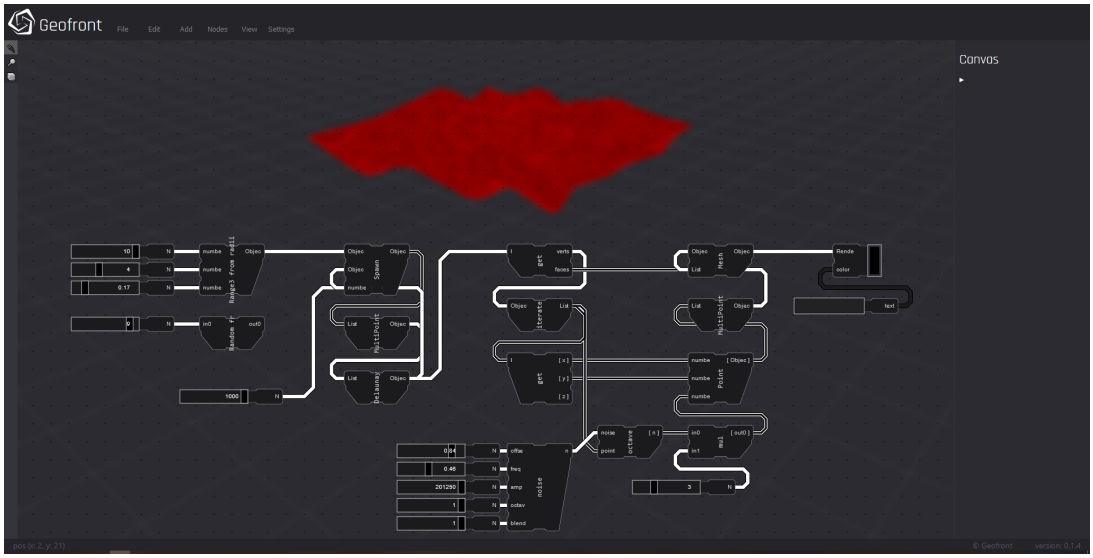


Figure 5.1: The Geofront Application

VIII `util`: Utility functions.

What follows is a clarification of some of these categories.

5.1.1 Shims

Since Geofront is partially a programming language, a way was needed to reason about some of the features of a programming language, such as functions, types, variables, and modules / libraries / plugins. This is why the `FunctionShim`, `TypeShim`, `VariableShim` and `ModuleShim` classes exist, respectively (see Figure 5.2). For example, the `FunctionShim` offers the name, description, number of inputs, and number of outputs of a function, as well as ways to invoke the function it represents. These inputs and outputs have corresponding `TypeShims`, which are formal representations of TypeScript / JavaScript-based types. TypeShims can be structured recursively to define a `List of List of strings` for example. The `TypeShim` is also used to reason about type checking and type compatibility.

the shims are a set of classes used in almost all other categories within Geofront. The shim classes are designed using the Object Type design pattern (SOURCE: BIG FOUR). In short, this means that one function corresponds to exactly one `FunctionShim` instance, and that this instance is shared with any object wishing to use the function. The name `shim` was taken from `wasm-bindgen`'s (SOURCE) naming convention for the auto-generated `JavaScript` and `TypeScript` files. These classes were first created as a representation of those shims.

5.1.2 Nodes Canvas

From the Shim models, the main model of the visual language can be constructed. The `nodes-canvas` category contains all logic & visualization of the visual programming graph,

```
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 5.2

and writing this category was equivalent to developing the visual programming language itself. The other categories of the repository can be understood as auxiliary to this one.

Representation

The architecture of Geofront's visual programming language is first and foremost a Model View Controller setup. The Model is at its core a Directed Acyclic Graph (DAG). This DAG is an object-oriented, graph-like representation of the data flow of a regular programming language.

This architecture is implemented using multiple classes. The Viewer and Controller are represented by the `NodesCanvas`, which has access to a `Graph` Model, containing `Nodes` and `Cables`.

Nodes are analogous to the "Functions" of normal programming languages. As such, a Node knows about the function they invoke 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` (SEE PICTURE). 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.

The Cables are Geofronts analogy to normal "Variables". 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 (SEE PICTURE).

From these ingredients, a graph can be constructed. To reason about this graph as a whole, the `Graph` class was introduced. It contains methods and functions to add or remove nodes, to add and remove cables between nodes, to parse the graph from and to a json, and to calculate

(a) (b) (c)

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

the graph. This study chose to centralize most logic to this Graph, instead of adding complex logic to individual Nodes. This centralized approach was deemed more clear: Many of the required functionalities cover multiple nodes, and this is better presented from an "overview" perspective of theb 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. Even though the JavaScript runtime is garbage collected, it is still subject to memory leak issues or runtime errors. Substituting references with ids makes it easier to prevent these types of problems.

Finally, this entire graph object is accessed by a larger `NodesCanvas` object, and this canvas has the responsibilities of "View" and "Controller".

RenderingE

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

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

The disadvantage of the Canvas API is that it uses many CPU-based draw calls. it is primarily CPU based, Making it much less fast and efficient than WebGL for instance, which is based on the GPU. This is quickly noticeable within interactive applications which need to redraw often, such as Geofront. The Canvas is refreshed and redrawn often, and does not distinguish between unchanged and changed features. This comes down to a performance linear to the amount of nodes and cables drawn. For the current implementation and scale of Geofront, this performance is acceptable.

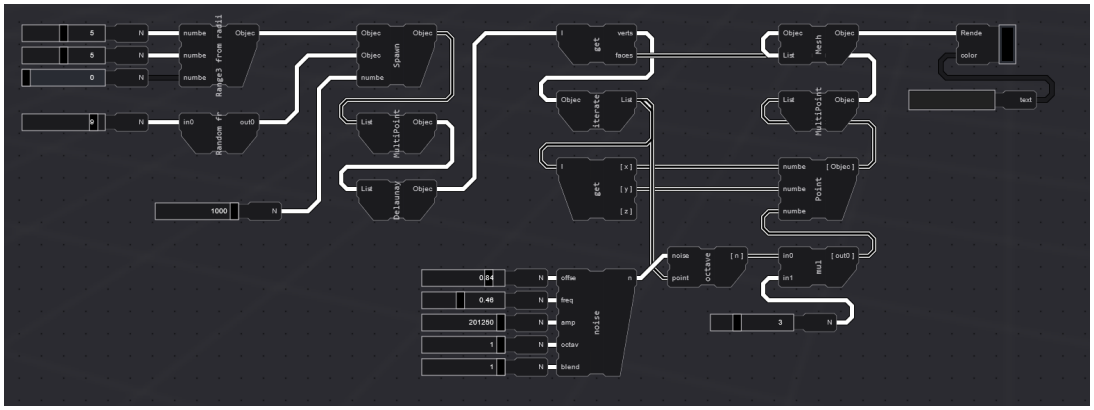


Figure 5.4: A complete Geofront Script

Style

Special care has been put into the stylization of the graph. The visuals take inspiration from various geometry VPLS, such as Blender's GeometryNodes, McNeel's Grasshopper, and Ravi Peter's GeoFlow (SOURCES). A few notable exceptions, however. Firstly, the entire graph is placed on a grid, and all nodes strictly 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 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, as an attempt to make connections between nodes more readable. This takes some additional inspiration from subway maps, as well as the design of computer chips. This makes for a good fit, since both these spatial configurations and the Geofront Script are focussed on organizing connectivity.

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).

Graph Manipulation

In order to support these features, especially undo / redo support, we are required to explicitly track the history of the graph. This is why a Command Pattern (Big Four, SOURCE) was implemented. Instead of directly editing the graph, all actions are represented by `Action`

```
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 5.5: Khan's algorithm in pseudo code

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 we decouple the graph and controller, only allowing interaction with the graph by giving this bridge Action objects. The Bridge maintains a stack of undo and redo actions, which represents this history.

Calculation

When regarding the Geofront graph, 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 we wish to calculate the result of the graph, these dependencies must be taken into account. We must sort the functions the graph in such a way that all dependencies are known before a function is calculated. This problem is known as a topological sorting problem, and can be solved using Kahn's algorithm (Section 5.5):

Using this algorithm for calculation has several important qualities. First of all, it detects cyclical graph patterns without getting trapped within such a loop. Graphs implemented on the basis of an event-system suffer from this drawback, and models like these 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. It was this aspect which enabled

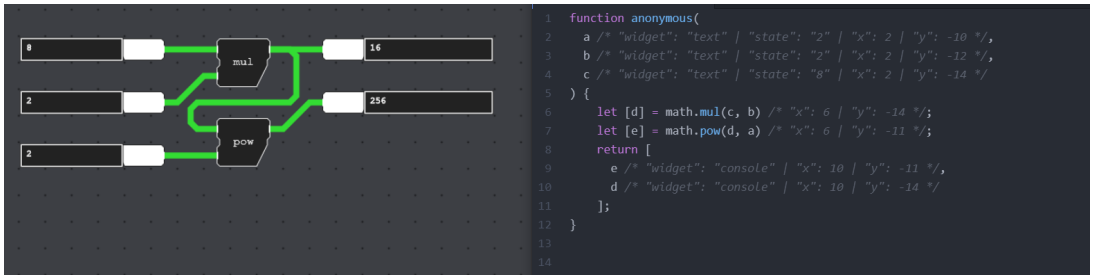


Figure 5.6: An early build of geofront, showing compilation to javascript

an earlier version of Geofront to compile a Geofront Script to Javascript at runtime (see Figure 5.6).

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 required functions will recalculate.

5.1.3 3D Viewer

Finally, the VPL requires some way of visualizing 3D geometry, so that in-between products containing spatial data can be viewed.

This study sought to test the browser itself, rather than to test the functionalities of applications layered on top of it, in order to make benchmarks as 'clear' as possible.

This is why the application makes use of a 3D engine written from scratch to visualize the geometry. It goes beyond the scope of this thesis to explain its implementation details, but it can be seen as similar to Three.js (Source). 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.

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 yet implemented, for these are reliant upon the compilation of existing geocomputation libraries.

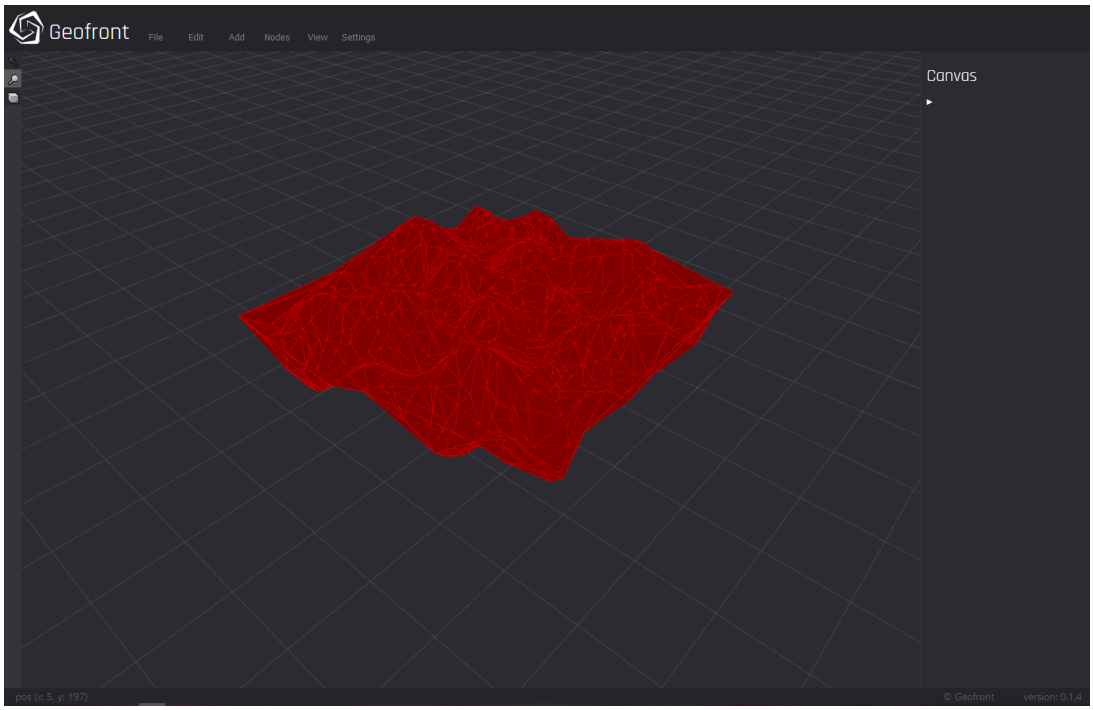
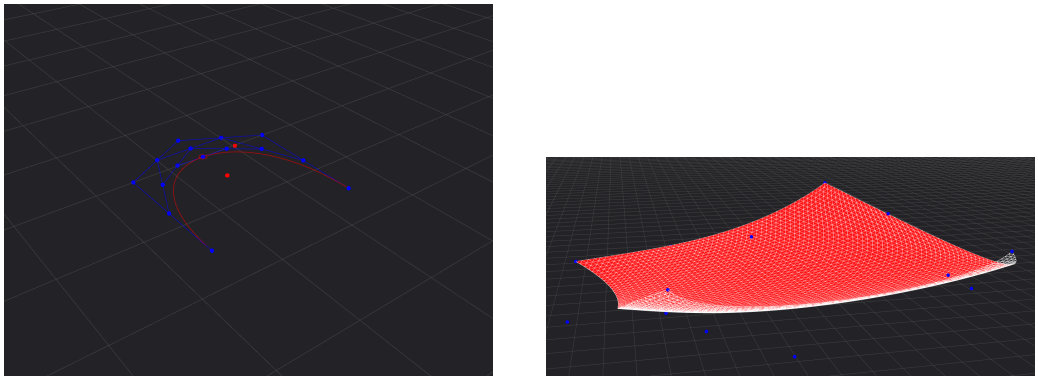


Figure 5.7: The Geofront Viewer



(a) (b)

Figure 5.8: Some a bezier curve and surface visualized using the geofront viewer

5.2 Web-ready geoprocessing libraries

5.2.1 First part

```
TODO write a piece of code in three ways, compare them
```

5.2.2 Second part

```
Explain the process of taking a C++ / Rust library and putting it onto a web-browser.
```

C++ & CGAL

```
- sub-dependencies: boost, ..., ...
```

Rust & startin

```
Startin is by no means a replacement of CGAL,
but it does use robust geometric predicates, and a high precision kernel (f64).
```

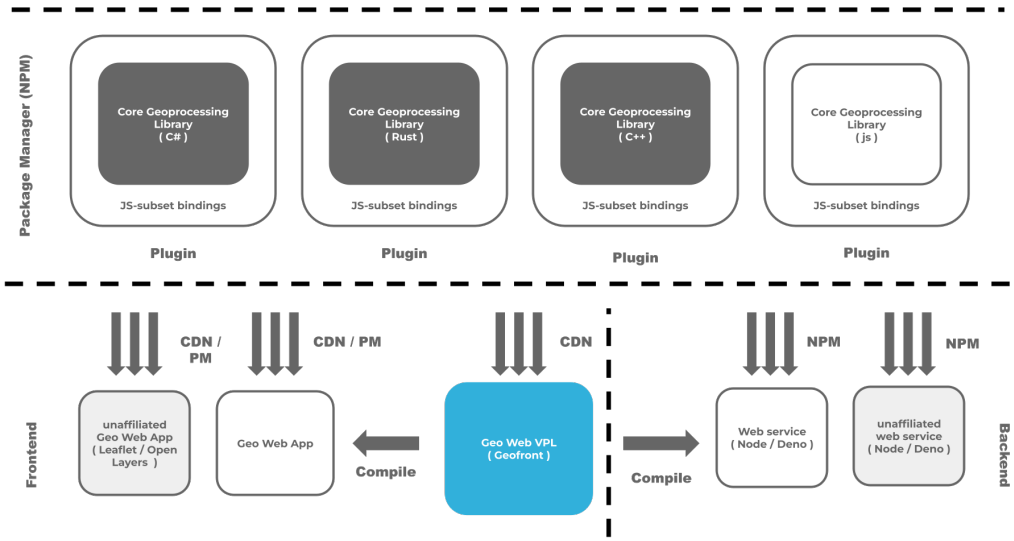


Figure 5.9: Plugin Model

5.3 The plugin model

5.3.1 Geofront's Plugin Loader

Geofront's plugin loader was implemented according to the specifications layed out at Section 4.4. This specification notes how certain **required** and **optional** information has to be extracted from each imported library, in order to properly use them.

The automated extraction of all **required** information is implemented 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 Geofront, this header file could be loaded and interpreted to find all basic information, including the names, the namespace location of where to find a functions, and all input and output types. This extraction of types was crucial, since these are not present in javascript source code, and types are crucial in explaining to the end-user how to use a function, and in making Geofront typesave.

With the extracted information from the "d.ts" file, its corresponding Javascript file could now be traversed and read. javascript's nature as a scripting language proved to be highly effective 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 allowed the plugin loader to localize and collect all functions within a library. And lastly, the "first-class function support" allowed these functions to be referenced by the nodes on the Geofront Canvas.

Because Geofront is 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

Figure 5.10: TODO: show importer side-by-side

Figure 5.11: TODO: show more achieved functionality

to use. Geofront's loader supports 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 was exposed by using a 'magic methods' strategy, influenced by the python programming language (SOURCE: PYTHON MAGIC METHODS). The library loader of the VPL will load certain functions, types, and methods in a special way, indicated by a naming convention. These functions are 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.

5.3.2 Achieved functionality

A loaded library can now be used inside Geofront to create nodes on the canvas. These nodes have inputs connectors for each input argument, and output connectors per return value (Multiple outputs connectors are created from tuples).

The result of this loader architecture, is that libraries can now be created which can be used by both Geofront and (web) applications in general: Just one API is able to satisfy both. This also means that Geofront can use the existing infrastructure of 'normal' javascript libraries, and that sometimes, Geofront can use a library which was never intended to be used in a visual manner.

5.3.3 Limitations

The plugin loader can technically load any javascript / typescript library. In practice, however, there are some limitations 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. In practice, the abundance of javascript bundlers do not make this limitation a "dealbreaker", and also most wasm compilers work in a manner compatible 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 even classes, not all types of libraries and programming styles are supported. Functions using callbacks, 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.

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 (Source). This wrapper handles type conversions and memory safety.

  However, rust libraries compiled to the web use have to be initialized. 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 5.3) 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 library.

Figure 5.12: TODO: show the achieved workflow visually

- **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.

5.3.4 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 "triples" as also a native geoprocessing library. The Geofront standard library is implemented using workflow with Rust. This workflow turned out to be very useful for the purpose of rapid experimentation.

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

Using Rust:
1. Write 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'.
3. Compile to `.wasm` + `d.ts` + `.js`.
4. publish to npm (`wasm-pack publish`)

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'*.
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:
4. Reference the CDN (content delivery network) address of this node package.
5. Use the library!

* these parts contain many caveats, explained in Section 5.2
```

5.4 Utilization

```
TODO: I might want to add some example applications.
Then those will be added here.
```

6 Analyses

In this chapter the various software implementations and design choices made in Chapter 5 are analyzed. First, Section 6.1. Second, Section 6.2. Then, Section 6.3. Finally, Section 6.4.

6.1 The base application

This section of the analysis covers the question of Representation, and aims in particular to answer the question: *Per requirement, to what extend can core browser features be used to implement it?*.

We group the requirements listed at Section 4.2 in a group of base features, a group of dataflow features, and a group of geometry features.

6.1.1 Base dataflow VPL Features

```
% \begin{enumerate}[-]
%   \item a visual language
%   \item an interface to configure this visual language
%   \item a representation of the 'variables' and 'functions' of the visual language
%   \item a way to provide input data
%   \item a way to execute the language
%   \item a way to display or save output data
% MAKING TYPESAFE CONNECTIONS
% EXPLAINING THINGS TO THE USER

TODO elaborate upon each of these features
Add type checking.

"Types in general have not been solved across all languages, so we cannot just do
    that here. Also, how to communicate ideas about types in an end-user-friendly
    manner is also a sizable topic"
```

Geofront was able to implement all normal, basic features of a dataflow VPL in the browser using JavaScript and HTML5 features.

The Canvas API turned out to be a crucial feature to make the base UI functional. The same level of interactivity would have been difficult to achieve using just plain html. However, The CPU-based rendering can get slow at a high number of components.

6.1.2 Geometry VPL Features

```
% should have:
% \begin{enumerate}[-]
%   \item A method to preview 3D data used throughout the flowchart

%   \item multiple ways to determine input data (text fields, sliders)

%   \item multiple ways to view output data (text displays, 3D viewers, etc.)
% \end{enumerate}

  TODO elaborate upon each of these features
  Add type checking.

  "Types in general have not been solved across all languages, so we cannot just do
      that here. Also, how to communicate ideas about types in an end-user-friendly
      manner is also a sizable topic"
```

Geofront was able to implement most features of a geometry-focused VPL using browser features, however there are some caveats.

To support features like loading or streaming a file, or inspecting data, the browser needs to take the full file into memory. Additionally, internal representations of geometry suffer from Javascript's limited type support. Typescript has been used to mitigate some of these shortcomings, this still leads to trouble when type-checking at runtime. The prototype-based objects also led to classes which could not be properly reflected at runtime. These two aspects together makes interpreting what geometry to visualize and when a difficult exercise.

however, what Geofront gets in return, is the fact that make aspects such as WebGL do not need to be included within the source code of Geofront, making it much more lightweight than an application which carries its render API with itself.

6.2 Compilation Process

6.2.1 First part

```
[Show the performance benchmarks in isolation. Also show them in action on the VPL
    canvas]
```

6.2.2 Second part

```
REQUIREMENT: libraries containing pure functions exclusively
- no side effects, thats the whole point


HOWEVER
- existing geo-libs:
  - file IO focussed, (because of streaming \& big data)
  - templates
  -

[Analyze the web-exposed geoprocessing libraries]


The catch 21 between C++ and Rust.

**C++ -> wasm is difficult**
```

```
- requires a lot of trivia knowledge.
  - Often, many subdependencies need to be traversed, and makefiles need to be
      manually edited.
  - This cannot be done often, or easely.
- Documentation is behind compared to Rust.

- Libraries cannot be called. emscripten prefers a cli-type interface, and prefers
    you to write all the operations you wish to publish as separate command line
    applications.

https://www.hellorust.com/demos/add/index.html
https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html


**rust-> wasm is powerful and feature-rich.**

- it functions like a normal compiler. compilation was done easely.
- If a library cannot be compiled, the compiler statements are clear enough to
    identify which library causes the incompatibility.
- Many Rust libraries also conveniently offer a 'no-std' option,
  And many crates (Rust equivalent of a library) include the web as one of their
      core targets.
- The compile tool 'wasm-bindgen' has excelent support for converting libraries.
  - You can create rust libraries which from the outside look like a normal
      javascript library.


  - figure out the difference in load time and performance between startin's tin,
      cgal's tin, and my dumb triangulator.
```

```
where C++ requires emscripten, rust does that almost out of the box, (rustup target
    add wasm32-unknown-unknown), NOTE C++ also does that with CLANG and LLVM

What we actually need to compare is not emscripten and wasm-bindgen, but 'embind'
    and 'wasm-bindgen'.
```

6.3 Loading plugins

In this section is meant to assess to what extend Geofront's plugin loader mitigates the need for explicit configuration. This analysis is based on the achievements and limitations layed out by Section 5.3.

To what extend does the plugin loader mitigate the need for explicit configuration of plugin libraries?

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 5.3.3 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, well, optional, one could argue that some of these properties are in fact required. Libraries without 'human-readable' names and descriptions are harder to utilize on the Geofront Canvas. 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.

Assess to what extend this creates seamless interoperability between textual programming libraries, and VPL libraries?

Because of the reasons outlined above, it is save 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 scenario's, 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 incompatible types, or have an incompatible architecture. In some cases, a library might be able to load, but is then functionality impaired by the interface. Figure 6.1 is an example of such a case.

6.4 Usability

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

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

```
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 6.1: This typescript file technically loads correctly into Geofront, but cannot be reasonably used on the Geofront canvas. TODO: show what this looks like on canvas

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.

Suggestion for improvement: re-implemented the 'compile to js' procedure.

(image of abstracting away a javascript subset)

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.

(image of iteration problem)

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.

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.

Suggestion for improvement: Stabilize the api of the Geofront Standard Library.

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 mathematical calculations.

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

(image of complex / simple mathematical calcluation in javascript and in geofront)

Suggestion for improvement: These situations could be prevented by allowing scriptable components.

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 2.3.3, 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. Therefore,

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.

(Image: Show example)

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?'"

sizeable phenomenon

grootste boosdoener: looping is now a simple boolean toggle within a component. This makes it very non-explicit,

This leads to another problem: the problem of declarative iterations within a vpl.

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.

Suggestion for improvement: Provide a way to annotate: create groups, write comments, etc. **Suggestion for improvement:** Type colors would also be nice.

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.

Suggestion for improvement: 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. As the question implies, by not making the code dispersed,

7 Conclusion & Discussion

This chapter consists of the answers to the study. questions that were given in the first chapter (Section 7.1), a summary of the most significant contributions (Section 7.2) and the limitations of these contributions (Section 7.3), a discussion of the value and quality of this study (Section 7.4), a number of theorized implications of this study (Section 7.5), and lastly, a self reflection (Section 7.6).

7.1 Conclusion

This paragraph 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. *"To what extend is the browser capable of **representing** a generic dataflow-vpl for processing 3D geometry?"*

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, and the analysis in Section 6.1. The browsers biggest advantage for an application like this is the sheer 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.

JavaScript can also be used to represent the data structure and logic needed to make a VPL functional. Javascript's flexibility proved to be crucial to support features like dynamically loading and using libraries at runtime. The disadvantage was due to JavaScript's very limited type support, and limited precision in general. While the language does offers powerful options for reflection, this cannot truly be used if javascript itself makes no distinction between different number types (`int, float, double`) for example. Typescript has been used to mitigate some of these shortcomings.

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

Based on the experiments and analysis in Section 6.2, 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 the focus of the emscripten compiler. The tool can be

used to compile full-scale C++ applications, and even includes an emulation of a POSIX environment. However, it severely lacks support for compiling libraries themselves, compared to other wasm-library compilers. libraries generated with emscripten's 'embind' tool severely lack ergonomics, and thus would not be directly loadable in a geo-web-vpl. 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 using pure functions. Additionally, many scientifically oriented C++ libraries like CGAL make extensive use of meta programming and template programming, paradigms which do not translate well to an environment outside of C++. Finally, the study was able to recognize some discrepancies between the novelty of the WebAssembly format, juxtaposed to 50 year legacy of the C++ language.

Despite all of 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 'catch 21' of some sorts. Rust is for the forseeable future a better choice for writing easily consumable, platform-independent libraries, but does not have a 30+ year legacy of existing geocomputation libraries. On the other hand, C++ does have these libraries, but lacks proper wasm-library compilation options.

To overcome this, 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.

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

Based on the method described in Section 4.4 and Section 5.3, and the analysis of Section 6.3, 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. *"To what extend can a 'geo-web-vpl' be **used** to create geodata pipelines?"*

Based on the analysis of Geofront in Section 6.4, it can be concluded that a geo-web-VPL can be used for geocomputation to a sufficient extend. 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 a VPL be used to support and execute existing geo-computation libraries in a browser?"

A VPL can support existing geocomputation libraries if and only if these libraries are able to be *compiled*, *loaded*, *represented*, and *utilized* in a 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 VPL capable of constructing scripts from 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 sufficiently *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.

7.2 Contributions

The full extend of the results of this study is represented by this section, as well as Section 7.3.

- **A new implementation of a web-based visual programming language for geocomputation**

  By providing the full source code of the application and all libraries used within the application, 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.

  Additionally, by having conducted this study on the intersection of all three of those fields, the study shows in what way these fields can be of help to one another.

- **A novel workflow of publishing and using native libraries on the web**

  Using the environment, you can take a rust-based geo-computation function or library, and without very many steps, use it within a visual programming environment. In software development, this can be used to lower the delta between "it works for me" and "it works for someone else". The environment also be used to:

  Visually debug,

  fine tune parameters of unintuitive / empirical algorithms,

  Compare performance of similar libraries,

  Load libraries with a minimum of configuration steps: Any Javascript, Typescript or Rust library which satisfies the conditions layed out in Section 5.3.3, automatically works in Geofront.

  And, unique to this environment, do this all online, in a 'published' format: the full configuration can be shared using a URL.

  This combination of features makes Geofront unique among both geo-VPLS and web-VPLS.

7.3 Limitations

The contributions mentioned above to have limitations, which can be described as follows:

- **Only Rust, Js & Ts library support** For now, only 'Rust' and 'JavaScript / TypeScript' can be properly used as libraries. However, most libraries relevant to geo-computation are C++ based. While C++ has excellent support for compiling full, self contained applications to WebAssembly using the 'emscripten' toolset, it lacks rust's level of support in compiling existing libraries.

  In Section 6.2, The issues between compiling C++ and Rust libraries were given. Since a stable method of using C++ can not be provided for at the current moment,

- **In practice, not all libraries can be used** - While it is indeed possible to use and run any rust library with 'wasm-bindgen' annotations or any js + ts library, in order to properly communicate, visualize, and make data interoperable, special 'config' functions and methods are needed.

- **Only small-scale geodata is possible** The 'near native performance' has some caveats, as explained in Section 6.2, but is not a problem at large. What is a problem is the fact that geodata is in general large. - the environment uses browser-based calculations, so it cannot be used properly for big data, or other expensive processes. Future work: compile the flowchart, run it headless on a server for large datasets.

- **Implementation shortcomings** Its still a prototype, and has many usability shortcomings, explained in Section 6.4. In particular, many geocomputation-specific aspects are missing.

7.4 Discussion

Questions on the decisions made during the study, and the answer this study is able to provide.

Q: Geodata is big data. Will this web application be scalable to handle big datasets?

One of the problems to address when considering the ergonomics of geocomputation, is the fact that geodata is almost always big data. A web application cannot be expected to process huge datasets. So how does geofront address this fundamental aspect of geoprocessing?

First of all, the purpose of this study was only to get geocomputation libraries to the web, and inside of a vpl format. Scaling the application up to handle big data was not part of this study, and had to be left to future work.

But still, lets give the devil his due: Even when processing "smaller" datasets of, lets say 4 GB, most of the 'flowchart niceties' of geofront will cease to be useful. Inspecting this data will take more time than its worth, and reconfiguring the flowchart will take a long time. This can be mitigated by using web workers, but it will still not be very ergonomic to work with.

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 Node.js or Deno, without any dependencies to Geofront itself. While it seems convoluted to compile a native library to WebAssembly and then execute it natively again, it is actually a sound strategy for building scalable, containerized programs for the cloud. The 'WASI' project is a good example of this (Source: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/). 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 constrains.

Q: Why not build everything as a native application, and publish the entire thing as wasm?

This would indeed be more performant, especially if the native version of the application is also distributed. Users could choose themselves if they wish to sacrifice the performance and native experience for accessibility on the web.

However, many features key to the solution and workflows proposed would be lost in a setup like that. Geofront would then be 'just' a geo-VPL. (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. Additionally, all features Geofront gets 'for free', like the DOM, the Canvas API, and WebGL, would have to be included within the application.

Q: Where was this 'barrier' the methodology spoke of?

Recall that the methodology states how the barrier preventing geo-web-VPLS from adapting native libraries, must be because of issues encountered during **compilation**, **Loading**, **Representation**, or **used**, or a combination of these factors.

After the conclusion of this study, three major issues can be identified, which indeed occupy spaces in between the above four factors.

1. Compilation and Loading

Firstly, it turned out to be challenging to expose existing libraries in a way usable by a dataflow-vpl. - "just compiling to wasm" was not enough. - Webassembly is a double edged sword. Interfacing with wasm binaries from javascript is slow: lots of duplication of data. -¿ catch 21 beyween rust and C++

2. Disconnect between textual and visual programming

Secondly, to turn a 'normal' function into a component usable in a dataflow graph, additional metadata needs to be specified. This leads to specific config files and classes, which in turns creates a barrier between regular programming libraries, and vpl libraries. Section 5.3 represents the attempted solution to this problem.

3. Web interfacing

And lastly, having limited access to the file system really hurts the usefulness of the vpl as a data processing application. This aspect has not been properly addressed during this study. The cloud-native geospatial movement might pose a solutions to this aspect, as well as the brand new

This study recommends cloud-native as a solution to this problem, and has added

Q: Is a geo-web-vpl the same as a 3D vpl with existing geoprocessing functionalities attached? In other words, is geoprocessing nothing else than procedural modelling?

No, but it is a good start. A geo-web-vpl is *at the very least* a Geometry VPL. Ofcourse, an actual geocomputation vpl might require more features, such as a global CRS, support of base-maps, more control on precision, etc. These important aspects must be left for future work.

Q: Usage: Who benefits from a web-geo-vpl, and how?

This study proposes 4 use-cases:

- Educational Sandbox
- Web Demo Environment
- End-user geoprocessing environment
- Rapid prototyping environment

Q: Is this environment truly accessible?

Based on the analysis given at Section 6.4, it is safe to say that based on its features, Geofront is about as accessible as comparable geo-vpls, like geoflow or grasshopper. However, this analysis is only based on the achieved functionality and features. Actual user-testing is required to assess The true accessibility of the tool.

Q: Is this environment truly a competitor to native / other methods of geoprocessing?

In theory, yes. Using the workflow as described, native geocomputation libraries could be written, and these same tools could be used on the web at near native performance. Additionally, the web offers enough functionality so that even sizable, local datasets could be processed this way. In practice, the 'catch 21' problem 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 to be 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 earlier, very simple version of the Geofront script had the ability to be compiled to normal javascript (see (Figure 5.6)). All libraries were converted to normal import statement, all nodes were replaced by function calls, and the cables substituted by functions. This way, the application could be run headless (without the gui) either the browser or on a server, using a "Node,js" (Source) or Deno interpreter (Source).

A future study could re-implement this feature, opening up the possibility for deployment and scalability: Scripts created with geofront could then deployed as web applications of themselves, or as a web processing services (WPS, SOURCE). 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 alternative to similar tools like the Google Earth Engine, and FME cloud compute.

7.5.2 Streamed, on demand geocomputation

The study showed that browser-based geocomputation using Rust is entirely possible. This possibility might allow for an entirely new type of geoprocessing workflow, which could replace some use-cases that now require big-data processing and storage. Instead of storing the sizable, pre-processed results of some geocomputation, an application could 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

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 crucial for browser-based geocomputation. However, there are two 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. The second one is that Rust compiled to WebAssembly sees extensive use on both cloud and edge servers. This could be very interesting to the "cloud-native geospatial" movement. This web 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 interoperable, more exchangeable, and Rust compiled to WebAssembly forms an excellent candidate.

Therefore, studying Rust-based geocomputation for cloud native and edge computing, would be an excellent 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-bias 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 amount of time was spend with both languages. If an author was more familiar with one of the two languages, this might have lead to a bias result.

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.

Bibliography

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

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. pages 107–120.

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.

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

Mozilla (2013). asm.js.

OGC, O. G. C. (2015). Web Processing Service.

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

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.

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.

Colophon

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