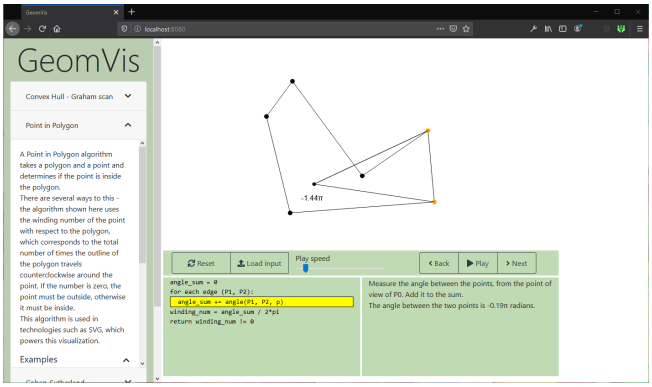
TÉCNICO LISBOA



GeomVis: WebApp for Computational Geometry Algorithm Visualization

Ricardo Miguel Tiago Farracho

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Alfredo Manuel dos Santos Ferreira Júnior

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Alfredo Manuel Dos Santos Ferreira Júnior
Member of the Committee: Prof. Prof. Daniel Jorge Viegas Gonçalves

October 2019

Acknowledgments

I would like to thank my advisor Alfredo Ferreira for his guidance during the development of this work.

I extend my gratitude to my friends and colleagues who helped me along the course of my studies, in particular to my friend Gabriel Maia with whom I could share ideas and get feedback for this work, and many other long-time friends who were greatly encouraging.

I would also like to thank the volunteers who helped me test GeomVis and evaluate the results of my work. I would also like to acknowledge the authors of previous work in this area whose works served as inspiration my own.

Finally, I extend my deepest gratitude to my family, who made it possible for me to further my studies in this field and without whom this work would not have been possible.

Resumo

As ferramentas de visualização têm o potencial de melhorar a perceção da informação. Este potencial pode ser estendido aos algoritmos. Atualmente, existem já várias ferramentas que se aproveitam desse facto.

Estudámos as ferramentas existentes de visualização de algoritmos e identificámos as principais funcionalidades dessas aplicações, como execução passo-a-passo, disponibilidade do código fonte e suporte para input e output de ficheiros. Analisámos também os algoritmos disponíveis nessas ferramentas e identificámos uma classe pouco representada de algoritmos: os da geometria computacional. Nomeadamente, não encontramos quase nenhuma ferramenta para visualizar esses algoritmos.

Neste trabalho, apresentamos uma nova ferramenta de visualização para algoritmos de geometria computacional que desenvolvemos, baseada em tecnologias Web, que está disponível online. A nossa aplicação tem três objetivos principais: ajudar a entender estes algoritmos através da visualização; suportar a execução com dados arbitrários para permitir que os utilizadores explorem livremente o comportamento dos algoritmos; e fornecer uma implementação modificável com código fonte disponível de forma livre (open-source). A nossa aplicação foi testada com utilizadores, e os resultados são promissores.

Palavras-chave: Visualização de Algoritmos, Geometria Computacional, Computação Gráfica, Applicação Web

Abstract

Visualization tools hold the potential to improve the understanding of information. This potential can be extended to algorithms. Several tools already exist that take advantage of this fact.

We studied existing algorithm visualization tools and identified key features for these applications, such as step-by-step execution, source code availability and support for file input and output. We also looked at the algorithms featured in these tools and identified an underrepresented class of algorithms: those from computational geometry. Namely, we find nearly no tools for visualizing these algorithms.

In this work, we present a new web-based visualization tool for computational geometry algorithms we developed, which is available online. Our application targets three main goals: to aid understanding of these algorithms via visualization; to support execution with arbitrary inputs to allow users to explore algorithm behavior freely; and to provide a modifiable implementation with freely available source code. We tested our application with users, and the results are promising.

Keywords: Algorithm Visualization, Computational Geometry, Computer Graphics, Web Application

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Visualization has long been a technique for presenting data and information in a way that is clear and easy to understand, in applications such and diagrams, maps, and data plots.

Visualization tools are widely used for diverse applications, particularly in the fields of data and information visualization – in such domains as meteorology and, increasingly, medicine – and help us grasp large amounts of data at a glance. We will look at several such existing tools for visualizing algorithm execution.

1.1 Motivation

When studying something, be it concepts, facts, or other things, having visual aids can be very useful. Our vision is our most essential and used sense, and so the visual aspect helps our brains take in information.

This concept also applies to algorithms. Indeed, in computer science classes, both online and in the classroom, it is common to see lecturers and authors use hand-drawn or digital diagrams to illustrate program execution while explaining how those algorithms work and what steps they take. A specific example of concrete visualizations are those often used for sorting algorithms. Often, bars are used to represent the items being sorted, with the sort criteria being the height of the bar. The bars are then sorted as per the algorithm, and the steps can be observed. This gives the observer a spatial intuition as to how the algorithm is operating, and can also help compare its speed with others. It also does not require pseudo-code or source code to be understood.

1.2 Problem

Algorithm visualization tools are, to some extent, becoming commonplace. However, we believe that more algorithms exist that are lacking such tools and that can benefit from them. As we will see, most of these tools focus on sorting algorithms, graph-based algorithms, and data structures. For our work, we will instead focus on geometric algorithms. These algorithms have even more to gain from visualization

as geometry is a field that adapts itself well to visualization, due to its nature as the study of planes, spaces, and other such intuitively visual concepts.

Our solution aims to fill the gap in algorithm visualizations by focusing on a few algorithms from computational geometry of varying complexity. As we will see, tools in this area generally have different focuses. As such, we have created a specialized application for visualizing this set of algorithms. We tackled the challenges of developing such an application so that it is accessible and usable by those interested in learning these algorithms. We also approached the issue of designing the application such that parts of it can be reused and the application itself can be expanded easily - for example, by adding more algorithms.

1.3 Objectives

Our solution addresses the problems and motivations by the medium of a web application for algorithm visualization. For this purpose, we set the following objectives:

Deploy a web application that runs in-browser for portability across operating systems and will leverage well-understood and well-supported standardized web technology for this purpose. This allows users to use the application without having to download or install specific new software[1]. This also helps support the other goals below.

Illustrate algorithm execution to facilitate understanding and studying algorithms for those interesting in learning, or for use as a teaching tool. The tool will illustrate the execution step-by-step and will permit the user to change the inputs to the algorithms, to visualize how execution differs based on those inputs. This supports our primary goal of facilitating **understanding** of the algorithms.

Support arbitrary inputs. We want to allow the user to have the algorithms execute with their inputs, such that they can freely explore the algorithms. For this, we made it possible to input data either via the interface or through simple formats such as text files.

Make the code available and open for reuse and derivative works. We want to contribute to the growing body of open source software and to allow others to improve the tool or use portions of it to develop works based on it, for example, by adding more algorithms besides the ones we implemented.

1.4 Thesis Outline

The rest of this thesis is structured as follows. First, we analyze existing tools (related work), in Chapter 2, including identifying key features in such tools. Then, we describe GeomVis, our solution to the problems we identified, in Chapter 3. In Chapter 4, we describe the specific algorithms included in our implementation. In Chapter 5, we analyze the results of testing GeomVis with users. Finally, in Section 6, we will present our conclusions.

[1] aside from a web browser, which is present in nearly all computers nowadays.

Chapter 2

Related Work

In the following sections, we present several existing tools and websites that provide algorithm visualization. Afterwards, we compare these tools, discuss their characteristics, and present our conclusions.

2.1 VisuAlgo

VisuAlgo [1] is a website containing a number of different visualizations for algorithms and data structures. There are many such visualizations (around a couple dozen) and they mostly focus on data structures. The front page is shown in Fig. 2.1, showcasing an appealing design with preview images for the algorithms.

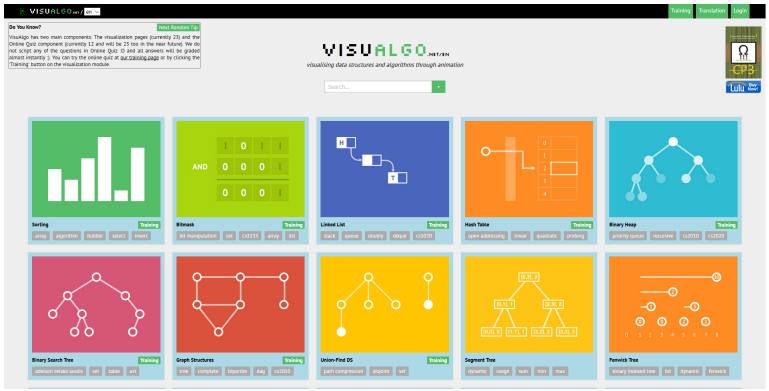


Figure 2.1: VisuAlgo front page

Upon selecting a visualization from the front page, we are shown lengthy textual explanations written for computer science students, including motivations for the data structures, operations on those data structures and brief explanations of the algorithms. Dismissing the explanations allows the user to access the visualization screen where a specific operation can be selected (for example, inserting a key in a hash table). Then, the input can be changed before starting the operation.

Fig. 2.2 shows the application's visualization midway through a hash table insertion. We can see that pseudo-code is shown and the current step is highlighted. Additionally, a textual explanation of

the current step is displayed above the pseudo-code. Finally, we can see speed controls as well as a pause/play button.

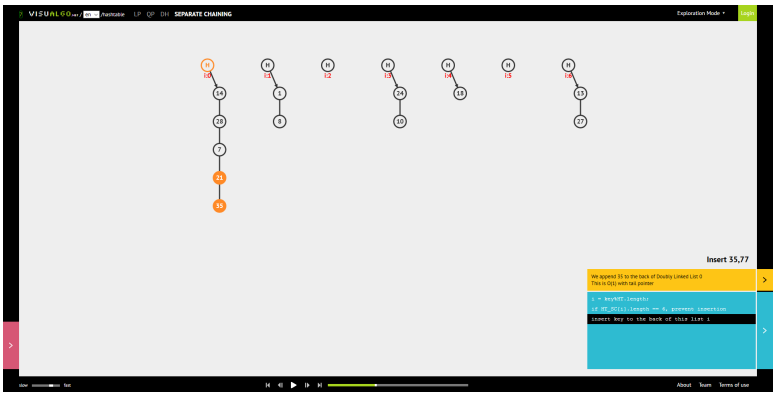


Figure 2.2: VisuAlgo visualization of an insert operation on a hash table

This tool has some limitations. For example, editing the data has to be done using the predefined operations (that is, the same ones being illustrated) so it is difficult to configure the inputs beyond what is shown by default. Additionally, the specification of the inputs varies with the algorithms and has some complexity. Finally, the algorithm results are merely visual, with no option to export the results.

2.2 Algorithm Animations and Visualizations

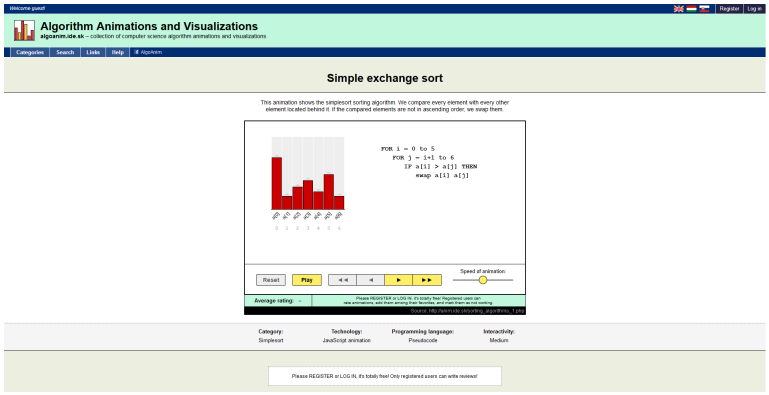


Figure 2.3: Algoanim's illustration of a sorting algorithm

Algorithm Animations and Visualizations ("Algoanim") is a "collection of computer science algorithm animations and visualizations" [2]. Like VisuAlgo, the website features numerous small animations of particular operations and algorithms, mostly around searching and sorting data structures. Fig. 2.3 shows one such visualization.

The visuals are relatively simplistic and disparate in appearance. Some use Javascript, some Adobe Flash and yet others are videos. Only a portion of the animations feature step-by-step or speed controls, pseudo-code, or explanations. The inputs are also pre-defined, which limits the usefulness of the animations, as only the given examples can be visualized.

2.3 Algo-visualizer

Algo-visualizer [3] is a web application featuring visualizations for several types of algorithms, such as graph search, sorting, substring search, and cryptography. For each algorithm, a small description is shown, then the real implementation code is presented side-by-side with an illustration. The app features step-by-step execution and speed controls and highlights the lines of the code as they are executed. Furthermore, the app provides APIs for the included visualizations and allows the user to change both the code being visualized and the visualization code itself. Algo-visualizer is very advanced and allows

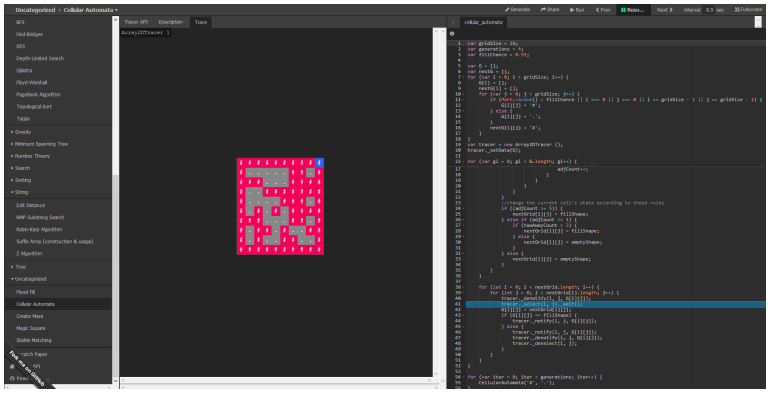


Figure 2.4: Algo-visualizer visualizing a cellular automata algorithm. The highlighted call to `tracer._select` showcases use of the visualization API

technical users to tinker both with the inputs and the actual code. This level of complexity is quite high for a tool to be used for learning but it is the most flexible in features as not only can the inputs be changed but the algorithms themselves. Additionally, the app is open-source, with an open invitation to the user to fork it on GitHub shown in the corner. A running example is shown in Fig. 2.4. The page is divided into three sections: a list of algorithms, the visualization, and code panels. For some of the algorithms, the visualization also shows additional logging output.

2.4 sorting.at

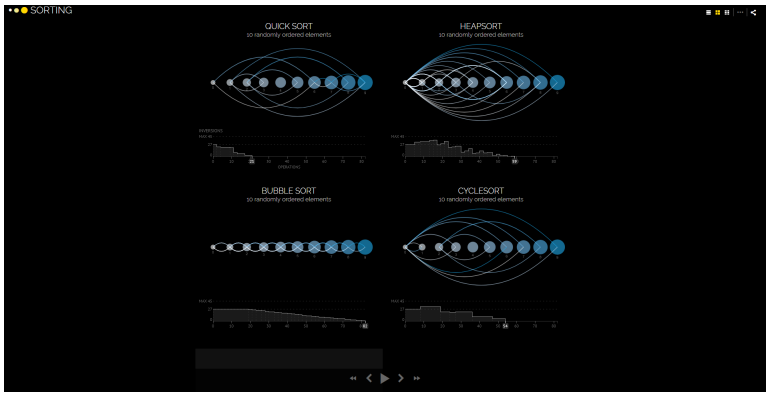


Figure 2.5: sorting.at showing the ”shapes” of various sorting algorithms

sorting.at [4] is a website that illustrates several sorting algorithms in an appealing way. It shows that

the different algorithms can have distinct ”shapes” based on how they switch items, as shown in Fig. 2.5. As the items are sorted, the visualization draws trails representing the paths of the items, leaving a distinct shape. The items are simple circles that are sorted by size. A graph showing the number of operations is also shown under each display, to compare the total work done by each algorithm.

The user cannot select the inputs directly, only choose some aspects, such as the total number of items and initial condition, from a limited set of options. The comparison aspect is not relevant for our work, but it might be worth taking into account the elegant and simple design when implementing our solution so that it is user-friendly and easy to use.

2.5 PathFinding.js

PathFinding.js [5] is a single-page web application for simple visualization of a set of path-finding algorithms. It uses a grid representation for the space of the search, with a green dot representing the starting point and a red dot representing the goal destination. The grid starts with an empty example with no walls; the user must click and drag to draw walls. An example is shown in Fig. 2.6.

There are several choices of algorithm and the parameters are configurable (for example, there several

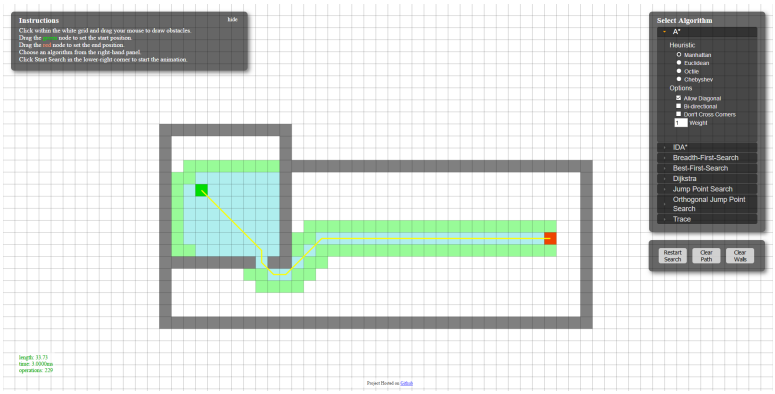


Figure 2.6: Example of an A* search illustrated by PathFinding.js

heuristics to choose from). As the algorithm runs, the cells that are explored are highlighted in different colors. There is no step-by-step feature or speed controls, although the search can be paused. Unfortunately, the lack of such features makes it hard to understand what is happening as the cells are colored and uncolored very quickly, making this application more suitable for illustration than study. The app also uses three different colors for explored cells but does not explain their meanings, so it is hard to understand what the algorithms are doing with the cells unless the user already knows the algorithms well.

2.6 Vamonos

Vamonos [6] is yet another collection of visualizations of algorithms, mostly sorting and graph algorithms. Fig. 2.7 shows an example of one of the visualizations. The interface seems study-oriented, as it

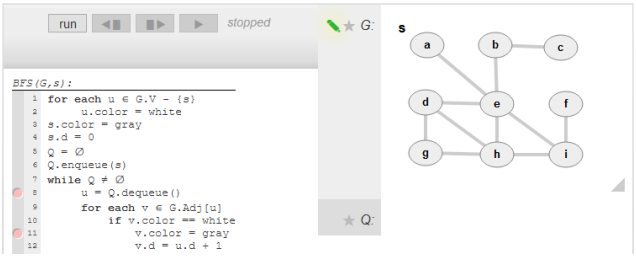


Figure 2.7: Breadth-First Search as visualized by Vamonos

features full pseudo code with step highlighting and step-by-step controls, and precise illustration of the states of the graph nodes. In the particular case of this example, the queue is also illustrated under the graph display and the nodes can be physically rearranged using the mouse. Nodes can also be created, connected, disconnected, or removed before the algorithm is run, although the editing interface is not very intuitive. Interestingly, the pseudo-code panel allows the user to set breakpoints at any line of the pseudo-code and these will be used for the step-by-step controls. This is interesting as it allows the user to skip steps they might not be interested in seeing.

2.7 Data Structure Visualization

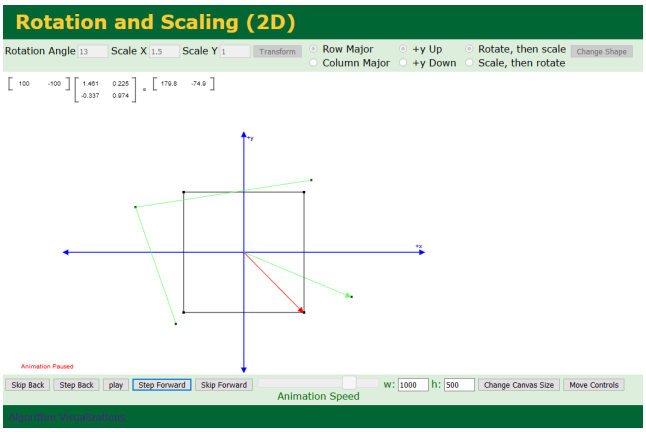


Figure 2.8: 2D rotation and scaling visualization from Data Structure Visualization

Data Structure Visualization [7] is a website featuring a collection of visualizations for very diverse algorithms from the usual sorting and graph algorithms to indexing, recursion and some geometric algorithms. The latter only include basic matrix transformations, however. An example is seen in Fig. 2.8. For some of the visualizations, the input can be changed, but only randomly within a set of fixed inputs. Others have free-form input. This tool also has step-by-step controls like most of the others.

2.8 Algomation

Algomation [8] is an online platform for sharing algorithm visualizations. Users may develop and submit their own visualizations using the provided API, which is a useful feature for advanced users who want

to easily create and share visualizations.

Fig. 2.9 shows one such visualization. The left half contains a textual description of the algorithm and the source code that drives the visualization. The right half illustrates the execution and contains the controls. In the cases we analyzed, the API seems to be quite verbose, which makes the algorithm difficult to study from the code as it is extensive and mixed with visual logic. There also seems to be no mechanism to step through code, only through full iterations. Additionally, user input also requires manually editing the code, which is difficult for beginners and non-programmers. The quality of the visualizations themselves varies depending on the author.

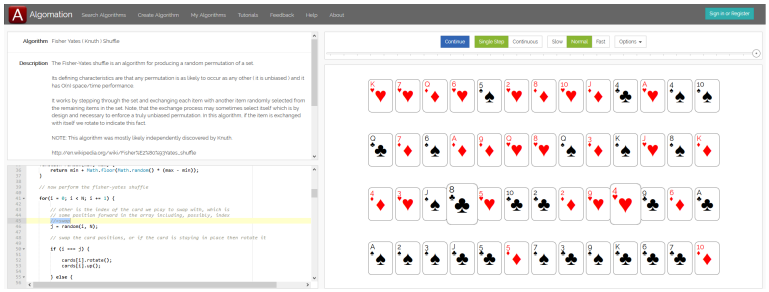


Figure 2.9: Visualization of the Fisher-Yates Shuffle algorithm in Algomation

2.9 Introduction to A*

Red Blob Games's Introduction to A* [9] is a detailed explanation of the A* search algorithm, an algorithm commonly used for pathfinding purposes. The explanation is accompanied by several visualizations. It starts with visualizations for Breadth-First Search and Djikstra's Algorithm; these allow the user to edit the obstacle and goal locations. These visualizations aid the explanation written in text form much like traditional illustrations and diagrams would but have the added benefits of allowing the user to modify the scenario and look at every step of it, which would be impractical with illustrations.

Fig. 2.10 shows one of the visualizations in the article. In this specific instance, the coverage of the space by the algorithm is shown, comparing the result between no optimization and using the early exit optimization, which immediately terminates the algorithm when the goal is found. Walls can be added or removed and the goal location, marked with an X, may be changed through direct input using the mouse. Several samples of Python implementations are shown in the article text.

2.10 Visualizing Algorithms

Visualizing Algorithms [10] is an article on using visualization for diverse classes of algorithms. It posits that algorithms are "a fascinating use case for visualization."

The article introduces problems from different fields, such as sampling, shuffling, and maze generation, and explains the backgrounds to those problems. It then introduces algorithms designed to tackle them. For example, the article explains the concept of sorting (comparing it to the reverse problem of

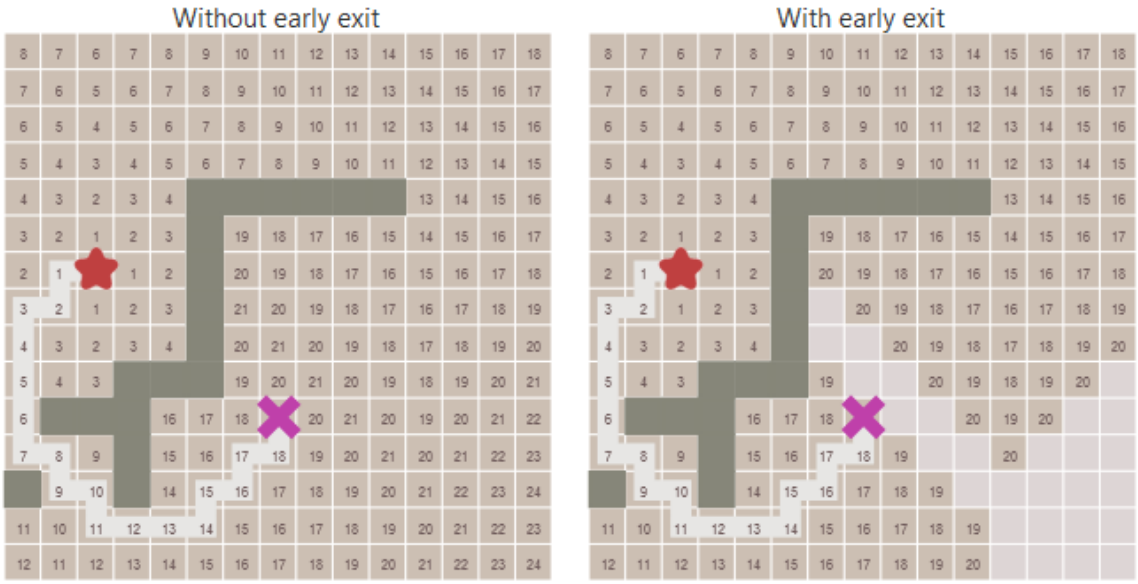


Figure 2.10: Comparison between Breadth-First Search with and without early exit, from Introduction to A*

shuffling), then shows several algorithms that perform sorting. Fig. 2.11 shows part of an example from that section.

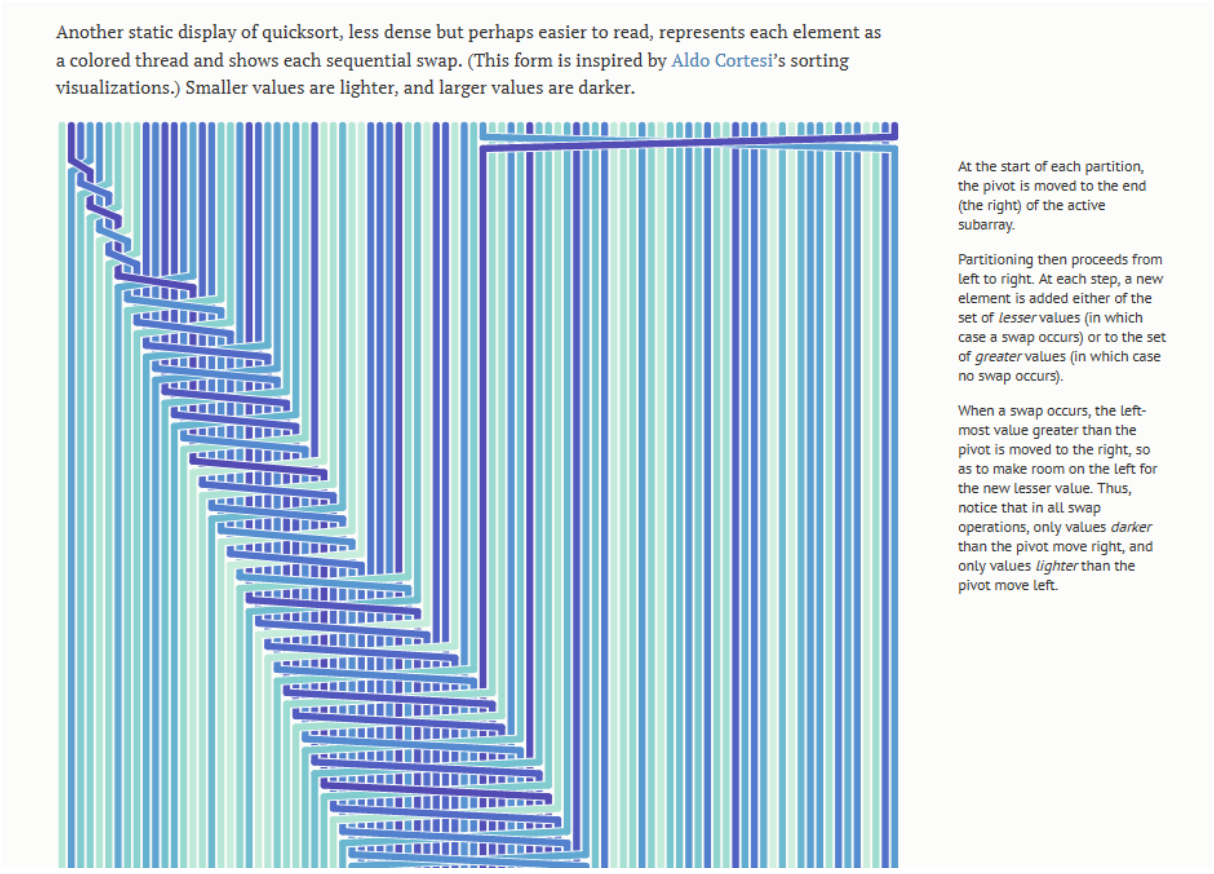


Figure 2.11: Portion of a static visualization of quicksort from Visualizing Algorithms.

A number of very diverse techniques are showcased as possibilities for the different algorithms, and the article concludes by discussing the concept of using vision to think. The visualizations are mostly hard-coded to predefined values are not customizable or extensible, so, overall, they serve more as a starting points for work on algorithm visualization, which is what the article seems to be intended as.

2.11 Explorable Explanations

Explorable Explanations [11] is a general directory of visualizations for problems in different fields, such as biology, social science, and math. The website aggregates links to other websites where visualizations of different formats can be accessed.

The front page (Fig. 2.12) begins with an introduction and features three random visualizations from the library, followed by a set of links to pages where lists of visualizations in particular fields can be explored. The website itself does not contain the visualization tools or implementations, it merely links to other websites and pages where these can be found, for example, the Introduction to A* page discussed above in Section 2.9. There is an extremely diverse set of these from different sources, so it is difficult to take conclusions on what features are generally supported. Most of the pages we looked at seem to fall into the category of Introduction to A*, where a text explanation is aided by specific visualizations targeting individual facets of algorithms spread throughout.

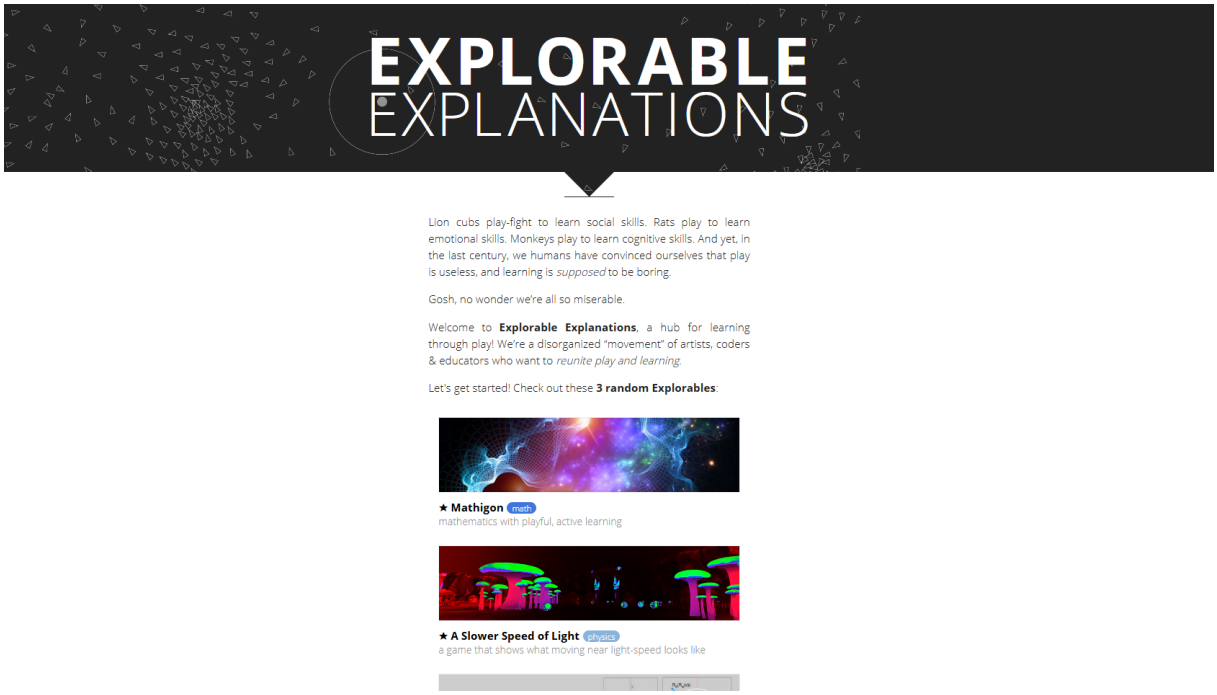


Figure 2.12: Front page of Explorable Explanations.

The website also aggregates links to tools and tutorials for making visualizations, which we recommend to any reader of this work also interested in this particular field.

A related website to Explorable Explanations is the author's personal website, `ncase.me`, which lists visualization works and tools made by the author himself and is subsumed by Explorable Explanations

in that respect.

2.12 Awesome Explorables

Similarly to Explorable Explanations, Awesome Explorables is a curated list of visualizations pertaining to different fields, in line with the trend of *awesome lists* - community-curated lists of items centered around a certain topic or problem. The list also points to tools for creating visualizations and literature about information visualization in general. There is significant overlap between this list and Explorable Explanations, and thus we take the same conclusions as in the previous section.

2.13 Discussion

We now compare the platforms by looking at a number of attributes. The comparison is summarized in Table 2.1, with columns for each attribute and rows for each analyzed tool. We will now discuss these attributes in turn and how they are relevant for our work.

The 'code' attribute indicates how the tool explains what type of code is used to explain the algorithm, if any. 'Source code' indicates that the tool displays and steps through code corresponding to the algorithm implemented in a real programming language, such as Javascript or Python. 'Pseudo-code' indicates that the tool displays an informal, higher-level description of the algorithm that does not correspond to a programming language. An empty field indicates neither is used. We note that the tools use a mix of either source code and pseudo-code, with several others not having any code at all. The tools in the latter case are mostly oriented towards simple visualization that does not aim to make it easy to study the algorithms. The tools using source code mostly do so as the source code is also what drives the visualization, or the objective is to provide the user with code they can copy and run. The other cases use pseudo-code for clarity in explanation of the algorithm itself. Due to the objectives of our work, we believe pseudo-code would be the best choice, although providing both pseudo-code and source code is an option to consider.

Nearly all tools in our analysis support 'step-by-step' execution and 'speed control' - they allow the user to control the visualization in discrete steps and adjust the speed at which it continuously plays. This is important for the user to be able to understand what is happening at their own pace.

The 'user-defined input' describes how flexible the tools are in what inputs are allowed. The checkmark indicates tools which allow arbitrary input via the interface, 'limited' indicates tools which allow the user to select from a set of pre-made or otherwise limited inputs, and empty cells indicate a single input example is available. We note that most tools allow the user to change the input, with some limitations. We believe a combination of both arbitrary input and pre-made examples is the best solution and the tools we analyzed were lacking in this respect. This approach allows the user to explore freely while also presenting interesting specific cases of the algorithms.

	Code	Step-by -step	Speed control	User-defined input	Open source
VisuAlgo	Pseudo-code	✓	✓	✓	
sorting.at		✓		Limited	
Algo-visualizer	Source code	✓	✓	✓	✓
Algoanim	Pseudo-code*	✓*	✓*		
PathFinding				✓	✓
Data Struct. Vis.		✓	✓	Limited	✓
Algomation	Source code	✓	✓	✓	✓*
Vamonos					
Intro to A*	Source code	✓*	✓*	✓	
Visualizing Algorithms	Source code				

Table 2.1: Comparison between applications and tools. ✓* indicates the subject partially fulfills the attribute.

We also note that only some of the tools are open-source, as indicated by the last column in Table 2.1. We believe these sorts of educational tools would be better suited to open source models so that they can be extended and modified for the specific needs of those who wish to use them. Therefore, having an open-source solution to this problem would be a plus.

Based on our research we concluded that we should create a solution that illustrates the algorithms supporting step-by-step execution and speed control, allows user-defined input in flexible ways, presents pseudo-code or source code (or both) for the algorithms, and is open source.

Chapter 3

GeomVis

GeomVis[1] is a tool we developed for visualizing computational geometry algorithms. It is a web application that runs entirely in the web browser and follows web standards. GeomVis has been tested in Mozilla Firefox and Google Chrome. The tool is licensed under the well-known open-source MIT License[2]; the source code is hosted on GitHub[3].

GeomVis aims to teach the user via augmenting descriptions of algorithms with animated visual representations of their steps. In this chapter, we describe how GeomVis is implemented, starting with a general architecture overview in Section 3.1, a look at each component in Sections 3.3 and 3.5, and an explanation of the implementation of the visualization system itself in Section 3.4.

3.1 Architecture Overview

GeomVis is a single-page web application designed to run in the browser and served by a simple file server. GeomVis uses the standard HTML5 and CSS languages for layout and styling, and SVG[4] for the algorithm visualizations. The algorithms, interaction logic and file processing are implemented in TypeScript[5]. The application makes heavy use of the svg.js library[6] to simplify SVG canvas manipulation. `npm`[7] is used for development.

An architecture overview of GeomVis is shown in Fig. 3.1. Our design uses a main user interface component and a set of algorithm modules. These modules are described in more detail in section 3.2 below.

Algorithms are implemented by adding Algorithm Modules. Each Algorithm Module contains the algorithm implementation, the necessary logic to initialize the user interface for the types of inputs (e.g. lines, points, polygons) required by the algorithm, and the logic for parsing uploaded input files for that algorithm.

[1] http://3dorus.tecnico.ulisboa.pt/geomvis/
[2] https://opensource.org/licenses/MIT
[3] https://github.com/ABCRic/GeomVis
[4] Scalable Vector Graphics, https://www.w3.org/TR/SVG2/
[5] https://www.typescriptlang.org/
[6] https://svgjs.com/
[7] https://www.npmjs.com/

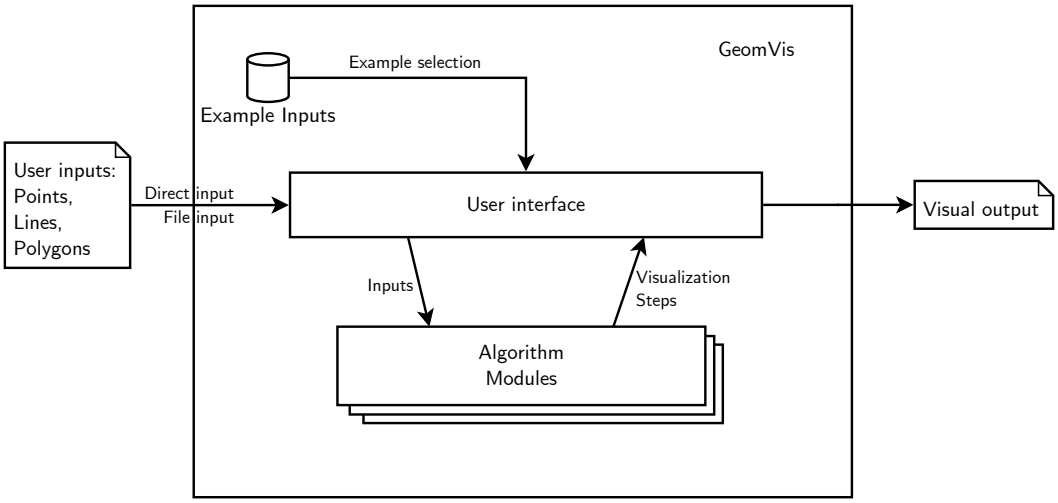


Figure 3.1: GeomVis system architecture

Visualizations are implemented as a sequence of **steps**. Each step contains a set of actions that change the visualization. These actions are, for example: adding a polygon to the visualization, changing the color or position of an element, removing a line from the visualization, and so on. These steps produce the on-screen visualization. The sequence-of-steps representation allows the user to visualize the algorithms either continually or step-by-step, with forward and backwards stepping being supported. This gives the user more control over the visualization.

The Algorithm Module for each algorithm is responsible for calculating the sequence of steps, given the algorithm input provided via the interface.

GeomVis also includes a set of example inputs, in the form of SVG files, which the User Interface contains references to in order to display them to the user and be able to load them.

3.2 Components

The GeomVis architecture has two main components, one responsible for algorithm execution and the other for interacting with the user. We describe them below.

3.2.1 Algorithm Module

The Algorithm Modules are responsible for executing the algorithm in response to the user interface. The algorithm inputs are received via the user interface. Upon user request, the interface instructs the active Algorithm Module to compute the sequence of visualization steps that correspond to the input. The module then executes its algorithm to compute the visualization.

The modules implement the necessary logic for configuring the user interface component to receive

the required inputs for the algorithm, namely points, lines, and polygons. The modules also implement the parsing mechanism for interpreting user-uploaded files when using file input. The input files use the SVG format for all the currently implemented algorithms, but any textual format is supported.

The implementations of the algorithms produce a list of steps, where each step contains a set of actions to be applied to the visualization canvas, a reference to a specific line of the pseudocode for that algorithm, and optionally extra text to be displayed to the user (such as computed results) via the bottom panel of the user interface. The actions in each step are designed so that they support both stepping forwards and backwards through the visualization, to give the user step-by-step control and allow reviewing previous steps. This representation also allows the interface to autoplay the visualization at a configurable speed.

The base application provides a set of flexible action types, such as adding an element (e.g. a line or polygon) to the visualization, changing a property of an element (e.g. color, size, position, etc.), removing an element, adding an element for a single step and then removing it, and others. New action types may be defined by implementing the appropriate interface. The base action specification allows defining code that should be executed 1) when coming in from a previous step; 2) when coming in backwards from the next step; 3) when exiting the current step into the next one; and 4) when exiting the current step backwards into the previous one. In practice, most actions are symmetrical (the same action is performed when entering from either direction, and when exiting to either direction) or entry-only (transitions to the next step do nothing); tools are provided to simplify the implementation of these types of actions.

3.2.2 User Interface

The User Interface component handles the display and user input layer of the application. This module displays the interface to the user and processes user input on that interface. The user interface is described in more detail in section 3.5 below.

This module renders the interface to the user and is responsible for the visual component of the application. The rendering process is done using simple manipulation of the HTML document, or the SVG canvas, via Typescript code; the browser rerenders the document and canvas to the screen as changes are made. The interface allows the user to select an algorithm, which changes the currently active Algorithm Module. User input received via the interface is passed to the active algorithm's module. The interface also displays pseudocode and textual descriptions provided by the Algorithm Module and a set of pre-made examples for each algorithm which the user can choose from.

The main component of the user interface is the canvas upon which the visualization is rendered. The canvas takes user input, namely mouse actions and keyboard presses, and passes them to the active Algorithm Module. The sequence of steps produced by that module are applied to the canvas during visualization.

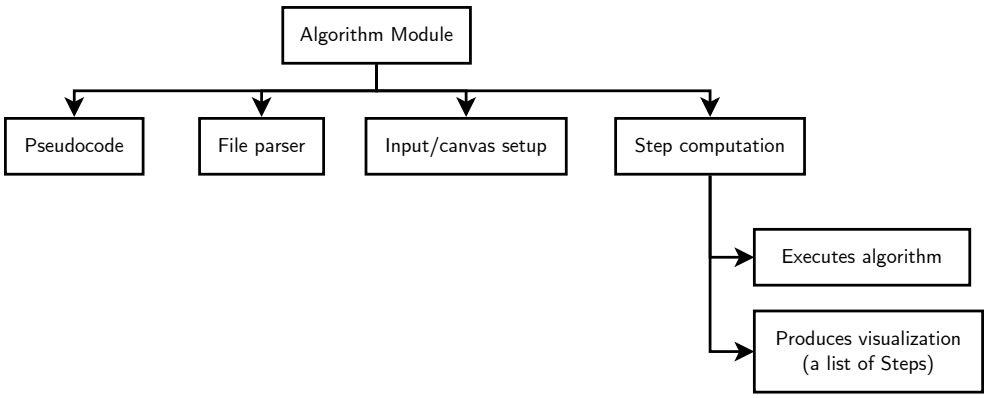


Figure 3.2: Composition of an Algorithm Module

3.3 Algorithm Modules

GeomVis includes five algorithm modules, one for each of the implemented algorithms: Point in Polygon, Graham scan (convex hull), Bentley-Ottmann (line segment intersection), Cohen-Sutherland (line clipping) and Sutherland-Hodgman (polygon clipping). We describe these algorithms in chapter 4.

A decomposition view of the general structure for an algorithm module is shown in 3.2. The components are as follows:

- The **pseudocode** portion, which consists of an ordered list of *lines*. Each *line* has a step of the algorithm encoded in a pseudocode representation, and a textual explanation of that step. Both items are displayed in the lower panel of the interface during algorithm visualization; the pseudocode is shown on the left portion of the panel, and the textual explanation beside it. As we will see in the next section, each step of the visualization points to one of the *lines* of pseudocode so that it is highlighted in the user interface.

- The **file parser**. This is implemented as a method that receives the input file as plain text and interprets the data within to add it to the canvas. For all currently implemented algorithms, this involves using standard javascript API methods to parse files in the SVG format, then extracting the relevant elements for the selected algorithm: in the case of the Bentley-Ottmann (line segment intersection), the line elements[8] in the file are extracted and added to the canvas. The file loading mechanism for both example files and local files is handled by the user interface; the algorithm modules need not implement that logic. The current implementation allows any plain text format to be used, not just SVG, and can be trivially modified to accept binary formats if necessary for some future algorithm implementation.

- The **Input and canvas setup** methods. These methods are executed when the algorithm is se-

[8] In practice, some SVG editing software saves lines as `path` elements, so those are parsed and transformed into lines.

lected via the user interface. The canvas setup method adds the necessary supporting elements to the canvas, if any. For example, in the case of Cohen-Sutherland (line clipping), the clipping rectangle is added to the canvas. The input setup method is responsible for hooking into input events via browser APIs so that the application can accept direct inputs from the user. Using the Graham Scan (convex hull) algorithm as an example, its input setup method hooks into the mouse click event to allow users to add points.

- **Step computation**. This is the main point of interest. In this portion of the module, a full implementation of the algorithm itself is included. The implementation is modified so that at every relevant step of the algorithm a set of actions, and some other information, is emitted. This allows the application to produce a visualization. The set of actions describes the transformations to be applied to the canvas, thus being shown to the user, during visualization. The structure of this set of actions and adjacent information is described in Section 3.4 below.

3.4 Visualization

In the previous sections, we described that GeomVis calculates a list of ”steps” before visualizing an algorithm. In this section, we describe in detail what a ”step” is in the context of our solution, how the visualization is modeled, and how that model maps to what the user sees.

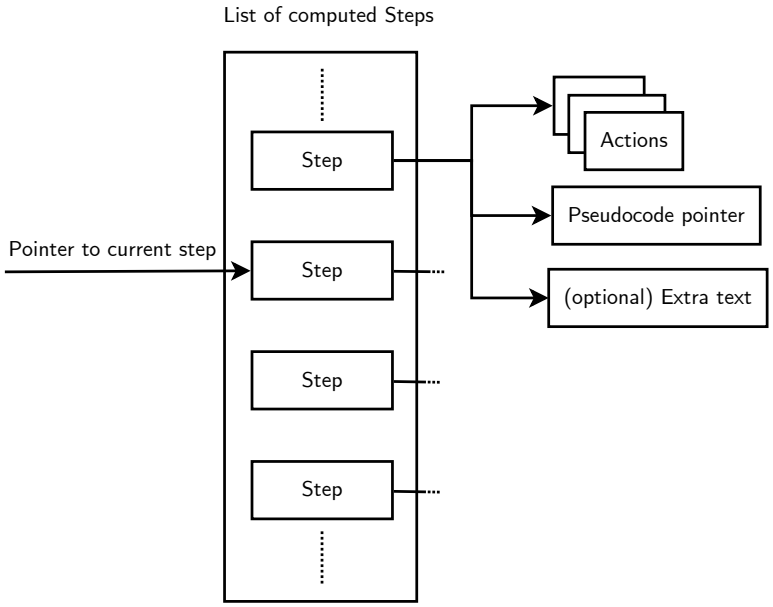


Figure 3.3: Diagram of the visualization data structures

Figure 3.3 shows a diagram of our visualization steps model during execution. The execution is modeled as an ordered list of `n` *steps*. Each *step* generally corresponds to a logical step in the algorithm whose visualization is being modeled, the way one might think of an algorithm as a sequence of steps. As mentioned in Section 3.3 above, each algorithm contains a pseudocode representation as an ordered list of lines, each corresponding to one such logical step. The correspondence is not always one-to-one, as adjustments are made where they benefit the visualization, but in the general case, this idea applies.

Each *step* contains a pointer to a particular line of that **pseudocode**. The pseudocode is shown in block form to the user at the bottom of the user interface, as can be seen in 3.5 and as described in Section 3.5 on the user interface. The pointer in each step is used to highlight the particular line of pseudocode so that it is more prominently displayed to the user, to illustrate which of the steps is the one currently being visualized. We believe this correspondence between the visualization and the pseudocode aids comprehension - indeed, it is common practice in both the related work we looked into and in traditional teaching of algorithms in classes. This highlighting also triggers the display in the user interface of the textual explanation corresponding to that step. Additionally, steps may include extra text - this text is displayed alongside the text corresponding to the line of pseudocode and, unlike it, may be generated during step computation, whereas the text accompanying the pseudocode is set beforehand.

3.4.1 Actions

Steps* also contain a list of *actions. Each *action* is a transformation to be applied to the canvas for the purposes of illustration. Examples of actions included in GeomVis are:

- Adding a point to the canvas;
- Drawing a line between two points;
- Changing the color of a circle;
- Inserting or changing text on the canvas;
- Hiding elements;
- and so on.

These actions are represented as objects with (Javascript) callbacks. The code in those callbacks applies the transformations to the canvas when ran. Each action has four callbacks:

- `stepFromPrevious`, called for the actions of a step when the visualization transitions forwards into it
- `stepToPrevious`, called for the actions of a step when the visualization transitions backwards out of it
- `stepToNext`, called for the actions of a step when the visualization transitions forwards out of it
- `stepFromNext`, called for the actions of a step when the visualization transitions backwards into it

These callbacks form the four possible transitions in and out of a step, if one were to imagine the sequence of steps as a state machine.

We recognize that for many actions, some of the callbacks might be duplicate code, or unused - we observed this ourselves during implementation of the bundled visualizations. In our implementation, we define two additional types of actions that only have two callbacks each: symmetrical actions (`SymmetricalVizAction`), for which transitions into the step are collapsed into a single callback, and so are transitions out of the step; and entry-only actions (`EntryOnlyVizAction`), where only the transitions coming to or going to the previous step are relevant and the others are unused. Figure 3.4 illustrates all callbacks for the three types of actions.

As mentioned both in our introduction and in our conclusions on related work, our solution should have, as a central feature, step control, both forwards and backwards. As such, our model of canvas transformations is partly based on state machine transitions. For a given transformation, we want to support its reversal so that the user can step back and forth at will, to aid comprehension. We considered two main options.

The first option was to use a model where we save every intermediate state of the computation, use those states to display elements on the screen, and interpolate between the states to generate transitions between steps. This is the most straightforward model to use from an algorithm implementer's

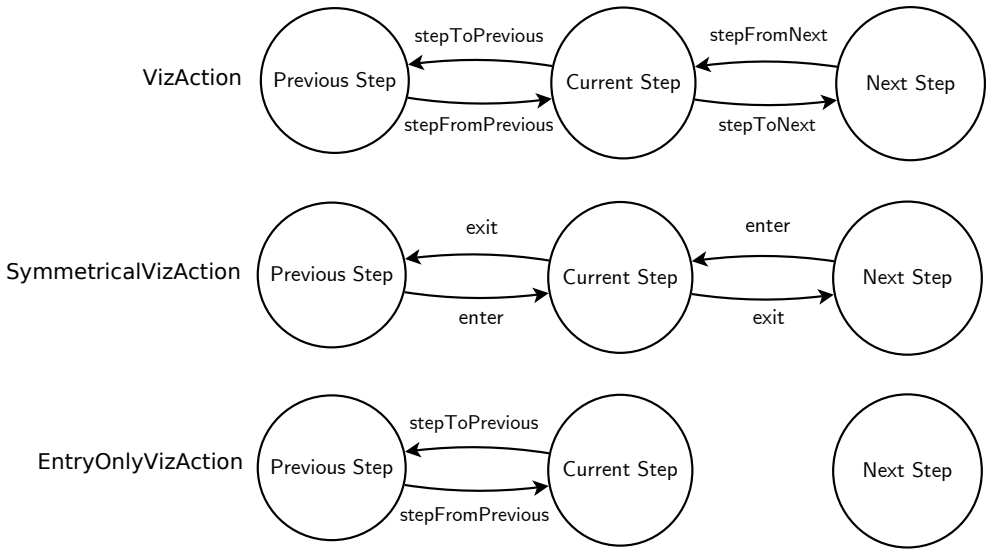


Figure 3.4: Visualization action types

standpoint, but it is extremely difficult to develop in this case due to the nature of the algorithms we wish to cover. As an example, we believe that VisuAlgo[1] (analyzed in section 2.1) is using this model specifically for some of its visualizations, namely the ones pertaining to or based on (vertex-and-edge) graphs. This model rapidly gets out of hand when very different algorithms must be implemented as its complexity balloons due to the variety of possible types of operations (versus the limited number of operations possible in graphs), and it would limit the capabilities to those implemented, which would hinder the possible visualizations.

The second option, and the one we went with, is to use a sort of ”linear” state machine model with user-defined transitions (the user, in this case, being the API consumer, i.e. each algorithm module). In this case, and as was implemented, each *step* represents the possible transitions in and out of a state. The actions contained within the *steps* are like transitions between states in a state machine, with the constraint that the states are arranged in a sequence, with no branching or loops.

With this model in mind, we realize that having *all* actions be completely user-defined (again, the user in this case being the writer of the algorithm module) leads to some needless boilerplate and so we include several common actions out of the box to simplify the step generation code. These actions are used in the included algorithm modules and can also be used effortlessly in any additional algorithm modules developed in the future.

3.5 User Interface

The GeomVis user interface is that of a single-page web application. There is a single, main view where most operations and user interaction take place; some parts of the interface can be hidden and others

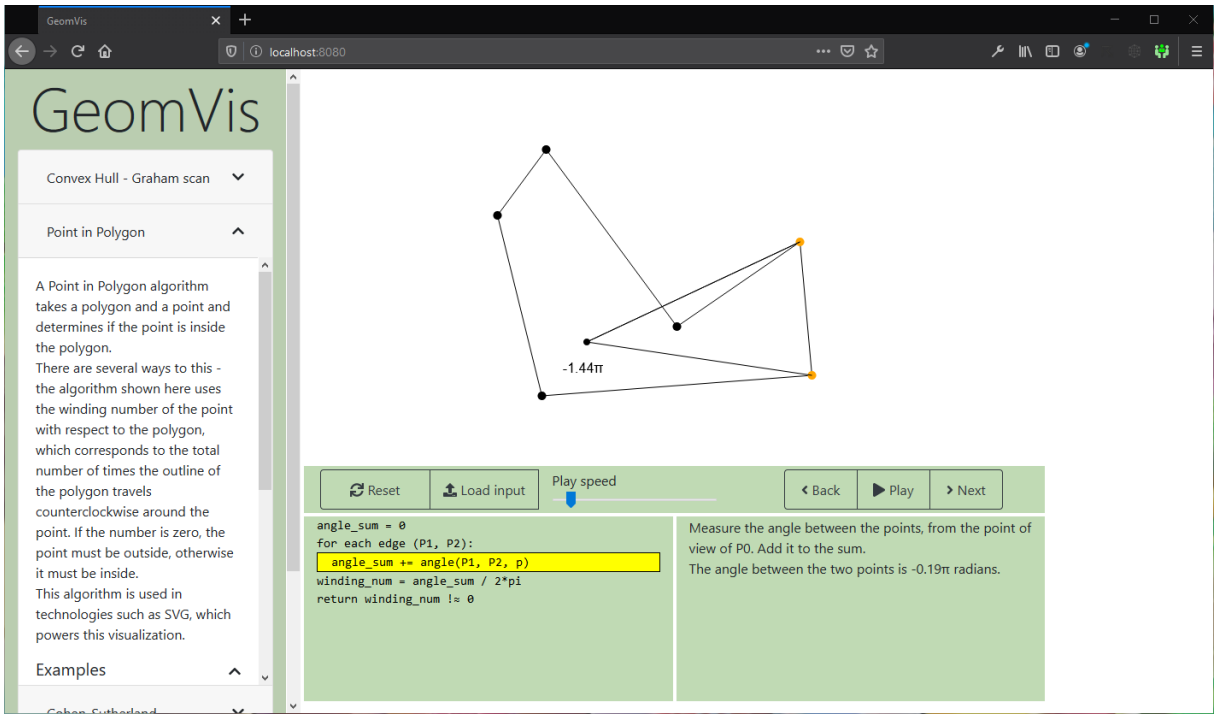


Figure 3.5: The GeomVis user interface.

appear when relevant. Fig. 3.5 shows a screenshot of GeomVis running in a modern browser.

The panel on the left features the app title and a list of the algorithms to choose from. Upon selecting an algorithm, its box expands to reveal a textual description of the algorithm, and the visualization for that algorithm becomes active in the main panel. Additionally, a table of examples is shown, each of which can be clicked to load it onto the main panel to be visualized. Fig. 3.6 shows a closeup of the Graham scan algorithm on the list, with its three associated examples.

The main panel, which encompasses most of the page, contains the visualization for the currently selected algorithm. The animations and visuals that illustrate the algorithms are shown in this panel. These visualizations include lines, circles, polygons and other geometric elements. The animations include changing the color of the elements, moving them to different positions, and altering their size and shape. Each algorithm uses animations and elements representative of its steps in processing the input - for example, the Cohen-Sutherland algorithm highlights the line being processed in the current iteration in a different color and clips that line as appropriate.

At the bottom of the main panel is an additional support panel containing a pseudocode representation of the algorithm, with the current step of the animation highlighted, and a text panel, with an explanation of the current step of the algorithm. At the top of the support panel is a row of elements containing speed and step-by-step controls and canvas control buttons. The reset button resets the canvas to a blank state and can be used to exit running visualizations. The load input button prompts the user to open a file with the inputs for the algorithm, which then loads onto the canvas. The Back, Play/Pause and Next buttons control the visualization steps.

The interface has two main modes of operation: editing mode and visualization mode. In editing mode, the canvas can be edited by the user, to alter the inputs via direct modification using the mouse.

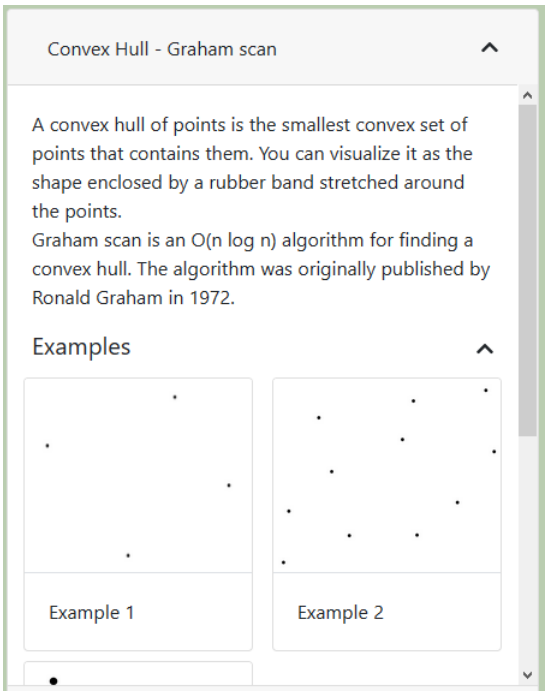


Figure 3.6: Graham scan description and examples on the interface's algorithm list

The specific ways in which input is modified depend on the currently active algorithm. An example of editing mode for the Sutherland-Hodgman visualizer is shown in Figure 3.7. In visualization mode, the algorithm visualization is taking place, and the user can press the Back and Next buttons to change between steps. The Play/Pause button can be used to enable automatic step playback, which causes GeomVis to automatically advance to the next step using a regular interval. The user may change the interval by adjusting the Play Speed slider. Clicking any of the Back, Play/Pause and Next buttons changes the interface to visualization mode if it was in editing mode. Clicking Reset, loading an input file, loading an example, or changing algorithm all change the interface back to editing mode.

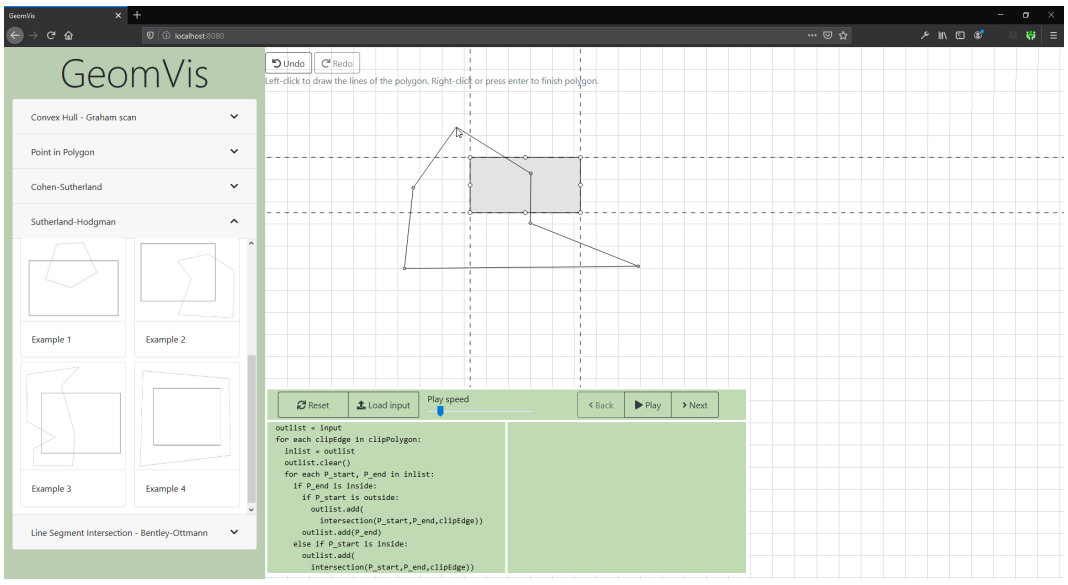


Figure 3.7: GeomVis interface while editing input for the Sutherland-Hodgman algorithm

Currently, the interface features the five algorithms we implemented. Additional algorithms can be trivially added to the interface after a corresponding Algorithm Module is implemented for them. We describe the already included algorithms in the next chapter.

Chapter 4

Use Cases

In Chapter 2, we looked at related tools and identified a gap in coverage for computational geometry algorithms. For this work, we picked five such algorithms that we found relevant for their use in education, particularly in the area of Computer Graphics, and for their relative diversity in the types of inputs and outputs, to showcase the possibilities of GeomVis. We implemented visualizations for these five algorithms as part of our work. The algorithms are described below, followed by a discussion of other algorithms we identified as potential future use cases to be implemented in GeomVis.

4.1 Point in Polygon (Winding Number) Algorithm

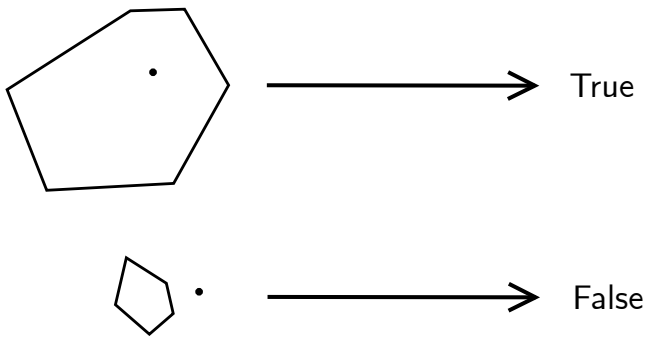


Figure 4.1: Point in Polygon algorithm input and output examples

A Point in Polygon algorithm takes a point of the plane and a polygon and determines whether the point lies inside the polygon. Two examples are shown in Fig. 4.1. This problem is generally simple to solve by sight but is nontrivial to implement in geometric terms, with implementations varying between using such methods as winding numbers and ray casting.

The variant of the algorithm we implemented is known as the *winding number algorithm.* The algorithm computes the winding number of a point relative to a polygon. The winding number represents the number of times that the line of the polygon travels around the point. The algorithm determines this number by sequentially summing the angles formed with each edge of the polygon. It can be shown [12]

that the winding number is zero for points outside the polygon and non-zero for points inside the polygon. See Algorithm 1 for a pseudocode description of the algorithm.

Algorithm 1: Winding number point in polygon algorithm

Input: A polygon with edges E, and the point being tested p

Output: A boolean value indicating whether the point is inside the polygon

```
1 anglesum ← 0
2 for each edge in E with points P_1, P_2 do
3     anglesum ← anglesum + ∠P_1pP_2
4 windingnumber ← anglesum ÷ (2 × π)
5 return windingnumber ≠ 0
```

4.2 Convex Hull (Graham's Scan) Algorithm

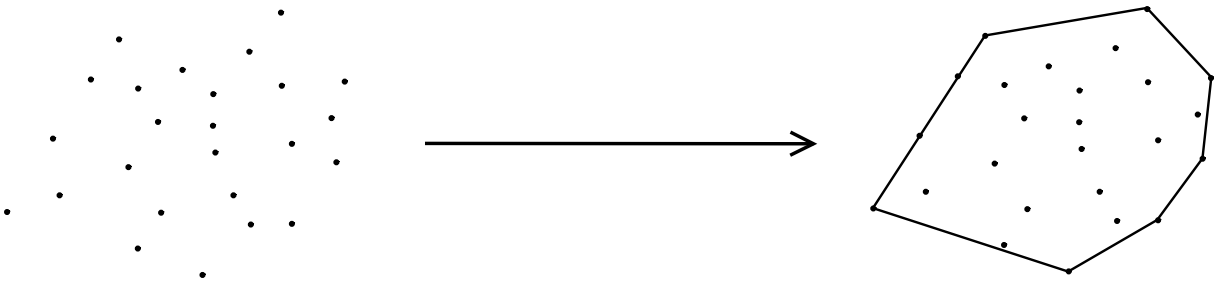


Figure 4.2: Convex Hull algorithm input and output example

A (2-dimensional) convex hull of a set *X* of points is the smallest convex set of points in the plane that contains the points in *X*. As a simple analogy for the 2-dimensional case, it is the shape that results from stretching an elastic band such that it contains all the given points in *X* and is then released to tighten around them. Fig. 4.2 illustrates an example.

Convex hulls have several applications. In artificial intelligence, for example, the convex hull of a robot is used to simplify collision detection, as if the convex hull avoids collisions then so does the robot. There are also several applications in image processing [13].

We implemented Graham's Scan, published by Ronald Graham[14]. Graham's scan is an $O(nlogn)$ time complexity algorithm for finding a convex hull given a set of points. As a top-level description, the algorithm takes a point interior to the hull, sorts the points by angle counterclockwise about that point,

and grows the hull incrementally around the set of points. The steps are illustrated in Algorithm 2.

Algorithm 2: Graham's scan algorithm

Input: A set of points P

Output: A stack of vertices that form the convex hull for the points in P

```
1 s ← empty stack
2 P_0 ←lowest point in P
3 sort the points in P by angle between x-axis and P_0
4 push P_0 into s
5 for each p in P do
6     while s has more than 1 point ∧ right_turn() do
7         stack.pop()
8     stack.push(p)
9 return s
```

4.3 Line Segment Intersection Algorithm

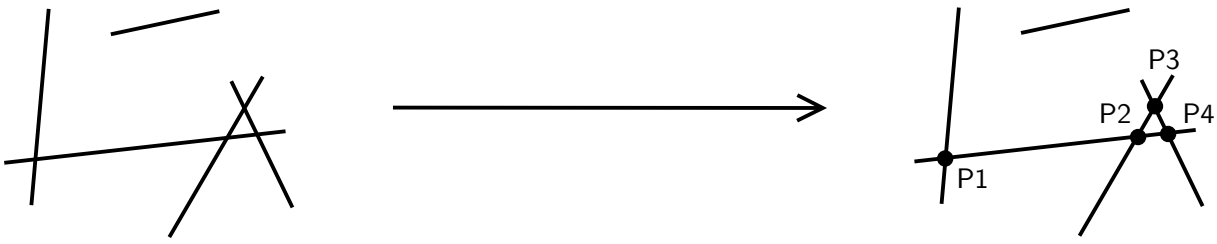


Figure 4.3: Line Segment Intersection algorithm input and output example

A Line Segment Intersection algorithm takes a set of line segments and determines at what points those segments intersect, as shown in Fig. 4.3. This is useful in several scenarios. One such scenario is determining where a road crosses a country or region border. As both roads and borders can be represented simply as sets of segments, we can use such an algorithm for finding these crossing points. Particularly, note that curves may be approximated by a number of small segments and the same algorithm applied to them as well.

A simple, brute-force algorithm would be to check, for every pair of line segments, whether there is an intersection. This clearly requires $O(n^2)$ time. However, in most cases, line segments do not intersect, so a more sophisticated algorithm, with running time proportional to the output, would be more efficient in most cases. We implemented a visualization for the Bentley-Ottman algorithm [15], which takes a sweep line approach to finding the intersection points. The implementation is based on [16].

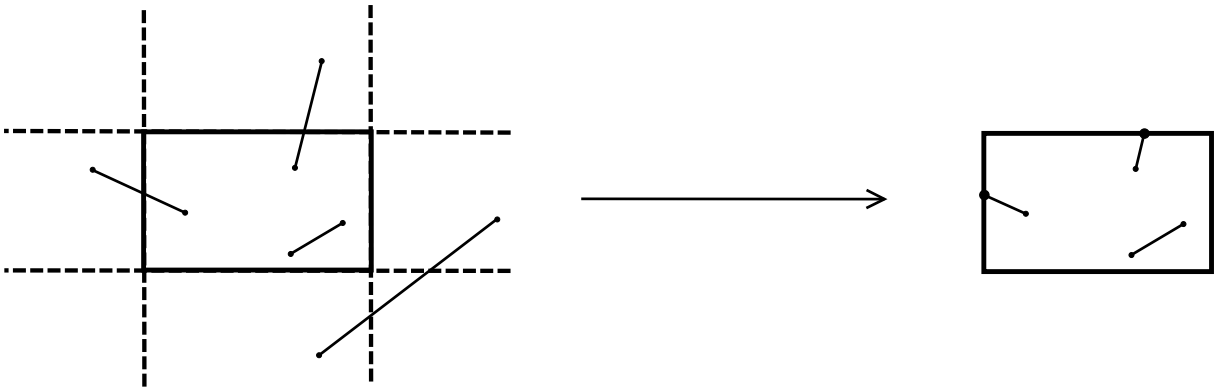


Figure 4.4: Cohen-Sutherland algorithm input and output example

4.4 Cohen-Sutherland Algorithm

The Cohen-Sutherland algorithm [17] is used in computer graphics for line clipping. It divides a 2-dimensional space into nine regions and then determines which lines and portions of lines are visible in the center region. An example is shown in Fig. 4.4. During the process of rasterization, this is useful to determine which lines or portions of lines should be considered for rendering.

Algorithm 3 shows a stepwise description of the algorithm. This algorithm takes input lines, testing their endpoints for which of the nine regions they are contained in, and giving them 'outcodes' (4-bit numbers). Each bit represents a cardinal direction and is set to 1 if the endpoint lies outside the viewport. If both endpoints are inside ($\vee$ of outcodes $= 0$) they can be trivially accepted, if endpoints share any non-visible regions ($\wedge$ of outcodes $\neq 0$) they can be trivially rejected. In the other case an endpoint that is outside is selected to be replaced by the intersection with the extended viewport borders. This last step is repeated until one of the trivial cases occurs.

Algorithm 3: Cohen-Sutherland algorithm

Input: A clipping rectangle R and a set of lines L

Output: A set of lines resulting from clipping the lines in L inside R

```
1  for each line l with endpoints P_1, P_2 in L do
2      C_1 ← outcode(P_1)
3      C_2 ← outcode(P_2)
4      if C_1 | C_2 = 0 then
5          goto next line
6      if C_1 & C_2 ≠ 0 then
7          delete line
8          goto next line
9      P_o ← a point from P_1, P_2 that is outside R
10     replace P_o with the rectangle intersection point
11 return L
```

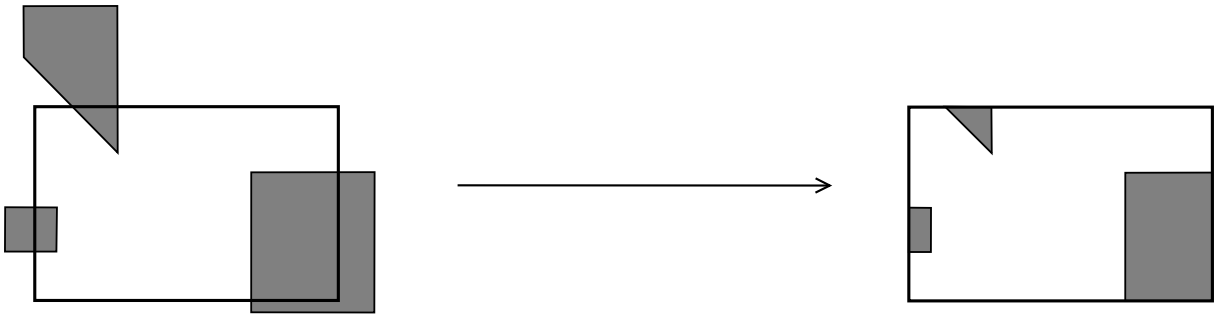


Figure 4.5: Sutherland-Hodgman algorithm input and output example.

4.5 Sutherland-Hodgman Algorithm

The Sutherland-Hodgman algorithm [18] is, similarly to the Cohen-Sutherland algorithm, used for clipping into a viewport. In this case, however, the subject of clipping is a polygon. The edges and points of the polygon are iterated to clip lines that intersect the viewport and discard vertexes that fall outside its bounds. An example can be seen in Fig. 4.5.

This algorithm is used by 3D rendering applications to prevent the unnecessary work of rendering portions of the world that would not be visible through the viewport. The algorithm works by extending each of the borders of the viewport and the polygon to be clipped is traversed. The points of the subject polygon on the visible side of the border are placed on the output list, and new points are generated where the polygon intersects the border (the points on the non-visible side of the border are discarded). After iterating through the entire border, the polygon is now clipped. Algorithm 4 illustrates a pseudocode description of the algorithm.

Algorithm 4: Sutherland-Hodgman algorithm

Input: A clipping convex polygon C and an input polygon I
Output: I clipped by R

```
1  outlist ← I
2  for each edge E in C do
3      inlist ← outlist
4      outlist.clear()
5      for endpoints P_start, P_end of each edge in inlist do
6          if P_end is inside C then
7              if P_start is outside C then
8                  outlist.add(intersection(P_start, P_end, E))
9              outlist.add(P_end)
10         else
11             if P_start is inside C then
12                 outlist.add(intersection(P_start, P_end, E))
13 return outlist
```

4.6 Discussion

Due to time constraints, we settled for implementing the five algorithms above for our work. Below, we list other related algorithms that we could not implement but could be added in the future, using the tools already provided by GeomVis.

A Delaunay triangulation [19] is a triangulation of a set of points such that none of the points are contained in any of the triangles' circumferences. Delaunay triangulations are used, for example, for generating smooth geometry for modelling terrain from a sample of points. An algorithm for generating Delaunay triangulations could be added, such as S-hull [20].

An algorithm for polygon triangulation would also be a suitable candidate for inclusion, for example, the one described in [21], which partitions a simple polygon into monotone polygons in $O(n \cdot log(n))$ time and then triangulates each of the results in linear time.

The Surveyor's formula [22] is an iterative formula for calculating the area of polygons with an arbitrary number of sides. A visualization of this formula in action could be implemented.

Finally, one other visualization that could be implemented is the intersection of non-convex polygons algorithm described in [23, pp.266-268]. In particular, this algorithm is an extension of the Bentley-Ottmann algorithm, which is already implemented, so parts of it could be reused.

Chapter 5

Evaluation

To test the effectiveness of GeomVis in teaching algorithms, we performed a round of effectiveness tests with users. The users were split into a control group with access to a traditional text and diagram description of the algorithm - i.e., as could be found in a book - and an experimental group that could use GeomVis. We describe the testing procedure in detail and its results in this chapter.

With our tests, we intend to prove our hypothesis that using GeomVis and the techniques it leverages is better for users to learn algorithms than traditional static text and diagram explanations. Therefore we tested this null hypothesis:

> H_0 = using GeomVis is equal or worse than using a traditional text and diagram explanation, in terms of how well users learn the algorithm.

To that end, we performed tests as described in the following section. We then follow-up in Section 5.2 with an analysis of the results.

5.1 Methodology

The tests were carried out at two similar locations at Instituto Superior Técnico in Information Technologies Buildings I and III (Pavilhão de Informática I & III) at Lab 15 in the former and an open studying area in the latter. The locations are relatively quiet at the times the tests were held and the tests were monitored to prevent disturbances. The equipment used for the tests was a 17-inch laptop with a wireless mouse and Wi-Fi internet access. The laptop was used both for running the application, and for the users to fill in a questionnaire and a profile survey.

The users were split into two groups: Group T (the control group, which used the text and diagram explanation) and Group G (the experimental group, which used GeomVis). Each user was received individually, assigned a group, and then taken through the test process. The user testing script prepared for the tests is included in Appendix D (in Portuguese). We relay the same information below.

The test process was as follows. First, the user was welcomed and received an introduction to what GeomVis is and an explanation of the tasks they would be asked to do. Then, users were asked to

Figure 5.1: User performing the exam

sign a consent form for the use of information provided by them during the test and fill in a short survey to collect profile information (Appendix A). Both groups were offered an explanation of what a convex hull is; concretely, that a convex hull of a set of points can be visualized as the shape of a rubber band stretched around that set of points - it is the smallest convex shape that contains the points. Afterwards, the users were given a task depending on the group they were in. For Group T, the control group, the users were given seven minutes to autonomously learn the Graham Scan algorithm for Convex Hull generation (described in section 4.2) using a static text and diagram explanation based on a common, openly accessible, and easy to find source, Wikipedia (see Appendix C). This explanation is the most likely result on search engines, and fits well in the time constraints of testing, especially when compared to explanations in literature, which usually go into deep theoretical background. Users in that group were told to learn the algorithm on their own using the included descriptions, diagrams, and pseudocode. For Group G, the users were also given seven minutes to autonomously learn the same algorithm, but using GeomVis instead of the text and diagram explanation. These users were told that they could use the example inputs included in GeomVis or their own. Both groups were allowed to use their time freely; users that felt confident they were done before the timer ran out were encouraged to keep going. After the seven-minute time passed, users received a multiple-choice exam with a five minute time limit, shown in Appendix B, to evaluate the knowledge they acquired. Figure 5.1 shows a user undertaking the exam.

To measure effectiveness, we looked at the following metrics during our testing: the number of correct answers in the exam, which tells us if users were able to learn the algorithm; the time taken to finish the exam, which hints at how confident users were in their acquired knowledge; and the number of indecision events while answering the exam (the amount of times users changed their answer), which also hints at the user's confidence in their knowledge. Figure 5.2 shows an example of a question from the exam.

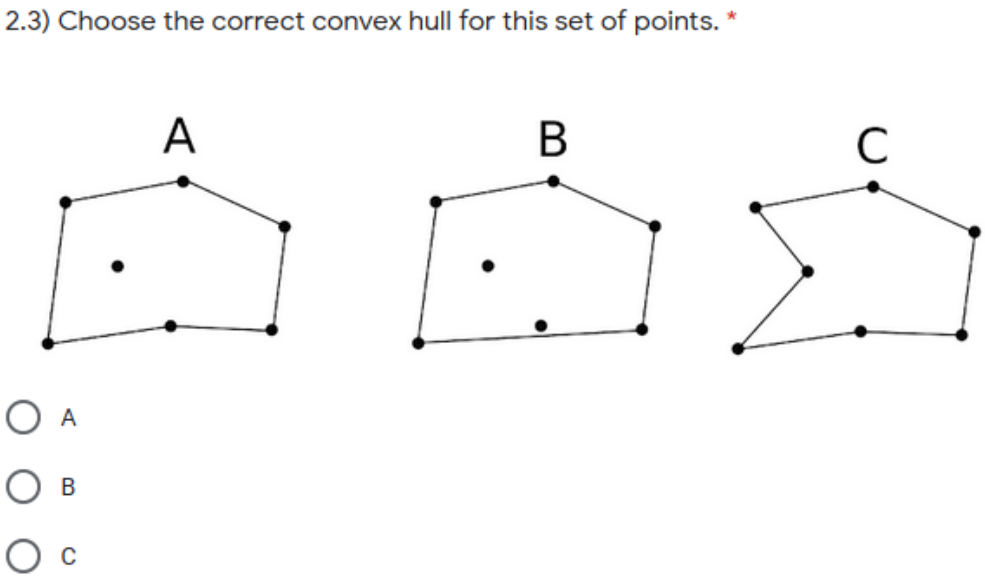


Figure 5.2: Example question from the exam

5.2 Results

The tests were performed by twenty users who went through all the steps indicated above in 5.1. Here, we analyze the results of the profile survey and of the exam itself.

5.2.1 User profiles

Testing was performed with twenty users; users were randomly split into the testing groups. As GeomVis is aimed at an audience of people who are interested in or currently studying Computer Science and similar subjects, we found it an appropriate choice to rely mostly on students at Instituto Superior Técnico for undergoing testing. As such, nearly all users were of usual university age, as can be seen in Fig. 5.3.

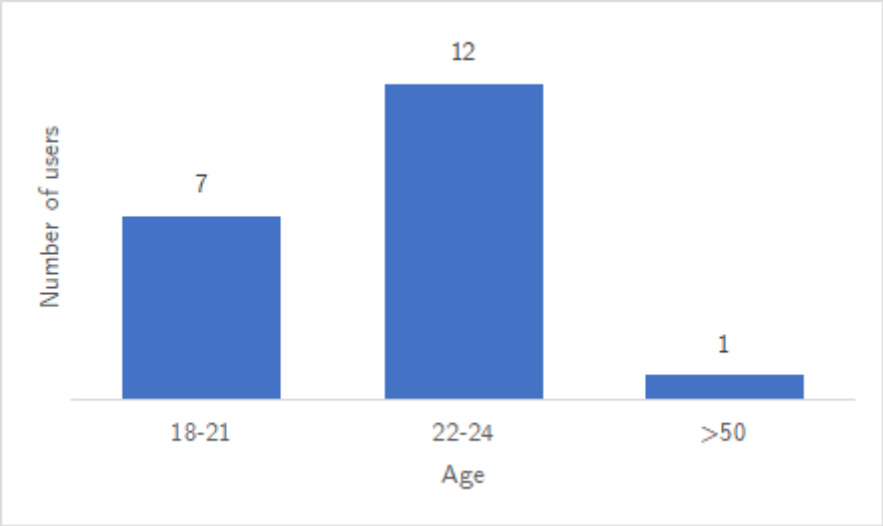


Figure 5.3: Number of users per age group

We also noted that our sample was split down the middle between Bachelor's and Master's degree students. Additionally, most of them had some experience or knowledge of several general algorithms and computational geometry algorithms (see Fig. 5.4). This is expected, as many of them were in our target audience. We believe this is an acceptable bias for our sample as it reflects that target audience. Additionally, none of the users were familiar with the specific algorithm being tested (users that did would be disqualified, although in practice this situation did not occur) so that previous knowledge of it did not affect the results.

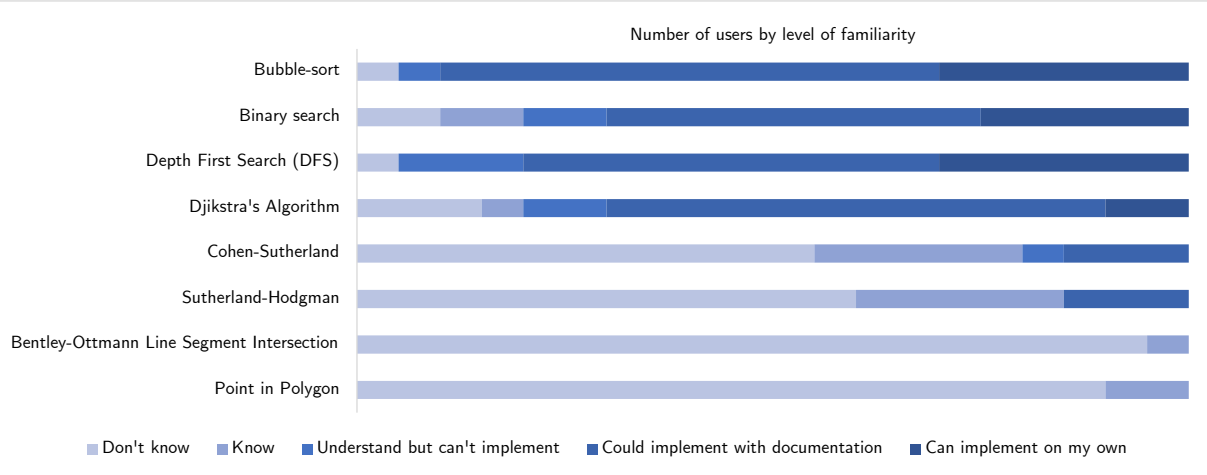


Figure 5.4: Test users' familiarity level with several algorithms

We note in particular that users were generally not familiar with some of the algorithms covered in this work, while being mostly comfortable with more popular algorithms such as bubble-sort and binary search.

5.2.2 Exam Results and Analysis

As the last step in the testing process, we had users undergo an exam with multiple-choice and true/false questions, nine in total. The exam had a time limit of five minutes, which the users were told before the start of the exam. We measured the following three metrics during and after the exam:

- the exam score - the number of correctly-answered questions in the exam
- the exam time - how long it took each user to complete the exam
- the number of indecision events - how many times each user changed his or her answer (on any question) while answering the exam.

We now analyze the results of the data we collected. We subjected the results of the exam score and time metrics to Student's t-test to check for statistical significance, as we are comparing, for each metric, two sets of data for significant difference. As we expect that variances may be different (and, indeed, they seem to be from results), we use an heteroscedastic version of the t-test, that is, one that does not assume the variance is equal for both sets. Note that Student's t-test assumes samples are taken from

a normal distribution. We used the Shapiro-Wilk normality test for exam scores and times and the test passed for $p < 0.05$. For the number of answer changes, the sample does not pass the normality test, there we use the Mann-Whitney U nonparametric test instead.

Regarding the exam scores, we found that most participants from the experimental group, who exclusively used GeomVis, had successfully acquired enough knowledge to answer nearly every question in the exam perfectly. All but two users achieved a perfect score; those two users failed a single question. The average was 8.8 out 9 points. The control group had lower and more spread-out scores, with a lower average of 7.4 out of 9, and two users with low scores of 5 points. Figure 5.5 illustrates the distribution using a box plot. We found statistically significant difference ($p = 0.01329$) between the two groups. We conclude that GeomVis was surprisingly effective in teaching users, due to the high scores in the exam, and more effective than the control method.

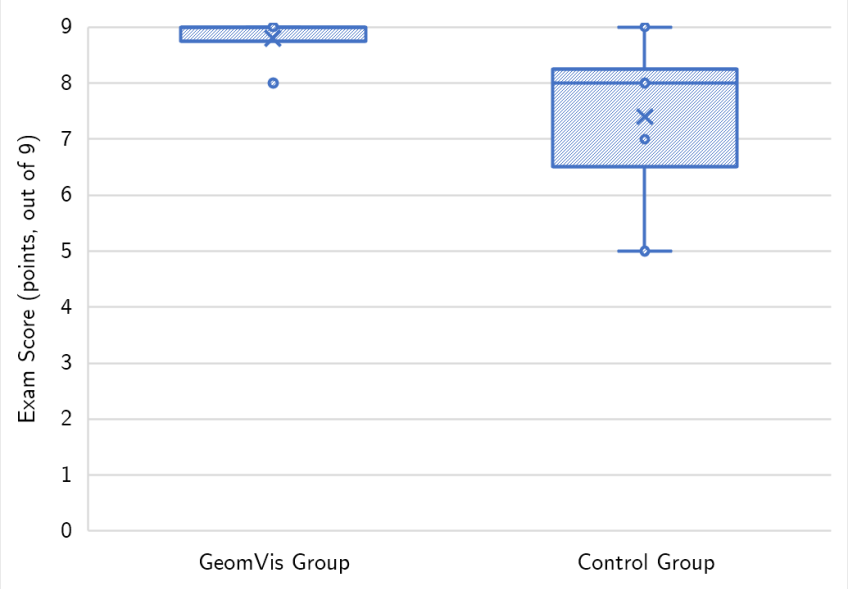


Figure 5.5: Boxplot of exam result scores

Exam times showed similar differences between the two groups. The average exam duration between the two groups showed a marked gap of $\delta = 1m23s$, with the average for the GeomVis group being 2m39s (53% of the time limit), and the average for the control group being 4m03s (81% of the time limit). This is a significant difference at $p < 0.01$ ($p = 0.00025$) and marks a notable improvement. The variation of the results from each group was slightly similar between groups, with a standard deviation of 32 seconds in the GeomVis group and 47 seconds in control. Fig. 5.6 shows the distribution and average values. During testing, we subjectively found that users in the GeomVis group were markedly confident in their newly-acquired knowledge of the algorithm, whereas the control group participants felt that they did not quite understand the full reasoning behind the algorithm. We believe that the significant difference between the averages in this metric, together with the number of answer change events, are objective validation of that observation.

Finally, for our third metric, we also found statistically significant difference ($U = 19$, $p = 0.02$) between the two groups, with the experimental group having on average 0.3 answer changes during the

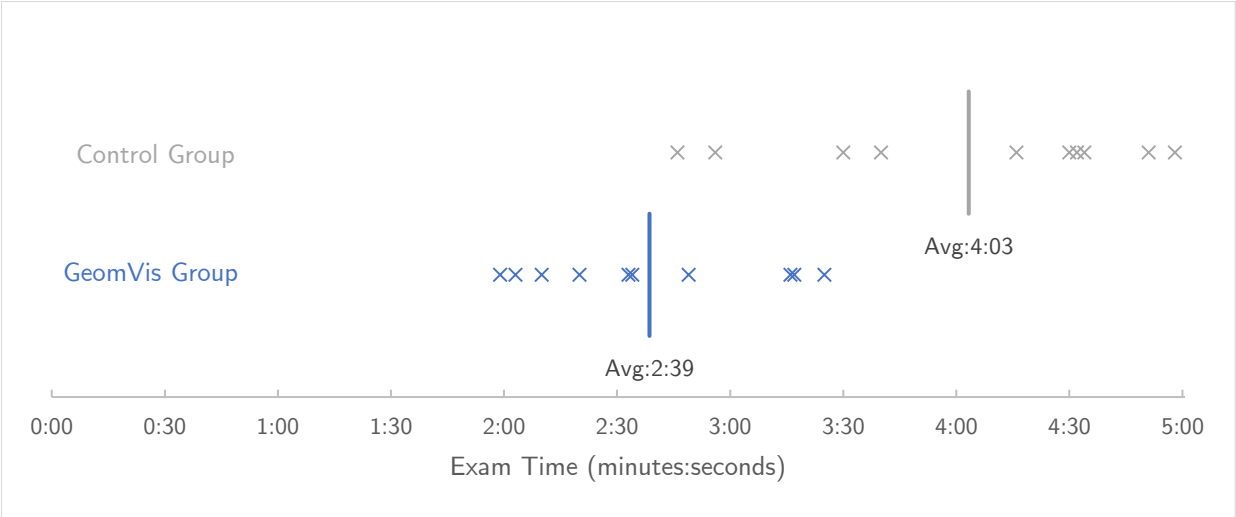


Figure 5.6: Exam execution time distribution

exam and the control group having 1.2 on average. Fig. 5.7 shows the distribution. Together with the results for test time, we conclude that users in the experimental group were more certain in their answers and as such took less time and were less likely to change them. We note that, from observation, the main source of answer changes was questions farther along the test causing the user to rethink answers to previous questions.

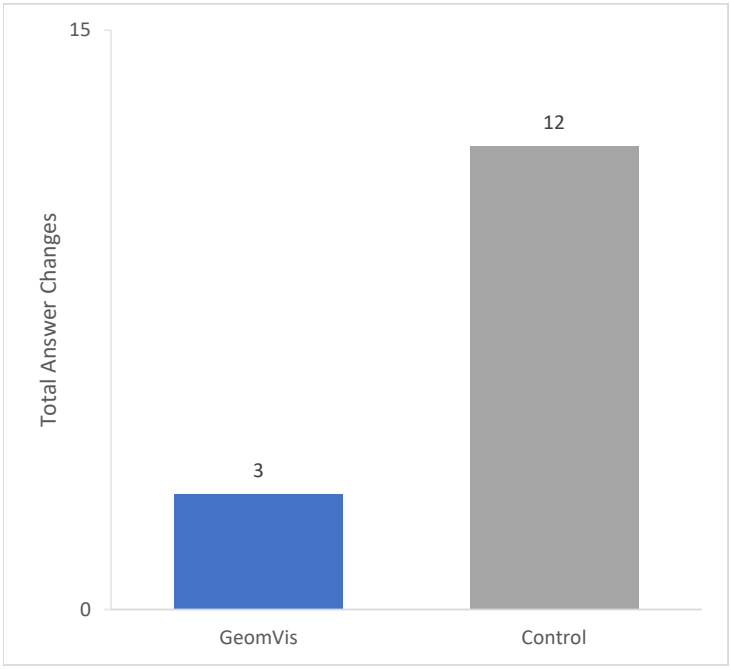


Figure 5.7: Total answer changes per group. Note that both groups had equal numbers of users

The results of the three metrics are summarized in Table 5.1. Overall, we believe these are promising results and reject our null hypothesis H_0, posed at the start of this chapter, in favor of our initial hypothesis that using GeomVis facilitates learning algorithms, more so than traditional text and diagram explanations. Due to time and resource constraints, we were unable to perform tests on a larger scale,

test multiple algorithms, or use different types of tests such as knowledge retention testing. Nevertheless, we find these results quite positive and that use of GeomVis in real world applications is worth considering.

	GeomVis Group	Control Group
Average Exam Score (points, out of 9)	8.8	7.4
Exam Score Standard Deviation (points)	0.42	1.43
Average Exam Time (min:s)	2:39	4:03
Exam Time Standard Deviation (min:s)	0:32	0:47
Average Answer Changes	0.3	1.2
Answer Changes Standard Deviation	0.48	0.79

Table 5.1: Averages and standard deviations of exam scores and times.

Chapter 6

Conclusions and Future Work

Computer-aided visualization is a field that is growing as more and more people have access to capable devices, whether they be traditional computers and laptops or smartphones. Its use in teaching algorithms is developing, and no doubt will flourish in the coming years.

While looking at current solutions in this field, we found a number of tools already available that harness visualization for aiding in the understanding of algorithms. We found disparities in functionality in the available solutions, and notably, we found a lack of such tools for algorithms from the field of computational geometry.

At the beginning of our work, we identified key features in applications that target that particular use, and found a void to be filled in algorithm coverage. As such, we set out to create a web application to visualize computational geometry algorithms in order to aid in learning these algorithms, with valuable features such as step-by-step control, and support for arbitrary user input. We created a framework for representing algorithm execution for visualization, and developed concrete implementations for five use cases. We created an extensible design and made the code available for reuse and derivative works, so that it can more easily be extended and adapted by those interested in using it for teaching. We deployed our application online and it is publicly accessible at `http://3dorus.tecnico.ulisboa.pt/geomvis/`.

We tested our solution with users from our target audience, students of computer science. Our testing showed that GeomVis was effective in teaching algorithms, as test subjects were able to easily answer exam questions on limited time constraints, with a statistically significant improvement over the traditional text and diagram explanation. Users in the experimental group demonstrated confidence in their newly acquired knowledge even in the short time that was given to them for studying the algorithm.

While developing our work, we found some limitations we could not address, whether due to time or resource constraints. We thus propose the following items as possible future work:

Test GeomVis in classes. We are confident GeomVis will prove useful, from our tests with users. Nevertheless, we believe no test is better than use in real-world conditions, with use by professors in classes. Such use can demonstrate the real usefulness of GeomVis, and of algorithm visualization in general, in enhancing the teaching and learning of algorithms.

Implement more algorithm visualizations. GeomVis was designed to be extensible, and we hope

that will pave the way for more algorithm visualizations, so that it can be of greater use to more people. We listed some potential candidates in Section 4.6.

Offer algorithm implementations. It would be useful for implementations of the included algorithms in several programming languages to be provided, licensed under permissive terms such that users could integrate them easily in their own programs.

Extend the visualization to 3D technologies. The current implementation is coupled to an SVG canvas where visualizations are displayed. However, the algorithm execution model could conceivably be adapted to other systems. It would be interesting to create a modified version of it using, for example, WebGL, a 3D graphics standard for the web. Such a modified version could be used for showing algorithms that operate in 3D space.

In conclusion, we believe that GeomVis, and future developments of it, are worthwhile. There is much more that could be done, some of which we suggested above. However, evaluation results show that this approach is effective in aiding our goal - teaching algorithms better.

Bibliography

[1] F. H. Steven Malim. Visualgo - visualising data structures and algorithms through animation. https://visualgo.net/, 2011. Accessed: Oct. 2018.

[2] L. Végh. Algorithm animations and visualizations. http://algoanim.ide.sk/. Accessed: Nov. 2018.

[3] J. Park. Algorithm visualizer. https://algorithm-visualizer.org/. Accessed: Nov. 2018.

[4] C. Zapponi. Sorting - visualizing sorting algorithms. http://sorting.at/, 2014. Accessed: Nov. 2018.

[5] X. Xu. Pathfinding.js. https://qiao.github.io/PathFinding.js/visual/. Accessed: Oct. 2018.

[6] M. Rosulek. Vamonos: Dynamic algorithm visualization in the browser. https://rosulek.github.io/vamonos/demos/. Accessed: Nov. 2018.

[7] D. Galles. Data structure visualizations. https://www.cs.usfca.edu/~galles/visualization/about.html. Accessed: Nov. 2018.

[8] D. Meech. Algomation - animated algorithms. http://www.algomation.com/, 2014. Accessed: Nov. 2018.

[9] A. Patel. Introduction to the a* algorithm. https://www.redblobgames.com/pathfinding/a-star/introduction.html, 2016. Accessed: Nov. 2018.

[10] M. Bostock. Visualizing algorithms. https://bost.ocks.org/mike/algorithms/, 2014. Accessed: Mar. 2019.

[11] N. Case. Explorable explanations. https://explorabl.es/. Accessed: Oct. 2019.

[12] A. A. Kai Hormann. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131–144, November 2001. https://doi.org/10.1016/S0925-7721(01)00012-8.

[13] H. F. M. A. Jayaram. Convex hulls in image processing: A scoping review. *American Journal of Intelligent Systems*, 6(2):48–58, 2016. doi: 10.5923/j.ajis.20160602.03.

[14] R. L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, June 1972. https://doi.org/10.1016/0020-0190(72)90045-2.

[15] T. O. J. Bentley. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, September 1979. https://doi.org/10.1109/TC.1979.1675432.

[16] G. Golikov. bentley-ottman. `https://github.com/ggolikov/bentley-ottman`, 2019.

[17] J. Foley, F. Van, A. Van Dam, S. Feiner, J. Hughes, J. Hughes, and E. Angel. *Computer Graphics: Principles and Practice*. Addison-Wesley systems programming series. Addison-Wesley, 1996. ISBN 9780201848403. URL `https://books.google.pt/books?id=-4ngT05gmAQC`.

[18] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, Jan. 1974. ISSN 0001-0782. doi: 10.1145/360767.360802. URL `http://doi.acm.org/10.1145/360767.360802`.

[19] B. N. Delaunay. Sur la sphère vide. *Bull. Acad. Sci. URSS*, 1934(6):793–800, 1934.

[20] D. Sinclair. S-hull: a fast radial sweep-hull routine for delaunay triangulation, 2016.

[21] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer Berlin Heidelberg, 2008. ISBN 9783540779742. URL `https://books.google.pt/books?id=9nhHUZpKzeEC`.

[22] B. Braden. The surveyor's area formula. *The College Mathematics Journal*, 17(4):326–337, 1986. ISSN 07468342, 19311346. URL `http://www.jstor.org/stable/2686282`.

[23] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2^{nd} edition, 1994. ISBN:978-0521649766.

Appendix A

Profile Questionnaire

GeomVis

Estamos a desenvolver uma ferramenta para visualizar algoritmos de forma a facilitar a sua aprendizagem e compreensão.
O inquérito é anónimo e toda a informação que nos der será usada exclusivamente para fins académicos.
As suas respostas ajudarão a melhorar o visualizador e a perceber se será útil no ensino.
Obrigado pela sua participação!

* Required

Perfil

1. **Idade** *
   Mark only one oval.

   - ◯ Menos de 18
   - ◯ 18-21
   - ◯ 22-24
   - ◯ 25-30
   - ◯ 31-40
   - ◯ 41-50
   - ◯ Mais de 50

2. **Sexo** *
   Mark only one oval.

   - ◯ Masculino
   - ◯ Feminino
   - ◯ Outro

3. **Habilitações literárias (se está a realizar alguma, escolha essa)** *
   Mark only one oval.

   - ◯ Ensino Básico
   - ◯ Ensino Secundário
   - ◯ Licenciatura
   - ◯ Mestrado
   - ◯ Doutoramento ou Pós-doutoramento

4. **Classifique a sua familiaridade com cada um dos algoritmos abaixo** *
Mark only one oval per row.

	Não conheço	Conheço	Compreendo mas não sei implementar	Consigo implementar recorrendo a documentação	Consigo implementar sozinho
Bubble-sort					
Binary search					
Depth First Search (DFS)					
Djikstra's Algorithm					
Cohen-Sutherland					
Sutherland-Hodgman					
Bentley-Ottmann Line Segment Intersection					
Point in Polygon					

Appendix B

Evaluation Exam

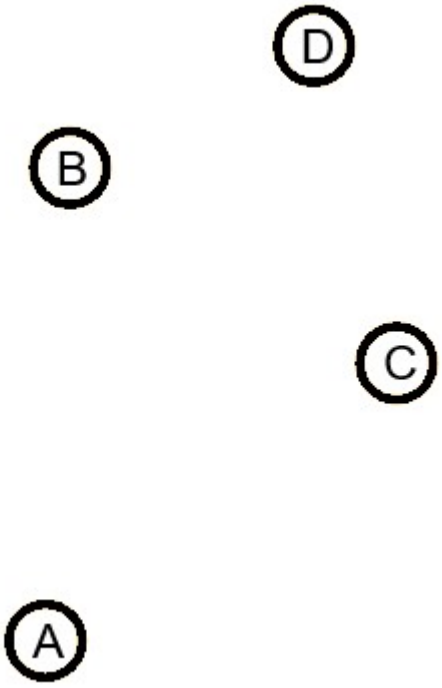
GeomVis - Teste

* Required

1. **(grupo - a preencher pelo observador)**
   Mark only one oval.

   - T
   - G

Exercise 1 - answer the following questions using this image.



2. **1.1) In the first step of the algorithm, the points other than the first one are sorted. What is the correct order in this case?** *
   Mark only one oval.

   - D, B, C
   - C, D, B
   - B, C, D
   - B, D, E

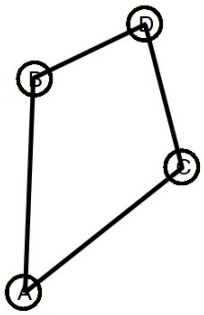
3. **1.2) What points are initially pushed to the stack?** *

Mark only one oval.

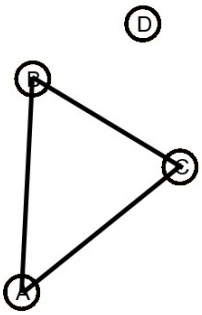
- A, B
- B, D
- A, A
- A, C

4. **1.3) Choose the correct convex hull for this set of points.** *

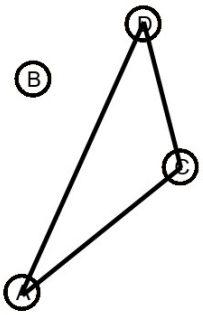
1



2



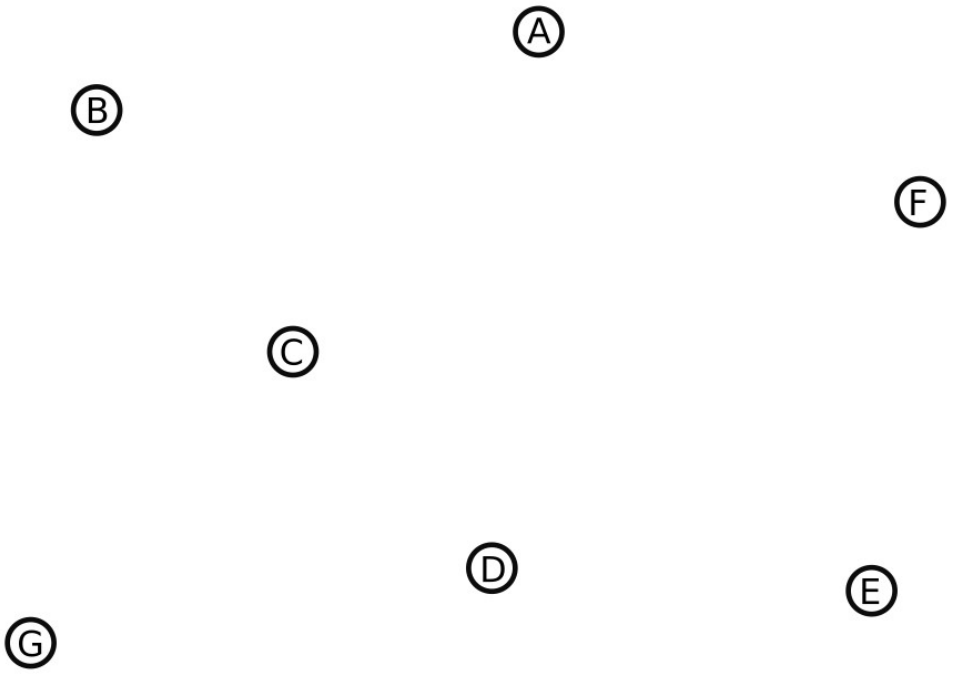
3



Mark only one oval.

- 1
- 2
- 3

Exercise 2 - answer the following questions using this image.



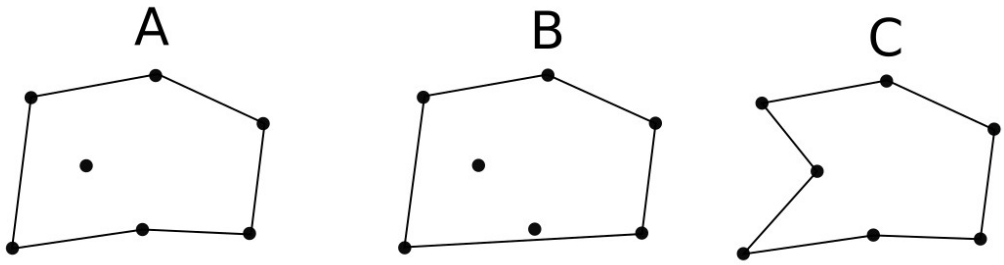
5. **2.1) Mark each statement as true or false.** *
Mark only one oval per row.

	True	False
Point A is added before point B.	◯	◯
E->D->F is a valid part of the final convex hull.	◯	◯
A->B->G is a right turn.	◯	◯
C is pushed to and popped from the stack during execution.	◯	◯

6. **2.2) Which points are initially pushed to the stack?** *
Mark only one oval.

- ◯ A, B
- ◯ G, B
- ◯ G, D
- ◯ G, E

7. **2.3) Choose the correct convex hull for this set of points.** *



Mark only one oval.

- ◯ A
- ◯ B
- ◯ C

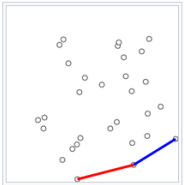
Appendix C

Control Group Text

Graham scan

From Wikipedia, the free encyclopedia

Graham's scan is a method of finding the convex hull of a finite set of points in the plane with time complexity O(n log n). It is named after Ronald Graham, who published the original algorithm in 1972.[1] The algorithm finds all vertices of the convex hull ordered along its boundary. It uses a stack to detect and remove concavities in the boundary efficiently.



A demo of Graham's scan to find a 2D convex hull.

Algorithm [edit]

The first step in this algorithm is to find the point with the lowest y-coordinate. If the lowest y-coordinate exists in more than one point in the set, the point with the lowest x-coordinate out of the candidates should be chosen. Call this point P. This step takes O(n), where n is the number of points in question.

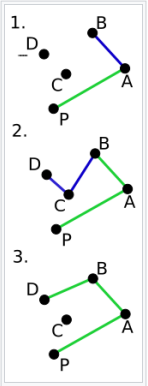
Next, the set of points must be sorted in increasing order of the angle they and the point P make with the x-axis. Any general-purpose sorting algorithm is appropriate for this, for example heapsort (which is O(n log n)).

Sorting in order of angle does not require computing the angle. It is possible to use any function of the angle which is monotonic in the interval $[0, \pi]$. The cosine is easily computed using the dot product, or the slope of the line may be used. If numeric precision is at stake, the comparison function used by the sorting algorithm can use the sign of the cross product to determine relative angles.

The algorithm proceeds by considering each of the points in the sorted array in sequence. For each point, it is first determined whether traveling from the two points immediately preceding this point constitutes making a left turn or a right turn. If a right turn, the second-to-last point is not part of the convex hull, and lies 'inside' it. The same determination is then made for the set of the latest point and the two points that immediately precede the point found to have been inside the hull, and is repeated until a "left turn" set is encountered, at which point the algorithm moves on to the next point in the set of points in the sorted array minus any points that were found to be inside the hull; there is no need to consider these points again. (If at any stage the three points are collinear, one may opt either to discard or to report it, since in some applications it is required to find all points on the boundary of the convex hull.)

Again, determining whether three points constitute a "left turn" or a "right turn" does not require computing the actual angle between the two line segments, and can actually be achieved with simple arithmetic only. For three points $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$, compute the z-coordinate of the cross product of the two vectors $\overrightarrow{P_1P_2}$ and $\overrightarrow{P_1P_3}$, which is given by the expression $(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$. If the result is 0, the points are collinear; if it is positive, the three points constitute a "left turn" or counter-clockwise orientation, otherwise a "right turn" or clockwise orientation (for counter-clockwise numbered points).

This process will eventually return to the point at which it started, at which point the algorithm is completed and the stack now contains the points on the convex hull in counterclockwise order.



As one can see, PAB and ABC are counterclockwise, but BCD isn't. The algorithm detects this situation and discards previously chosen segments until the turn taken is counterclockwise (ABD in this case.)

Pseudocode [edit]

The code below uses a function ccw: ccw > 0 if three points make a counter-clockwise turn, clockwise if ccw < 0, and collinear if ccw = 0. (In real applications, if the coordinates are arbitrary real numbers, the function requires exact comparison of floating-point numbers, and one has to beware of numeric singularities for "nearly" collinear points.)

Then let the result be stored in the `stack` .

```
let points be the list of points
let stack = empty_stack()

find the lowest y-coordinate and leftmost point, called P0
sort points by polar angle with P0, if several points have the same polar angle then only keep the farthest

for point in points:
    # pop the last point from the stack if we turn clockwise to reach this point
    while count stack > 1 and ccw(next_to_top(stack), top(stack), point) < 0:
        pop stack
    push point to stack
end
```

Now the stack contains the convex hull, where the points are oriented counter-clockwise and P0 is the first point.

Here, `next_to_top()` is a function for returning the item one entry below the top of stack, without changing the stack, and similarly, `top()` for returning the topmost element.

This pseudocode is adapted from *Introduction to Algorithms*.

Categories: Convex hull algorithms

Appendix D

User Tests Script

Guião de testes GeomVis

O que vai ser avaliado

A eficácia de uma ferramenta de visualização de algoritmos, que permite aos utilizadores criar os seus próprios casos e percorrer o funcionamento passo-a-passo, na aprendizagem desses algoritmos.

O sistema

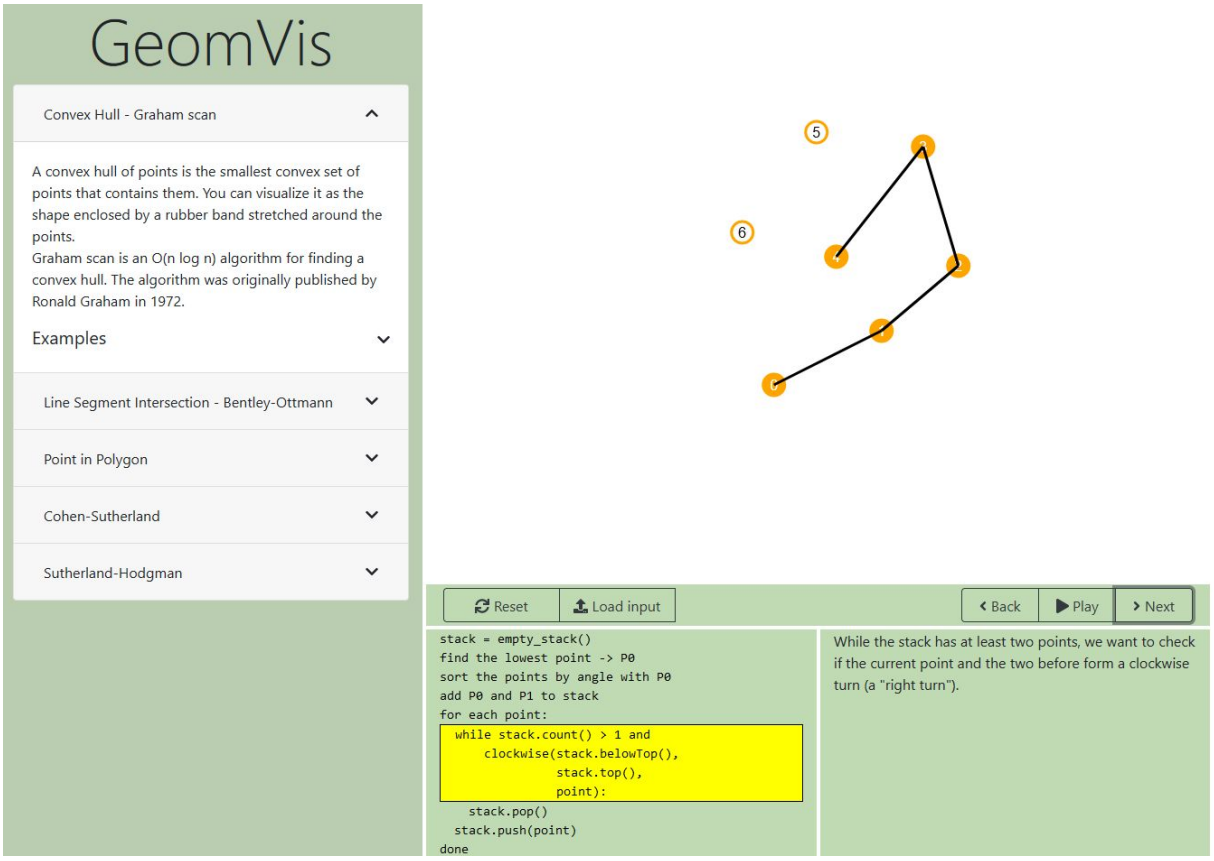


Fig. 1: interface durante a execução do algoritmo *Graham scan*

O sistema a avaliar é um visualizador de algoritmos de geometria computacional que corre num web browser. A aplicação corre no browser usando tecnologias web e permite controlo passo-a-passo da visualização. A interface permite inserir os dados de input dos algoritmos através de ações simples - no caso ilustrado na Fig. 1, o algoritmo *Graham scan*, cliques no canvas inserem pontos. Botões na parte inferior da interface permitem iniciar e controlar a visualização do algoritmo.

Ambiente

Os testes serão realizados numa sala do Pavilhão de Informática III do Instituto Superior Técnico (IST), ou em outros espaços do IST caso necessário. O material usado será um portátil com acesso à internet no qual um browser estará a correr a aplicação.
Durante os testes haverá um observador a gerir o teste; o observador não irá auxiliar o utilizador nas tarefas autónomas ou de avaliação. O observador tomará notas subjetivas sobre a experiência dos utilizadores. Serão tiradas algumas fotografias aos utilizadores durante os testes. Não será recolhido áudio ou vídeo.

Metodologia dos testes

Serão realizados testes intergrupos, sendo os utilizadores divididos em grupos A e B. Os passos serão os seguintes:

1. Acolhimento ao utilizador. Apresentação do projeto e explicação das tarefas que o utilizador deverá seguir. (2 min)
2. Preenchimento pelo utilizador de um questionário para registo de perfil e informação sobre o utilizador e assinatura do formulário de consentimento (2 min)
3. (Grupo A) Aprendizagem autónoma do algoritmo Graham Scan pelo utilizador, usando a explicação textual preparada (7 min)
   (Grupo B) Aprendizagem autónoma do algoritmo Graham Scan usando o GeomVis pelo utilizador, incluindo executar os vários exemplos providenciados (7 min)
4. Realização de um teste de avaliação de conhecimentos, com duas questões de várias alíneas, pelo utilizador. (5 min)

Os resultados do questionário e do teste de avaliação serão recolhidos de forma anónima usando o mesmo computador em que o teste foi realizado, através do Google Forms. O estado inicial do sistema será a aplicação já carregada no browser Firefox.

Métricas

- Número de respostas certas no teste de avaliação
- Tempo de realização do teste de avaliação
- Número de indecisões nas respostas ao teste (o utilizador mudar de resposta)

Formulário de consentimento informado

Esta avaliação é realizada no âmbito do Mestrado em Engenharia Informática e de Computadores do Instituto Superior Técnico e é parte da tese de mestrado “WebApp para Visualização de Algoritmos de Geometria Computacional”. O objetivo da avaliação é compreender a utilidade de um visualizador de algoritmos no ensino desses algoritmos no contexto da Geometria Computacional. O seu contributo será uma grande ajuda.

O trabalho será apresentado no final de 2019 e pode dar origem à publicação de artigos ou apresentação em conferências. Em qualquer um dos casos, a informação será usada de forma anónima.

A sua participação na avaliação é voluntária. Pode recusar participar ou retirar-se em qualquer altura.

Contacto para informações/pedidos: ricardofarracho@tecnico.ulisboa.pt

	Sim	Não
Dou consentimento para que os dados recolhidos sejam guardados pelos autores do trabalho até um período máximo de 1 ano após a conclusão e avaliação do trabalho.	☐	☐
Dou consentimento para que os dados recolhidos sejam usados, para fins científicos, de forma exclusivamente anónima.	☐	☐
Dou consentimento para que sejam recolhidas fotografias durante a realização da avaliação.	☐	☐

Assinatura: _____ Data: _____