# GeomVis: WebApp for Computational Geometry Algorithm Visualization

Ricardo Farracho
ricardofarracho@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

**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

## 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.

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 (on a whiteboard, for example) or digital diagrams to illustrate program execution while explaining how those algorithms work and what steps they take.

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 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.

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

**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.

## 2. Related Work

For our work, we analyzed several existing tools and websites that provide algorithm visualization. Here we look at those tools. Afterwards, we compare them, discuss their characteristics, and present our conclusions.

### 2.1. VisuAlgo

VisuAlgo [10] 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.

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. 1 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.

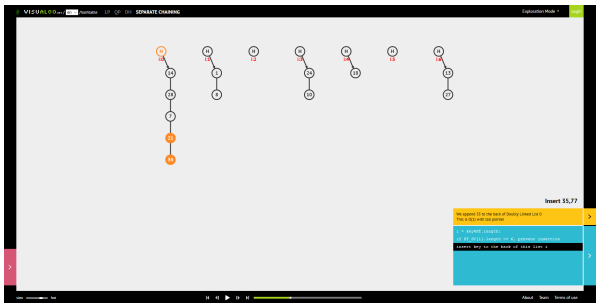

Figure 1: VisuAlgo visualization of an insert operation on a hash table

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 Animation and Visualizations

Algorithm Animations and Visualizations ("Algoanim") is a "collection of computer science algorithm animations and visualizations" [12]. Like VisuAlgo, the website features numerous small animations of particular operations and algorithms, mostly around searching and sorting data structures. The visuals are relatively simplistic and disparate in appearance, with the available features differing significantly between algorithms. The inputs are pre-defined, which limits the usefulness of the animations, as only the given examples can be visualized.

### 2.3. Algo-visualizer

Algo-visualizer [7] 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 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. The page is divided into three sections: a list of algorithms, the visualization, and code panels.

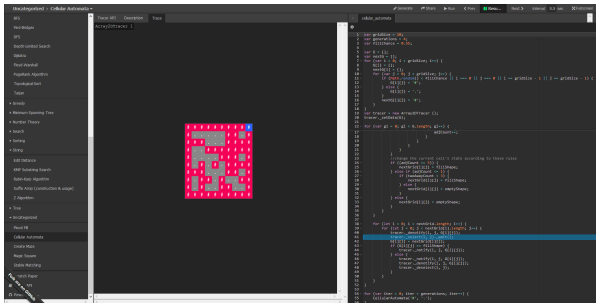

Figure 2: Algo-visualizer visualizing a cellular automata algorithm

### 2.4. sorting.at

sorting.at [14] 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 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 site supports step-by-step playback but not custom inputs.

### 2.5. Pathfinding.js

PathFinding.js [13] 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. Playback consists of the grid being painted by the algorithm's exploration of the space. The visualization is confusing as there is no explanation of what the colors of the squares mean, and step by step controls are not available.

### 2.6. Vamonos

Vamonos [9] is yet another collection of visualizations of algorithms, mostly sorting and graph algorithms. The interface features full pseudo code with step highlighting and step-by-step controls, and precise illustration of the states of graph nodes for graph algorithms. Nodes can also be created, connected, disconnected, or removed before the algorithm is run, although the editing interface is not very intuitive.

### 2.7. Data Structure Visualization

Data Structure Visualization [2] 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. For some of the visualizations, the input can be changed, but only randomly within a set of fixed inputs. This tool also has step-by-step controls like most of the others.

### 2.8. Algomation

Algomation [6] 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. 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.

### 2.9. Introduction to A*

Red Blob Games's Introduction to A* [8] 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. 3 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 an optimization which immediately terminates the algorithm when the goal is found. Walls can be added or removed and the goal location may be changed through direct input using the mouse.

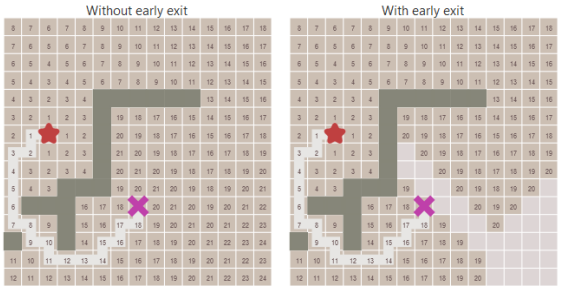


Figure 3: Comparison between Breadth-First Search with and without early exit, from Introduction to A*

### 2.10. Discussion

We now compare the platforms by looking at a number of attributes. The comparison is summarized in Table 1, with columns for each attribute and rows for each analyzed tool. We will 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. 'Pseudocode' 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 pseudocode, 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.

We also note that only some of the tools are open-source, as indicated by the last column in Table 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.

## 3. GeomVis

GeomVis[2] is a tool 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. GeomVis is licensed under the open-source MIT License; 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 section, we describe how GeomVis is implemented, starting with a general architecture overview in Section 3.1, a look at each component in Sections 3.2 and 3.5, and an explanation of the implementation of the visualization system itself in Section 3.3.

### 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, CSS and SVG tecnologies for the algorithm visualizations. The algorithms, interaction logic and file processing are implemented in TypeScript. Our design uses a main user interface component and a set of algorithm modules. These modules are described in more detail in sections to come 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.

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. 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 Section 4.

Each module has four components. These 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.

[2] http://3dorus.tecnico.ulisboa.pt/geomvis/
[3] https://github.com/ABCRic/GeomVis

| | 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 | ✓* | ✓* | ✓ | |

Table 1: Comparison between applications and tools. ✓* indicates the subject partially fulfills the attribute.

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.

The **Input and canvas setup** methods. These methods are executed when the algorithm is selected via the user interface. Adding required elements for visualization and hooking into browser input events to support direct input.

**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 3.3 below.

### 3.3. 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 4 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 3.2 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.

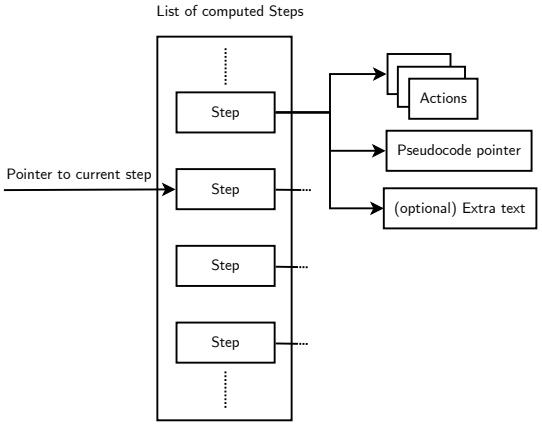


Figure 4: Diagram of the visualization data structures

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 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.

### 3.4. 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 circle to the canvas; drawing a line between two points; 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`; `stepToPrevious`; `stepToNext`; and `stepFromNext`. These callbacks form the four possible transitions in and out of a step.

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. To this effect, we 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). As 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 appear when relevant. Fig. 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. 6 shows a closeup of the Graham scan algorithm on the list, with its three associated examples.

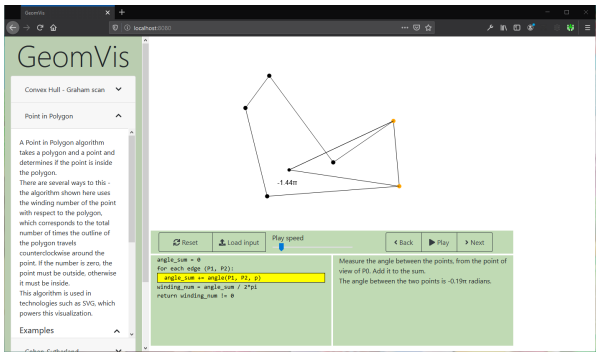


Figure 5: The GeomVis user interface.

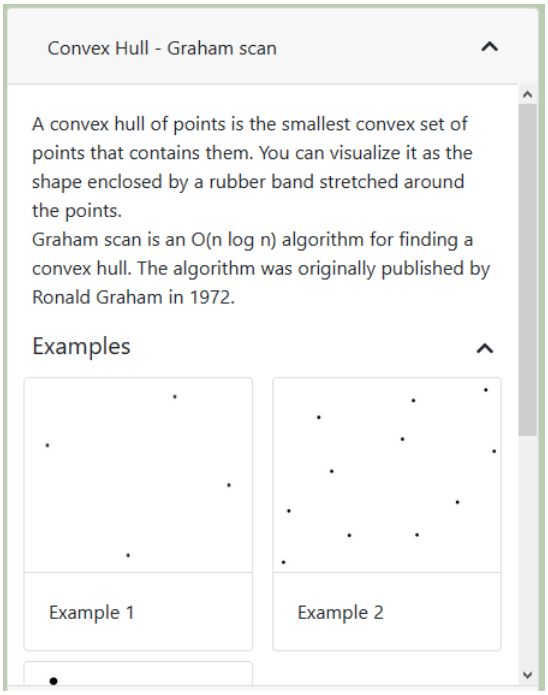


Figure 6: Graham scan description and examples on the interface's algorithm list

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 al-

gorithm 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. 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 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.

## 4. Use Cases

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 implemented visualizations for them.

A Point in Polygon algorithm takes a point of the plane and a polygon and determines whether the point lies inside the polygon. We implemented the variant of the algorithm known as the *winding number algorithm*, which computes the winding number of a point relative to a polygon [5].

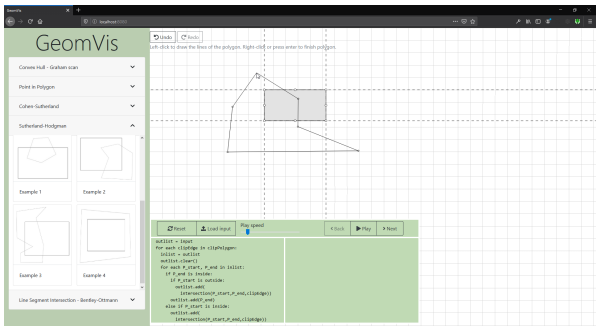


Figure 7: GeomVis interface while editing input.

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 $X$. We implemented a visualization for Graham's Scan [3], a $O(nlogn)$ time complexity algorithm for finding a convex hull given a set of points.

A Line Segment Intersection algorithm takes a set of line segments and determines at what points those segments intersect. We implemented a visualization for the Bentley-Ottman algorithm [4], which takes a sweep line approach to finding the intersection points.

The fourth algorithm we included is the Cohen-Sutherland algorithm [1], which is used in computer graphics for line clipping. This algorithm is useful in computer graphics to determine which lines or portions of lines should be considered or discarded for rendering.

The final algorithm implemented in GeomVis is the Sutherland-Hodgman algorithm [11], which clips polygons into a viewport. This algorithm is used by 3D rendering applications to prevent the unnecessary work of rendering portions of the scene that are not currently visible.

## 5. Results

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 section.

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.

Tests were carried out with twenty users at studying locations at IST. 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 test process was as follows. First, the user was welcomed and received an introduction to what

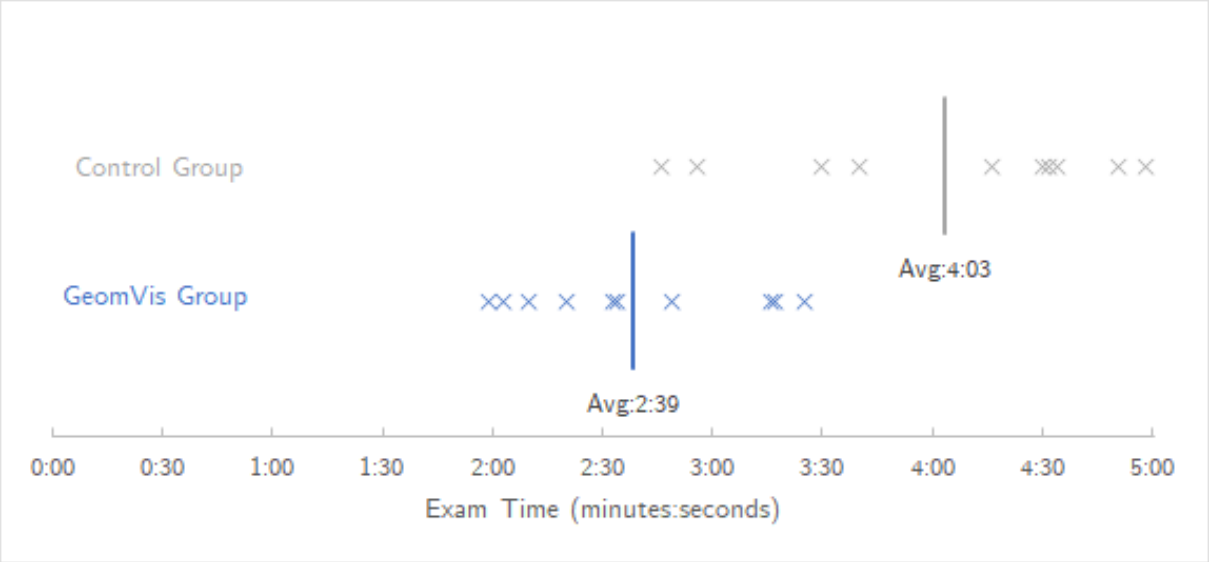


Figure 8: Exam execution time distribution

GeomVis is and an explanation of the tasks they would be asked to do. Then, users were asked to 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. 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) using a static text and diagram explanation based on a common, openly accessible, and easy to find source, Wikipedia. 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 to evaluate the knowledge they acquired.

We note 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. 9).

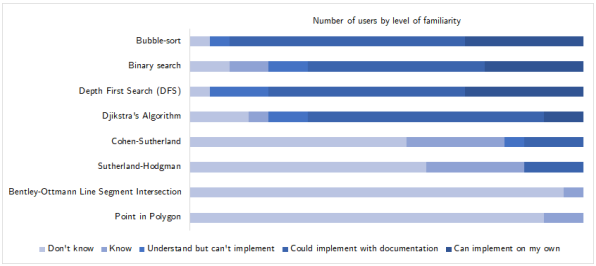


Figure 9: Test users' familiarity with algorithms

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. None of the users were familiar with the specific algorithm being tested (users that did would be disqualified from testing) so that previous knowledge of it did not affect the results.

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); and 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.

| | 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 2: Averages and standard deviations of exam scores and times.

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.

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. 8 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 exam and the control group having 1.2 on average. 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.

The results of the three metrics are summarized in Table 2. Overall, we believe these are promising results and reject our null hypothesis $H_0$, posed at the start of this section, 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.

## 6. Conclusions

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.

**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.

**Offer algorithm implementations.** It would be useful for implementations of the included algorithms in several programming languages to be provided.

**Extend the visualization model to technologies other than SVG.** 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 e.g. WebGL, for 3D graphics.

In conclusion, we find that the application we developed for this work, and future developments of it, are worthwhile. Although there is much more that could be done, some of which we suggested above, our results were positive that our current solution is effective in aiding our goal - teaching algorithms better.

**Acknowledgements**

I would like to thank my advisor Alfredo Ferreira for his guidance during the development of this work, along with my family and friends for their support. I would also like to thank the volunteers who helped me test GeomVis and validate my work. Finally, I would also like to acknowledge the authors of previous work in this area whose works served as inspiration my own.

**References**

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

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

[11] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, Jan. 1974.

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

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

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