

INSTITUT FÜR INFORMATIK

Masterarbeit

**LiveCG: a Framework for Interactive Visualization of
Algorithms from Computational Geometry**

Sebastian Kürten
Matrikelnummer: 4205308
sebastian.kuerten@fu-berlin.de

Erstgutachter:
Prof. Wolfgang Mulzer

Zweitgutachterin:
Prof. Agnès Voisard

Berlin, den 17. März 2014

Abstract

The field of Computational Geometry is devoted to geometric problems and to finding efficient algorithms for solving them. Given that the problems of the field are typically concerned with geometric objects and manipulations of such, it appears natural to visualize the geometric context of a problem when studying it. This thesis describes the design and implementation of a framework for the visualization of algorithms and data structures from computational geometry. This framework can assist scientists and students in creating useful visualizations with reduced effort by profiting from the ready to use components of the system. It also serves as a tool for people interested in learning about specific algorithms as a good visualization can be a helpful instrument in understanding complex problems. For some basic algorithms of the field, animated visualizations have been created that will be presented as well.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Berlin, den 17. März 2014

Contents

1	Introduction	1
1.1	Structure of the Document	2
1.2	Objectives and Contribution	2
2	Background	4
2.1	Computational Geometry	4
2.1.1	The Sweep Line Technique	4
2.1.2	The Doubly-Connected Edge List (DCEL)	5
2.1.3	General Position	5
2.2	Algorithm Visualization	5
2.2.1	History	7
2.2.2	User Roles in Visualization Systems	7
2.2.3	Overview of Available Visualization Systems	8
2.2.4	Other Visualization Material	9
3	LiveCG	10
3.1	Relevant Software and Technology	10
3.2	Project Details	12
3.3	Data Model	13
3.4	File Format	14
4	User Perspective	17
4.1	Main Graphical User Interface (GUI)	17
4.2	Example: a Visualization Dialog	20
4.3	Command-Line Interface (CLI)	22
4.4	Advanced Configuration	23
5	Developer Perspective	25
5.1	Implementing Visualizations	25
5.2	Rendering Subsystem	28
5.2.1	Drawing Layer	29
5.2.2	Drawing Back Ends	31
5.3	Text Output Module	33

6	Implemented Visualizations	35
6.1	Doubly-Connected Edge List (DCEL)	35
6.2	Fortune's Sweep Line Algorithm for Computing Voronoi Diagrams	36
6.3	Partitioning Polygons into Monotone Pieces	38
6.4	Triangulating Monotone Polygons	39
6.5	Shortest Paths in Polygons	41
6.6	Chan's Algorithm for Computing the Convex Hull	43
6.7	Fréchet Distance	44
6.7.1	Free Space Diagram	45
6.7.2	Distance Terrain	46
6.8	Buffer Regions	47
7	Conclusions and Future Work	49
7.1	Summary	49
7.2	Future Work	49
	References	51
	Acronyms	55
A	Source Code	57
A.1	Painter Interface	57

Using graphical visualizations for modeling abstract topics is a widespread method applied across virtually every discipline. Computer science is no exception to this and visualizations are widely used for explaining concepts, data structures and algorithms to make them easier to grasp. We regularly employ drawings to illustrate the properties of data structures such as lists, trees or graphs and the operations that are defined upon them. For many topics such as automata or graph theory, the graphical abstraction is ubiquitous. Other forms of visual abstraction such as diagrams are also very popular. Entity-relationship diagrams are used to describe data models and the Unified Modeling Language can be used to describe various aspects of complex software or hardware systems.

Computational geometry deals with problems that usually come from a geometric domain or are representable in it. Utilizing graphical representations is thus an even more natural approach than with topics where the graphical depiction is purely abstract. For many topics of the field it is certainly a good idea to start explaining them by drawing a suitable picture, because it may serve best for illustrating a geometric problem or an algorithm's general ideas. Moreover, explaining individual steps of an algorithm often requires a sketch of the geometric objects involved. Analysing degenerate cases may also be easier when actually seeing what is happening in a special situation.

However, creating proper drawings by hand may be exceptionally hard for many algorithms and data structures. In contrast, the pictures created by a computer system can usually be arbitrarily accurate even for complex situations. Therefore, we are interested in creating such drawings using software. Such software should allow us to inspect the behaviour of an algorithm in general or for specific situations of interest. A suitable program could be a valuable tool for teaching, self-study and for the creation of publications.

Even though they could be quite helpful, there are only few software-generated visualizations available in the field of computational geometry. This motivates the development of more visualizations, especially for algorithms where none exist so far. Because there are many recurring tasks to be solved across different visualization implementations, a framework should be a useful tool in making the process of creating visualizations simpler and the results

more widely usable. This thesis describes the design and implementation of such a framework called *LiveCG*, which is an acronym for *Live Interactive Visualization Environment for Computational Geometry*. Furthermore, it also presents the visualizations that have been implemented using this framework.

1.1 Structure of the Document

In the next section, I will first define objectives for this thesis and anticipate which of them have been achieved. The next chapter will then provide background information on relevant topics and technology and clarify terminology used throughout the thesis. First of all, computational geometry and some of its core concepts will be introduced briefly (Section 2.1). Next, algorithm visualization will be defined and I will give an outline of its history and the landscape of visualization software systems (Section 2.2). The following chapters present LiveCG: Chapter 3 introduces the framework in general; Chapter 4 presents it from a user’s perspective; and Chapter 5 focuses on the visualization developer’s point of view. Chapter 6 will then illustrate the visualizations that have been implemented. The relevant algorithms are introduced briefly, but the focus is on the way their respective visualizations display insightful information about them. The last chapter aggregates the results and gives an outlook on what could be done in the future.

1.2 Objectives and Contribution

No modern visualization framework specialized on computational geometry could be found (see Section 2.2.3). Although some of the general-purpose systems implement a few algorithms of the field as well, they do not seem to fulfill the typical requirements imposed on such a system. Hence, no system could be identified that provides a good starting point for further development of geometric visualizations.

In order to improve this situation, our main objective is to create a new framework for the visualization of data structures and algorithms of the field. Although technologically obsolete, the Workbench for Computational Geometry [Eps+94] looks like an archetype of the kind of system we have in mind. Together with pioneering general purpose systems, like BALSAs [BS84], it serves as an inspiration for our system.

To summarize our contributions, a flexible and modern system has been created that could serve as a basis for the visualization of various algorithms from computational geometry. For the end user, the system leverages interactivity, features intuitive graphical user interfaces and offers multiple export options for visualization graphics. Although graphical representations are central to the framework, the system is designed to support textual explanation to in-

crease the educational value. From the visualization developer's perspective it offers many reusable components that make the development of visualizations more efficient than by starting from scratch.

The framework focuses on pure algorithm visualization and there are no attempts to integrate features of program visualization. Likewise, it does not offer any form of scripting environment that would allow end users to easily experiment with the implementation of new algorithms. Although it tries to simplify the developer's job as far as possible, there are still a number of programming tasks that have to be completed for creating new visualizations.

For the sake of simplicity, we decided to focus on problems with 2-dimensional geometric input. Furthermore, the scope of geometric objects is limited to those consisting of a finite number of straight line segments. To enable the user to generate the input for the visualizations in terms of such data, we have created a software component for the manipulation of basic geometric objects.

The system implements a number of visualizations. The major algorithms are Fortune's Sweep for computing Voronoi diagrams, an algorithm for computing shortest paths in polygons and Chan's algorithm for computing the convex hull. Concerning polygons, there are also visualizations for partitioning them into monotone pieces and for triangulating them. As a building block for others, a reusable visualization of the DCEL is available. Also, there are interactive diagrams that are related to the computation of the Fréchet distance and to the computation of buffer regions.

2.1 Computational Geometry

Computational geometry is a relatively young discipline that emerged in the 1970s [Ber+00]. Nowadays, a large community of researchers holds conferences and writes dedicated journals on the subject. According to de Berg et al. [Ber+00, p.2] it can be defined as “the systematic study of algorithms and data structures for geometric objects, with a focus on exact algorithms that are asymptotically fast”.

Typical topics considered in introductory books [PS85; Ber+00] include geometric searching, convex hulls, intersections of geometric objects, Voronoi diagrams, triangulations, the geometry of rectangles, linear programming and robot motion planning. Results of the field have broad applications in various areas such as computer graphics, robotics, geographic information systems (GIS), and computer aided design (CAD), engineering (CAE) and manufacturing (CAM).

Efficient algorithms for geometric problems often employ general algorithmic techniques also used in other fields of computer science such as divide-and-conquer, balancing, recursion or dynamic programming. Well-known data structures such as priority queues or those for representing ordered or unordered sets are also frequently used. However, there are also important recurring techniques and data structures that are rather unique to the domain of geometry.

2.1.1 The Sweep Line Technique

The *sweep paradigm* [PS85, p.11] is an important technique that serves as a design pattern for geometric algorithms. A line on the plane, the *sweep line*, is moved along the plane, and the steps of the algorithm are carried out at important positions of this line called *event points*. Usually, two data structures are involved: an *event point schedule* (EPS) and the *sweep line status* (SLS). The EPS organizes the event points. Depending on the algorithm, they may be computed in a pre-processing step or change dynamically during execution. The SLS maintains information about the intersection of the sweep line with the geometric objects involved and it is modified at the event points. Using

this structure, the algorithm computes the actual output, either on the fly or in a post-processing step.

2.1.2 The Doubly-Connected Edge List (DCEL)

The *Doubly-Connected Edge List (DCEL)* [MP78] is an essential data structure that is used in many algorithms. It represents subdivisions of the plane as a graph whose nodes correspond to its vertices, edges and faces. The nodes are interconnected so that an efficient traversal of the subdivision is possible. For example, we can traverse the edges around a face or iterate over all edges that coincide in a given vertex. As the name suggests, edges play a central role within the data structure and are organized similar to the elements of doubly-linked linear lists.

2.1.3 General Position

When talking about algorithms from Computational Geometry, it is often useful to make assumptions about input data that makes them easier to formulate and explain. Such input data is said to be in *general position*. What exactly that means depends on the algorithm. For example, it may mean that no three points lie on a line or that no four points lie on a circle. Another typical assumption is that no two points have the same y -coordinate.

An algorithm is then designed under the assumption that the input is in general position. Usually, it is possible to modify it afterwards to work with previously ignored special cases without changing the asymptotic worst case running time and space requirements. Either these cases have to be treated in the implementation specifically, or generic frameworks like the *perturbation technique* [BMS94] can be used to deal with degeneracies transparently. Although it appears to be debatable whether the latter concept is applicable in practice [BMS94], authors frequently leave out the details about degenerate cases to reduce complexity and to avoid distraction from an algorithm's major ideas.

2.2 Algorithm Visualization

An *algorithm visualization* communicates information about what an algorithm does and how it works. Price et al. [PBS98] point out that “visualization is a word that is often misunderstood, even by experienced English speakers”. They say many people believed it had only to do with drawing pictures although it had rather to do with forming a mental image of something not actually present to the sight. They further argue that this could be achieved by different forms of sensory input, not only through input to the visual field.

Hence, algorithm visualization is about forming a mental image of an algorithm, and different techniques can be used to accomplish this goal. From this perspective, describing the steps of an algorithm with natural language or via pseudocode is a form of visualization as well. Other methods may be more surprising, such as algorithm auralization where sound is used to illustrate an algorithm's behaviour. In spite of this broad definition, pictures are an important and widely used form of visualization as well. They can capture a lot of information in a comprehensible way — especially in the field of computational geometry where many objects and operations have a direct graphical representation.

An *algorithm animation* is a dynamic visualization that lets us observe the behaviour of an algorithm. Although applicable to sound as well, this usually refers to graphical illustrations where animation effectively results in a motion picture. In the simplest case the animation is a sequence of snapshots from the algorithm in subsequent states of execution. However, some animations go one step further and animate the transitions between individual steps of the algorithm.

A visualization or animation becomes *interactive* when the user, i.e. the person who is watching it, is able to interact with certain aspects of the visualization. This may mean that one can select the input to the visualized algorithm, navigate through the animation, or configure parameters of the algorithm or its presentation.

In this thesis I use the term *visualization* as an umbrella term that covers both animation and interaction as well as visual and non-visual techniques. However, algorithm visualization should be distinguished from the more general concept of *software visualization* which includes the former but which also includes *program visualization*. Program visualization is concerned with the lower level implementation of algorithms, as opposed to the higher level abstraction in which algorithms are usually described. Visualizing programs imposes requirements on a visualization framework that are not within the scope of this thesis. For example, it requires a display of a program's source code and a presentation of the runtime environment at the current state of execution similar to that of a typical interactive debugger.

Although there is not much scientific evidence whether algorithm animations really help in teaching and learning, they have been used for that purpose for many years now [SL98]. It appears to be popular belief that algorithm animation is a useful tool in learning about algorithms. For example, Byrne et al. [BCS96, p.2] write about the pros of algorithm animation:

The use of pictures and visualizations as educational aids is accepted practice; textbooks are filled with pictures, and instructors often diagram concepts on the blackboard to assist an explanation.

Animation goes one step further. While static visualizations can provide people with the essence of how something looks, is laid out, or is constituted, animation appears better able to explain a dynamic, evolving process.

Furthermore, it is often argued that interaction makes visualizations even more effective [Jef98].

2.2.1 History

With the above understanding of *algorithm visualization*, it has a very long history. Ever since people have studied algorithms, they have used natural and formal language and visual abstraction for describing them. Algorithm animation, on the other hand, became feasible only with advancements in technology and their availability to researchers. Interactive visualization became possible through the rise of workstations and personal computers. The first short films illustrating algorithm behaviour were published in conferences in the 1970's [BP98]. Nevertheless, many authors state that the field has its roots in movies and systems released in the early 1980's [SBL93; TD95; BP98]. The movie *Sorting Out Sorting* by Baecker and Sherman [BS81] is often portrayed as an important milestone and one of the first movies to illustrate algorithms dynamically. It compares nine sorting algorithms and presents original techniques that influenced the whole field.

Many publications can be found on the topic from the 80's and 90's. They are often concerned with the presentation of new systems for algorithm visualization. One of the first and most prominent of those is the *Brown Algorithm Simulator and Animator (BALSA)* [BS84] developed at Brown University in the 80's. The software has been designed for teaching purposes in an electronic classroom. Apparently, numerous algorithms have been implemented for this system. Depending on the algorithm, the user was able to inspect different aspects of it via specific views producing graphics of the algorithm's state. The user was able to step through the execution and inspect in parallel a number of views on underlying data structures or other useful displays.

2.2.2 User Roles in Visualization Systems

Different types of users are involved with an algorithm visualization framework. Price et al. [PBS93] identify the following roles: system developer, programmer, visualizer and user. The *system developer* creates the visualization system that all other parties use. The *user*, or *end user* to be more specific, is the person who views a visualization and interacts with it. This visualization has been implemented by the *visualizer* based on an algorithm implementation that a *programmer* provides. Visualizer and programmer have to cooperate to make a sophisticated visualization possible. In fact, the tasks of both roles are so

tightly coupled, that, for our considerations, they are one. Further along in the text, *visualization developer*, *developer* for short, shall stand for the merger of the visualizer and programmer.

2.2.3 Overview of Available Visualization Systems

Various different visualization systems developed by others in the past have been considered as a starting point for our work. Shaffer et al. [Sha+10] provide an overview of existing systems. Most of the older systems, such as BALSAs, Zeus, XTANGO or JCAT are no longer available due to technological obsolescence and their often extensive content is effectively lost. Yet, the systems have been presented in publications so that information about their features is conveyed.

Systems that are available today include Animal, JAWAA, JHAVÉ, AIViE, Matrix Pro and Jeliot 2000. Most of them are implemented in Java and can be run as standalone applications. Individual visualizations are often additionally made available through the web in the form of applets. For instance, TRAKLA2 is an e-learning platform that integrates the visualizations available through Matrix Pro and offers automatic assessment of student exercises to instructors. To increase accessibility, some projects try to switch to more browser-friendly programming environments by moving to HTML- and Javascript-based solutions. The OpenDSA project aims to create an interactive web-based textbook that includes interactive animations instead of static figures. The authors use a Javascript visualization library [KS13] to integrate well with modern browser technology. Similarly, the Data Structures Visualization project¹ provides a rich collection of visualizations that are available through the browser.

However, the focus of these systems is often primarily on fundamental computer science data structures and algorithms and the systems do not provide special means that are essential for working with geometric data. Historically, some systems have focused on computational geometry: the *XYZ GeoBench* [Sch90], *GeoLab* [RJ93], the *Workbench for Computational Geometry* [Eps+94], *GASP* [TD95], *GeomView* [Ame+95] and *GeoWin* [BN02]. Unfortunately, none of them qualifies as a valid starting point for further development. They are either not available today due to technological obsolescence or are only available under a commercial licensing model (GeoWin). An exception is GeomView, but this software is merely a sophisticated viewer for 3-dimensional geometric data and not a fully-featured algorithm visualization environment.

¹<http://www.cs.usfca.edu/~galles/visualization/>

2.2.4 Other Visualization Material

Apart from visualization systems, there are numerous videos and interactive animations available through different sources. Often, individuals created those visualizations and published them on their homepage. Fortunately, the AlgoViz project [Sha+11] strives to build and maintain a database of all available material. They categorized the visualizations they have collected and usually for each visualization a link is provided to where it can be executed. Their collection covers not only independent visualization projects but also the visualizations available through the aforementioned systems.

Shaffer et al. [Sha+10] have analyzed their collection and identified that the majority of visualizations are created for recurring topics, such as sorting algorithms, linear lists, trees and graphs, with few visualizations for specialized areas such as computational geometry.

Through additional research, more visualizations not yet listed by AlgoViz have been found. Most notably the GeometryLab², which provides an extensive collection of computational geometry visualizations. Another important source, primarily for videos presenting algorithm animations, is the *Annual Video Review of Computational Geometry*³ which is held yearly within the *ACM Symposium on Computational Geometry* since 1992. According to Hausner and Dobkin [HD99] it is “the main vehicle for dissemination of techniques in the field”. The conference website hosts the videos that have been submitted since 2003.

Unfortunately, only few authors publicly release their visualization’s source code and even fewer do so under an open source license that would allow it to be reused [Coo+14].

²<http://www.geometrylab.de/>

³<http://computational-geometry.org>

LiveCG is a framework for the creation of interactive visualizations of data structures and algorithms from computational geometry. To its users, the framework acts as a collection of visualizations and as an experimenting environment. It provides an extensive user interface that makes it easy to access the available visualizations for varying inputs (see Chapter 4). For developers, the framework provides software components that ease the process of creating visualizations. The challenge is to design them in such a way that they are reusable across different algorithms and visualizations to maximize the benefit for the developer (see Chapter 5).

While the following chapters focus on these different perspectives, this chapter addresses basic topics concerning the framework. We first describe relevant software and technologies (see Section 3.1) and then provide some general information about the project (see Section 3.2).

The framework's problem domain are algorithms operating on geometric data, and their implementation typically requires some abstract data types to represent the geometric objects they manipulate. Different algorithms may need different data structures, but the framework should provide the most common ones such that developers may save the effort of implementing those data structures themselves (see Section 3.3).

It should be possible to store collections of geometric objects for later usage. This allows for creating educationally helpful instances of input data that may be shared among instructors and students, made available on the web or included as examples into visualization applications. For this purpose, a file format for geometric data has been designed (see Section 3.4).

3.1 Relevant Software and Technology

Java

Java is an object-oriented programming language that compiles to Java byte-code instead of native machine code. This bytecode can be run on any operating system for which a *Java Virtual Machine (JVM)* is available, making Java programs executable on all major desktop operating systems without modification.

Swing

Swing is a library and widget toolkit for graphical user interfaces for Java that ships with its standard distribution. Similar to the language itself, Swing is platform independent and allows the programmer to create user interfaces regardless of the underlying operating system.

AWT

The *Abstract Window Toolkit (AWT)* is the original toolkit for graphical user interfaces for Java. Although it has been superseded by Swing, many parts of the library are still relevant. Most notably, the rendering engine (class `java.awt.Graphics2D`) and the geometry library (package `java.awt.geom`) have not been replaced with the introduction of Swing.

XML

The *Extensible Markup Language (XML)* is a language for describing structured data in plain text, which makes it both machine-readable and human-readable. It is superior to simpler plain text formats because it is easily possible to design robust file formats for storing complex information whose structure is still almost self-explanatory.

CGAL

The *Computational Geometry Algorithms Library (CGAL)* is an open source geometry library with a strong academic background written in C++. It covers a wide range of topics¹ and is probably the most extensive collection of implemented geometric algorithms. The design of many packages is very flexible to support different number types, precision models and distance metrics. When applicable, the algorithms are usually implemented to work with two, three or even arbitrary dimensions.

Due to its extensive coverage of topics, CGAL would naturally be the first choice among the available geometry libraries. Unfortunately, it is not directly usable from Java. Although language bindings are being developed², their usage would imply severe restrictions concerning the portability of the framework.

JTS

The *JTS Topology Suite (JTS)* is a geometry library for Java. When compared to CGAL, it covers only a small range of topics and is less flexible. On the other hand, it has a less complicated API.

The library has its roots in the Geographic Information System (GIS) context and the implemented features are designed to fit the requirements of that field. Its data model supports 2-dimensional geometric objects consisting of

¹<http://doc.cgal.org/latest/Manual/packages.html>

²<https://code.google.com/p/cgal-bindings/>

non-intersecting straight line segments. Supported features include Boolean set operations and spatial predicates for polygons; convex hulls; Voronoi diagrams and Delaunay triangulations; length and area measurements; curve simplification; and spatial indexing methods. It also offers robust basic tools for working with geometric data, such as orientation and collinearity tests; intersection computations with extended precision; and common data structures such as a quad-edge data structure that serves a similar purpose as the DCEL.

3.2 Project Details

LiveCG is free software and its source code is released under the terms of the General Public License³ (GPL) in order to encourage contributions from other developers. As such, it is available as a project⁴ on the Github⁵ platform.

For every software, the choice of the programming language is an important one. LiveCG has been developed in Java, which is still one of the most popular programming languages today⁶. Its prominence, especially in the academic context, should allow many potential developers to familiarize fast with the framework. In the past, the majority of visualizations has been programmed in Java [Coo+14], which facilitates porting existing visualizations to LiveCG.

Furthermore, the major functional requirements can be fulfilled using Java. On the one hand, graphical user interfaces are an integral component of our framework. Java features, through Swing, a rich toolkit for creating such interfaces that can be used without modifications on all major desktop platforms. On the other hand, our framework provides a number of programming libraries. Java's object-oriented design and strong type system enable the creation of reusable and clear programming interfaces. With JTS, there is also a geometry library available, which can be relied upon for robust computation of basic geometric operations and predicates. We can thus omit to implement them on our own, which is good, because even the simplest geometric operations tend to be tricky to implement robustly.

Arguably, when trying to reach as many users as possible, the web-browser is nowadays the obvious choice of technology [KS13]. In this respect, Java may not be the best choice after all, because the classical approach of bringing Java-based applications to the browser — applets — is unable to compete with modern, browser-specific technology, such as HTML5 and JavaScript. Unlike applets, they run on any modern browser and thus are not only available on the desktop, but also on mobile devices like smartphones and tablets.

Nevertheless, I think Java is still a viable choice for this project. On the

³<https://www.gnu.org/copyleft/gpl.html>

⁴<https://github.com/sebkur/live-cg>

⁵<https://github.com>

⁶<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, accessed 3/16/2014

one hand, LiveCG is primarily designed as a classical desktop application for which Java and Swing are an excellent platform. On the other hand, it is indeed possible to produce HTML5 and JavaScript based front ends using Java, when utilizing frameworks such as the Google Web Toolkit (GWT). It compiles Java source code to JavaScript and provides an API for creating browser based user interfaces. Special measures have been undertaken to enable compatibility of our system with GWT, which does not support all packages from the Java Standard Edition. Particularly, the package `java.awt.geom`, which contains useful implementations of the `Shape` interface, is not available there. Its usage has therefore been replaced with a compatibility layer. For this purpose, a small separate project called NoAWT⁷ has been created, which basically provides a refactored, GWT-compatible version of this package.

3.3 Data Model

A data model is needed to represent the basic geometric objects that the algorithms operate on in their implementation. A number of geometric objects is relevant to the domain: coordinates, nodes, line segments, lines, polygonal chains, polygons, planes, halfplanes, curves, different kinds of splines, triangles, rectangles, circles, ellipses and polyhedrons.

While there are in theory many types of objects that one might want to support through appropriate data structures, complexity increases with the number of supported object types. On the one hand, the more data types available, the more geometric problems could be modeled. Consequently, more visualizations could be implemented and hence, the more complete the framework. On the other hand, every extra data type supported results in additional complexity in various components of the framework: the file format has to support them, the user interface must provide means for manipulating them and the visualization implementations have to handle them. Facing this trade-off of complexity versus completeness, only a small subset of geometric objects has been chosen for implementation. The selection has been made such that all planned visualizations could be realized.

The basic geometric objects are: coordinates, nodes, polygonal chains and polygons (see Figure 3.1 for examples). A coordinate is a simple data structure that merely groups a tuple of floating point values to represent a point on the plane. A node has a coordinate and represents a point on the plane as well. The difference between a node and a coordinate is that the more complex objects, namely chains and polygons, consist of the former. Polygonal chains model a connected list of line segments and are implemented as a list of nodes. They may be closed in which case a segment exists from the last back to the first node

⁷<https://github.com/sebkur/NoAwt>

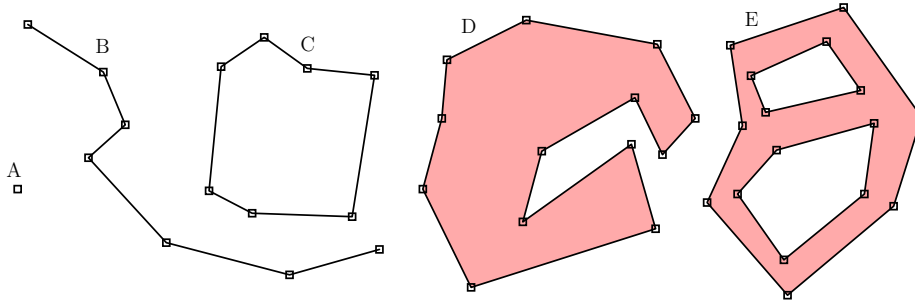


Figure 3.1: Supported geometric objects: node (A), polygonal chain (B), closed polygonal chain (C), polygon (D) and polygon with holes (E)

of the chain. A closed polygonal chain is also called a polygonal ring, or ring for short. Polygons model a bounded area in the plane where the boundary of the polygon is made up of connected line segments. A polygon is implemented such that it consists of a polygonal ring defining the outer boundary of the polygon plus optionally a number of polygonal rings that define holes in the polygon.

The data structures have been designed such that it is possible to explicitly model sets of objects as a network. That means for instance that two polygonal chains may not only coincide in individual nodes, they may actually share instances of the same node as part of their definition. If the coordinate of that node is altered, the shape of *all* chains that contain that node will change. Furthermore, the node “knows” that it is part of those chains, so that it is easy to traverse the network of geometric objects. To accomplish this, the data structures have been implemented with some redundancy. More precisely, apart from the chains storing references to the containing nodes, the nodes store additional references to the chains that they are part of. This redundancy allows for easy and fast examination and traversal of the network of objects, but comes at the cost of more expensive and complicated object manipulation.

The data types described above are those that are available as input to algorithms. For programmers, some additional data structures are provided that can be helpful when implementing visualizations. For example, there are abstract data types to represent rectangles, line segments and the DCEL.

3.4 File Format

It should be possible to store scenes of geometric objects in files so that they are available for later usage, exchange or publication. Evidently, some file format is required for storing the data described in the previous section. The encoded files should be editable with a simple text editor, i.e. be plain text and should have a descriptive and yet simple and compact structure. Apart from using the framework’s input editor, this will allow users to manipulate

files with their favorite tools and makes it easy to port existing data to the file format.

For various reasons, it is advisable to use an already established file format. For instance, this would make it possible to use existing viewers for inspecting those files or advanced editors for manipulating them. Also, there might already be data available that could be reused in the project. Since geometric data is widely used in many applications, a lot of different file formats exist that could possibly serve as a data format for LiveCG. Possibly viable candidates include the file format used by the OpenStreetMap project (`.osm` files), the Object File Format (`.off` files), used for example by the CGAL project, the Geography Markup Language (`.gml` files) or the Well-known-text (WKT) representation of geometric objects.

However, no format that appeared appropriate for our data of concern and all of our requirements could be found. Hence, a simple XML format has been designed. The recommended file name extension is `.geom` and it is inspired by some of the aforementioned formats. The basic structure is similar to that of `.osm` files, but it has been made less verbose with ideas borrowed from the way complex objects are described in `.off` and `.gml`.

Although the structure of our file format is probably most precisely described by its Document Type Definition (DTD), an example may be easier to understand. Listing 3.1 defines the geometric objects shown in Figure 3.1. Like any `.geom` file, it contains a `scene` XML element that defines width and

```

1 <scene width="570.0" height="180.0">
2   <node id="1" x="235.841" y="145.987" />
3   <node id="2" x="183.689" y="160.733" />
4   <node id="3" x="112.211" y="142.069" />
5   ...
6   <node id="42" x="419.0" y="70.0" />
7   <node id="43" x="397.0" y="27.0" />
8   <node id="44" x="321.0" y="13.0" />
9   <chain closed="false">1 2 3 4 5 6 7</chain>
10  <chain closed="false">8</chain>
11  <chain closed="true">9 10 11 12 13 14 15</chain>
12  <polygon>
13    <shell>16 17 18 19 20 21 22</shell>
14    <hole>23 24 25 26 27</hole>
15    <hole>28 29 30 31</hole>
16  </polygon>
17  <polygon>
18    <shell>32 33 34 35 36 37 38 39 40 41 42 43 44</shell>
19  </polygon>
20 </scene>

```

Listing 3.1: A file defining some geometric objects

height of the document. This element contains children defining the geometric objects on the scene. Three types are available in correspondence to the data

model: nodes, chains and polygons. A node stores a coordinate with attributes `x` and `y` and has an identifier `id` so that it can be referenced by other objects. Chains contain an ordered list of identifiers that reference nodes. The referenced nodes define the segments of the polygonal chain. The property `closed` can be used to mark a chain as being a ring. Polygons contain a mandatory child element `shell` that defines the outline of the polygon and optionally a number of `hole` elements that define the holes of the polygon. Both `shell` and `hole` are represented the same way as polygonal chains, only that they are implicitly closed.

Should the data model of LiveCG be extended in the future, it would be easy to extend the file format accordingly as well. To support more different types of objects, we could simply define corresponding XML elements. The system's file parser has been implemented to be forward compatible, i.e. it is able to handle files that contain unknown elements. It ignores such elements and informs the user about the unhandled data.

From the user's point of view, an important component — and also the starting point for any visualization — is the main user interface (see Section 4.1). With this tool, the user selects or designs the input for an algorithm and launches visualizations in separate dialogs (see Section 4.2). Apart from this graphical interface, the user can use a command-line interface for various purposes (see Section 4.3).

4.1 Main Graphical User Interface (GUI)

Algorithms usually transform some input into output. Consequently, offering the user a means for defining input data for algorithms is a basic challenge that every visualization faces. More often than not, standalone visualizations provide only rudimentary methods for this task, which affects negatively the overall usefulness of the visualization. In contrast, LiveCG provides a sophisticated software component that can be used for the creation and manipulation of geometric objects and their corresponding data structures. The main user interface puts this task at its center and is organized around it. At its core, the application maintains a current scene of geometric objects. Such scenes can be loaded from and stored to files, and can be manipulated by the user with different tools. The scene's objects can then be used as input to the different visualizations that are available through the **Visualizations** menu. The main window (see Figure 4.1) contains a menu bar, a toolbar, a status bar, and, at its center, the geometry editor.

Geometry Editor

The geometry editor displays the currently loaded scene of geometric objects. Basic operations on the scene are available via the **File** menu and via buttons in the toolbar: a new, empty scene can be created and it can be stored to and loaded from file.

Using different tools, the scene's objects can be manipulated. Creating, altering and deleting objects is primarily achieved by using the mouse pointer. There are six different mouse modes, selectable via the toolbar or via keyboard shortcuts:

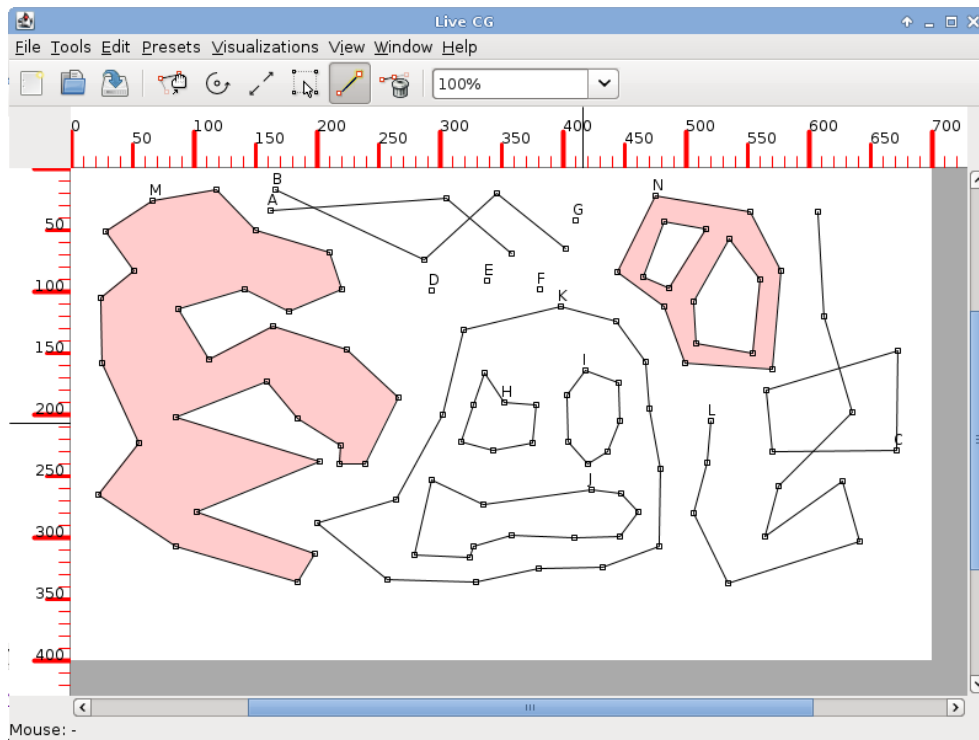




Figure 4.1: The geometry editor

- **Select and Move.** Objects can be selected in this mode. Clicking additional objects while holding **Shift** will add them to the current selection, and clicking already selected objects will deselect them. Selected objects can be moved around the scene by clicking one of them and dragging the mouse. When moving nodes, a snapping mode can be enabled by holding **Ctrl**, which will snap a dragged node to other nodes on the scene, thereby merging them to a single node.
- **Rotate.** Selected objects can be rotated by dragging the mouse around the initial drag position of the mouse.
- **Scale.** Selected objects will be surrounded by a rectangle with eight anchor points that can be used to scale those objects in either direction.
- **Rectangular Selection.** Clicking and dragging creates a rectangle on the screen. After releasing the mouse button all objects within the rectangle will be selected.
- **Draw Segments.** This mode is used for creating geometric objects. When no object is currently selected, a new object is created once the mouse is clicked. If a single node is selected, clicking will convert the node to a polygonal chain by creating an initial segment that connects the previously selected node and the current mouse position. If a polygonal chain and one of its endpoints are selected, clicking will add an additional

segment to the active chain. When doing so, the chain remains selected and the new node becomes selected, so that further segments can be added by subsequent clicks. A polygonal chain can be closed by holding  while clicking anywhere on the scene.

- **Delete.** Objects or individual nodes can be deleted from the scene by selecting them with the mouse. When deleting a node from a polygonal chain, the previous node will normally be connected to the next node. When holding  while deleting a node, the polygonal chain will be split in two separate chains instead.

Object dialog

The *Object dialog* can be reached via the menu (**Window** → **Object dialog**). It displays information and controls depending on the currently selected objects, thereby offering means to operate on them. When a single node is selected, its position can be changed through text fields displaying its coordinates. In addition, the process of snapping and thereby merging nodes can be reversed. By splitting a merged node, several independent nodes reappear on the scene that can be manipulated individually. When a polygonal chain is selected, a toggle button controls whether it is closed or not.

Operations on multiple selected objects can be performed as well: a filter menu allows the user to narrow the selection down to specific types of objects. For example it is possible to remove all nodes from the current selection, or to keep nothing in the selection except for closed chains. One or more rings can be converted to a polygon provided that one of them contains all other selected rings. Reversing that operation converts a polygon to its constituting closed polygonal chains. As a means for generating convex polygons, the dialog also offers a button for creating the convex hull of all currently selected objects and inserting it onto the scene as a polygon.

Apart from the currently implemented features, several more would certainly be useful: for example, it would be convenient to have controls for performing Boolean set operations on multiple polygons, or for simplifying polygonal chains.

Menu and Toolbar

The first three buttons in the toolbar provide the basic scene operations — new, save and load — and the next six buttons allow the user to select the current mouse mode. The next component is a drop down menu attached to a text field that can be used to alter the zoom used for displaying the scene. When the scene plus some extra margin does not fit into the window, the viewport can be moved, which is indicated by the scrollbars at the sides being

enabled. Dragging the scene while the right mouse button is pressed moves the scene within the viewport.

The menu **Preset**s offers access to the database of scenes bundled with LiveCG. When opened, it displays a number of available scenes that can be loaded into the editor. The scenes are organized hierarchically, grouped by the type of objects contained or by the visualization they have been designed for.

Opening the **Visualizations** menu gives access to the implemented visualizations of the system. This menu is also organized hierarchically, grouping the available visualizations by their topic. Choosing one of the visualizations entries will launch it in a separate dialog. The input for the visualization is taken from the scene currently opened in the editor.

4.2 Example: a Visualization Dialog

Once a visualization has been selected in the menu, a visualization dialog will appear. Figure 4.2 shows an example — the dialog for the Shortest Paths in Polygons visualization (see Section 6.5 for details about the visualization).

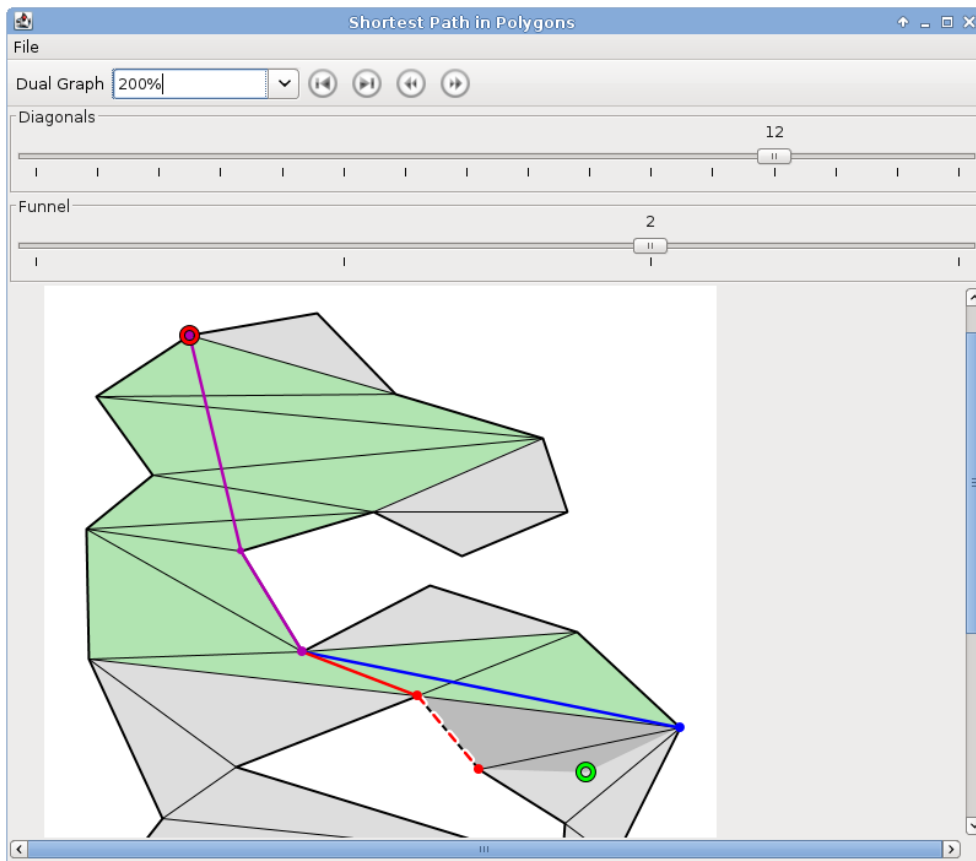


Figure 4.2: Dialog for Shortest Paths in Polygons

A dialog typically contains a menu bar, a toolbar, some control widgets and a visualization panel. Some visualizations may also show secondary di-

alogs that provide additional information, for example about involved data structures. The visualization panel displays the algorithm's current status in a meaningful way.

Usually, the algorithm's progress can be controlled by buttons in the toolbar and through specific control widgets. In this example, two sliders exist to control the two nested loops of the algorithm. The outer loop of the algorithm is over the diagonals of the polygon's sleeve. Moving the first slider controls which diagonal is currently under consideration. For each diagonal, the inner loop is executed, with which the algorithm traverses the nodes of the current funnel to find a valid funnel for the next step of the iteration. The second slider controls the progress of this traversal of the funnel.

The arrow buttons in the toolbar can be used for controlling the algorithm's status as well. The first two buttons control the major steps, i.e. move to the previous and next diagonal, while the next two buttons control the minor steps of the algorithm, i.e. the funnel traversal.

The toolbar usually contains some additional controls for manipulating the visualization's appearance. Typically, the visibility of certain features can be toggled. In this example only one such option is available, which toggles the visibility of the dual graph of the triangulation of the polygon. As with most of the visualizations, there is also a zoom control widget that lets the user define the magnification of the visualization panel. In our example, the magnification has been set to 200%.

Exporting Images

Each visualization can be exported to different image file formats. This functionality is available through the menu (**File** → **Export...**). The saved image will contain a snapshot of the algorithm at the same point of execution as in the dialog and with the same display settings as selected using the configuration options of the user interface. Currently the graphics can be stored as raster images (PNG), vector graphics (SVG), L^AT_EX-figures (TikZ) and Ipe files. Section 5.2.2 explains the available formats in detail.

Textual Explanation

Some algorithms feature textual explanations. When an algorithm supports this, the *Description dialog* (see Figure 4.3) updates whenever the algorithm's status changes and displays an explanation for the current step. The dialog logs all messages that have already been displayed before and highlights the current message with a boldface font.

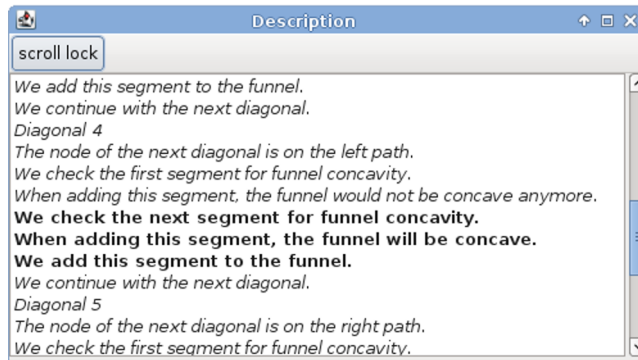


Figure 4.3: The Description dialog showing some explanations for Shortest Paths in Polygons

4.3 Command-Line Interface (CLI)

There are three different executables of the framework currently available: `livecg-ui`, `livecg-visualization` and `livecg-create-image`. All of them are available as bash and batch files for usage on Unix-based systems and on Microsoft Windows respectively.

The script `livecg-ui` launches the main user interface that has been presented in Section 4.1. It accepts as an optional argument a path to a geometry file to open as the initial scene:

```
> livecg-ui my-input.geom
```

To open a specific visualization for a geometry file directly, the `livecg-visualization` command may be used. The `visualization` flag is used to select the desired user interface to launch. Another argument is required which specifies the path to a geometry file to use as input for the visualization:

```
> livecg-visualization my-polygon.geom
    -visualization triangulation
```

The script `livecg-create-image` can be used to export an image file for a specified input file using one of the available visualization implementations. The user needs to specify input and output file, the visualization, and the desired output format. In addition it is possible to configure the visualization's appearance via command-line switches and to set the algorithm's status:

```
> livecg-create-image -input my-chains.geom
    -output my-freespace.ipe
    -visualization freespace -output_format ipe
    -Dreachable-space=true -Dfreespace-markers=true
    -Depsilon=100
```

A nice feature would be an extension to the export functionality of LiveCG to support several operations for sequences of visualization artifacts. These features could then be integrated in both the main UI as well as the CLI. The execution of an algorithm on some input yields a number of interesting states, and for each state an image of the visualization can be generated. This sequence of images could be combined into a motion picture that would capture the dynamics of the algorithm in a single movie and which could be easily published on the web for example. Additionally, the images could be assembled to \LaTeX or HTML documents. Such documents could be enhanced by adding descriptions generated by the text output module. Combined, this should provide a nice presentation of the algorithm in a printable or browsable form.

4.4 Advanced Configuration

Most visualizations allow the user to partially modify their appearance in terms of the features that are being displayed. For example, in the Shortest Paths in Polygons visualization, it is possible to toggle whether the dual graph of the triangulation should be displayed. Control widgets for such properties are typically located in the toolbar of the visualization dialog and they can also be controlled via the CLI.

However, there are typically many additional aspects of a visualization that are in principle configurable. This concerns primarily lower level visual properties, such as colors used for drawing and the sizes of individual elements. A configuration file has been introduced that allows developers to factor out those properties to a central place. As a result, users can configure them by modifying this file, without looking into the visualization's actual source code.

The configuration file uses a custom file format that basically defines a dictionary of named properties. Each property is identified by a verbose, dot-separated name that maps to an associated value. The format is similar to that of standard Java `.properties`-files, but names of properties can be hierarchically structured to make the file easy to read and maintain.

Listing 4.1 shows an excerpt of the configuration that defines properties for the Shortest Paths in Polygons visualization. It defines colors for some of the involved objects and the widths used for drawing line segments. It defines for instance that the background of the visualization is white, while the polygon is drawn in light gray. Moreover, it defines that the polygon's outer boundary is drawn with lines with a width of 2 units, while the lines to display the diagonals have a width of only 1 unit.

To increase user-friendliness in the future, a means for manipulating the lower level properties should be integrated into the user interface. Since only a limited number of value types occur, generic components could be implemented

```
1 algorithm.polygon.shortestpath {
2   ...
3   width {
4       polygon = 2.0
5       diagonals = 1.0
6       dual_graph = 1.5
7       path = 2.5
8       substatus = 2.5
9       substatus.bg = 6.5
10  }
11  ...
12  colors {
13      background = #ffffff
14      boundary = #000000
15      diagonals = #000000
16      dualgraph = #eeee00
17      polygon = #dddddd
18      sleeve = #bbbbbb
19      sleeve.done = #3300ff00
20      ...
21  }
22  ...
23 }
```

Listing 4.1: Some properties for Shortest Paths in Polygons

that would allow for easy configuration of individual properties. All properties relevant to a single visualization could then be aggregated into a configuration dialog.

Developer Perspective

For developers, LiveCG is a framework for creating visualizations. One benefit of using our system is that they do not have to care about how the end user defines the input for their algorithm, because the framework already provides a sophisticated editor for this purpose (see Section 4.1). Instead they can immediately focus on their actual endeavor of creating the visualization itself.

To support the developer with this task, the framework provides a number of data structures, software components and utilities. They aim at saving the developer's time or at accomplishing valuable features easily. This chapter explores those facilities. It starts with a guideline for the implementation of visualizations, thereby explaining relevant components of LiveCG (see Section 5.1). We will then look at the rendering subsystem (see Section 5.2) and the text output module (see Section 5.3).

5.1 Implementing Visualizations

The *Model-view-controller (MVC)* approach is the standard pattern used to design graphical user interfaces with Java Swing. The pattern is also well suited for the implementation of algorithm visualizations, and in consequence, many components of LiveCG are geared towards this scheme. Typically, it is a bit difficult to clearly separate view and controller as they are often quite interweaved. As Eckstein et al. [ELW98] point out, Swing itself mostly uses a modified version of the pattern in which both components are implemented as a single object. Although the same difficulty applies to visualizations, there are components that are primarily concerned with the presentation, while others are primarily concerned with the manipulation of the model.

Model-View-Controller in LiveCG

Figure 5.1 gives an overview of the MVC pattern used in LiveCG. In a typical interactive situation the *model* is the algorithm logic, the *view* is the visual representation of the current algorithm state and the *controller* are some UI components that let the user manipulate the algorithm's status. On the other hand, when exporting an image file on the command-line the situation may look a bit differently. The *model* will still be the algorithm, but the *view* may

then be a component that renders the visualization to a file. Moreover, the *controller* is now a module that parses command-line options and sets the status of the algorithm to a user specified state to select the snapshot of the animation that the image will represent.

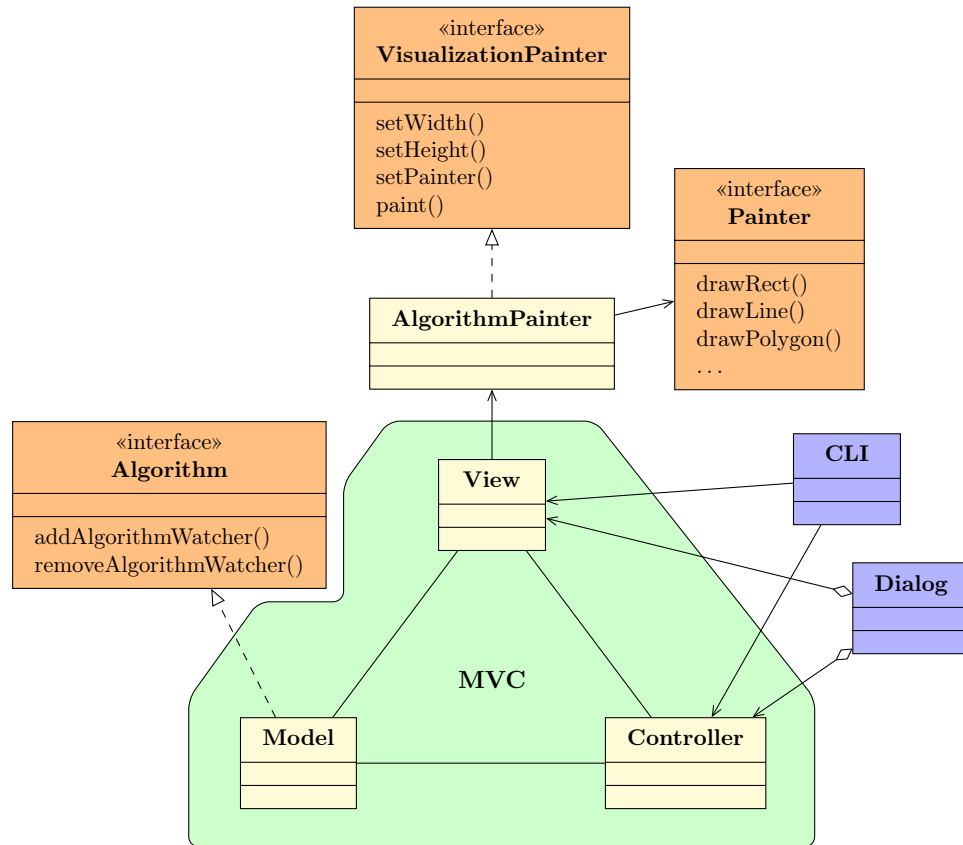


Figure 5.1: Class diagram for creating visualizations with the MVC pattern

Model

At the core of any visualization is an implementation of the algorithm or the data structure of interest. In terms of MVC, the algorithm is the *model*, and the view will be connected to the algorithm through the *Observer* pattern. To enable this functionality consistently, LiveCG provides the **Algorithm** interface, which algorithm implementations should implement. The interface declares methods that can be used for subscribing listeners on changes to the algorithm's status. Since the implementation of these methods tends to be the same for all algorithms, the class `DefaultAlgorithm`, which already implements the **Algorithm** interface, can be used as a superclass.

View

Another central component of the visualization is the *view*, which presents an algorithm's current status to the user. For geometric algorithms the most in-

teresting part is typically a graphical depiction of involved geometric objects. One design goal of LiveCG is that these graphics can not only be displayed on the screen, but can also be exported to files in different formats for postprocessing.

To achieve this, we need to encapsulate the logic for creating the graphics as a module that can be used on different output targets. Therefore, we create an `AlgorithmPainter` class which implements the actual visualization on top of an abstraction layer. In practice this means that we program the visualization against the `Painter` interface which provides the necessary drawing primitives (see Section 5.2.1). For the different use cases, we can then create lightweight views that internally utilize the `AlgorithmPainter` to create the desired output through an appropriate back end (see Section 5.2.2). The central interface that the `AlgorithmPainter` has to implement is the `VisualizationPainter`. Conforming to this interface establishes compatibility with various components of LiveCG.

A simple Swing view can be created using a `VisualizationPainter` with only a few lines of code. However, enabling advanced features such as magnification and panning, which allow the user to zoom into details of the view, requires more work than that. Since such functionality is expected from virtually every visualization panel, we provide an extendable base class for this case that already implements this functionality transparently.

Controller

For a complete integration into LiveCG, two kinds of controllers are necessary for steering an algorithm's flow of execution and the visual appearance of the visualization: a UI controller and a command-line parser.

The UI controller will be embedded into the visualization dialog and consists of ordinary Swing widgets. Ready-made components such as buttons, toggle buttons and sliders typically suffice, but sometimes it is reasonable to use more specific non-standard widgets. LiveCG does not enforce any policies on widget usage and does not provide specific reusable widgets yet. However, for consistency of appearance it is advised to reuse the icons that ship with the application, e.g. for controls that steer an algorithm's execution (⏸, ⏹, ⏺, ⏻, ⏼, ⏽, ⏾).

The command-line parser is relevant for the CLI that is capable of creating snapshots of algorithms from the terminal or via scripts. To integrate a visualization into the CLI, a factory component that implements the `VisualizationSetup` interface has to be created. This object receives the geometric input and configuration options from the CLI and essentially creates a `VisualizationPainter` instance in the user specified configuration and state of execution. For this purpose, the command-line utilities accept a `status` switch

to select the algorithm's status in an algorithm-dependent argument format. Additionally, all options that are prefixed with a capital **D** will populate a **Properties** object that will be passed to the parser as well. These properties can be used to adjust the visualization's configuration that is available through the UI in an interactive setup.

Miscellaneous

Some additional problems have to be solved for each visualization.

To make it available to users, it has to be accessible through LiveCG's main UI and the CLI. Therefore, we have to create an entry in the **Visualizations** menu that launches the respective dialog. Also, we have to add an additional case to the CLI's parsing routine that instantiates the appropriate visualization factory.

In order to make advanced configuration (see Section 4.4) possible, we have to factor out certain rendering parameters to the configuration file. Developers can simply add custom properties to this file and access them using the framework. Convenience methods are available that receive as a parameter a property name and return values such as numbers or colors.

To let the user export the content of the canvas to the various image formats, we have to provide the export entries for the **File** menu. These menu entries can be created through a helper class which creates them generically based on an arbitrary **VisualizationPainter** instance. By design, every visualization's **AlgorithmPainter** implements this interface and can thus be passed to the helper methods.

5.2 Rendering Subsystem

The graphical part of a visualization may be used for several purposes. Hence, we would like the visualizations to produce artifacts in various output formats, be it raster or vector graphics or special file formats that allow for convenient postprocessing. The framework allows creators of visualizations to abstract from the details of those formats and create their visualizations with one simple yet sufficiently powerful API. This is achieved through the definition of an abstraction layer. It consists of a drawing interface against which visualization have to be programmed (see Section 5.2.1) and for which several different back ends exist. As a result, the graphics can not only be displayed on screen, but can also be exported to different file formats. Currently supported file types are PNG, SVG, TikZ and Ipe (see Section 5.2.2).

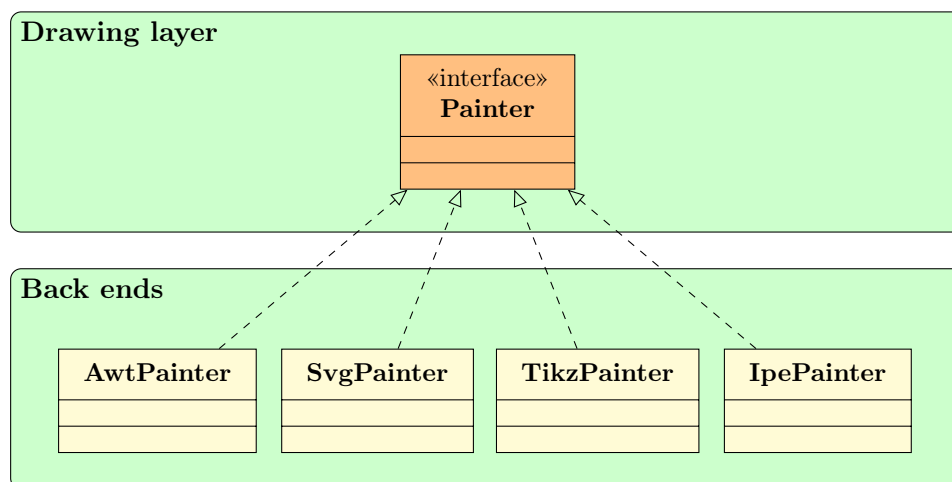


Figure 5.2: Components of the rendering subsystem

5.2.1 Drawing Layer

The drawing layer¹ provides a means for developers to realize the graphical output of their visualization. It comes as an interface called **Painter** which exposes a set of methods that are very similar to those provided by many graphics frameworks such as the *Java 2D Graphics API*² or the *Cairo*³ graphics library. Although it is quite extensive, it does not offer all methods that are typically supported by those frameworks. Instead, it has been kept deliberately simpler to keep the effort for the realization of several back ends manageable.

Two paradigms of drawing guide the concept of most operations: drawing paths and filling shapes. Hence, each operation usually comes in two similar forms — one for drawing and another for filling. For example, there is a method **drawRect** that draws a rectangle and a method **fillRect** that fills it. There are similar methods for other primitive geometric objects such as circles or line segments. The visual result of every operation depends on the style that has been configured in advance. For both types of operations, the color can be selected, and for drawing, the width and other style attributes of the stroke can be defined through appropriate methods.

Complex Objects

In addition to the methods for drawing primitive geometric objects, there are also methods for drawing more complex shapes. On the one hand, specialized methods have been defined for the data structures from LiveCG’s data model: there are methods for drawing polygonal chains and for drawing or filling polygons. These methods have been added to the interface for convenience, because programmers of visualizations will often be working with such

¹see Appendix A.1

²<http://docs.oracle.com/javase/tutorial/2d/>

³<http://cairographics.org/>

objects and thus they should be easily drawable. In general it is advisable to use the Java Collections Framework⁴ when working with Java. To facilitate working with those classes, there is also a method for drawing a path defined by an ordinary list of coordinates.

On the other hand the **Shape** class of the AWT package included in the standard JVM provides powerful methods for creating complex shapes for drawing purposes. For instance, **Shape** implementations for various primitive objects can be created, Boolean set operations can be applied to them and they can be modified using affine transformations. Drawing them is supported directly as programmers that are used to working with Java's standard graphics library should find this convenient. Also, porting existing AWT-based visualizations to the framework is hereby simplified in case they rely on **Shape** classes.

Stroke styles

When drawing objects, the appearance of the created strokes can be influenced. First of all, the width of the stroke can be configured. Moreover, drawing operations can be carried out using different stroke styles. Our drawing layer currently supports continuous and dashed strokes. By default, strokes are continuous. Dashed strokes are defined by a dash pattern which consists of a list of numeric values. Each value defines, in alternating order, a solid or a gap section of the stroke. Strokes are then drawn by applying this pattern to the requested path repeatedly. Additionally, a phase value has to be supplied which defines a start pointer into that pattern and which can be used to shift the pattern along the drawn path.

Drawing frameworks usually provide a means for influencing the stroke's cap-style and join-style. The cap-style defines the shape of the pen tip and the join-style controls the way adjacent line segments will be joined. Our framework currently does not offer control over those options. Instead, it instructs the back ends to use both a round cap-style and a round join-style.

Text

Rudimentary support for creating text has been implemented. However, no support for configuring the text style is available, because this would not be well mappable to some of the supported back ends. For example, setting font-family and font-size would not be applicable to formats like TikZ and Ipe.

Raster Images

Raster images can be drawn onto the canvas as well. On the one hand, we can illustrate drawings using bitmap icons, and on the other hand, we can create

⁴<http://docs.oracle.com/javase/tutorial/collections/>

and include diagrams that cannot be well represented as vector graphics. For example, a heat map would best be created as a raster image and then drawn onto the canvas as such.

Clipping and Transformations

All operations mentioned above are subject to both clipping and transformations that have been applied to the canvas before using them. Clipping can be used to restrict the area to which other drawing operations apply. For example, we can apply a rectangular clip. Now, for each shape that is filled, only the intersection of the shape with the clipping rectangle becomes visible in the drawing. Furthermore, arbitrary shapes can be used for clipping, and subsequent calls to the clipping methods will further restrict drawing operations to the intersection of all supplied shapes.

Transformations interfere with the other operations in a similar way. Every object that is drawn using one of the drawing methods is first transformed using the current transformation. Using transformations allows us to easily reuse already implemented drawing code. For example, we can scale and move the output of a complex drawing operation to a different region on the canvas without modifying the operation itself or calling it with different parameters.

5.2.2 Drawing Back Ends

Different back ends exist for the rendering subsystem. They delegate calls to the drawing layer to appropriate operations on their respective output targets.

AWT Back End

The AWT back end utilises the Java 2D Graphics API. It can be used for implementing UI components and for generating raster images.

Custom Swing UI components such as panels for displaying a visualization's graphics are generally implemented by extending a basic class such as `JPanel` and overwriting its `paint()` method. This method receives as a parameter a `Graphics2D` object that is used to apply drawing operations to the canvas. Our AWT back end accepts a `Graphics2D` instance as a parameter as well, and delegates all drawing operations to that object.

Raster images can be created in Java by constructing `BufferedImage` instances. They represent in-memory images and offer means for obtaining a `Graphics2D` instance that operates on the underlying image. Our back end can be instantiated using this object to render the visualization to the image. Using the `ImageIO` API, they can be exported to various image file formats, such as BMP, JPEG, PNG and GIF. By default, the framework stores images in the PNG format, because this is a widespread and open file format with lossless compression.

SVG Back End

Scalable Vector Graphics (SVG) is an open, XML-based vector image file format. Unlike raster images, vector graphics can be scaled arbitrarily without causing pixelation effects. The SVG back end has been implemented using Batik⁵, a framework for creating SVG graphics with Java. Using Batik's abstraction layer for the XML file's *Document Object Model (DOM)*, the required SVG elements can be created using Java's standard DOM interface⁶.

PGF/TikZ Back End

The *Portable Graphics Format (PGF)*⁷ is a drawing framework for T_EX/L^AT_EX. It is a low-level graphics library that can be used in T_EX documents for embedding graphics. It is portable in the sense that it can be used in conjunction with the different publishing file formats for T_EX documents, such as DVI, PS and PDF. To achieve this, it operates in turn with several back ends. TikZ is a front end for PGF that exposes a higher level interface of T_EX-macros that can be used for drawing images. Our back end creates simple text files that consist of such macros that can then be included into normal T_EX documents within a `tikzimage` environment.

Ipe Back End

Ipe⁸ is a vector drawing program developed and maintained by Otfried Cheong [Sch95]. It specializes in the creation of drawings for publications in computational geometry. The graphics are intended to be used in the L^AT_EX environment and in fact the resulting PDF files that can be included into T_EX documents are in turn produced using L^AT_EX.

Ipe uses an XML file format to store its drawings. Our back end produces compatible XML files using Java's standard DOM interface. These files can then be converted to PDF using command-line utilities that ship with the Ipe distribution. Alternatively, they can be opened and postprocessed using the Ipe editor and stored as PDF from within the program.

Implementation Notes

Mapping basic drawing operations to any of the underlying frameworks is a straight-forward process, because the respective APIs are usually quite similar. What tends to be more complicated is the implementation of clipping and transformation operations and the support for raster images.

⁵<http://xmlgraphics.apache.org/batik/>

⁶<https://jaxp.java.net/>

⁷<http://www.ctan.org/pkg/pgf>

⁸<http://ipe7.sourceforge.net/manual/manual.html>

Most formats support transformations natively, but with Ipe for example, this functionality could not be found in the documentation. To support transformations anyway, this deficiency has been compensated by actually applying the transformations to the shapes before inserting them into the document. In contrast, with SVG for example, transformations can be specified in the markup, and they will not be applied to the shapes until a renderer displays the file.

Clipping is natively supported by all back ends. However, with TikZ, errors occurred regularly when postprocessing files that make extensive use of this technique. It turned out that the L^AT_EX rendering engines have problems handling shapes that extend far beyond the image boundaries, even when they are clipped to some area within that boundaries using the native clipping operations. To avoid such problems, all shapes are additionally clipped to a safety rectangle which is slightly larger than the image itself. To actually apply the clipping to the shapes, a clipping algorithm for arbitrary shapes had to be found⁹. As a consequence, transformations had to be explicitly applied to all shapes as with the Ipe back end. Otherwise, it would not have been possible to clip them to a rectangle that is defined in untransformed coordinate space.

Except for TikZ, all back ends are able to embed raster images into their respective file format directly using some special encoding technique. For example, with SVG raster images can be included into the XML markup as Base64-encoded PNG files. With TikZ however, this is not possible, and raster images can only be stored in separate files that are then included into the drawing by referencing that file within the file system. Although this works, this means that a TikZ file that uses raster images will be accompanied by a directory containing the referenced image files. Hence, users have to take care when handling such files to keep the references intact.

5.3 Text Output Module

The text output module enables visualization implementations to explain their graphics using text. It provides generic tools that can be used to display explanations to the user and it defines interfaces that algorithms have to implement so that those tools can be used with them. When an algorithm supports explainability, the *Description dialog* presented in Section 4.2 can be used to display explanation corresponding to the current state of the algorithm's execution.

To make a visualization compatible with this module, the algorithm has to implement the `Explainable` interface. It defines a method that returns the explanation for the current step of the algorithm as a list of strings. The

⁹<http://javagraphics.blogspot.de/2007/04/shapes-clipping-to-rectangle.html>, accessed 3/16/2014

Description dialog will display each of those strings in a separate line of its text area.

The dialog requests explanations from the algorithm on demand and keeps already displayed explanations visible. Depending on the UI, the user may navigate through the algorithm's execution in unpredictable order, skipping individual steps for example, or rewinding to previously skipped steps. However, the dialog inserts new explanation snippets at the position within the stream of explanations at which they would naturally occur when explaining the complete algorithm. To maintain the correct order of all snippets, the algorithm has to implement another interface, which provides a means for determining an algorithm's current status in form of a string. This string has to be unique for each of the algorithm's steps and will be used by the Description dialog to establish the order between individual explanations. Thus, these status strings have to be mutually comparable and their lexicographic order has to reflect the order of the steps of the algorithm.

Implemented Visualizations

A number of visualizations have been developed for data structures and algorithms. Some of them are rather static and merely provide a drawing of a data structure, the result of an algorithm or an important concept that it uses. Most algorithm visualizations are interactive animations. They provide an interface for stepping through the stages of the algorithm and display drawings of individual snapshots. Each of the implemented visualizations will be described in the following sections in some detail. The illustrations have been produced with the respective visualization using the Ipe back end.

6.1 Doubly-Connected Edge List (DCEL)

The visualization for the DCEL [MP78] is a static visualization that focuses on the depiction of halfedges. Figure 6.1 shows an example of the graphics produced by the visualization (b) for a small subdivision (a). As a basic layer,

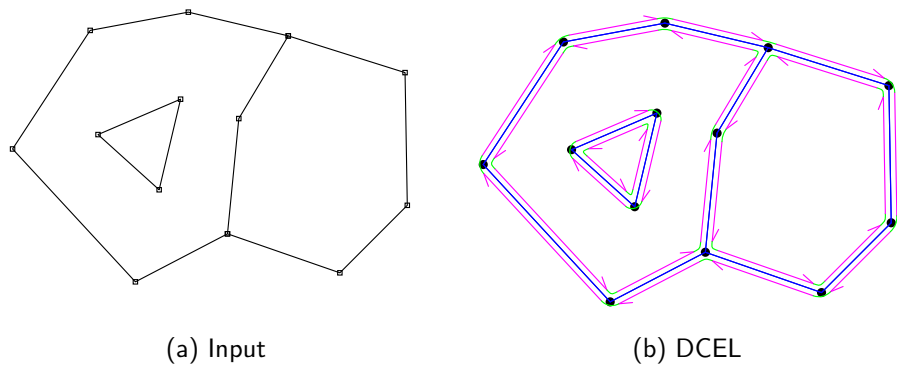


Figure 6.1: DCEL visualization

vertices and edges of the subdivision are displayed in the drawing as black disks and blue line segments. On top of that, the halfedges (magenta) and their *next* and *previous* pointers (green) are being displayed. The halfedges are depicted as arrows parallel to their corresponding edge. They are slightly shifted into the direction where their incident face is so that individual halfedges are clearly visible. The arrowhead is reduced to a single line to prevent visual overload: because the second line of the arrowhead is missing, the padding between edge and halfedge can be reduced so that it becomes less likely for arrows

to overlap. Each arrow is a little shorter than its corresponding edge, which avoids overlapping of arrows as well and gives some space for displaying the pointers to other halfedges. Two halfedges that are connected to one another with their respective next and previous pointers are visually connected with a curve that connects the tip of one arrow to the tail of the next.

This visualization can be used for inspecting the structure of the DCEL for an arrangement itself, but can in particular be used as a building block for more advanced visualizations that involve a DCEL data structure.

6.2 Fortune's Sweep Line Algorithm for Computing Voronoi Diagrams

Fortune's Sweep [For87] is a sweep line algorithm that computes the *Voronoi diagram* for a set of input points in $O(n \cdot \log n)$ time where n is the number of points. While the original algorithm by Fortune is actually more complicated because it involves geometric transformations, the simpler variant presented here has been given by Seidel [Sei88] and in more detail by Guibas and Stolfi [GS88].

The Voronoi diagram is a subdivision of the plane into polygonal regions such that exactly one region exists for each input point, which are called sites in this context. The *Voronoi region* corresponding to a site p contains all points on the plane that are nearer to p than to any other site. Figure 6.2 shows input points (a) and the resulting Voronoi diagram (b) for a set of eight sites. The blue lines are the *Voronoi edges* which describe the boundary of the Voronoi regions. Endpoints of Voronoi edges are called *Voronoi vertices*.

A typical data structure for representing the Voronoi regions is the DCEL. The underlying implementation of the algorithm constructs the DCEL and the user of the visualization is able to switch between a simple display (b) and a display that also shows the DCEL (c).

The dual problem to computing the Voronoi diagram is the computation of the *Delaunay triangulation*. It is a triangulation of the input point set that maximizes the minimum angle of the constituting triangles. Both problems are tightly connected and thus it makes sense to integrate the Delaunay triangulation into this visualization. When the user enables the respective option, the triangulation will be shown with gray line segments as in Figure 6.2d. Two points will be connected in the triangulation if and only if the respective Voronoi regions are neighbors, i.e. they share a common Voronoi edge.

The visualization for Fortune's Sweep is an interactive animation. As the algorithm uses the plane sweep paradigm, the dimension of animation is the position of the sweep line. In this implementation it is a vertical line that moves from left to right. Different methods for navigating through the algorithm have

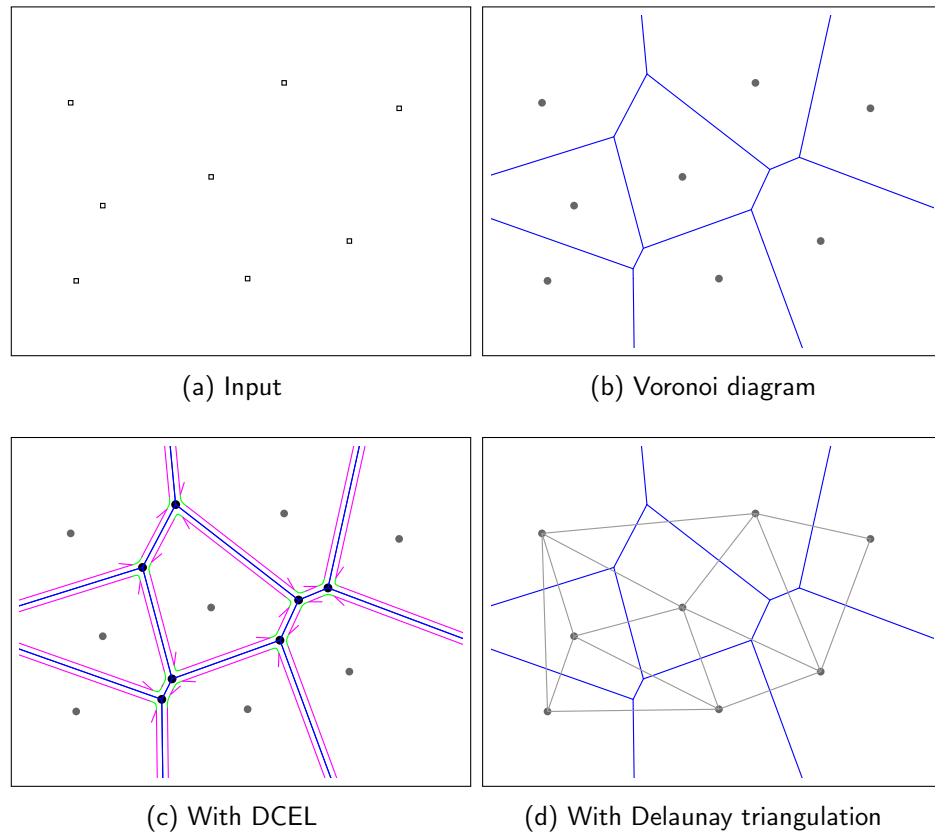


Figure 6.2: Fortune's Sweep visualization

been implemented: a widget similar to a slider can be used to adjust the sweep line position by dragging a handle with the mouse. Buttons can be used to move it to the next or the previous event point or to the next or the previous pixel on the screen. A playback mode is available that plays the animation like a motion picture in either forward or backward direction. Figure 6.3 showcases the graphics that the visualization produces for different positions of the sweep line, which is displayed as a red vertical line that extends from the top to the bottom. At the initial position (a) only the sweep line and the input points are visible. In the other pictures, we can see a partially computed Voronoi diagram with its DCEL and the so-called *wavefront* which consists of parabolic arcs and extends from the top to the bottom of the diagram. The area left to the wavefront is the area for which we have computed the Voronoi diagram up to now.

A *site event* (b) occurs when the sweep line reaches one of the input points. When this happens, a new parabolic arc will be inserted into the wavefront and a new edge of the Voronoi diagram emerges. This happens at the position where a horizontal ray shot from the site to the left hits the wavefront. While the sweep line moves forward, the wavefront's shape transforms, capturing more and more of the plane. Although an efficient implementation of Fortune's Sweep would only consider the event points itself, the visualization can display

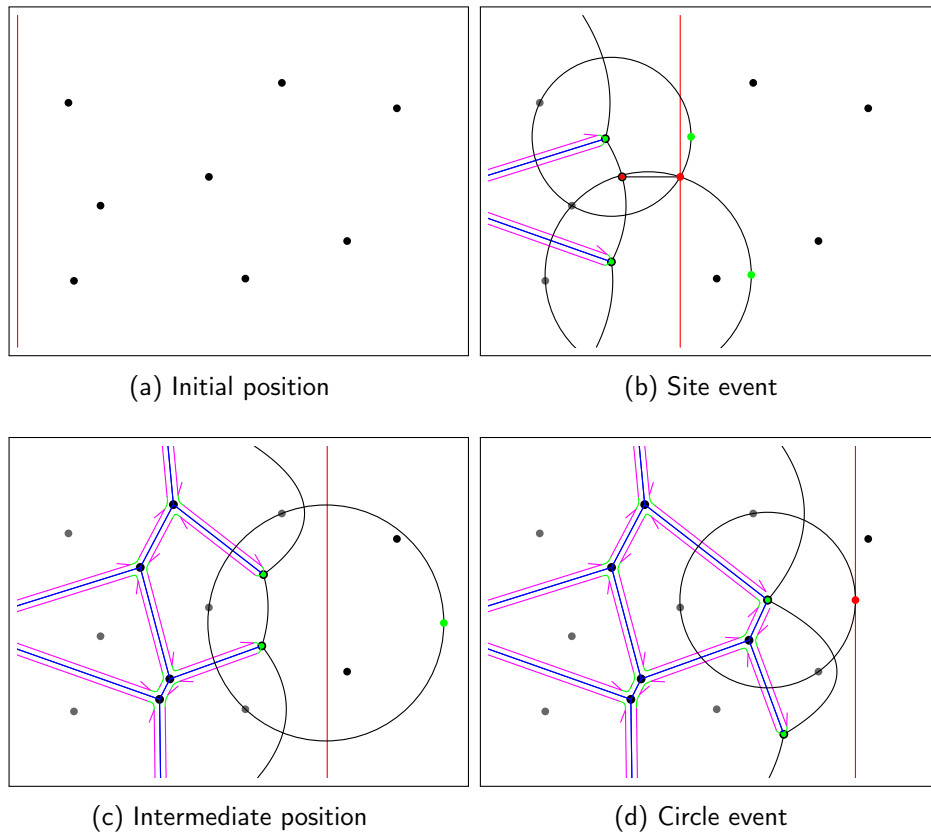


Figure 6.3: Fortune's Sweep: various positions of the sweep line

intermediate positions (c) to illustrate the transition between consecutive event points. During the transformation of the wavefront, an individual parabolic arc may disappear from it because it gets covered by its neighbors completely. Such events are called *circle events* (d). Whenever this happens, two incomplete Voronoi edges will be joined, thereby creating a Voronoi vertex and a new incomplete Voronoi edge.

The visualization has been developed based on an implementation¹ by Allan Odgaard and Benny Kjær Nielsen that has been created at the University of Copenhagen in 2000 and is available through Github². The original implementation was designed as a Java Applet using the AWT framework and has been ported to LiveCG.

6.3 Partitioning Polygons into Monotone Pieces

This visualization is about a plane sweep algorithm by Garey et al. [Gar+78] that partitions a simple polygon without holes into y -monotone pieces. It takes $O(n \cdot \log n)$ time where n is the number of vertices of the polygon. In combination with an algorithm for triangulating monotone polygons it can be

¹<http://www.diku.dk/hjemmesider/studerende/duff/Fortune>, accessed 3/16/2014

²<https://github.com/sorbits/visual-fortune-algorithm>

used to give an algorithm for triangulating arbitrary non-monotone polygons [Gar+78]. Our implementation, the visualization, and this description are guided by the presentation of the algorithm by de Berg et al. [Ber+00].

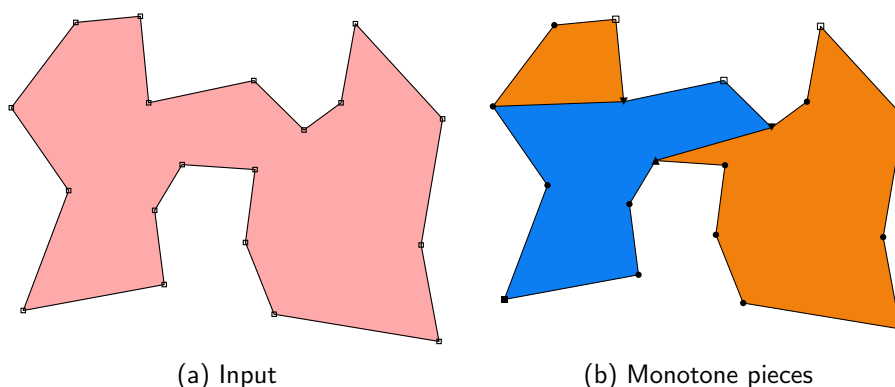


Figure 6.4: Partitioning a polygon into monotone pieces

Figure 6.4 shows a simple polygon (a) and the resulting monotone pieces (b). The algorithm classifies all vertices of the input polygon to be one of five types: start (\square), end (\blacksquare), split (\blacktriangle), merge (\blacktriangledown) or regular (\bullet). The plane sweep method is then used to partition the polygon into pieces that do not contain any split or merge vertices anymore. Such pieces are y -monotone. The event points of the sweep algorithm are the vertices of the polygon. Whenever a split or merge vertex is encountered, a diagonal will be inserted by connecting it to another vertex of the polygon. During the algorithm's execution we make sure that we always know which vertex we can connect such a diagonal to. The implemented visualization is static: it only produces an illustration of the resulting partition of the polygon. We can identify the inserted diagonals and easily distinguish the different subpolygons as adjacent pieces will be filled with visually contrastive colors. A future extension that animates the algorithm's steps would be desirable.

6.4 Triangulating Monotone Polygons

This algorithm triangulates a monotone polygon in linear time. It has been presented by Garey et al. [Gar+78] to give, in combination with the one from the previous section, an algorithm for triangulating arbitrary simple polygons in $O(n \cdot \log n)$ time.

Although the algorithm could be extended to support any monotone polygon, we assume here that the polygon is y -monotone. We use the sweep paradigm to add diagonals until we obtain a triangulation. The event points are the vertices of the polygon, ordered from top to bottom by their y -coordinate. Building a sorted list of those coordinates in linear time is possible because the polygon is monotone: we basically have to merge the two mono-

tone chains of the polygon which are already sorted with respect to y due to their monotonicity.

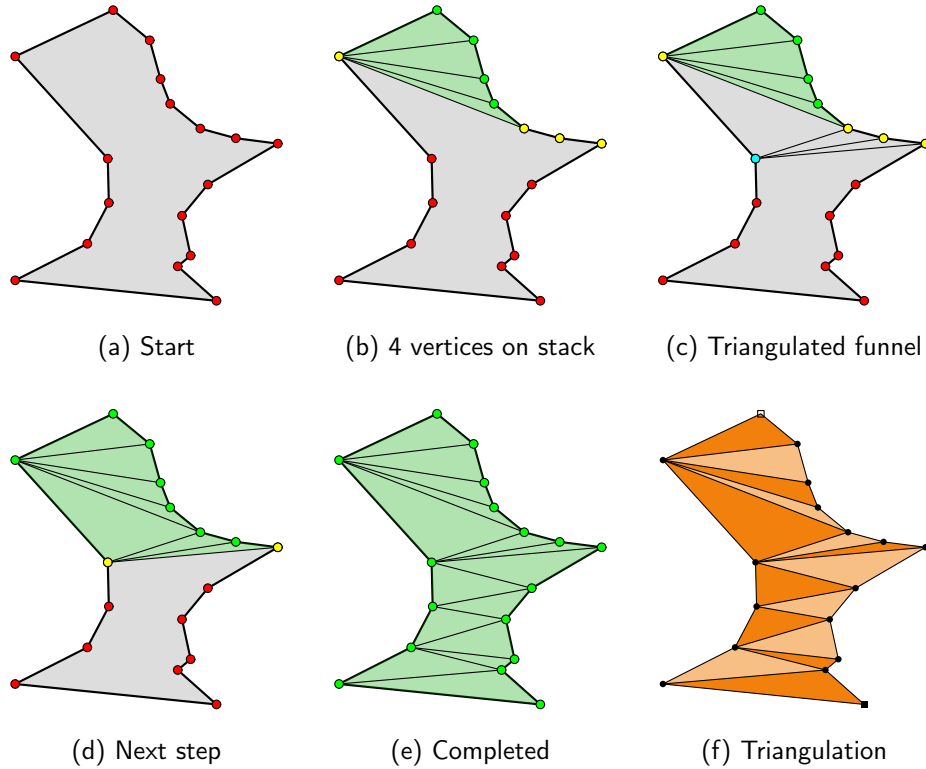


Figure 6.5: Triangulating a monotone polygon

Figure 6.5 shows the kind of graphics that the visualization produces. During the algorithm, we maintain a stack containing at least two previously encountered vertices of the polygon that define a funnel-shaped subregion of the polygon. Everything above that funnel is readily triangulated (green area). Vertices are colored depending on their status. Either they have not been considered yet (red), are currently on the stack (yellow) or will not be needed anymore (green). The current vertex is shown in cyan.

At each event point, we remove some of the vertices from the stack and add diagonals by connecting each of them to the current vertex. When the current vertex and the one at the top of the stack are on opposite chains, the complete funnel can be triangulated (see (b) \rightarrow (c) \rightarrow (d)). When they are on the same chain, it may happen that only part of the funnel can be triangulated in that step. After adding all possible diagonals, we push the two vertices of the lastly added diagonal onto the stack and continue with the next vertex.

The resulting triangulation (e) can also be displayed with more contrast (f), emphasizing the individual triangles of the triangulation. Adjacent triangles are then filled with different colors and since the dual graph of the triangulation is a tree, only two different colors are needed for a valid coloring. Two different shades of the same color are used here. This has the effect that the same

coloring can be used when combining this visualization with the one for partitioning a polygon into monotone pieces (see Figure 6.6). Adjacent monotone

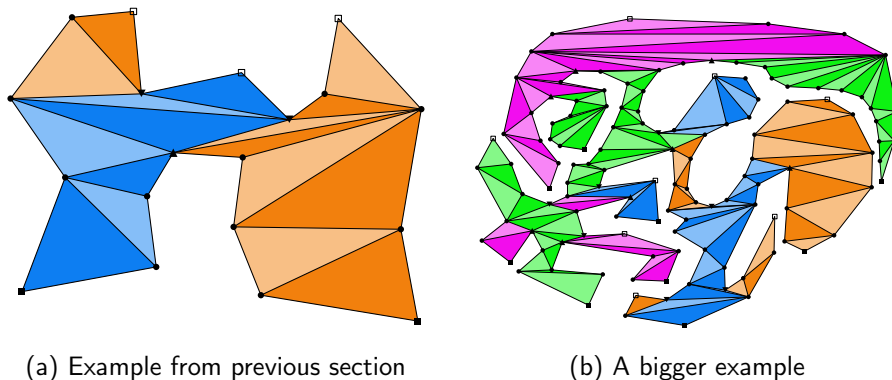


Figure 6.6: Triangulated monotone pieces

pieces are colored with contrasting colors while the triangles of the individual triangulations use shades of the same color. Hence, the monotone pieces can be spotted easily along with their triangulation: a single picture shows us the result of both algorithms applied for an arbitrary simple polygon.

6.5 Shortest Paths in Polygons

This visualization is of an algorithm for finding the shortest path between two points, s and t , within the interior of a polygon. The path must lie within the polygon, i.e. it is not allowed to cross its boundary. Lee and Preparata [LP84] have given an algorithm for computing this shortest path in $O(n \cdot \log n)$ time. When the triangulation of the polygon is given, the algorithm takes only $O(n)$ time.

The polygon is assumed to be simple and does not contain any holes. We triangulate the polygon and locate the triangles Δ_s and Δ_t containing s and t respectively. Next, we look at the dual graph of the triangulation and determine a path within that graph from Δ_s to Δ_t . Since the polygon does not contain any holes, the dual graph is a tree. Hence, the path is unique and can be found in linear time. The corresponding triangles of this path constitute the triangulation of another polygon called the *sleeve*. Figure 6.7 shows a polygon (a) and the corresponding triangulation (b): s and t are marked with a red and a green circle respectively. The dark gray triangles are part of the sleeve while the light gray triangles belong only to the original polygon's triangulation.

An important property of the sleeve is that the dual graph of its triangulation is a chain in which Δ_s is the first node and Δ_t is the last. The shortest path from s to t starts within Δ_s , goes through all triangles of the sleeve and ends in Δ_t . On its way, the path has to cross all diagonals of the sleeve's

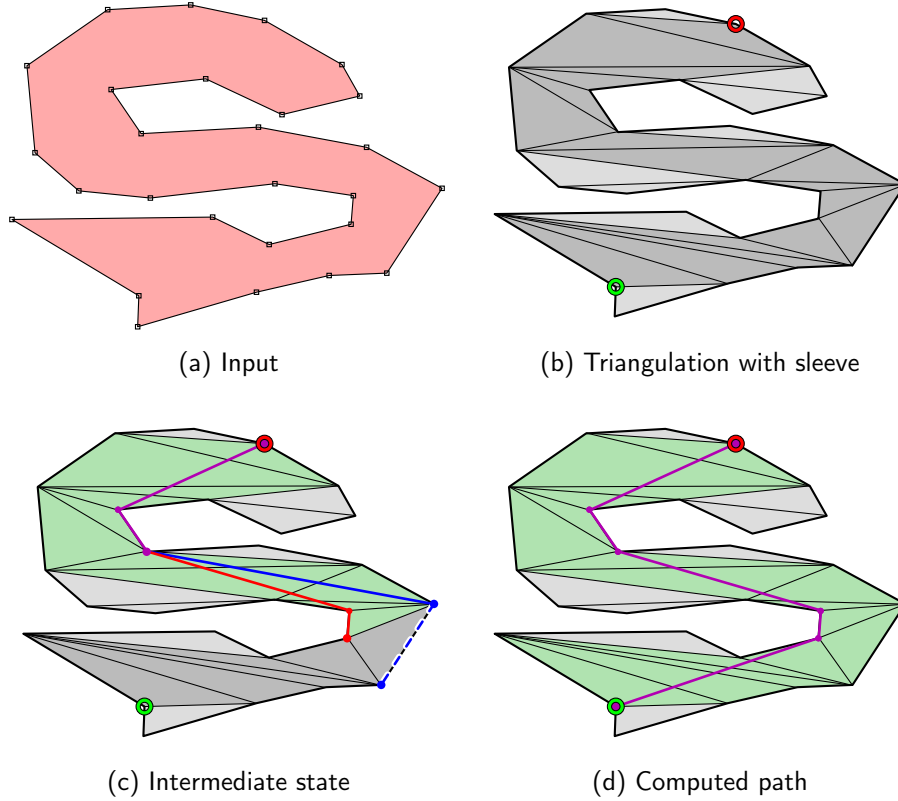


Figure 6.7: Shortest path in a simple polygon

triangulation. The algorithm makes use of this observation and considers the diagonals one after the other, thereby computing the shortest paths from s to both of each diagonal's endpoints. Thus, the main loop of the algorithm is over the diagonals of the sleeve. Each diagonal splits the sleeve into two parts, one of which contains all diagonals that have been considered already and the other containing all remaining diagonals. In the visualization, the former part is depicted in light green to emphasize the part of the sleeve that we are already done with: in (c) we see the visualization in an intermediate state and (d) shows the algorithm when it is completed.

The shortest paths from s to a diagonal's right and left endpoints are displayed in red and blue respectively. Both paths are identical from s through to a vertex called the *apex*. This common part is displayed in purple. The diverging parts of the paths have the shape of a funnel. In each iteration, the algorithm determines the shortest path to one of the next diagonal's endpoints by maintaining this special shape. To achieve this, the algorithm traverses the vertices of the funnel and at each position, examines a candidate segment. Such segments are displayed as dashed lines so that the user can observe the construction of the funnel based on that of the previous iteration.

6.6 Chan's Algorithm for Computing the Convex Hull

Chan [Cha96] has developed an optimal, output-sensitive algorithm for computing the convex hull of a set of points on the plane. It runs in $O(n \cdot \log h)$ time, where h is the number of vertices of the convex hull. It improves over previous algorithms with the same time bounds by being relatively simple, because it does not use complex data structures or frameworks.

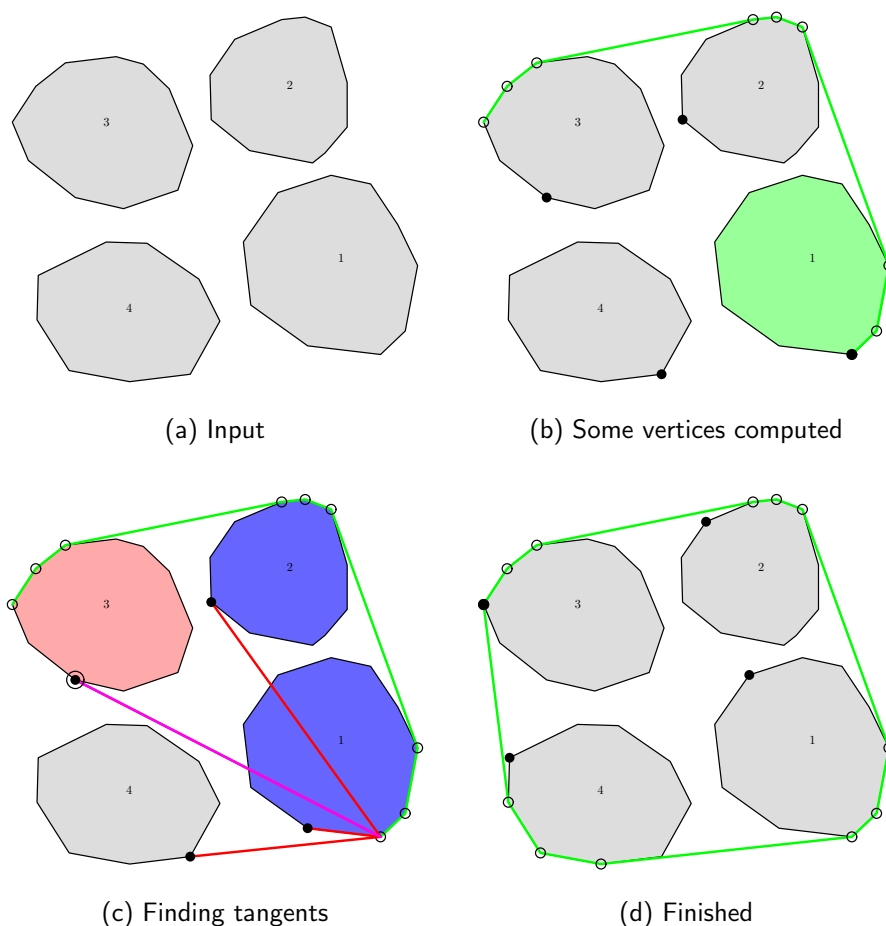


Figure 6.8: Chan's algorithm performing gift-wrapping with four small convex hulls

It is inspired by Jarvis's march [Jar73] and uses the gift-wrapping technique [CK70; Sha78]. The algorithm pre-processes the input by dividing the points into groups of size m for which the convex hull will be precomputed with any algorithm with running time $O(n \cdot \log n)$. It then uses gift-wrapping to find the global convex hull: we start with the leftmost of all points and find one vertex of the hull after the other. In each step, we consider the tangents through the last vertex of the convex hull that has been found to each of the precomputed polygons. Of those tangents, we select the one that maximizes the interior angle of the polygon corresponding to the emerging convex hull.

The problem is that m is not known in beforehand, so the algorithm tries different values subsequently. When it finds that m is too small, it stops

the computation and continues with the next larger value. Subsequent values of m are chosen to increase super-exponentially to obtain the desired overall asymptotic running time of the algorithm.

The visualization (see Figure 6.8) focuses on the gift-wrapping step that the algorithm performs on a set of convex polygons. The input is thus not a set of points, but actually a set of convex polygons instead (a). The partial convex hull is drawn with green lines and its vertices are depicted with black circles (b). In each wrapping step, the algorithm determines the tangents to the polygons through the current hull vertex (c). In the implemented variant, it does so by walking around each polygon in clockwise order, starting at the tangent vertex of the previous iteration, until the tangent vertex is found. Tangents are displayed in red; tangent nodes are depicted with black disks; and the polygons for which the tangents have already been found are blue. The tangent that we are currently looking for is magenta; the tangent node that is currently being moved is highlighted with an additional, larger black circle; and the respective polygon is light red. Once all tangents have been found, the algorithm can determine the next vertex of the convex hull. In that step, the small polygon that this vertex belongs to is depicted in light green to emphasize this decision (b).

6.7 Fréchet Distance

The Fréchet distance is a metric for the resemblance of curves. The interesting property of this metric is that it takes into account the shape of the curves. This distinguishes it from other measures such as the Hausdorff distance where the curves are considered as sets of points and their shape is disregarded.

Algorithms for computing the Fréchet distance for two given polygonal curves make use of different types of diagrams. For two such diagrams visualizations have been implemented.

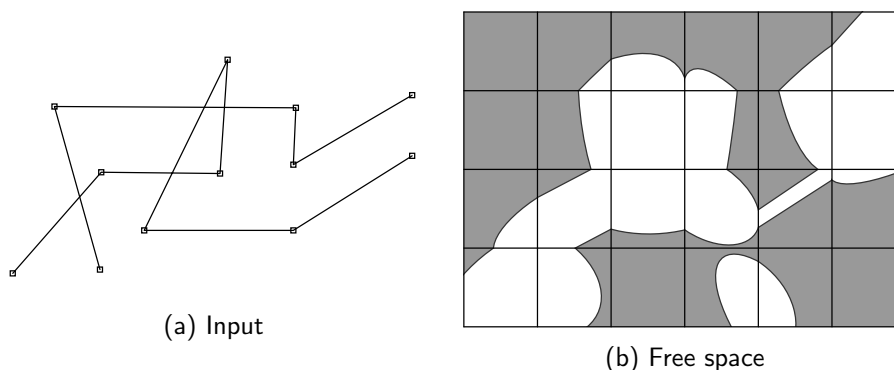


Figure 6.9: The free space diagram

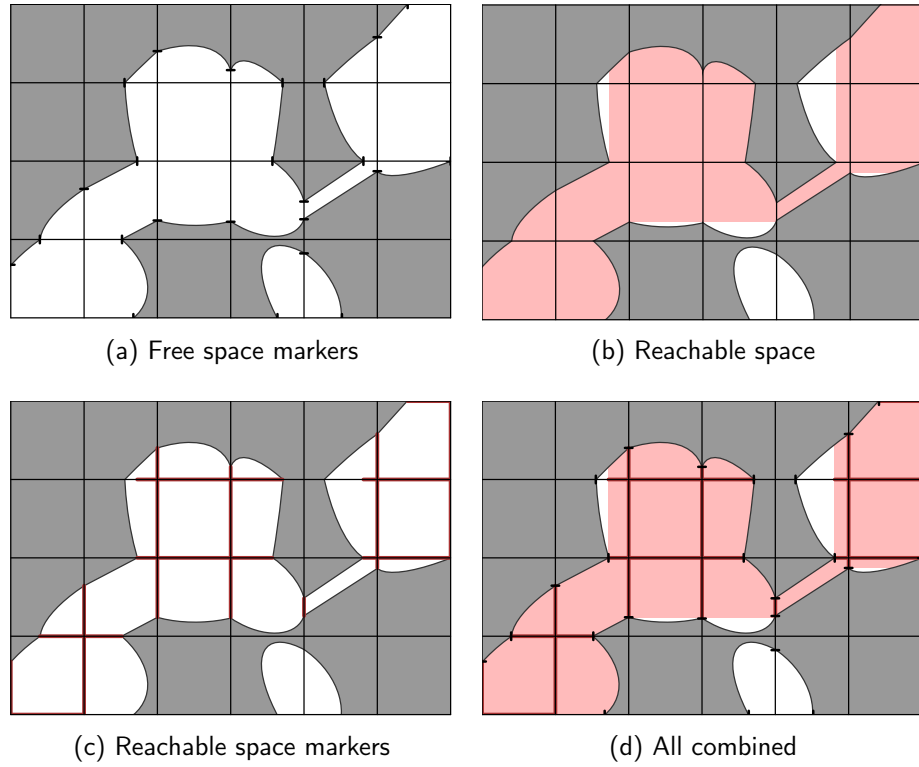


Figure 6.10: Free space visualization components

6.7.1 Free Space Diagram

Alt and Godau [AG95] have given the initial algorithm for computing the Fréchet distance for polygonal chains. Their algorithm uses the *free space* to solve the decision problem for a given distance ϵ . By solving the decision problem for certain critical values, the actual Fréchet distance can be computed. To visualize the free space, the *free space diagram* can be used. Figure 6.9 shows an example of such a diagram: for a fixed ϵ it displays a white area that represents the parameter subspace of the curves where the distance between the curves is $\leq \epsilon$. Each point in the diagram represents a position on both of the input curves. When the diagram is white at such a point, it means that the distance between the two corresponding points at the respective positions on the curves is $\leq \epsilon$.

The diagram consists of cells that corresponds to pairs of segments from both input curves. In this example, one chain has six segments, the other four so the diagram has 6×4 cells. Within each cell the free space is the intersection of the cell rectangle with an ellipse whose equation depends on the corresponding line segments. The intersection of the free space with the four sides of a cell are (possibly empty) intervals. Those intervals are important for the algorithm and thus it is possible to show their endpoints in the diagram as in Figure 6.10a.

For solving the decision problem, we compute the *reachable space*, a subset

of the free space. Starting at the bottom left corner of the diagram, working our way through it until we reach the top right corner, we compute the reachable space using dynamic programming. The user can toggle the reachable space visibility resulting in a picture like Figure 6.10b. The real algorithm only deals with the intersections of the reachable space with the cell boundaries rather than the reachable space as an area. As with the free space, those intersections are line segments, which can also be shown in the diagram as is Figure 6.10c. Enabling all components simultaneously will result in an image like Figure 6.10d.

The user interface provides a slider for configuring ϵ so that one can inspect the diagram for different parameters. This makes it easy to observe how free space and reachable space evolve with a changing value for ϵ .

6.7.2 Distance Terrain

Buchin et al. [Buc+13] have developed another algorithm for computing the Fréchet distance. It is different from the previous algorithm because it does not repeatedly solve a decision problem for a finite set of critical values. Instead, it computes the Fréchet distance directly by considering the *distance terrain*. The distance terrain (see Figure 6.11) is a generalization of the free space diagram.

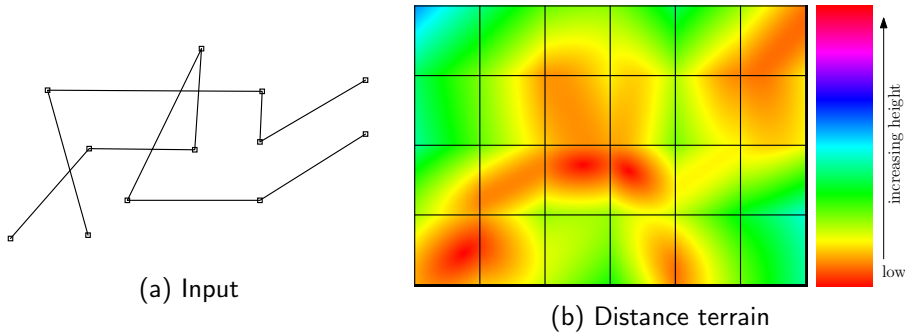


Figure 6.11: The distance terrain

As with the free space, each point on the terrain corresponds to positions on the input chains. However, instead of encoding binary information, each point represents the distance between the corresponding points on the curves. When interpreted as a height, this gives a 3-dimensional terrain. The visualization shows a 2-dimensional view of that terrain where the color encodes its elevation.

Their algorithm examines bimonotone paths from the bottom left to the top right corner of the diagram. Across the possible paths to the boundaries of the cells, they optimize a criterion of that path: its maximum height through the terrain should be as low as possible.

The visualization of the distance terrain is static. To achieve a clear depiction of the elevation of the terrain it has been mapped to hue values on the color wheel. An arbitrary distance interval maps to the complete color wheel

resulting in periodic ambiguity of colors. Which value is useful for this mapping depends on the input and influences the overall appearance of the graphic. Figure 6.12 shows the same distance terrain for three different mapping values. To give the user control over this value, the user interface provides a slider that adjusts it.

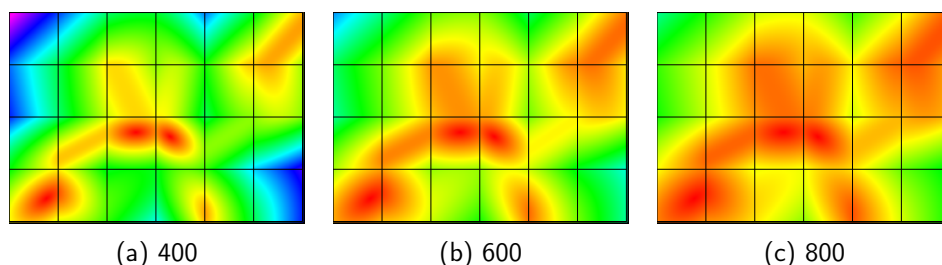


Figure 6.12: The distance terrain with different scale values

6.8 Buffer Regions

A common geometric operation in *Geographic Information Systems (GIS)* is the computation of so-called buffer regions. Buffers are typically defined as the Minkowski sum of some geometric features such as lines or polygons and a disk centered at the origin [Kre06]. Their computation involves the computation of *offset curves* [Hof89] which are derived from input curves by shifting them by a given distance d . Each point p' on an offset curve is derived from a point p on the input curve by translating it by a vector with length d whose direction is orthogonal to the direction of the input curve at point p .

JTS implements buffering that can be applied to points, lines and polygons for a specified buffer distance d . The resulting curved shapes are approximated with straight line segments with configurable accuracy. The JTS implementation applies the concept of end-caps and join-styles to buffers, which are otherwise known from computer graphics in the context of drawing paths with different kinds of stroke styles (see Figures 6.13 and 6.14 for the available end-cap and join styles). The shape of the buffer then looks as if the outline of

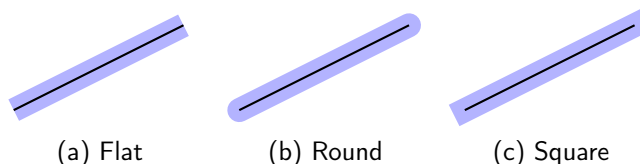


Figure 6.13: Cap styles

the input geometries had been drawn with the respective styles. Figure 6.15 shows some example buffers for the input (a). Using a round end-cap and a round join style (b) yields the classical buffer corresponding to the Minkowski sum with a disk, but the styles can be combined arbitrarily (c).

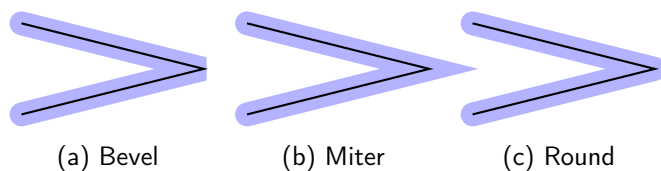


Figure 6.14: Join styles

Usually, the buffer distance is positive, but negative values can also be supplied (d). Points and polygonal chains do not contribute to the output in that case and polygons are shrunk by the respective distance.

Our visualization lets the user select end-cap and join styles through combo boxes and the buffer distance can be adjusted using a slider.

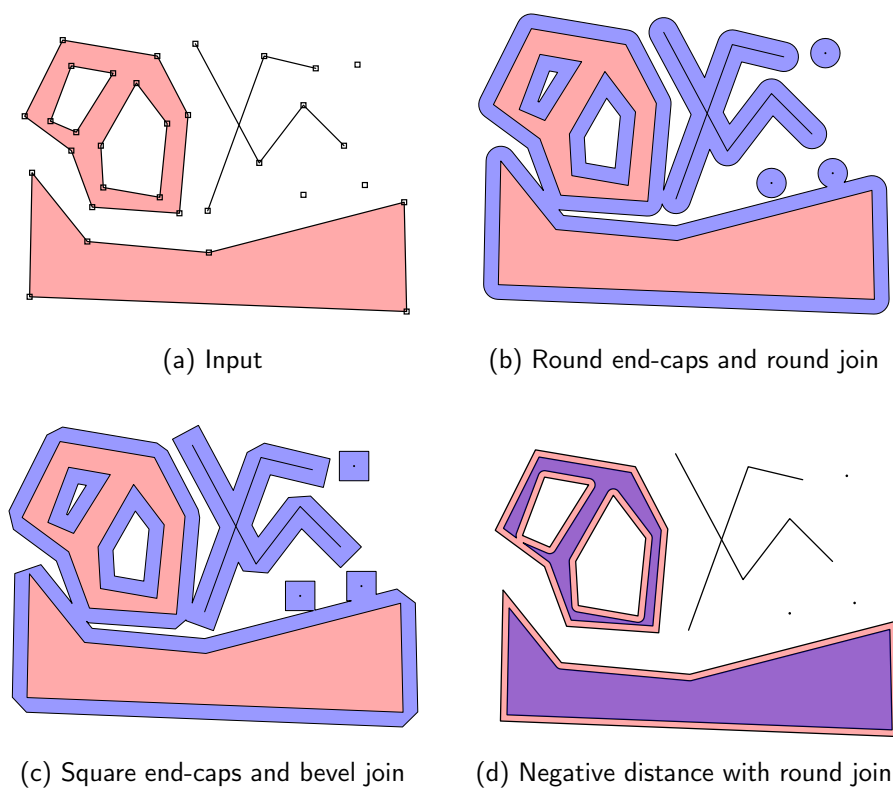


Figure 6.15: Buffer regions

Conclusions and Future Work

7.1 Summary

Our main objective was to create a modern framework for visualizing algorithms and data structures from computational geometry. The software that has been developed accomplishes this goal: end users can explore data structures and the behavior of algorithms, and, through the rich export functionality, they can use the output of visualizations for various purposes. With the geometry editor, there is an experimenting environment for designing algorithm input. Additionally, the program ships with a database that contains examples for each implemented algorithm. Developers benefit from the facilities that the framework provides and it allows them to create new visualizations easily. The framework's flexible design makes it possible to reuse many components to reduce the developer's workload.

Our collection of implemented visualizations already provides partial coverage of the topics considered in introductory courses on computational geometry plus some material on more specialized problems. Hence, it could be used as a complementary tool to assist teaching such courses. Through its open source nature, other developers could participate in the process of creating a thorough collection of animations. This could be done by researchers themselves, but the implementation of visualizations could also be a suitable topic for student's software projects.

7.2 Future Work

LiveCG's main user interface is quite comprehensive, but it could still be improved. For instance, the various visualization-dependent settings described in Section 4.4 could be made accessible through appropriate generic user interfaces. This would allow the user to change these settings conveniently at runtime. Also, not having to edit configuration files by hand with a text editor would increase usability.

Some functionality that one usually expects from desktop applications is still missing. The geometry editor should support undo/redo facilities to let the user revert misguided or accidental changes to the input. In this regard,

it would also be useful if the editor was aware of unsaved changes when the current scene is about to be dismissed either by loading another scene or by closing the program. Users should then be warned about potential data loss and given a chance to save their work.

The visualization for partitioning polygons into monotone pieces (Section 6.3) is currently only static. It should be animated to capture the dynamic behaviour of the algorithm.

To increase coverage of topics, more visualizations should be implemented or ported from existing applets. Obvious candidates are line segment intersection, trapezoidal maps, multiple algorithms for computing the convex hull, computing the closest pair of a point set, visibility problems and robot motion planning. Also, algorithms and data structures for geometric searching like R-trees and Quadtrees could be nicely visualized. To reduce the effort, it would be useful to get access to the source code of existing visualizations and port them to the framework.

In the long run, it would be desirable to support more types of geometric objects. For instance, for implementing visualizations on arrangements of lines, we would need the ability to add infinite lines or rays to the scene. To support more topics from computer graphics, different types of splines should be added. Of course, 3-dimensional objects for input would allow many more algorithms or generalizations of 2-dimensional variants to be implemented. However, supporting them would be a rather big task, as the input and manipulation of such objects require sophisticated user interfaces that do not have much in common with their 2-dimensional counterparts.

Using the currently available back ends, a number of interesting output formats could be implemented on top of them. For instance, combining the PNG output module with the text output could be used for generating static HTML websites that explain an algorithm in detail. In a similar fashion, text output could be combined with Ipe images to produce explanatory \LaTeX documents. On the other hand, combining sequences of images could also be used for the creation of video files that would bundle the dynamics of an algorithm's execution into a single artifact.

Another interesting idea is to bring the visualizations to the web, not only statically, but with support for user interaction. Therefore, a GWT-based back end could be implemented. With appropriate user interfaces it should be possible to build an experimenting environment similar to the Swing UI while sharing a common codebase for the visualizations themselves.

References

- [AG95] Helmut Alt and Michael Godau. „Computing the Fréchet distance between two polygonal curves“. *International Journal of Computational Geometry & Applications* 5.01–02 (1995), pp. 75–91.
- [Ame+95] Nina Amenta, Stuart Levy, Tamara Munzner, and Mark Phillips. „Geomview: A System for Geometric Visualization“. In: *Proceedings of the 11th Annual ACM Symposium on Computational Geometry (SoCG)*. SCG '95. Vancouver, British Columbia, Canada: ACM, 1995, pp. 412–413.
- [BCS96] Michael Dwyer Byrne, Richard Catrambone, and John T Stasko. *Do algorithm animations aid learning?* Tech. rep. GIT-GVU-96-18. Georgia Institute of Technology, Aug. 1996.
- [Ber+00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*. Springer, 2000.
- [BMS94] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. „On degeneracy in geometric computations“. In: *Proceedings of the 5th annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 1994, pp. 16–23.
- [BN02] Matthias Bäßen and Stefan Näher. „GeoWin A Generic Tool for Interactive Visualization of Geometric Algorithms“. In: *Software Visualization*. Ed. by Stephan Diehl. Vol. 2269. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 88–100.
- [BP98] Ronald Baecker and Blaine Price. „The early history of software visualization“. *Software Visualization* (1998), pp. 29–34.
- [BS81] Ronald M Baecker and David Sherman. *Sorting Out Sorting. 16mm color sound film*. Shown at SIGGRAPH. 1981.
- [BS84] Marc H. Brown and Robert Sedgewick. „A System for Algorithm Animation“. *SIGGRAPH Computer Graphics* 18.3 (July 1984), pp. 177–186.

- [Buc+13] Kevin Buchin, Maike Buchin, Rolf van Leusden, Wouter Meulemans, and Wolfgang Mulzer. „Computing the Fréchet Distance with a Retractable Leash“. In: *Proceedings of the 21st Annual European Symposium on Algorithms (ESA)*. Springer, 2013, pp. 241–252.
- [Cha96] Timothy M Chan. „Optimal output-sensitive convex hull algorithms in two and three dimensions“. *Discrete & Computational Geometry* 16.4 (1996), pp. 361–368.
- [CK70] Donald R Chand and Sham S Kapur. „An algorithm for convex polytopes“. *Journal of the ACM (JACM)* 17.1 (1970), pp. 78–86.
- [Coo+14] Matthew L. Cooper, Clifford A. Shaffer, Stephen H. Edwards, and Sean P. Ponce. „Open source software and the algorithm visualization community“. *Science of Computer Programming* (2014).
- [ELW98] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O’Reilly & Associates, Inc., 1998.
- [Eps+94] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. „A workbench for computational geometry“. *Algorithmica* 11.4 (1994), pp. 404–428.
- [For87] Steven Fortune. „A sweepline algorithm for Voronoi diagrams“. *Algorithmica* 2.1-4 (1987), pp. 153–174.
- [Gar+78] Michael R Garey, David S Johnson, Franco P Preparata, and Robert E Tarjan. „Triangulating a simple polygon“. *Information Processing Letters* 7.4 (1978), pp. 175–179.
- [GS88] Leonidas J Guibas and Jorge Stolfi. *Ruler, Compass and Computer: The Design and Analysis of Geometric Algorithms*. Springer, 1988.
- [HD99] Alejo Hausner and David P Dobkin. In: Jörg-Rüdiger Sack and Jorge Urrutia. *Handbook of Computational Geometry*. Elsevier, 1999. Chap. Making Geometry Visible: An Introduction to the Animation of Geometric Algorithms.
- [Hof89] Christoph M Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, 1989.
- [Jar73] Ray A Jarvis. „On the identification of the convex hull of a finite set of points in the plane“. *Information Processing Letters* 2.1 (1973), pp. 18–21.
- [Jef98] Clinton L. Jeffrey. In: John Stasko, Marc Domingue, Marc H Brown, and Blaine A Price. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1998. Chap. A Menagerie of Program Visualization Techniques.

- [Kre06] Marc van Kreveld. „Computational geometry: Its objectives and relation to GIS“. *Nederlandse Commissie voor Geodesie (NCG)* (2006), pp. 1–8.
- [KS13] Ville Karavirta and Clifford A. Shaffer. „JSAV: The JavaScript Algorithm Visualization Library“. In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '13. Canterbury, England, UK: ACM, 2013, pp. 159–164.
- [LP84] D. T. Lee and F. P. Preparata. „Euclidean shortest paths in the presence of rectilinear barriers“. *Networks* 14.3 (1984), pp. 393–410.
- [MP78] David E. Muller and Franco P. Preparata. „Finding the intersection of two convex polyhedra“. *Theoretical Computer Science* 7.2 (1978), pp. 217–236.
- [PBS93] Blaine A Price, Ronald M Baecker, and Ian S Small. „A principled taxonomy of software visualization“. *Journal of Visual Languages & Computing* 4.3 (1993), pp. 211–266.
- [PBS98] Blaine Price, Ronald Baecker, and Ian Small. In: John Stasko, Marc Domingue, Marc H Brown, and Blaine A Price. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1998. Chap. An Introduction to Software Visualization.
- [PS85] Franco P Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.
- [RJ93] P. de Rezende and W. Jacometti. „Animation of Geometric Algorithms Using GeoLab“. In: *Proceedings of the 9th Annual ACM Symposium on Computational Geometry (SoCG)*. SCG '93. San Diego, California, USA: ACM, 1993, pp. 401–402.
- [SBL93] John Stasko, Albert Badre, and Clayton Lewis. „Do Algorithm Animations Assist Learning? An Empirical Study and Analysis“. In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: ACM, 1993, pp. 61–66.
- [Sch90] Peter Schorn. „An object-oriented workbench for experimental geometric computation“. In: *Proceedings of the 2nd Canadian Conference on Computational Geometry (CCCG)*. 1990, pp. 172–175.

- [Sch95] Otfried Schwarzkopf. „The Extensible Drawing Editor Ipe“. In: *Proceedings of the 11th Annual ACM Symposium on Computational Geometry (SoCG)*. SCG '95. Vancouver, British Columbia, Canada: ACM, 1995, pp. 410–411.
- [Sei88] Raimund Seidel. *Constrained Delaunay triangulations and Voronoi diagrams with obstacles*. Tech. rep. 260. IIG, TU Graz, June 1988, pp. 178–191.
- [Sha+10] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. „Algorithm Visualization: The State of the Field“. *ACM Transactions on Computing Education* 10 (Aug. 2010), pp. 1–22.
- [Sha+11] Clifford A. Shaffer, Monika Akbar, Alexander Joel D. Alon, Michael Stewart, and Stephen H. Edwards. „Getting Algorithm Visualizations into the Classroom“. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. SIGCSE '11. Dallas, TX, USA: ACM, 2011, pp. 129–134.
- [Sha78] Michael Ian Shamos. „Computational geometry.“ PhD thesis. Yale University, 1978.
- [SL98] John Stasko and Andrea Lawrence. In: John Stasko, Marc Domingue, Marc H Brown, and Blaine A Price. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1998. Chap. Empirically Assessing Algorithm Animations as Learning Aids.
- [Sta+98] John Stasko, Marc Domingue, Marc H Brown, and Blaine A Price. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1998.
- [TD95] Ayellet Tal and David Dobkin. „Visualization of geometric algorithms“. *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 194–204.

Acronyms

API	Application programming interface
AWT	Abstract Window Toolkit
BALSA	Brown Algorithm Simulator and Animator
BMP	Bitmap image file
CAD	Computer aided design
CAE	Computer aided engineering
CAM	Computer aided manufacturing
CG	Computational geometry
CGAL	Computational Geometry Algorithms Library
CLI	Command-line interface
DCEL	Doubly-connected edge list
DOM	Document Object Model
DTD	Document type definition
DVI	Device independent file format
EPS	Event point schedule
GIF	Graphics interchange format
GIS	Geographic information system
GPL	General Public License
GUI	Graphical user interface
GWT	Google Web Toolkit
HTML	HyperText Markup Language
JPEG	Joint Photographic Experts Group

JTS	JTS Topology Suite
JVM	Java Virtual Machine
LiveCG	Live Interactive Visualization Environment for Computational Geometry
MVC	Model-view-controller
PDF	Portable Document Format
PGF	Portable Graphics Format
PNG	Portable Network Graphics
PS	PostScript
SLS	Sweep line status
SVG	Scalable Vector Graphics
UI	User interface
URL	Uniform resource locator
WKT	Well-known text
XML	Extensible Markup Language



Source Code

A.1 Painter Interface

Manipulating the Drawing Appearance

```
public void setColor(Color color);

public void setStrokeWidth(double width);

public void setStrokeNormal();

public void setStrokeDash(float[] dash, float phase);
```

Drawing Primitive Objects

```
public void drawRect(int x, int y, int width, int height);

public void drawRect(double x, double y,
                    double width, double height);

public void fillRect(int x, int y, int width, int height);

public void fillRect(double x, double y,
                    double width, double height);

public void drawLine(int x1, int y1, int x2, int y2);

public void drawLine(double x1, double y1,
                    double x2, double y2);

public void drawCircle(double x, double y, double radius);

public void fillCircle(double x, double y, double radius);
```

Drawing Polygonal Chains

```
public void drawPath(List<Coordinate> points,  
                    boolean close);
```

Drawing Geometric Objects

```
public void drawChain(Chain chain);  
  
public void drawPolygon(Polygon polygon);  
  
public void fillPolygon(Polygon polygon);
```

Drawing AWT Shapes

```
public void draw(Shape shape);  
  
public void fill(Shape shape);
```

Drawing Text

```
public void drawString(String text, double x, double y);
```

Manipulating the Clip

```
public Object getClip();  
  
public void setClip(Object clip);  
  
public void clipRect(double x, double y,  
                   double width, double height);  
  
public void clipArea(Shape shape);
```

Applying Transforms

```
public AffineTransform getTransform();  
  
public void setTransform(AffineTransform t);
```

Drawing Raster Images

```
public void drawImage(BufferedImage image, int x, int y);
```