A Geometric Workbench for Degree-Driven Algorithm Design

Clinton Freeman
freeman@cs.unc.edu

Jack Snoeyink
snoeyink@cs.unc.edu

June 2, 2014

Abstract

Millman built a C++ library (DDAD) to facilitate implementing algorithms with low degree predicates. Our workbench extends DDAD with a visual event system and provides a standalone GUI that can generate input data and render algorithm execution.

1 Introduction

Geometric algorithms are often difficult to implement and convey to others. Algorithm *implementers* must correct programming errors and handle degenerate situations correctly. Traditional debuggers provide only textual or numerical representations of geometric data structures, and generating degenerate geometric input is a nontrivial task for which there is often little assistance. Algorithm *presenters* must convey their ideas to audiences of researchers and students. Many presenters use static depictions and verbally explain algorithm mechanics, a type of presentation that does not fully capture the dynamic nature of geometric algorithms.

A *geometric workbench* aids algorithm implementers and presenters by providing facilities to dynamically visualize geometric algorithms. Implementers can visually inspect geometric relationships and properties of data structures, quickly recognizing erroneous computations. Presenters can produce animations of their algorithms, more clearly conveying essential ideas to their audience. Both types of geometers can interactively control the flow of execution and easily generate degenerate input data.

Degree-driven algorithm design encourages robust geometric computing by attempting to minimize an algorithm's arithmetic precision with its running time and space [14]. Millman built a C++ library (DDAD) to facilitate implementing algorithms with low degree predicates. Our workbench extends DDAD with a visual event system and provides a standalone GUI that can generate input data and render algorithm execution. Figure 1 captures the workbench rendering a terrain mesh built by incremental delaunay triangulation.

Section 2 reviews previous work. Section 3 recalls a general framework for designing software visualization systems. Section 4 uses the framework to explain our workbench desiderata. Section 5 provides an architectural overview of the workbench and section 6 runs through a case study of animating a convex hull algorithm. Section 7 concludes by explaining how current workbench architecture supports our desiderata and identifies areas in which it can improve.

2 General Workbench Desiderata

From their thorough review of geometric workbench systems, Dobkin and Hausner extracted a set of design decisions that must be made by new systems [11]. However, a geometric workbench is a specific type of *software visualization* (SV) system. The SV literature contains several well-developed taxonomies that are designed to evaluate SV systems [6]. In particular, Price *et al.* outline six major categories in which they judge SV systems: scope, content, form, method, interaction, and effectiveness [17]. We found that this detailed taxonomy provided a more structured way of organizing the desiderata of new workbench systems.

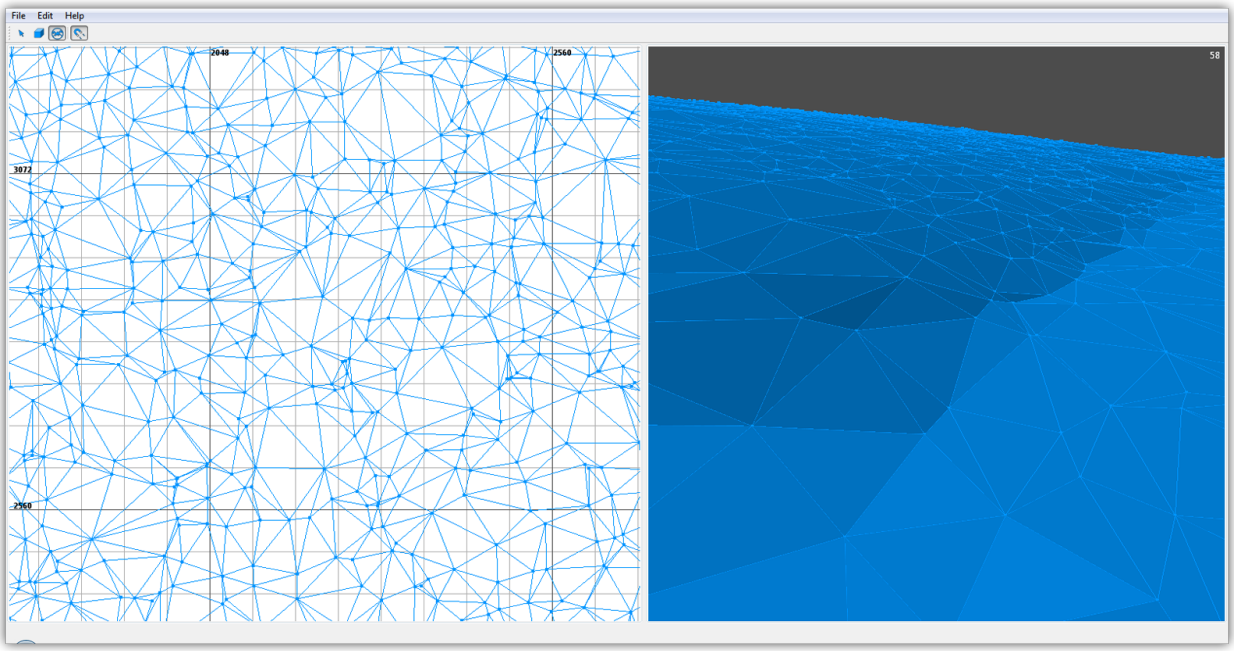

Figure 1: Our workbench animates incremental terrain mesh generation.

This section recalls the taxonomy for evaulating SV systems and relates each category to the specific case of workbench systems. The taxonomy is a hierarchy of categories, each of which may be described in terms of a question about the system. This makes the task of describing our specific desiderata quite easy because we may simply answer each of these questions in turn.

2.1 Scope

What is the range of programs that the SV system may take as input for visualization?

1. Generality. Can the system handle a generalized range of programs or does it display a fixed set of examples?
    (a) Hardware. What hardware does it run on?
    (b) Operating System. What operating system is required to run it?
    (c) Language. What programming language must user programs be written in?
        i. Concurrency. If the programming language is capable of concurrency, can the SV system visualize the concurrent aspects?
    (d) Applications. What are the restrictions on the kinds of user programs that can be visualized?
        i. Specialty. What kinds of programs is it particularly good at visualizing (as opposed to simply capable of visualizing)?
2. Scalability. To what degree does the system scale up to handle large examples?
    (a) Program. What is the largest program it can handle?
    (b) Data Sets. What is the largest input data set it can handle?

1. Scope. What is the range of programs that the SV system may take as input for visualization?

(a) Generality. Can the system handle a generalized range of programs or does it display a fixed set of examples?

i. Hardware. What hardware does it run on?

ii. Operating System. What operating system is required to run it?

iii. Language. What programming language must user programs be written in?

A. Concurrency. If the programming language is capable of concurrency, can the SV system visualize the concurrent aspects?

iv. Applications. What are the restrictions on the kinds of user programs that can be visualized?

A. Specialty. What kinds of programs is it particularly good at visualizing (as opposed to simply capable of visualizing)?

(b) Scalability. To what degree does the system scale up to handle large examples?

i. Program. What is the largest program it can handle?

ii. Data Sets. What is the largest input data set it can handle?

2. Content. What subset of information about the software is visualized by the SV system?

(a) Program. To what degree does the system visualize the actual implemented program?

i. Code. To what degree does the system visualize the instructions in the program source code?

A. Control Flow. To what degree does the system visualize the flow of control in the program source code?

ii. Data. To what degree does the system visualize the data structures in the program source code?

A. Data Flow. To what degree does the system visualize the flow of data in the program source code?

(b) Algorithm. To what degree does the system visualize the high-level algorithm behind the software?

i. Instructions. To what degree does the system visualize the instructions in the algorithm?

A. Control Flow. To what degree does the system visualize the flow of control in the algorithm instructions?

ii. Data. To what degree does the system visualize the data structures in the algorithm?

A. Data Flow. To what degree does the system visualize the flow of data in the algorithm?

(c) Fidelity and Completeness. Do the visual metaphors present the true and complete behavior of the underlying virtual machine?

i. Invasiveness. If the system can be used to visual concurrent applications, does its use disrupt the execution sequence of the program?

(d) Data Gathering Time. Is the data on which the visualization depends gathered at compile-time, at run-time, or both?

i. Temporal Control Mapping. What is the mapping between 'program time' and 'visualization time'?

ii. Visualization Generation Time. Is the visualization produced as a batch job (post-mortem) from data recorded during a previous run, or is it produced live as the program executes?

3. Form. What are the characteristics of the output of the system (the visualization)?

(a) Medium. What is the primary target medium for the visualization system?

(b) Presentation Style. What is the general appearance of the visualization?

i. Graphical Vocabulary. What graphical elements are used in the visualization produced by the system?

A. Colour. To what degree does the system make use of colour in its visualizations?

B. Dimensions. To what degree are extra dimensions used in the visualization?

ii. Animation. If the system gathers run-time data, to what degree does the resulting visualization use animation?

iii. Sound. To what degree does the system make use of sound to convey information?

(c) Granularity. To what degree does the system present coarse-granularity details?

i. Elision. To what degree does the system provide facilities for eliding information?

(d) Multiple Views. To what degree can the system provide multiple synchronized views of different parts of the software being visualized?

(e) Program Synchronization. Can the system generate visualizations of multiple programs simultaneously?

4. Method. How is the visualization specified?

(a) Visualization Specification Style. What style of specification is used?

i. Intelligence. If the visualization is automatic, how advanced is the visualization software from an AI point of view?

ii. Tailorability. To what degree can the user customize the visualization?

A. Customization Language. If the visualization is customizable, how can the visualization be specified?

(b) Connection Technique. How is the connection made between the visualization and the actual software being visualized?

i. Code Ignorance Allowance. If the visualization system is not completely automatic, how much knowledge of the program code is required for a visualization to be produced for the user?

ii. System-Code Coupling. How tightly is the visualization system coupled with the code?

5. Interaction. How does the user of the SV system interact with and control it?

(a) Style. What method does the user employ to give instructions to the system?

(b) Navigation. To what degree does the system support navigation through a visualization?

i. Elision Control. Can the user elide information or suppress detail from the display?

ii. Temporal Control. To what degree does the system allow the user to control the temporal aspects of the execution of the program?

A. Direction. To what degree can the user reverse the temporal direction of the visualization?

B. Speed. To what degree can the user control the speed of execution?

(c) Scripting Facilities. Does the system provide facilities for managing the recording and playing back of interactions with particular visualizations?

6. Effectiveness. How well does the system communicate information to the user?

(a) Purpose. For what purpose is the system suited?

(b) Appropriateness and Clarity. If the automatic (default) visualizations are provided, how well do they communicate information about the software?

(c) Empirical Evaluation. To what degree has the system been subjected to a good experimental evaluation?

(d) Production Use. Has the system been in production use for a significant period of time?

3 Previous Work

A geometric workbench combines a geometric algorithm library with a geometric algorithm visualizer. *Geometric algorithm libraries* provide a broad selection of algorithms and the substrate of types upon which they are built. *Geometric algorithm visualizers* provide a GUI capable of animating and visually debugging those algorithms implemented in the library. They borrow heavily from techniques used in general algorithm animation systems [3, 21, 20, 22]. Previous works may thus be categorized as libraries [12, 9, 15, 8], visualizers [16, 10, 1, 2], and full workbench systems [18, 5, 4, 7, 23, 19, 24].

3.1 Geometric workbench systems

XYZ GeoBench (1991) XYZ (eXperimental geometrY Zurich) GeoBench was a geometric workbench developed by Peter Schorn under the supervision of Jurg Nievergelt. Schorn's 1991 thesis focused on the question of how to produce good software for geometric computation, with a particular emphasis on geometric robustness [18]. Schorn describes the GeoBench as "a programming environment, implemented in an object oriented language, for the rapid prototyping of geometric software and a testbed for experiments," noting that "algorithm animation is used for demonstration purposes and debugging." The XYZ Library was built on top of the GeoBench and offered implementations of a large number of geometric algorithms.

There are several notable features about the GeoBench. The first is the use of interchangable arithmetic and software simulation of parameterized floating point numbers. In particular, algorithm implementations make reference to an abstract 2D point class that does not specify a particular number system for the coordinates. The abstract point class is extended to form concrete point types for single-precision (realPoint), long integer (longIntPoint), and parameterized floating point (floatPoint) coordinate types. Counterinuitively, the parameterized floating point arithmetic was not used to ameliorate robustness problems by increasing precision, rather it was used to simulate low precision floating point to make the robustness problems more pronounced.

Workbench (1991) Workbench is a system similar to XYZ GeoBench that focused on implementing complex geometric algorithms instead of robustness issues. The system is built in Smalltalk and composed of three main components: a library, visualization GUI, and tools for extending each. It focused heavily on empirical comparisons of different algorithms and using animations for teaching and demonstration. They define the minimum criteria for a geometric workbench as a system with: representations of geometric objects, geometric data types, non-geometric data types, algorithmic implementations that adhere to specification, different arithmetic types, and a GUI that provides animation and debugging facilities.

GeoLab (1993) de Rezende created GeoLab, a system that binds together support for software and algorithm development with realtime interaction [5, 4]. The system supports software development with built-in abstract data types for geometric objects, data structures, basic algorithms, and mechanisms for incorporating new components *without* recompilation. The system supports user interaction with the ability to construct input data, debug implementations visually, gather statistics, input/output geometric data, and customize algorithm animations. Although the system is implemented in object-oriented C++, algorithms are not necessarily member functions of the classes upon which they operate – they may be free functions. This echoes Meyer's advice that minimal use of member functions can often increase encapsulation.

GASP (1995) Tal and Dobkin describe the Geometric Animation System, Princeton (GASP) [23]. The system allows users to quickly create 3D visualizations, animate complex geometric algorithms, and visually debug implementations. The use of style files to control the visual aspects of the animation allows users to produce new renderings without recompiling the system.

GeoBuilder (2009) GeoBuilder is a cross-platform geometric workbench written in Java [24]. It features a novel mechanism for automatically positioning the 3D camera during algorithm execution and allows

users to collaborate on programming and visualization. Wei et al demonstrate the 3D tracking feature by constructing convex hulls and detecting line segment intersections. They note that automatically positioning a single view of an algorithm may not be sufficient to capture all changes in algorithm state, and identify automatic positioning of multiple cameras for future work.

3.2 Geometric algorithm libraries and visualizers

LEDA The library of efficient data structures and algorithms (LEDA) is often mentioned in the context of workbenches.

CGAL The computational geometry algorithms library (CGAL) is the current standard implementation among the geometry community - it used to provide support for GeomView but now just has its own visualization module.

Geomview Geomview is a geometric visualization system that focuses on rendering and manipulating geometric data in 3-space [16, 10, 1]. It is able to render both static geometry and dynamic geometry produced by an external program running concurrently. External programs that use Geomview in this fashion are called "external modules." It gained a large userbase after its initial release in 1991 due to its decoupled approach to animating geometric algorithms; users could implement algorithms however they wanted and simply interface with Geomview at runtime. Of particular note is its 4D visualization module that allow users to explore 4-dimensional geometry through various projections. CGAL provides a module for producing Geomview visualizations.

GeoWin GeoWin is a C++ data type that provides the ability to visualize sets of geometric objects, with a focus on two dimensions [2]. This functionality may be used to visualize the output of geometric algorithms or to animate the progression of algorithms. GeoWin defines a programming interface to define scenes and an interactive interface to define how the user may interact with the scene. The data type integrates directly with LEDA and CGAL while custom libraries must implement the interfaces, entailing a dependency on LEDA types.

4 DDAD Workbench Desiderata

Geometric objects on a digital computer are composed of two types of data: numerical and combinatorial. Examples of numerical data include the Cartesian coordinates of a point in 3-space, the length of a line segment connecting two such points, or the angle between two such line segments. Examples of combinatorial information include grouping two points as an edge, grouping a collection of edges as a face, or grouping a collection of faces as a surface.

Geometric algorithms that operate on geometric objects are best thought of as two types of operations: predicates and constructions. Predicates determine relationships between objects. A predicate might determine if a point is to the left, right, or is collinear with a line segment, determine if a point is inside, outside, or on a circle, or determine if a line intersects a plane in one, none, or infinitely many points. Constructions produce new geometric objects from existing geometric objects. A construction might produce the rotation of a point around an origin, produce the point of intersection between two line segments, or produce an offset of an algebraic curve.

Scope describes the range of programs the system can take as input for visualization. First, the system should visualize geometric algorithms implemented with primitives and predicates from the degree-driven algorithm design library (DDAD). Second, the system should provide example implementations and visualizations of algorithms that solve textbook computational geometry problems. Third, the system should be flexible enough to visualize new geometric algorithms from the facilities used in example visualizations.

Fourth, the system should run well given inputs large enough to verify the correctness of a particular implementation.

Content describes what subset of information about the software is visualized by the SV system. For algorithm implementers, the system should visualize information about the concrete implementation of the algorithm. This entails combining traditional debugging facilities with the new ability to visualize geometric structures and operations. For algorithm presenters, the system should visualize information about the high-level algorithm description. This entails mechanisms for displaying visualizations that do not necessarily correspond to the underlying implementation details.

Form describes the characteristics of the output of the system (the visualization). First, the output medium should be an interactive computer program. This is sufficient for implementers, but presenters may want to produce a digital recording of an algorithm animation. Fortunately, screen capture programs can facilitate producing this type of output. Second, the output should have a rich graphical vocabulary, displaying standard graphics primitives (points, lines, polygons), each with a variety of attributes (color, transparency, pattern, texture). Presenters in particular desire highly expressive output. Third, the output should contain both 2D and 3D elements. The third dimension can be used both for visualizing inherently 3D algorithms and to effectively display non-dimensional information. Fourth, the output should have multiple levels of information granularity, with the ability to elide and reveal specific levels as needed. Fifth, users should be able to view the same data in multiple ways. This capability is particularly pertinent in geometry where insight into a problem may be gained from viewing a dual formulation (*e.g.* viewing points as lines).

Method describes how the visualization is specified. Implementers desire highly automated specifications that minimally invade implementation source code. Increased automation increases the likelihood a tool will be used for debugging assistance. Useful automated visualizations require a high level of intelligence in order to correctly recognize and display high level data structures. In contrast, presenters are less concerned with automated visualization specifications, and accept a higher degree of invasiveness in return for customization and increased expressibility. The system should strike a balance between automation and customization.

Interaction describes how the user of the software visualization system interacts with and controls it. First, users should be able to control the speed of execution. This includes starting, stopping, speeding up, slowing down, and single stepping. Second, users should be able to control a camera to navigate around the scene and to focus on different objects. This is especially important for 3D algorithms and larger data sets. Third, users should be able to record and play back particular runs of a visualization. For presenters this could aid pedagogy, and implementers might record when an algorithm fails.

4.1 Current Capabilities

The workbench currently satisfies the desired functionality as follows. It extends the DDAD library with visual types and encapsulates these into a geometry kernel. The geometry kernel is paired with a visualizer that is capable of animating algorithm execution. The workbench strikes a balance between automated visualizations and expressive power by using object oriented design to encapsulate visual commands in lower level objects. The user is able to control the speed of execution and pause at important moments. They control two different views of the scene and can move these views interactively as the algorithm executes.

5 Case Study: Melkman's Algorithm

This section examines using our workbench to animate Melkman's convex hull algorithm. By providing a concrete example we hope to illuminate the major steps required to animate algorithms in general. Before we get started, let us briefly examine the algorithm in question.

A *polyline* P is a polygonal chain of vertices $p_1, p_2, \ldots, p_n$ connected by line segments $p_i p_{i+1}$ for $1 \leq i < n$. P is *simple* if the only intersection between segments is at their shared endpoints. Melkman's algorithm incrementally computes the convex hull of a simple polyline in $O(n)$ time [13]. It can also be used to compute the convex hull of arbitrary point sets if we first sort by x coordinate, breaking ties by y coordinate.

Algorithm Melkman

Input: Simple polyline $P = \langle v_1, \ldots, v_m \rangle$.
Output: CH(P).

```
1: procedure MELKMAN(P)
2:     CH(P).push(v_2); CH(P).push(v_1); CH(P).push(v_2);        ▷ Init hull
3:     for i = 3 ... m do
4:         derp
5:     end for
6: end procedure
```

Animating an algorithm using the workbench is composed of a number of tasks, namely

- Implement input data structures and instrument them with visualization code.
- Optionally modify the GUI to allow the user to create instances of the input data structure.
- Implement output data structures and instrument them with visualization code.
- Implement predicates.
- Implement the algorithm and instrument it with a small amount of visualization code.
- Optionally modify the GUI to allow the user to run the algorithm on selected input data.

5.1 Data Structures

polychain_2r
    polygon_2r

5.2 Generating Input Data

Create button and implement click handler. buttongroup to model mutually exclusive input states.
    define input states. configmanager singleton.
    Handle ortho widget mouse clicks forward signals from ortho to scene observer

5.3 Predicates

Given an oriented line pq and a point r, the 2D orientation predicate ORIENT2D(p, q, r) answers the question, "is r to the left, right, or on pq?" It is often written as the sign of the 2-by-2 determinant,

$$\text{ORIENT2D}(p, q, r) = \text{SIGN}\left(\begin{vmatrix} p_x - r_x & p_y - r_y \\ q_x - r_x & q_y - r_y \end{vmatrix}\right).$$

6 Workbench Architecture

The workbench comprises two major components: a geometric algorithm library and a geometric algorithm visualizer. The *geometry library* implements geometric algorithms. It defines types for arithmetic, linear algebra, and geometric primitives (e.g. points, line segments, and triangles). These are building blocks with which it defines more complex geometric objects (e.g. polygons and polytopes). All of these may be

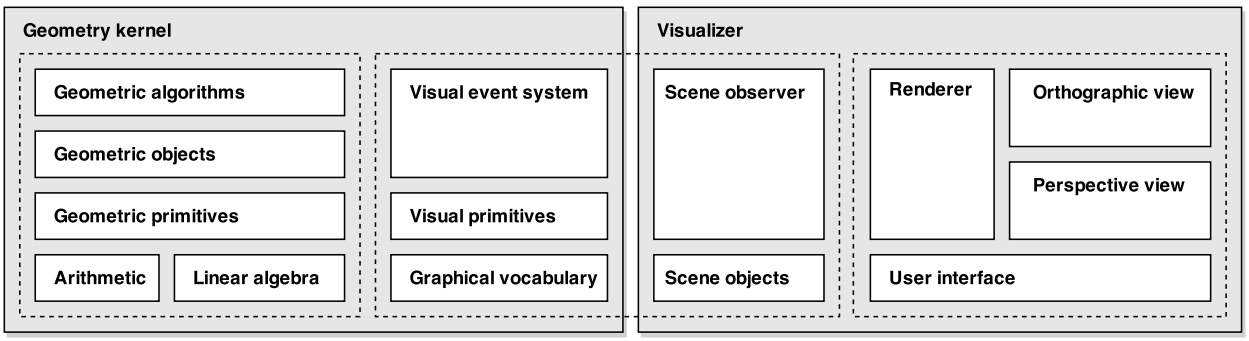

Figure 2: An abstract overview of the workbench architecture.

used to implement geometric algorithms. The *visualizer* displays the execution of geometric algorithms. It is structured using the model-view-controller (MVC) design pattern. In particular, it maintains a visual model, several views of the visual model, and a user interface that functions as the controller. In this section, we are concerned with explaining how the kernel and visualizer work together to produce algorithm visualizations.

6.1 The Geometry Library

The *arithmetic* and *linear algebra* components are orthogonal to our discussion of visualization. The *graphical vocabulary* component is necessary to define briefly but is straightforward to understand and depends only on fundamental language types. In particular, it consists of types to specify color, transparency, and a lighting model.

Geometric primitives are geometric objects of constant size. We are concerned with three: points, line segments, and triangles[1]. Points are the basic unit of visualization. They are equipped with unique identifiers (UIDs), assigned by registering with the visual model. Line segments and triangles are combinatorial groupings of two and three points, respectively. Thus, they are implicitly equipped with unique identifiers given by the combinatorial grouping of their points' UIDs.

Visual primitives are types that define the subset of graphical vocabulary applicable to a corresponding geometric primitive. For example, a visual point type might specify that points may be assigned a color and rendered as either a circular or square sprite; a visual segment type might also specify a color attribute but have no notion of a sprite shape.

The *visual event system* defines the notion of observable geometry. *Observable geometry* objects notify their observers of changes in their visual state by emitting signals. *Geometry observers* handle these notifications by implementing a corresponding slot. All higher level geometric types that wish to be rendered by the visualizer must implement the observable geometry interface. If a geometric object is composed of other observable geometry objects, then it must also implement the geometry observer interface and subscribe to those other objects.

Many geometric types will be composed of other observable geometry objects, so we define an abstract base class that is both observable and capable of observing other objects. By default, this base class forwards any visual events it receives from the objects it observes onward to its own subscribers. In this way, visual events are passed through a chain of event handlers, eventually arriving at the scene observer in the visualizer.

[1]Other primitives include lines and rays but these are not directly supported by the visual interface.

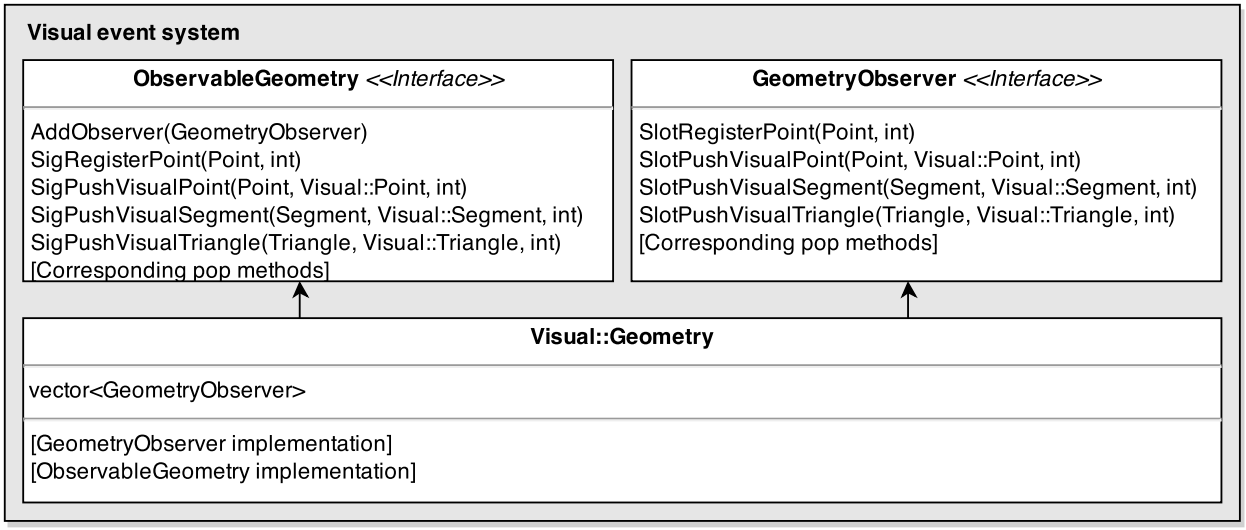

Figure 3: The visual event system embedded in the geometry kernel.

6.2 The Visualizer

Scene objects are wrappers around geometric objects that implement an interface required by the scene observer (e.g. being selectable or having a name). Scene objects observe the geometric object they wrap, and may be observed by the scene observer.

The *scene observer* is the ultimate destination for all visual events produced by geometric objects. It maintains three maps that pair unique geometric primitives with a stack of visual primitives, the top of which defines the current visual state. The scene observer is responsible for translating the graphical vocabulary defined in the kernel into OpenGL buffer objects. The scene observer lives inside of its own thread, and each of the visual events emitted from an observable object may be given an integer value that will pause this thread for a corresponding number of milliseconds.

The *renderer* is responsible for managing OpenGL data and state.The *orthographic view* provides a top-down orthographic projection of the scene. The user may pan around the plane and zoom in/zoom out. The *perspective view* provides a perspective projection of the scene. The user may arbitrarily orient the camera and travel forward and backward along the view direction.

7 Case Study: Incremental Delaunay Triangulation

8 Lessons Learned and Future Directions

Implementing a geometric algorithm workbench is a challenging task with a rich set of problems encompassing a variety of disciplines. We converged on an interesting events design in which the user annotates observable objects to signal changes in their visual state. The user specifies visual state on points, line segments, and triangles, determines delay length between animation sequences, and directs the viewing camera while the animation runs.

Many opportunities for improvement and further exploration exist. First, implementing existing plans of debugging controls affords an immediate improvement in the tool's usefulness. Second, allowing the presenter to specify sounds to accompany interesting events provides another means of conveying information. Finally, creating better abstractions of common data structures and investigating integration with existing debuggers reduces system invasiveness for the implementer.

References

[1] N. Amenta, S. Levy, T. Munzner, and M. Phillips. Geomview: A system for geometric visualization. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 412–413. ACM, 1995.

[2] M. Bäsken and S. Näher. Geowin a generic tool for interactive visualization of geometric algorithms. In *Software Visualization*, pages 88–100. Springer, 2002.

[3] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 177–186, New York, NY, USA, 1984. ACM.

[4] P. de Rezende and W. Jacometti. Animation of geometric algorithms using geolab. In *Proceedings of the ninth annual symposium on Computational geometry*, pages 401–402. ACM, 1993.

[5] P. J. de Rezende and W. R. Jacometti. Geolab: An environment for development of algorithms in computational geometry. In *CCCG*, pages 175–180, 1993.

[6] S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer, 2007.

[7] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11(4):404–428, 1994.

[8] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The cgal kernel: A basis for geometric computation. In *Applied Computational Geometry Towards Geometric Engineering*, pages 191–202. Springer, 1996.

[9] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, et al. On the design of cgal, the computational geometry algorithms library. 1998.

[10] A. J. Hanson, T. Munzner, and G. Francis. Interactive methods for visualizable geometry. *IEEE Computer*, 27(7):73–83, 1994.

[11] A. Hausner and D. P. Dobkin. Animation of geometric algorithms. *Handbook of Computational Geometry*, page 389, 1999.

[12] K. Mehlhorn and S. Näher. Leda a library of efficient data types and algorithms. In *Mathematical Foundations of Computer Science 1989*, pages 88–106. Springer, 1989.

[13] A. A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25(1):11–12, 1987.

[14] D. L. Millman. Degree-driven design of geometric algorithms for point location, proximity, and volume calculation. 2012.

[15] M. H. Overmars. Designing the computational geometry algorithms library cgal. In *Applied computational geometry towards geometric engineering*, pages 53–58. Springer, 1996.

[16] M. Phillips, S. Levy, and T. Munzner. Geomview: An interactive geometry viewer. *Notices of the American Mathematical Society*, 40(8):985–988, 1993.

[17] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266, 1993.

[18] P. Schorn. *Robust algorithms in a program library for geometric computation.* PhD thesis, Diss. Techn. Wiss. ETH Zürich, Nr. 9519, 1991. Ref.: J. Nievergelt; Korref.: G. Gonnet, 1991.

[19] M. Shneerson and A. Tal. Gasp-ii: a geometric algorithm animation system for an electronic classroom. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 379–381. ACM, 1997.

[20] J. Stasko. Polka animation designers package. *Lite Version*, pages 30332–0280, 1995.

[21] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[22] J. T. Stasko. Samba animation designer's package. *Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA*, 1995.

[23] A. Tal and D. Dobkin. Visualization of geometric algorithms. *Visualization and Computer Graphics, IEEE Transactions on*, 1(2):194–204, 1995.

[24] J.-D. Wei, M.-H. Tsai, G.-C. Lee, J.-H. Huang, and D.-T. Lee. Geobuilder: A geometric algorithm visualization and debugging system for 2d and 3d geometric computing. *Visualization and Computer Graphics, IEEE Transactions on*, 15(2):234–248, 2009.