Point Location Visualization Demo

Patrick Stephen

April 24, 2017

1 Introduction

The problem of point location is one of the oldest problems in computational geometry, and since it has been considered, a number of algorithms of sufficient complexity have been developed that visualizing these algorithms in relation to each other and comparing their methods is a frequent project to aid in understanding the algorithms and deciding which of them to use in a given problem.

This project has developed a unified visualization demo for a number of point location algorithms, showing the construction and query process at variable rates of visualization for arbitrary DCEL structures. Unfortunately these algorithms are for the most part unstable, and require more time investment and more rigorous testing to be reusable in other applications.

Despite these problems, as this demo was built from scratch in a language where most of the constructs it developed did not yet exist in a well-documented package, it may be of use to the open source community. With some more work this project can develop into a very useful code base for general computational geometry, operations with generic search trees, and operations on DCELs.

2 Code Structure

The library developed can be found at github.com. The primary package, gocompgeo, stores a global set of error types, and a number of subpackages:

2.1 The Search Package

The search package contains interfaces for working with constructs related to searching. It describes methods required for objects to be searchable, for them to be keys and values in search structures, and methods describing persistent search structures.

Search's only subdirectory, tree, details binary search trees and persistent binary search trees. A BST can be either static or dynamic, where a static tree can be searched but not modified, and a dynamic tree can be modified. Although benchmarks currently show little distinction in query time between static and dynamic search trees, the motivation behind this distinction was to enable slow-to-query, pointer-based dynamic trees to be converted into array-based trees after some initialization.

Search defines Red-Black, AVL, and Splay trees, but currently only has Red-Black trees developed and tested. These trees are all defined on the same underly-

ing structure, taking advantage of function pointers to be able to define a tree type on functions that correct the tree's structure after inserts, deletes, and searches.

A limited implementation of Persistent Trees is also defined here, limited in that it uses full tree copies and does not allow for modifications to be made on time stamps prior to the latest timestamp it has seen. This structure does, however, satisfy what we need for Slab Decomposition.

2.2 The DCEL Package

The DCEL package implements Doubly-Connected Edge Lists in up to three dimensions. The contents of this package are for the most part helper functions for specific point location algorithms on a DCEL.

DCELs as described by this package are equivalent to those discussed in class, in that a DCEL is a list of Vertices, HalfEdges, and Faces. This implementation expects that each edge is either immediately followed or preceded by its twin in the HalfEdge list. This specification allows drawing a DCEL to be a little faster, only looping through half as many edges. DCELs also define explicitly that their 0th face is the infinite exterior face.

Constructing a DCEL by hand is demonstrated in the shapes.go file in the DCEL package, but as doing so takes a long time and a lot of effort, testing DCELs through manually defined DCELs is impractical. In order to properly test DCELs, DCEL has a subpackage, OFF, which describes the Object File Format structure and allows for OFF files to be converted into DCELs. [1] Unfortunately, as OFF files only define Faces with outer vertex chains, this code base is primarily tested on DCELs containing no faces with inner portions aside from the infinite exterior face.

2.3 The Geometry Package

The geometry package describes geometric primitives for use in point location and DCELs. These primitives are either structures, such as the concept of a span between some set of minimums and maximums, or utility functions on those structures, such as cross product comparisons or floating-point degeneracy helper functions.

2.4 The Point-Location Package

The point location package itself describes one thing– an interface for point location on a DCEL. It defines several subpackages to implement specific algorithms which return an object satisfying the point location interface:

2.4.1 Visualization

The visualization package controls a channel to send visualization instructions along to another process. The package is used by point location algorithms in combination with the demo application to draw to screen polygons and lines.

2.4.2 Slab Decomposition

Slab Decomposition uses one of the older methods of point location, using a persistent binary search tree to binary search on the x value of a query point followed by the y value of a query point. [2] Other than brute force, this is the most stable algorithm in the code base.

This does not use limited node copying, but rather full-tree copies, and so can be improved in terms of space efficiency. It also, while being the most stable, is not prepared for all inputs and will on occasion miss edges to be removed in constructing the persistent search tree.

2.4.3 Brute Force

The brute force package currently just holds a basic plumb line algorithm, but with credit to the simplicity of the algorithm seems stable. The plumb line algorithm manually checks every face in the polygon in strict order.

2.4.4 Trapezoidal Map

The trapezoidal map approach in this package is described in de Berg's textbook at page 128 [3]. This trapezoidal implementation is just one of many varieties of the concept of splitting the planar subdivision into trapezoids. This algorithm is very unstable, but doesn't crash and has a visualization which is helpful for figuring out what exactly it did wrong.

2.4.5 Kirkpatrick

The Kirkpatrick package holds the beginnings of the triangulation-tree approach introduced by Kirkpatrick. [4] This is unfinished and is thus disabled in the demo application.

2.4.6 Monotonization

The monotization package contains a methods to convert DCELs into y-monotone polygons and a method to convert the resulting polygons into a triangulation. This package is based off of segments of de Berg's text [3] and was built as an aid for the Kirkpatrick package, but due to the Kirkpatrick's package being incomplete, is untested.

2.5 The Application Package

The application package, "demo", defines the methods for rendering and manipulating a DCEL, along with running and visualizing point location algorithms. For instructions on using these operations in the running program, see the Application Usage.

DCELs are rendered as structures called Polyhedrons, which wrap around DCELs and hold color information for each face and edge of a DCEL. Polyhedrons support rotation and scaling operations on a DCEL, and have a number of assistant functions to properly transform faces into polygons, draw DCEL elements in depth-order, and correct the position of a moving DCEL to stay in the bounds of a rgba buffer before being uploaded to the window.

Polyhedrons could be more sophisticated in how they manipulate color information, but the underling engine running this application does not have access to graphics cards, and so things like projection, lighting, and anti-aliasing are foregone in order to reduce stress on the CPU.

In order to interact with polyhedrons on screen, they themselves are wrapped as structures called Interactive Polyhedrons, which store collision areas for Vertices in a DCEL to react to mouse clicks on these

[1]This itself is done through point location. It is not done through the algorithms coded in this package, as these methods were coded to work on DCELs and collision spaces are not DCELs. This internally uses my fork of an R tree package, rtreego. this work, found in the oak engine, was not done as a part of this

vertices.[1]

Another interesting structure in the demo package is the dynamic ticker. A dynamic ticker sends out a signal at some defined time interval. This time interval can be remotely changed, hence the ticker being dynamic. The structure uses Go's built in channels to communicate across processes. The demo uses a dynamic ticker to allow a user to modify the speed at which a point location algorithm runs, while it runs.

Remaining code in the demo package either performs some modification on a DCEL in response to a mouse click or key press, or interacts with the oak engine to control the flow of the application.

3 Application Usage

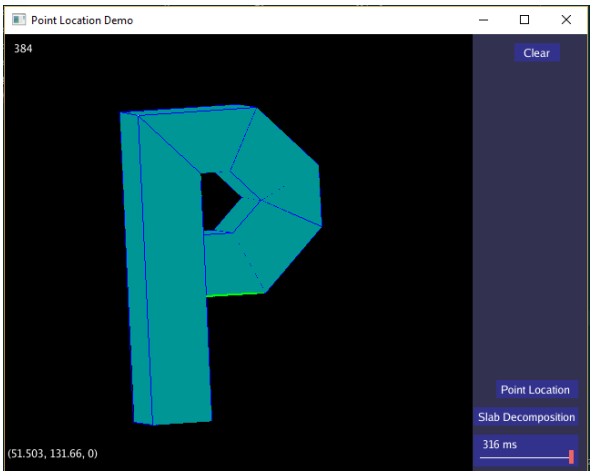

Figure 1: The application window. DCEL on left, control buttons on right.

3.1 Keyboard Commands

- Arrow Keys: Move the DCEL around the viewport. If shift is held, Up and Down will scale the DCEL larger or smaller.
- D and C: Change the depth value of the mouse position. Useful when moving points or defining faces of a DCEL.
- 1 through 4: Set the mouse mode to one of Rotate, Move Point, Point Locate, or Define Face.
- Shift: Some operations have alternate effects when shift is held.

3.2 On-Screen Buttons

These buttons are listed from top to bottom, on the right side of the screen.

- Clear: The clear button will reset the application so that it has no DCEL loaded and most application settings are reset to their starting setting.
- Step: When in stepwise visualization mode, this button will increment the step of the current point location algorithm.
- Mouse Mode: The second button represents the current mouse mode, as manipulated by the 1-4 keys. Pressing this button will increment the mouse mode.
- Point Location Mode: The third button represents the current point location mode. Pressing the button will cycle through modes. When this button is blue, there is no existing structure to run point location on, and running point location will begin with constructing the appropriate object. Otherwise, the button will be green and running point location will use an existing created structure. For

project.

some algorithms, this created structure may be meaningless, as is the case in the Plumb Line algorithm.

- Visualization Slider: The final element on the right side of the application is a slider which controls the visualization rate of the program. The leftmost value represents zero point location visualization, and the rightmost value represents stepwise visualization through the Step button. Other values range from 3 to 300 ms per visualization step.

3.3 Mouse Controls

Depending on which mouse mode the application is in, the mouse will do different things:

- Rotate: The mouse can click and hold around the DCEL to rotate the DCEL. If shift is held, horizontal mouse movement will rotate around the Z axis.
- Move Point: By clicking and holding on a vertex on a DCEL, that vertex can have its position moved around the viewport.
- Point Location: By clicking anywhere in the viewport, this will run the currently selected point location algorithm at the current visualization rate, with the mouse position as the query point.
- Define Face: By clicking anywhere in the viewport, this will begin a new DCEL face at the mouse position and change the mouseMode to Defining Face. When defining a face, left click will add a new point on the face and right click will close the face and return to Define Face mode.[2]

3.4 Command Line Input

- "c load [filepath]": Load will clear the viewport, most application settings, and load in the OFF file found at the input filepath, relative to where the application was run from. If the load fails, this is equivalent to clear.
- "c clear": Clear is equivalent to clicking on the clear button.
- "c reset": Reset is equivalent to running load with the existing file. Reset will reset the position and scaling of the DCEL and clear any rendering artifacts from the screen.
- "c visualize [time duration]": Visualize will set the visualization rate to the input time duration. This duration is some time-duration formatted string like "5s" for five seconds or "100ms" for one-hundred milliseconds.[3] The value input will not display on the visualization slider, is not bound to the limits of the visualization slider, and will overwrite the current value of the visualization slider.

[2]Define Face mode does not prevent the user from defining all varieties of invalid face. Invalid faces can cause point location algorithms to act unexpectedly or potentially crash. Define Face mode will attempt to connect defined faces to existing faces if existing vertices are clicked on, but its attempts to do so will cause point location algorithms to, generally, fail to recognize user-added faces. Loading off files is the recommended way to interact with the application at this time, otherwise defining simple faces that share no points.

[3]See golang.org/pkg/time for a precise description of how this is parsed.

If the slider is moved it will reinforce it's setting, overwriting the manual setting.

While point location is ongoing, most operations in the application will not run, as point location happens in a separate process which does not check back regularly for stop signals, and both manipulating the DCEL and closing the channel used to control visualization can cause point location and visualization respectively to act unexpectedly.

There are two other features of the application window which are non-interactive: in the top left, the window displays the current frames-per-second of the engine, i.e. how many times the screen is being drawn in one second. This value does not express how often objects are being updated, which is strictly capped to 60 times per second. To disable this from showing up, set "showFPS" to false in the oak.config file included with the application. The second non-interactive element is the current position of the mouse, which is in the bottom left corner of the screen.

4 Challenges

This project was largely unprepared for the issues with data degeneracy that arose through experimentation, and finding data that did not have some kind of unexpected quality the algorithms were unprepared for was surprisingly difficult.

Finding data in general was not easy, as most readily available files for geometric constructs are in structures far more complex than we need for representing geometry, including color, lighting, and texture information that we'd like to avoid having to parse. I was unable to find any file type that described both inner and outer edge chains for faces they described[4], and this in combination with the difficulty inherent in manually constructing DCELs leaves most of the code base completely unexposed to DCELs that contain faces with inner components.

Every algorithm present in the code base had significant portions of the algorithm left out of theoretical descriptions. Perhaps most egregious is the case of the trapezoidal map, where the description in de Berg's text does not describe at all how to deal with trapezoids other than the first and last trapezoid in a line of multiple trapezoids intersected by a single input edge. [3] Attempts to model after other code usually found said code untested, unreadable, or obviously leaving cases out.[5]

5 Conclusion

5.1 Next Steps

The code base is clearly incomplete in it's implementation of Kirkpatrick's point location algorithm [4], and remaining location algorithms all need tests to confirm that they are returning the proper faces. In order to properly form these tests, we need both a set of known results for known inputs and a reliable randomized way of performing fuzz tests, which will probably be through the R-Tree package used by the engine itself for mouse collision.

[4]Excluding one file type which required payment to inform me how the file was structured.

[5]This was particularly the case when dealing with Red Black trees, which inspired the intensive fuzz testing for RB-trees in this code base.

The next obvious gaps to fill are in the binary search tree package, so that AVL trees and Splay trees can be benchmarked against RB trees[6]. Structuring the point location algorithms in a way that allows the checks against visualization to be conditionally removed entirely would then enable a thorough suite of benchmarks to be created and tested on the algorithms.

5.2 Final Notes

Despite difficulties in implementing these algorithms, the demo application shows some interesting visualization of point location and the code base produced by this project is significantly developed and well-structured to base other projects off of.

For more specific documentation on exposed functions and structures in the code base, see godoc.org.

References

[1] R. Holmes, "Holmes3d." http://www.holmes3d.net/graphics/offfiles/.

[2] N. Sarnak and R. E. Tarjan, "Planar point location using persistent search trees," *Commun. ACM*, vol. 29, pp. 669–679, July 1986.

[3] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Santa Clara, CA, USA: Springer-Verlag TELOS, 3rd ed. ed., 2008.

[4] D. Kirkpatrick, "Optimal search in planar subdivisions," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 28–35, 1983.

[6] There's a convincing argument that Splay trees may perform better in an expected case on certain data sets, in that if there's any region which is chosen disproportionately more often than others, it may outperform other trees.