A Generic Tool for Interactive Visualization of Geometric Algorithms (GeoWin)

Matthias Bäsken and Stefan Näher
Fachbereich IV – Informatik
Universität Trier
54286 Trier, Germany
{baesken,naeher}@informatik.uni-trier.de

Abstract: In this paper we present GeoWin a generic tool for the interactive visualization of geometric algorithms. GeoWin is implemented as a C++ class that uses templates and object-oriented programming to be independent from the underlying geometric kernel. In this way, it can handle arbitrary types of geometric objects, as long as they provide a small set of basic operations. Currently, GeoWin is used as visualization tool for both the CGAL and LEDA projects.

Keywords: computational geometry, visualization, animation.

1 Introduction

The visualization of algorithms and data structures is very important in the design and implementation of algorithms, especially in the area of computational geometry. It is useful for the presentation of results, teaching, debugging and testing, the design of worst-case or degenerate inputs, and it often helps to discover problems and find new solutions. There are many visualization tools and systems available, e.g., GeomView[1], OpenGL[11] or GeoSheet[6]. However these systems have two serious disadvantages: they are very complex and it requires a lot of special knowledge to use them in a non-trivial way and, secondly, it is very difficult or even impossible to use them interactively. On the other hand, there have also been developed many interactive programs and systems for the animation of geometric algorithms, see [4, 13, 12] for examples. However, these systems have been designed and implemented in a very special closed environment and cannot be reused by other software systems.

In this paper we introduce GeoWin a generic tool for the interactive visualization of geometric algorithms.

By the use of templates and object-oriented programming, GeoWin is a generic visualization and animation tool that can be used for arbitrary types of geometric objects, as long as they provide a small set of basic operations. This makes GeoWin independent from any concrete geometric kernel. Currently, it is used as visualization tool for both the CGAL (see [5]) and LEDA (see [9]) projects.

The design and implementation of GeoWin was influenced by the popular LEDA graph editor GraphWin (see [9], chapter 12). Both editors follow the idea to support certain styles of programming that are often used in demo and animation programs. Examples of such programming styles are the edit & run approach and the use of result scenes which both will be discussed in section 5 of this paper.

2 The GeoWin Data Type

A GeoWin gw is an editor that maintains a collection of geometric scenes. Each geometric scene in this collection has an associated container of geometric objects whose members are displayed according to a set of visual attributes (color, line width, line style, etc.). A scene can be visible or invisible and one of the scenes in the collection
can be active. The active scene receives all editing input and thus can be manipulated through the interactive interface of GeoWin (see section 3).

Both the container type and the object type have to provide a certain functionality. The container type must implement the STL list interface ([10]), in particular, it has to provide STL-style iterators, and for any object type T the following functions and operators have to be defined before it can be used in GeoWin.

- I/O operators for streams, LEDA windows, and LEDA postscript files

\[
\begin{align*}
\text{ostream} & \quad \text{operator\langle\langle (\text{ostream}, \text{const } T \&)}; \\
\text{istream} & \quad \text{operator\rangle\rangle (\text{istream}, T \&)}; \\
\text{window} & \quad \text{operator\langle\langle (\text{window}, \text{const } T \&)}; \\
\text{window} & \quad \text{operator\rangle\rangle (\text{window}, T \&)}; \\
\text{ps\_file} & \quad \text{operator\langle\langle (\text{ps\_file}, \text{const } T \&)};
\end{align*}
\]

- translate and rotate operations

\[
\begin{align*}
\text{Translate} & \quad (T \& o, \text{double } dx, \text{double } dy); \\
\text{Rotate} & \quad (T \& o, \text{double } x, \text{double } y, \\
 & \quad \text{double } phi);
\end{align*}
\]

- basic geometric queries

\[
\begin{align*}
\text{BoundingBox} & \quad (\text{const } T \& o, \\
 & \quad \text{double } x0, \text{double } y0, \\
 & \quad \text{double } x1, \text{double } y1); \\
\text{IntersectsBox} & \quad (\text{const } T \& o, \\
 & \quad \text{double } x0, \text{double } y0, \\
 & \quad \text{double } x1, \text{double } y1);
\end{align*}
\]

Any combination of container and object type that fulfill these requirements for containers and objects, respectively, can be associated with GeoWin scene in a \texttt{gw.new.scene()} operation. Currently, GeoWin is used to visualize geometric objects and algorithm from both the CGAL ([5]) and LEDA ([9]) libraries. In this context typical examples for containers of a scene are

\[
\begin{align*}
\text{std::list} & \quad \text{CGAL::POINT_2\langle\text{cart}\rangle} > \\
\text{leda\_list} & \quad \text{rat\_circle} > C;
\end{align*}
\]

The first one uses an STL list from the standard C++ library and points from the cartesian geometry kernel of CGAL, the second example uses a LEDA list of circles from LEDA's rational geometry kernel.

\section{The Interactive Interface}

The interactive interface of a GeoWin \texttt{gw} is started by calling \texttt{gw.edit()} or \texttt{gw.edit(sc)}. The first variant opens a GeoWin with an empty set of scenes and allows the user to interactively create and activate scenes from a menu. The second variant makes the supplied scene \texttt{sc} the active scene, i.e., \texttt{sc} is the edited scene.

At the top of the main window there is a default menu bar containing menus for \texttt{File} and \texttt{Edit} operations, \texttt{Scenes} and \texttt{Window} setup, and other \texttt{Options} (Figure 1 shows a screenshot). This default menu can be changed and extended by user-defined menus and buttons.

In the same way, the following default actions associated with mouse and keyboard events occurring in the drawing area of the window can be replaced or extended by user-defined actions. The left mouse button creates a new object or scrolls the scene when the button is held down while dragging the mouse. The middle button selects a single object or a group of objects, and the right mouse button opens a context menu that allows to delete objects or to change individual attributes of objects.

Now we are ready to write our first GeoWin program. It just creates a scenes of polygons, changes the default filling color to grey and enters the interactive mode. You can see a screenshot of the program (after generating a hilbert polygon) in Figure 1.

```c
#include <LEDA/geowin.h>

int main() {
    GeoWin gw;
    list<polygon> L;
    geo_scene sc = gw.new_scene(L);
    gw.set_color(sc,black);
    gw.set_fill_color(sc,greyl);
    gw.edit(sc);
    return 0;
}
```

\section{The Programming Interface}

In this section we will discuss some of the most important operations of the data type \texttt{GeoWin}. For a complete list of operations see the CGAL manual[5].
Creating and opening a GeoWin

For the creation of a GeoWin you can use one of the following constructors

GeoWin gw(const char* label);
creates a GeoWin with frame label label.

GeoWin gw;
creates a GeoWin with default label.

gw.display(int x, int y);
opens a GeoWin with the left upper corner at position \((x,y)\).


gw.display();
opens a GeoWin at default position.

Starting the interactive mode

void gw.edit();
starts the interactive mode without specifying an active scene.

bool gw.edit(geo_scene sc);
starts the interactive mode for scene \(sc\).

Creating Scenes

Scenes are created by the \texttt{gw.new.scene()} Operation

geo_scene gw.new.scene(container<object>& C);
creates a new scene with the associated container \(C\)

geo_scene gw.new.scene(function_t func, geo_scene& sc_inp);
creates a so-called result scene whose contents is computed by applying function \(func\) to the objects of the input scene \(sc\_inp\). Result scenes will be discussed in more detail in section 5.

5 Animation with GeoWin

In this section we will describe three main approaches to interactive visualization.

5.1 Edit and Run

The \texttt{edit & run} style is very simple. It uses GeoWin only for constructing a certain input for the algorithm by entering the interactive mode \((\texttt{gw.edit()})\). Then the algorithm is applied to this input, the result is displayed, e.g., by changing the attributes of certain objects, or by simply producing a text output, and finally the program returns to the interactive mode. This loop can be interrupted by leaving \texttt{gw.edit} through the \texttt{Quit} button. The basic structure of an \texttt{edit and run} program is given below.

#include <LEDA/geowin.h>

int main() {

  list<object> L;
  geo_scene sc = gw.new_scene(L);

  while (gw.edit(sc))
    { ALGORITHM(L);
      "display or output result";
      return 0;
    }

  Figure 2 gives a more concrete example that displays the result of a closest pair algorithms by changing the color of the closest pair to red. Note that this program assumes that there is a \texttt{CLOSESTPAIR} function defined that takes a list of points (of some general type \texttt{point_t}) and writes a pair of points whose euclidean distance is minimal to the reference parameters \(a\) and \(b\).

5.2 Result Scenes

A result scene is a scene whose contents (container of objects) depends on one or more input scenes (in this extended abstract we only deal with the case of one input scene).

The dependance is defined by a function that is applied to the objects of the input scenes. The contents of the result scene is simply defined to be the output of this function. Whenever the input scene is modified, e.g., interactively (if the input scene is active), the contents of the output scene is recomputed. In this way, it is very easy to write animation programs that show the result of an algorithm on-line while the user edits...
the input to this algorithm, for example by moving points around, or by inserting or removing objects. The general structure of a program following the result scene approach is given below.

```cpp
void algo(const list<object1>& in,
           list<object2>& out);

geo_scene sc_in = gw.new_scene(L);
geo_scene sc_out = gw.new_scene(algo,
                               sc_in);
gw.edit(sc_in);
```

Figure 3 shows a more concrete example program for the result scene approach. It assumes that there is function INTERSECT that implements an algorithm for computing the point of intersection (of type point.t) for a given set of straight line segments (of type segment.t). Then we can create a the result scene that depends on an input scene sc_input of points by calling gw.new_scene(INTERSECT,sc_input). Most of the demo programs in LEDA and CGAL are written in this way. In particular, all algorithms working on an input set of points (e.g. all kinds of Voronoi and Delaunay diagrams) can be demonstrated very elegant in a single program. Then a particular algorithm or a combination of algorithms can be selected by simply changing the visibility of the scenes in the interactive interface.

Now, during the interactive mode, the user can grab and move the sweep line with the mouse, and for each triggered motion event the sweep handler function is called, with the relative distance vector of the motion. Note that the call-back function associated with move object events has a boolean return type. The result of this function is evaluated by GeoWin and controls whether the actual motion is really executed. In the sweep example we use this fact to prevent any backward motion of the sweep line.

```cpp
bool sweep_handler(GeoWin& gw, const line& sl,
                   double dx,
                   double dy)
{
  if (dx < 0) {
    // do not move backward
    return false;
  }
  "perform sweep by vector(dx,dy)"
  return true;
}
```

```cpp
int main()
{
  GeoWin gw("Sweep Demo");
  list<line> sweep_line;
  sweep_line.append(line(point(0,-100),
                        point(0,100)));

  geo_scene sc_sweep = gw.new_scene(sl);
  gw.set_color(sc_sweep,black);
  gw.set_visible(sc_sweep, true);
  gw.set_move_handler(sc_sweep, sweep_handler);
  gw.edit(sc_sweep);

  return 0;
}
```

The screenshot of Figure 4 shows a sweep line animation that uses the above described technique for the animation of Fortune's sweep algorithm for computing the Voronoi Diagram of a set of points in the plane. This animation allows to drag the sweep line across the plane while watching several different structures: the contructed Delaunay triangulation, the shore line bisector of parabolic arcs, and the circle events contained in the event queue.
6 Current and Future Work

- Support of three-dimensional geometry
  We are currently working on the support of three-dimensional objects and algorithms. Figure 5 shows the output of prototype 3d-viewer while animating a 3d convex hull algorithm. We will incorporate this viewer into GeoWin.

- Improved support for incremental algorithms
  Currently, incremental algorithms are not very well supported. We plan to associate incremental data structures directly with result scenes to make their visualization much more easy and elegant.

- Algorithm Animation
  We plan to build a small library of animation software supporting the teaching and the presentation of geometric algorithms.

References


Figure 1: GeoWin: Editing a scene of polygons.
double CLOSEST_PAIR(const list<point_t>& L, point_t& a, point_t& b);

int main()
{
    GeoWin gw;

    list<point_t> L;
    geo_scene sc = gw.new_scene(L);
    gw.set_color(sc, black);

    while (gw.edit(sc))
    {
        if (L.length() < 2) continue;

        point_t a, b;
        double dist = CLOSEST_PAIR(L, a, b);

        gw.set_color(sc, black);
        gw.set_obj_color(sc, a, red);
        gw.set_obj_color(sc, b, red);
        gw.message(string("distance: \%f", dist));
        gw.redraw();
    }
    return 0;
}

Figure 2: Edit and Run: Computing a closest pair.

void INTERSECT(const list<segment_t>&, list<point_t>&);

int main()
{
    GeoWin gw("Segment Intersection");
    list<segment_t> L;
    geo_scene sc_input = gw.new_scene(L);
    geo_scene sc_output = gw.new_scene(INTERSECT, sc_input);
    gw.set_color(sc_output, red);
    gw.set_point_style(sc_output, circle_point);
    gw.set_visible(sc_output, true);
    gw.edit(sc_input);
    return 0;
}

Figure 3: A result scene showing intersections of segments.
Figure 4: A sweep program

Figure 5: Visualization of a 3d convex hull algorithm.