CSCI–4190 Introduction to Robotic Algorithms, Spring 2004
Guide to DOLT and CGAL

The Drawable Object Library (DOLT) is a library, somewhat designed for implementing software for robotics, that simplifies writing interactive programs with two-dimensional graphics. The fundamental paradigm is that there are `drawableObject`s that represent some geometric shape but also contain information about how it is to be drawn on the screen. There are many kinds of `drawableObject`s: `drawablePoints`, `drawableSegment`s, `drawableCircle`s, and so on.

Geometric representations are therefore a fundamental part of DOLT. Although DOLT has built-in representations, it does not provide any geometric algorithms such as intersection. We will be using DOLT with the representations provided by the Computational Geometry Algorithms Library (CGAL). CGAL is very sophisticated library of geometric representations and algorithms. It is capable of much more than what we'll be using it for in this class, but it saves us the trouble of writing geometric algorithms.

Unfortunately, CGAL is complicated. This is somewhat mitigated by the way it has been incorporated into DOLT (see the `geometry.h` file), but you may still see horrendous error messages from the compiler because of its heavy use of templates.

This document is not a replacement but a supplement to the documentation for DOLT and CGAL. I would definitely recommend looking through the DOLT documentation; you may or may not want to look through the CGAL documentation. The purpose of this document is to introduce some of the "quirks" and technicalities of using CGAL with DOLT.

CGAL& geometric representations

The methods and classes described here are not comprehensive but should include everything that you'll need. For full details, see the CGAL documentation — I suggest going to the "2D and 3D Kernel" page, and then to the "Kernal classes and operations" reference page (in the rightmost column); scroll down to find the "kernel objects: two dimensional."

Points & vectors

In CGAL, a `Point` and a `Vector` are two distinct entities, even though ultimately they are both represented by the coordinates (x, y).

A `Point` can be created with its x and y coordinates: coordinates:

```
Point::Point();
Point::Point(double x, double y);
double Point::x();
double Point::y();
```

A `Vector` has the same constructors and accessors:

```
Vector::Vector();
Vector::Vector(double x, double y);
double Vector::x();
double Vector::y();
```

There are also additional constructors, such as `Vector::Vector(Segment s)`.

A `Vector` is the difference between two `Point`s. This distinction is enforced by the operators on points and vectors:

```
Vector operator- (Point p, Point q);
Point  operator+ (Point p, Vector v);
Point  operator- (Point p, Vector v);
```

If you have a point and want to find the vector from the origin to that point, you can subtract `CGAL::ORIGIN` from that point to get a vector.

Vector operations

Vectors can be combined using the stardard operations:

```
Vector Vector::operator+ (Vector w);
Vector Vector::operator- (Vector w);
Vector Vector::operator- ();         // negation
Vector Vector::operator/ (double s); // divide by scalar
Vector Vector::operator* (double s); // multiply by scalar from right
double Vector::operator* (Vector w); // dot product
```

There is no built-in function for the length of a vector, but of course you can take the square root of the dot product of the vector with itself.

A standard trick for getting a vector 90 degrees clockwise from a two-dimensional vector $\vec{v}$ is to take the cross product $\hat{z} \times \vec{v}$. However, CGAL provides the function:

```
Vector Vector::perpendicular(Orientation o);
```

where you can pass `CGAL::CLOCKWISE` or `CGAL::COUNTERCLOCKWISE` as the argument.

The cross product is still useful however for the property: $|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}| \sin\theta$. However, CGAL only provides a cross product function for three-dimensional vectors. For two dimensional vectors, recall that $|\vec{a} \times \vec{b}| = a_x b_y - a_y b_x$.

Numerical errors

Although CGAL does have exact representations, we are using an inexact representation (`doubles`) to store numbers. When you do calculations, especially those that use `double` approximations to irrational numbers, there will be errors due to rounding or quantization.

Therefore, it is never a good idea to check whether, for example, two `Points` are equal using the "`==`" operator. Instead, you should check whether the distance between the two points is less than some threshold. That threshold depends upon the magnitude of the numbers that you are using. When we are using numbers that represent distances on the order of, say, 10 meters, then `1e-4` should be sufficient, though I might tend to use something slightly smaller like `1e-6`.

It is these sorts of issues that make writing robust algorithms for geometric computation difficult.

Segments

A segment is a directed line segment from a "source" point to a "target" point:

```
Segment(Point source, Point target);
Point source();
Point target();
Vector to_vector(); // returns target - source
```

Bounding boxes

A bounding box is an axis-aligned rectangle. There are the standard constructors and accessors:

```
Bbox();
Bbox(double x_min, double y_min, double x_max, double y_max);
double xmin();
double ymin();
double xmax();
double ymax();
```

You can get a bounding box of any CGAL object by using the `bbox()` method:

```
Bbox Point::bbox();
Bbox Vector::bbox();
Bbox Segment::bbox();
Bbox Circle::bbox();
Bbox Polygon::bbox();
```

Note that `Rectangle`, `Square`, `Ellipse`, `Arc`, and `lineStrip`, are not CGAL classes and therefore do not have a `bbox()` method.

Two bounding boxes can be "added" to get a bounding box that bounds them both:

```
Bbox Bbox::operator+ (Bbox c);
```

Rectangles

The `Rectangle` class is provided as part of DOLT; it is not a CGAL class. The `Rectangle` class is a subclass of `Bbox`, so it is axis-aligned. (To draw a non-axis-aligned rectangle, you must use a `Polygon`.) `Rectangle` is the parent class of `drawableRectangle`. (There is no `drawableBbox`.) It provides the following constructors and methods:

```
Rectangle();
Rectangle(Point center, double width, double height);
Rectangle(Point lowerLeft, Point upperRight);
Rectangle(Bbox b);
Point getMinPoint(); // return lower left point
Point getMaxPoint(); // return upper right point
```

Squares

The `Square` class is provided as part of DOLT; it is not a CGAL class. The `Square` class is a subclass of `Rectangle`, so it is axis-aligned. `Square` is the parent class of `drawable-Square`. It provides the following constructors

```
Square();
Square(Point lowerLeft, double sideLen);
```

Circles

CGAL's `Circle` class provides the following methods:

```
Circle(Point center, double squared_radius);
Point center();
double squared_radius();
```

Ellipses

The `Ellipse` class is provided as part of DOLT; it is not a CGAL class. It provides the following constructors and methods:

```
Ellipse();
Ellipse(Point p, double orientation,
        double major_rad_len, double minor_rad_len);
void setPos(Point p);
void setOrientation(double o);
void setMajorRadius(double r);
void setMinorRadius(double r);
Point getPos();
double getOrientation();
double getMajorRadius();
double getMinorRadius();
```

Arc

The `Arc` class is provided as part of DOLT; it is not a CGAL class. It represents a circular arc and is therefore a subclass of `Circle`. It provides the following constructors and methods:

```
Arc()
Arc(Point center, double squared_radius,
    double minAngle, double maxAngle);
double getMinAngle();
double getMaxAngle();
void setMinAngle(double m);
void setMaxAngle(double m);
```

Both angles must be specified in radians, measured counterclockwise from the x axis. The arc is drawn from the "minAngle" to the "maxAngle".

Line strips

The `lineStrip` class is derived from `std::vector<Point>`. It is used to represent a connected sequence of line segments.

Polygons

A polygon represents a closed chain of line segments. It is fundamentally a list (actually, a `vector`) of points. The CGAL documentation for polygons can be found under the "Basic Library" on the "Polygons and polygon operations" reference page.

Points can be stored in a polygon using the `push_back` method:

```
Polygon P;
double x[5], y[5];
getPoints(x,y);
for (int i=0; i<5; ++i)
  P.push_back(Point(x[i], y[i]));
```

Points must be stored in *counterclockwise* order so that the interior of the polygon is on the left side as you visit the points on the boundary. This is very important for the intersection routines to work properly!

You can find out how many points (or edges) are in a polygon by using the method:

```
int size();
```

Since the underlying container is a `vector`, you can use random access methods for the edges and vertices:

```
const Point operator[] (int i); // return the i-th vertex
const Point vertex(int i);      // return the i-th vertex
Segment edge(int i);            // return the i-th edge
```

However, for performing some operation on all the points or edges of a polygon, it is more natural to use an iterator or a circulator.

Here is an example of a vertex iterator:

```
void printPolygon(const Polygon& P)
{
  cout << "The polygon consists of the points:" << endl;
  for (Polygon::Vertex_const_iterator k = P.vertices_begin();
       k != P.vertices_end(); ++k)
    cout << "  (" << (*k).x() << ", " << (*k).y() << ")" << endl;
}
```

You can also iterate using the edges of the polygon:

```
void printEdgeLengths(const Polygon& P)
{
  cout << "The lengths of the polygon edges are:" << endl;
  for (Polygon::Edge_const_iterator k = P.edges_begin();
       k != P.edges_end(); ++k) {
    // (*k) is a Segment
    Point s = (*k).source();
    Point t = (*k).target();
    Vector v = t - s;
    cout << "  " << sqrt(v * v) << endl;
  }
}
```

Circulators are like iterators except that they keep going around the polygon as you increment them. You can get circulators using the methods:

```
Vertex_const_circulator Polygon::vertices_circulator();
Edge_const_circulator   Polygon::edges_circulator();
```

Miscellaneous

Every CGAL object (and DOLT object derived from a CGAL object) has an output operator << defined. These operators simply print a list of numbers. For example:

```
Point P(3.5, 2.6);
Segment S(P,Point(4.2, 3.7));
cout << "P is: " << P << endl << "S is: " << S << endl;
```

produces:

```
P is: 3.5 2.6
S is: 3.5 2.6 4.2 3.7
```

Using DOLT

Dolt provides "drawable" versions of all the geometric representations above:

```
drawablePoint
drawableSegment
drawableLineStrip
drawablePolygon
drawableEllipse
drawableCircle
drawableRectangle
drawableSquare
drawableArc
drawableWedge
```

All drawable objects are derived from their corresponding geometric representation (except `drawableWedge` which is derived from `Arc`).

All drawable objects also inherit from one of the following classes:

```
class drawableObject;
class drawableLineObject     : public drawableObject;
class drawableEnclosedObject : public drawableLineObject;
```

The basic paradigm for DOLT is to create drawable objects, and then add them to an `objectList`. The `objectList`s allow you to group drawable objects together. The `objectList`s are then given to an instance of `Graphics` in order to be displayed.

`drawableObject`s are drawn in the order according to the `objectList`, and the order in which you `objectList`s are given to DOLT, using the `Graphics::addObjectList()` method, is the order in which the lists will be drawn.

Note that you may have an `objectList` and need to access specific attributes of each drawable object. Here is (as far as I know) the correct way to do this:

```
void printObjectList(const objectList &ol)
{
  for (objectList::iterator k = ol.begin();
       k != ol.end(); k++) {
    Polygon *pg;
    if ((pg = dynamic_cast<Polygon*>(*k)) != 0)
      cout << "Here's a polygon: " << *pg << endl;
    else {
      Point pt;
      if ((pt = dynamic_cast<Point*>(*k)) != 0)
        cout << "Here's a point:" << *pt << endl;
      else
        cout << "Not a point or polygon!" << endl;
    }
  }
}
```

The `dynamic_cast<...>(...)` operator will properly navigate the class hierarchy; if it cannot make the type conversion, it will return a null pointer.

Colors

For a list of predefined colors, see the `color.cpp` file.